“I picked up a copy of JavaScript by Example over the weekend and wanted to thank you for putting out a book that makes JavaScript easy to understand. I’ve been a developer for several years now and JS has always been the ‘monster under the bed,’ so to speak. Your book has answered a lot of questions I’ve had about the inner workings of JS but was afraid to ask. Now all I need is a book that covers Ajax and Coldfusion. Thanks again for putting together an outstanding book.”

—Chris Gomez, Web services manager, Zunch Worldwide, Inc.

“I have been reading your UNIX® Shells by Example book, and I must say, it is brilliant. Most other books do not cover all the shells, and when you have to constantly work in an organization that uses tcsh, bash, and korn, it can become very difficult. However, your book has been indispensable to me in learning the various shells and the differences between them...so I thought I'd email you, just to let you know what a great job you have done!”

—Farogh-Ahmed Usmani, B.Sc. (Honors), M.Sc., DIC, project consultant (Billing Solutions), Comverse

“I have been learning Perl for about two months now; I have a little shell scripting experience but that is it. I first started with Learning Perl by O’Reilly. Good book but lacking on the examples. I then went to Programming Perl by Larry Wall, a great book for intermediate to advanced, didn’t help me much beginning Perl. I then picked up Perl by Example, Third Edition—this book is a superb, well-written programming book. I have read many computer books and this definitely ranks in the top two, in my opinion. The examples are excellent. The author shows you the code, the output of each line, and then explains each line in every example.”

—Dan Patterson, software engineer, GuideWorks, LLC

“Ellie Quigley has written an outstanding introduction to Perl, which I used to learn the language from scratch. All one has to do is work through her examples, putz around with them, and before long, you’re relatively proficient at using the language. Even though I’ve graduated to using Programming Perl by Wall et al., I still find Quigley’s book a most useful reference.”

—Casey Machula, support systems analyst, Northern Arizona University, College of Health and Human Services
“When I look at my bookshelf, I see eleven books on Perl programming. *Perl by Example, Third Edition*, isn’t on the shelf; it sits on my desk, where I use it almost daily. When I bought my copy I had not programmed in several years and my programming was mostly in COBOL so I was a rank beginner at Perl. I had at that time purchased several popular books on Perl but nothing that really put it together for me. I am still no pro, but my book has many dog-eared pages and each one is a lesson I have learned and will certainly remember.

“I still think it is the best Perl book on the market for anyone from a beginner to a seasoned programmer using Perl almost daily.”

—Bill Maples, network design tools and automations analyst,  
*Fidelity National Information Services*

“We are rewriting our intro to OS scripting course and selected your text for the course. *UNIX® Shells by Example* is an exceptional book. The last time we considered it was a few years ago (second edition). The debugging and system administrator chapters at the end nailed it for us.”

—Jim Leone, Ph.D., professor and chair, Information Technology,  
*Rochester Institute of Technology*

“Quigley’s *PHP and MySQL by Example* acknowledges a major usage of PHP. To write some kind of front end user interface program that hooks to a back end MySQL database. Both are free and open source, and the combination has proved popular. Especially where the front end involves making an HTML web page with embedded PHP commands.

“Not every example involves both PHP and MySQL. Though all examples have PHP. Many demonstrate how to use PHP inside an HTML file. Like writing user-defined functions, or nesting functions. Or making or using function libraries. The functions are a key idea in PHP, that take you beyond the elementary syntax. Functions also let you gainfully use code by other PHP programmers. Important if you are part of a coding group that has to divide up the programming effort in some manner.”

—Dr. Wes Boudville, CTO,  
*Metaswarm Inc.*
Perl by Example

Fifth Edition
This page intentionally left blank
# Contents

*Preface*  xxv

1  **The Practical Extraction and Report Language**  1  
1.1  What Is Perl?  1  
1.2  What Is an Interpreted Language?  2  
1.3  Who Uses Perl?  3  
  1.3.1  Which Perl?  4  
  1.3.2  What Are Perl 6, Rakudo Perl, and Parrot?  4  
1.4  Where to Get Perl  6  
  1.4.1  CPAN (cpan.org)  6  
  1.4.2  Downloads and Other Resources for Perl (perl.org)  7  
  1.4.3  ActivePerl (activestate.com)  8  
  1.4.4  What Version Do I Have?  9  
1.5  Perl Documentation  9  
  1.5.1  Where to Find the Most Complete Documentation from Perl  9  
  1.5.2  Perl `man` Pages  10  
  1.5.3  Online Documentation  12  
1.6  What You Should Know  13  
1.7  What’s Next?  13  

2  **Perl Quick Start**  15  
2.1  Quick Start, Quick Reference  15  
  2.1.1  A Note to Programmers  15  
  2.1.2  A Note to Non-Programmers  15  
  2.1.3  Perl Syntax and Constructs  15  
    Regular Expressions  28  
    Passing Arguments at the Command Line  29
3 Perl Scripts 33

3.1 Getting Started 33
  3.1.1 Finding a Text Editor 34
  3.1.2 Naming Perl Scripts 35
  3.1.3 Statements, Whitespace, and Linebreaks 35
  3.1.4 Strings and Numbers 36

3.2 Filehandles 37

3.3 Variables (Where to Put Data) 37
  3.3.1 What Is Context? 38
  3.3.2 Comments 38
  3.3.3 Perl Statements 39
  3.3.4 Using Perl Built-in Functions 39
  3.3.5 Script Execution 40

3.4 Summing It Up 42
  3.4.1 What Kinds of Errors to Expect 43

3.5 Perl Switches 44
  3.5.1 The -e Switch (Quick Test at the Command Line) 45
  3.5.2 The -c Switch (Check Syntax) 46
  3.5.3 The -w Switch (Warnings) 46

3.6 What You Should Know 47

3.7 What’s Next? 47
  EXERCISE 3 Getting with It Syntactically 48

4 Getting a Handle on Printing 49

4.1 The Special Filehandles STDOUT, STDIN, STDERR 49

4.2 Words 51

4.3 The print Function 51
  4.3.1 Quotes Matter! 52
    Double Quotes 53
    Single Quotes 54
    Backquotes 54
    Perl's Alternative Quotes 55
Contents

4.3.2 Literals (Numeric, String, and Special) 59
   Numeric Literals 60
   String Literals 61
   Special Literals 63
4.3.3 Printing Without Quotes—The here document 66
   here documents and CGI 67

4.4 Fancy Formatting with the printf Function 69
   4.4.1 Saving Formatting with the sprintf Function 73
   4.4.2 The No Newline say Function 73

4.5 What Are Pragmas? 74
   4.5.1 The feature Pragma 74
   4.5.2 The warnings Pragma 75
   4.5.3 The diagnostics Pragma 76
   4.5.4 The strict Pragma and Words 77

4.6 What You Should Know 78

4.7 What’s Next? 79

EXERCISE 4 A String of Perls 79

5 What’s In a Name? 81
   5.1 More About Data Types 81
      5.1.1 Basic Data Types (Scalar, Array, Hash) 81
   5.1.2 Package, Scope, Privacy, and Strictness 82
      Package and Scope 82
      5.1.3 Naming Conventions 85
      5.1.4 Assignment Statements 86
   5.2 Scalars, Arrays, and Hashes 87
      5.2.1 Scalar Variables 88
         Assignment 88
         The defined Function 89
         The undef Function 89
         The $_ Scalar Variable 90
      5.2.2 Arrays 91
         Assignment 92
         Output and Input Special Variables ($, and "$) 93
         Array Size 94
         The Range Operator and Array Assignment 95
         Accessing Elements 95
         Looping Through an Array with the foreach Loop 97
         Array Copy and Slices 98
         Multidimensional Arrays—Lists of Lists 99
5.2.3 Hashes—Unordered Lists  99
Assignment  100
Accessing Hash Values  101
Hash Slices  102
Removing Duplicates from a List Using a Hash  103
5.2.4 Complex Data Structures  104

5.3 Array Functions  105
5.3.1 Adding Elements to an Array  105
The push Function  105
The unshift Function  106
5.3.2 Removing and Replacing Elements  106
The delete Function  106
The splice Function  107
The pop Function  109
The shift Function  110
5.3.3 Deleting Newlines  111
The chop and chomp Functions (with Lists)  111
5.3.4 Searching for Elements and Index Values  112
The grep Function  112
5.3.5 Creating a List from a Scalar  114
The split Function  114
5.3.6 Creating a Scalar from a List  118
The join Function  118
5.3.7 Transforming an Array  119
The map Function  119
5.3.8 Sorting an Array  121
The sort Function  121
5.3.9 Checking the Existence of an Array Index Value  124
The exists Function  124
5.3.10 Reversing an Array  125
The reverse Function  125

5.4 Hash (Associative Array) Functions  125
5.4.1 The keys Function  125
5.4.2 The values Function  126
5.4.3 The each Function  128
5.4.4 Removing Duplicates from a List with a Hash  129
5.4.5 Sorting a Hash by Keys and Values  130
Sort Hash by Keys in Ascending Order  130
Sort Hash by Keys in Reverse Order  131
Sort Hash by Keys Numerically  132
Numerically Sort a Hash by Values in Ascending Order  133
Contents

 Numerically Sort a Hash by Values in Descending Order  134
 5.4.6 The delete Function  135
 5.4.7 The exists Function  136
 5.4.8 Special Hashes  137
     The %ENV Hash  137
     The %SIG Hash  138
     The %INC Hash  139
 5.4.9 Context Revisited  139
 5.5 What You Should Know  140
 5.6 What’s Next?  141
    EXERCISE 5  The Funny Characters  141

6 Where’s the Operator?  145
 6.1 About Perl Operators—More Context  145
    6.1.1 Evaluating an Expression  147
 6.2 Mixing Types  148
 6.3 Precedence and Associativity  149
    6.3.1 Assignment Operators  151
    6.3.2 Boolean  153
    6.3.3 Relational Operators  154
       Numeric  154
       String  155
    6.3.4 Conditional Operators  156
    6.3.5 Equality Operators  157
       Numeric  157
       String  159
    6.3.6 The Smartmatch Operator  160
    6.3.7 Logical Operators (Short-Circuit Operators)  162
    6.3.8 Logical Word Operators  164
    6.3.9 Arithmetic Operators and Functions  166
       Arithmetic Operators  166
       Arithmetic Functions  167
    6.3.10 Autoincrement and Autodecrement Operators  172
    6.3.11 Bitwise Logical Operators  173
       A Little Bit About Bits  173
       Bitwise Operators  174
    6.3.12 Range Operator  175
    6.3.13 Special String Operators and Functions  176
 6.4 What You Should Know  178
 6.5 What’s Next?  179
    EXERCISE 6  Operator, Operator  179
7 If Only, Unconditionally, Forever  181
  7.1 Control Structures, Blocks, and Compound Statements  182
    7.1.1 Decision Making—Conditional Constructs  183
      if and unless Statements  183
      The if Construct  183
      The if/else Construct  184
      The if/elsif/else Construct  185
      The unless Construct  186
  7.2 Statement Modifiers and Simple Statements  188
    7.2.1 The if Modifier  188
    7.2.2 The unless Modifier  189
  7.3 Repetition with Loops  190
    7.3.1 The while Loop  190
    7.3.2 The until Loop  192
    7.3.3 The do/while and do/until Loops  194
    7.3.4 The for Loop (The Three-Part Loop)  196
    7.3.5 The foreach (for) Loop  198
  7.4 Looping Modifiers  202
    7.4.1 The while Modifier  202
    7.4.2 The foreach Modifier  203
    7.4.3 Loop Control  204
      Labels  204
      The redo and goto Statements  205
      Nested Loops and Labels  208
      The continue Statement  210
    7.4.4 The switch Statement (given/when)  212
      The switch Feature (given/when/say)  214
  7.5 What You Should Know  217
  7.6 What’s Next?  217
    EXERCISE 7 What Are Your Conditions?  218

8 Regular Expressions—Pattern Matching  219
  8.1 What Is a Regular Expression?  219
    8.1.1 Why Do We Need Regular Expressions?  220
  8.2 Modifiers and Simple Statements with Regular Expressions  221
    8.2.1 Pattern Binding Operators  222
    8.2.2 The DATA Filehandle  223
  8.3 Regular Expression Operators  225
    8.3.1 The m Operator and Pattern Matching  225
      The g Modifier—Global Match  229
      The i Modifier—Case Insensitivity  230
9 Getting Control—Regular Expression Metacharacters 245

9.1 The RegExLib.com Library 245

9.2 Regular Expression Metacharacters 247

9.2.1 Metacharacters for Single Characters 251

The Dot Metacharacter 251
The s Modifier—The Dot Metacharacter and the Newline 252
The Character Class 253
The POSIX Bracket Expressions 257

9.2.2 Whitespace Metacharacters 258

9.2.3 Metacharacters to Repeat Pattern Matches 261

The Greed Factor 261
Metacharacters That Turn off Greediness 267
Anchoring Metacharacters 269
The m Modifier 271
Alternation 273
Grouping or Clustering 273
Remembering or Capturing 276
Turning off Greed 280
Turning off Capturing 281
Metacharacters That Look Ahead and Behind 282

9.2.4 The tr or y Operators 285

The d Delete Option 288
The c Complement Option 289
The s Squeeze Option 290

9.3 Unicode 290

9.3.1 Perl and Unicode 291

9.4 What You Should Know 294

9.5 What’s Next? 295

EXERCISE 9 And the Search Goes On . . . 295
10 Getting a Handle on Files 297

10.1 The User-Defined Filehandle 297
  10.1.1 Opening Files—The open Function 297
  10.1.2 Opening for Reading 298
     Closing the Filehandle 299
     The die Function 299
  10.1.3 Reading from a File and Scalar Assignment 300
     The Filehandle and $_ 300
     The Filehandle and a User-Defined Scalar Variable 301
     “Slurping” a File into an Array 302
     Using map to Create Fields from a File 303
     Slurping a File into a String with the read Function 304
  10.1.4 Loading a Hash from a File 306

10.2 Reading from STDIN 307
  10.2.1 Assigning Input to a Scalar Variable 307
  10.2.2 The chop and chomp Functions 308
  10.2.3 The read Function 309
  10.2.4 The getc Function 310
  10.2.5 Assigning Input to an Array 311
  10.2.6 Assigning Input to a Hash 312
  10.2.7 Opening for Writing 313
  10.2.8 Win32 Binary Files 315
  10.2.9 Opening for Appending 316
  10.2.10 The select Function 317
  10.2.11 File Locking with flock 317
  10.2.12 The seek and tell Functions 319
     The seek Function 319
     The tell Function 322
  10.2.13 Opening for Reading and Writing 324
  10.2.14 Opening for Anonymous Pipes 326
     The Output Filter 327
     Sending the Output of a Filter to a File 329
     Input Filter 330

10.3 Passing Arguments 333
  10.3.1 The @ARGV Array 333
  10.3.2 ARGV and the Null Filehandle 334
  10.3.3 The eof Function 338
  10.3.4 The -i Switch—Editing Files in Place 340

10.4 File Testing 342

10.5 What You Should Know 344

10.6 What’s Next? 344

EXERCISE 10 Getting a Handle on Things 345
11 How Do Subroutines Function? 347
   11.1 Subroutines/Functions 348
      11.1.1 Defining and Calling a Subroutine 349
         Forward Declaration 351
         Scope of Variables 351
   11.2 Passing Arguments and the @_ Array 352
      11.2.1 Call-by-Reference and the @_ Array 353
      11.2.2 Assigning Values from @_ 353
         Passing a Hash to a Subroutine 355
      11.2.3 Returning a Value 356
      11.2.4 Scoping Operators: local, my, our, and state 357
         The local Operator 358
         The my Operator 358
      11.2.5 Using the strict Pragma (my and our) 361
         The state Feature 363
      11.2.6 Putting It All Together 364
      11.2.7 Prototypes 365
      11.2.8 Context and Subroutines 366
         The wantarray Function and User-Defined Subroutines 367
      11.2.9 Autoloading 369
      11.2.10 BEGIN and END Blocks (Startup and Finish) 371
      11.2.11 The subs Function 371
   11.3 What You Should Know 373
   11.4 What's Next? 373
      EXERCISE 11 I Can't Seem to Function Without Subroutines 374

12 Does This Job Require a Reference? 377
   12.1 What Is a Reference? 377
      12.1.1 Hard References 378
         The Backslash Operator 379
         Dereferencing the Pointer 379
      12.1.2 References and Anonymous Variables 382
         Anonymous Arrays 382
         Anonymous Hashes 383
      12.1.3 Nested Data Structures 383
         Using Data::Dumper 384
         Array of Lists 385
         Array of Hashes 387
         Hash of Hashes 389
      12.1.4 More Nested Structures 391
14 Bless Those Things! (Object-Oriented Perl) 447

14.1 The OOP Paradigm 447
  14.1.1 What Are Objects? 447
  14.1.2 What Is a Class? 448
  14.1.3 Some Object-Oriented Lingo 449

14.2 Perl Classes, Objects, and Methods—Relating to the Real World 450
  14.2.1 The Steps 451
  14.2.2 A Complete Object-Oriented Perl Program 451
    A Perl Package Is a Class 453
    A Perl Class 453
  14.2.3 Perl Objects 454
    References 454
    The Blessing 454
  14.2.4 Methods Are Perl Subroutines 456
    Definition 456
    Types of Methods 457
    Invoking Methods 457
    Creating the Object with a Constructor 458
    Creating the Instance Methods 460
    Invoking the Methods (User Interaction) 462
  14.2.5 Creating an Object-Oriented Module 464
    Passing Arguments to Methods 466
    Passing Parameters to Instance Methods 467
    Named Parameters and Data Checking 470
  14.2.6 Polymorphism and Runtime Binding 472
  14.2.7 Destructors and Garbage Collection 476

14.3 Anonymous Subroutines, Closures, and Privacy 478
  14.3.1 What Is a Closure? 478
  14.3.2 Closures and Objects 481

14.4 Inheritance 484
  14.4.1 The @ISA Array and Calling Methods 484
  14.4.2 $AUTOLOAD, sub AUTOLOAD, and UNIVERSAL 486
  14.4.3 Derived Classes 489
  14.4.4 Multiple Inheritance and Roles with Moose 496
  14.4.5 Overriding a Parent Method and the SUPER Pseudo Class 499

14.5 Plain Old Documentation—Documenting a Module 501
  14.5.1 pod Files 502
  14.5.2 pod Commands 504
    Checking Your pod Commands 504
14.5.3 How to Use the pod Interpreters  506
14.5.4 Translating pod Documentation into Text  506
14.5.5 Translating pod Documentation into HTML  507
14.6 Using Objects from the Perl Library  508
14.6.1 An Object-Oriented Module from the Standard Perl Library  509
14.6.2 Using a Module with Objects from the Standard Perl Library  511
14.7 What You Should Know  512
14.8 What’s Next?  513
EXERCISE 14 What’s the Object of This Lesson?  513

15 Perl Connects with MySQL  519
15.1 Introduction  519
15.2 What Is a Relational Database?  520
15.2.1 Client/Server Databases  521
15.2.2 Components of a Relational Database  522
The Database Server  523
The Database  523
Tables  523
Records and Fields  524
The Database Schema  527
15.2.3 Talking to the Database with SQL  528
English-like Grammar  528
Semicolons Terminate SQL Statements  529
Naming Conventions  529
Reserved Words  529
Case Sensitivity  529
The Result Set  530
15.3 Getting Started with MySQL  530
15.3.1 Installing MySQL  531
15.3.2 Connecting to MySQL  532
Editing Keys at the MySQL Console  533
Setting a Password  533
15.3.3 Graphical User Tools  534
The MySQL Query Browser  534
The MySQL Privilege System  536
15.3.4 Finding the Databases  537
Creating and Dropping a Database  538
15.3.5 Getting Started with Basic Commands  539
Creating a Database with MySQL  539
Selecting a Database with MySQL  541
Creating a Table in the Database  541
Data Types  541
### Adding Another Table with a Primary Key
543

### Inserting Data into Tables
544

### Selecting Data from Tables—The SELECT Command
546

#### Selecting by Columns
546

#### Selecting All Columns
547

#### The WHERE Clause
548

#### Sorting Tables
550

#### Joining Tables
551

#### Deleting Rows
552

#### Updating Data in a Table
553

#### Altering a Table
554

#### Dropping a Table
555

#### Dropping a Database
555

### 15.4 What Is the Perl DBI? 556

#### 15.4.1 Installing the DBD Driver
556

- **Without the DBD-MySQL with PPM** 556
- **Using PPM with Linux** 558
- **Installing the DBD::mysql Driver from CPAN** 558

#### 15.4.2 The DBI Class Methods
558

#### 15.4.3 How to Use DBI
560

#### 15.4.4 Connecting to and Disconnecting from the Database
561

- **The connect() Method** 561
- **The disconnect() Method** 563

#### 15.4.5 Preparing a Statement Handle and Fetching Results
563

- **Select, Execute, and Dump the Results** 563
- **Select, Execute, and Fetch a Row As an Array** 564
- **Select, Execute, and Fetch a Row As a Hash** 566

#### 15.4.6 Getting Error Messages
567

- **Automatic Error Handling** 567
- **Manual Error Handling** 567

#### 15.4.7 The ? Placeholder and Parameter Binding
571

- **Binding Parameters in the execute Statement** 571
- **Binding Parameters and the bind_param() Method** 574

#### 15.4.8 Handling Quotes
576

#### 15.4.9 Cached Queries
577

### 15.5 Statements That Don’t Return Anything 579

#### 15.5.1 The do() Method
579

- **Adding Entries** 579
- **Deleting Entries** 580
- **Updating Entries** 581

### 15.6 Transactions 583

#### 15.6.1 Commit and Rollback
583
### 16 Interfacing with the System 595

#### 16.1 System Calls 595

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1.1 Directories and Files</td>
<td>597</td>
</tr>
<tr>
<td>Backslash Issues</td>
<td>597</td>
</tr>
<tr>
<td>The File::Spec Module</td>
<td>598</td>
</tr>
<tr>
<td>16.1.2 Directory and File Attributes</td>
<td>599</td>
</tr>
<tr>
<td>UNIX</td>
<td>599</td>
</tr>
<tr>
<td>Windows</td>
<td>600</td>
</tr>
<tr>
<td>16.1.3 Finding Directories and Files</td>
<td>603</td>
</tr>
<tr>
<td>16.1.4 Creating a Directory—The <code>mkdir</code> Function</td>
<td>605</td>
</tr>
<tr>
<td>UNIX</td>
<td>605</td>
</tr>
<tr>
<td>Windows</td>
<td>605</td>
</tr>
<tr>
<td>16.1.5 Removing a Directory—The <code>rmdir</code> Function</td>
<td>607</td>
</tr>
<tr>
<td>16.1.6 Changing Directories—The <code>chdir</code> Function</td>
<td>607</td>
</tr>
<tr>
<td>16.1.7 Accessing a Directory via the Directory Filehandle</td>
<td>608</td>
</tr>
<tr>
<td>The <code>opendir</code> Function</td>
<td>609</td>
</tr>
<tr>
<td>The <code>readdir</code> Function</td>
<td>609</td>
</tr>
<tr>
<td>The <code>closedir</code> Function</td>
<td>610</td>
</tr>
<tr>
<td>The <code>telldir</code> Function</td>
<td>611</td>
</tr>
<tr>
<td>The <code>rewinddir</code> Function</td>
<td>611</td>
</tr>
<tr>
<td>The <code>seekdir</code> Function</td>
<td>611</td>
</tr>
<tr>
<td>16.1.8 Permissions and Ownership</td>
<td>612</td>
</tr>
<tr>
<td>UNIX</td>
<td>612</td>
</tr>
<tr>
<td>Windows</td>
<td>612</td>
</tr>
<tr>
<td>The <code>chmod</code> Function (UNIX)</td>
<td>614</td>
</tr>
<tr>
<td>The <code>chmod</code> Function (Windows)</td>
<td>614</td>
</tr>
<tr>
<td>The <code>chown</code> Function (UNIX)</td>
<td>615</td>
</tr>
<tr>
<td>The <code>umask</code> Function (UNIX)</td>
<td>616</td>
</tr>
<tr>
<td>16.1.9 Hard and Soft Links</td>
<td>616</td>
</tr>
<tr>
<td>UNIX</td>
<td>616</td>
</tr>
<tr>
<td>Windows</td>
<td>617</td>
</tr>
<tr>
<td>The <code>link</code> and <code>unlink</code> Functions (UNIX)</td>
<td>618</td>
</tr>
<tr>
<td>The <code>symlink</code> and <code>readlink</code> Functions (UNIX)</td>
<td>619</td>
</tr>
<tr>
<td>16.1.10 Renaming Files</td>
<td>620</td>
</tr>
<tr>
<td>The <code>rename</code> Function (UNIX and Windows)</td>
<td>620</td>
</tr>
<tr>
<td>16.1.11 Changing Access and Modification Times</td>
<td>620</td>
</tr>
</tbody>
</table>
16.4 Error Handling  664
   16.4.1 The Carp Module  665
      The die Function  665
      The warn Function  666
   16.4.2 The eval Function  666

16.5 Signals and the %SIG Hash  669
   16.5.1 Catching Signals  669
   16.5.2 Sending Signals to Processes  670
      The kill Function  670
      The alarm Function  671
      The sleep Function  672
   16.5.3 Attention, Windows Users!  672

16.6 What You Should Know  673

EXERCISE 16  Interfacing with the System  674

A  Perl Built-ins, Pragmas, Modules, and the Debugger  675
   A.1 Perl Functions  675
   A.2 Special Variables  705
   A.3 Perl Pragmas  708
   A.4 Perl Modules  710
   A.5 Command-Line Switches  716
   A.6 Debugger  718
      A.6.1 Getting Information About the Debugger  718
      A.6.2 The Perl Debugger  718
      A.6.3 Entering and Exiting the Debugger  719
      A.6.4 Debugger Commands  720

B  SQL Language Tutorial  723
   B.1 What Is SQL?  723
      B.1.1 Standardizing SQL  724
      B.1.2 Executing SQL Statements  724
         The MySQL Query Browser  725
      B.1.3 About SQL Commands/Queries  725
         English-like Grammar  725
         Semicolons Terminate SQL Statements  726
         Naming Conventions  727
         Reserved Words  727
         Case Sensitivity  727
         The Result Set  728
      B.1.4 SQL and the Database  728
         The show databases Command  728
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1.5</td>
<td>SQL Database Tables</td>
<td>729</td>
</tr>
<tr>
<td>B.2</td>
<td>SQL Data Manipulation Language (DML)</td>
<td>731</td>
</tr>
<tr>
<td>B.2.1</td>
<td>The SELECT Command</td>
<td>731</td>
</tr>
<tr>
<td></td>
<td>Select Specified Columns</td>
<td>732</td>
</tr>
<tr>
<td></td>
<td>Select All Columns</td>
<td>732</td>
</tr>
<tr>
<td></td>
<td>The SELECT DISTINCT Statement</td>
<td>733</td>
</tr>
<tr>
<td></td>
<td>Limiting the Number of Lines in the Result Set with LIMIT</td>
<td>734</td>
</tr>
<tr>
<td></td>
<td>The WHERE Clause</td>
<td>736</td>
</tr>
<tr>
<td></td>
<td>Using Quotes</td>
<td>737</td>
</tr>
<tr>
<td></td>
<td>Using the = and &lt;&gt; Operators</td>
<td>737</td>
</tr>
<tr>
<td></td>
<td>What Is NULL?</td>
<td>737</td>
</tr>
<tr>
<td></td>
<td>The &gt; and &lt; Operators</td>
<td>739</td>
</tr>
<tr>
<td></td>
<td>The AND and OR Operators</td>
<td>740</td>
</tr>
<tr>
<td></td>
<td>The LIKE and NOT LIKE Conditions</td>
<td>741</td>
</tr>
<tr>
<td></td>
<td>Pattern Matching and the % Wildcard</td>
<td>741</td>
</tr>
<tr>
<td></td>
<td>The _ Wildcard</td>
<td>743</td>
</tr>
<tr>
<td></td>
<td>The BETWEEN Statement</td>
<td>743</td>
</tr>
<tr>
<td></td>
<td>Sorting Results with ORDER BY</td>
<td>744</td>
</tr>
<tr>
<td>B.2.2</td>
<td>The INSERT Command</td>
<td>745</td>
</tr>
<tr>
<td>B.2.3</td>
<td>The UPDATE Command</td>
<td>746</td>
</tr>
<tr>
<td>B.2.4</td>
<td>The DELETE Statement</td>
<td>747</td>
</tr>
<tr>
<td>B.3</td>
<td>SQL Data Definition Language</td>
<td>748</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Creating the Database</td>
<td>748</td>
</tr>
<tr>
<td>B.3.2</td>
<td>SQL Data Types</td>
<td>749</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Creating a Table</td>
<td>751</td>
</tr>
<tr>
<td>B.3.4</td>
<td>Creating a Key</td>
<td>753</td>
</tr>
<tr>
<td></td>
<td>Primary Keys</td>
<td>753</td>
</tr>
<tr>
<td></td>
<td>Foreign Keys</td>
<td>755</td>
</tr>
<tr>
<td>B.3.5</td>
<td>Relations</td>
<td>756</td>
</tr>
<tr>
<td></td>
<td>Two Tables with a Common Key</td>
<td>756</td>
</tr>
<tr>
<td></td>
<td>Using a Fully Qualified Name and a Dot to Join the Tables</td>
<td>757</td>
</tr>
<tr>
<td></td>
<td>Aliases</td>
<td>758</td>
</tr>
<tr>
<td>B.3.6</td>
<td>Altering a Table</td>
<td>759</td>
</tr>
<tr>
<td>B.3.7</td>
<td>Dropping a Table</td>
<td>761</td>
</tr>
<tr>
<td>B.3.8</td>
<td>Dropping a Database</td>
<td>761</td>
</tr>
<tr>
<td>B.4</td>
<td>SQL Functions</td>
<td>761</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Numeric Functions</td>
<td>762</td>
</tr>
<tr>
<td></td>
<td>Using GROUP BY</td>
<td>763</td>
</tr>
<tr>
<td>B.4.2</td>
<td>String Functions</td>
<td>765</td>
</tr>
<tr>
<td>B.4.3</td>
<td>Date and Time Functions</td>
<td>766</td>
</tr>
<tr>
<td></td>
<td>Formatting the Date and Time</td>
<td>767</td>
</tr>
<tr>
<td></td>
<td>The MySQL EXTRACT Command</td>
<td>769</td>
</tr>
</tbody>
</table>

First of all, a lot has been happening since the release of Perl 5.10. Many of the ideas from Perl 6 have been backported to Perl 5 as we await the official release of Perl 6. And as new features are added, there have been a number of incremental version changes, the latest version number being Perl 5.21. In fact, version 5.10 was what has been called the beginning of “modern Perl.” CPAN has added a number of new modules that have spiked interest in Perl, among them Moose, Mojolicious, Dancer, DBIx::Class, and more; and Core Perl has gained many new modules as well, such as List::Util, Time::Piece, autodie, and so on. Those incremental changes to Perl 5 continue to enhance Core Perl and all the many new modules that deal with modern projects and technology. Perl 6 is still a work in progress. To see the roadmap for Perl 6 development, you can go to github.com or you can participate in the development process by going to perl6.org. But the fact is, we’re still entrenched in Perl 5 while we wait. This book addresses new features that have been added since the last edition, revitalizes and updates some of the older examples, and trims some of those topics that are not applicable in modern Perl.

As you read this, I am still teaching Perl University of California, Santa Clara (UCSC) extension in Sunnyvale, California, to groups of professionals coming from all around Silicon Valley. I always ask at the beginning of a class, “So why do you want to learn Perl?” The predominate response today: for automation and testing, not CGI or biotech, not even for completing a resume now that the Valley is on an upswing, but primarily for automation and testing. The legacy code remains for those companies that started with Perl, and it continues to grow. No matter what anyone tells you, Perl is still in demand. I know. I teach it, not only at UCSC, but to those major companies that use Perl and require their employees to learn it as part of their training path.
Perl by Example is not just a beginner’s guide but a complete guide to Perl. It covers many aspects of what Perl can do, from basic syntax to regular expression handling, files, references, objects, working with databases, and much more. Perl also has a rich variety of functions for handling strings, arrays, hashes, and the like. This book will teach you Perl by using complete, working, numbered examples and output with explanations for each line, and avoids veering off into other areas or using complicated explanations that send you off to your favorite search engine in order to figure out what’s going on. It helps if you have some programming background, but it is not assumed that you are an experienced programmer or a guru. Anyone reading, writing, or just maintaining Perl programs can greatly profit from this text.

The appendices contain a complete list of functions and definitions, command-line switches, special variables, popular modules, and the Perl debugger; a tutorial to introduce Moose for object-oriented programming; a tutorial covering the Web application framework, Dancer, to replace the need for the Common Gateway Interface; and a guide for using PerlBrew and CPAN (“the gateway to all things Perl”) and how to effectively download modules.

I was fortunate to have been introduced to Alastair McGowan-Douglas as the technical expert for reviewing and critiquing this edition. He went well beyond the line of duty and has contributed greatly to not only transforming this book, but to adding his own writing for the tutorials in the appendices, correcting errors, and introducing modern Perl practices. His extensive knowledge and dedication have been invaluable. When we started the project, Alastair wrote to me:

“. . . I should note that ‘modern Perl’ refers to the era since 5.10, where practices and conventions got a massive overhaul within the community, as Perl itself had a resurgence in development on it (the language and binary themselves). The previous edition, of course, predates this sea-change, which it seems like the rug has somewhat been swept out from under us.

No matter! We shall prevail, as they say.”

And that is precisely what this edition has attempted to do!

—Ellie Quigley

September 2014
Acknowledgments

I’d like to acknowledge the following people for their contributions to the fifth edition.

Thank you, Mark Taub, an editor-in-chief to be praised for being very cool in every step of the process from the signing of the contract to the final book that you have now in your hand. Mark has a way of making such an arduous task seem possible; he soft-talks impossible deadlines, keeps up a steady pressure, and doesn’t get crazy over missed deadlines, quietly achieving his goal and always with a subtle sense of humor. Thank you, Mark, for being the driving force behind this new edition!

Of course, none of this would have been possible without the contributions of the Perl pioneers—Larry Wall, Randal Schwartz, and Tom Christiansen. Their books are must reading and include Learning Perl by Randal Schwartz and Programming Perl by Larry Wall, Tom Christiansen, and Jon Orwant.

Thank you, Vanessa Moore, the project manager and compositor who has been working with me for the past 20 years on making the by Example books look beautiful. She excels in her ability to do editing, layout, and artwork, and also in her ability to find errors that most programmers wouldn’t see, not to mention an abundance of patience and sense of humor. Without her, this book would be like a painting without color. She’s the best!

Also a big thanks to Daniel Holmes from NetApp (RTP) who contributed to the sections on Moose and wrote the final example; and Alastair McGowan-Douglas whose technical expertise was invaluable.

And last, but certainly not least, a huge thanks to all the students, worldwide, who have done all the real troubleshooting and kept the subject alive.
This page intentionally left blank
This page intentionally left blank
What's in a Name?

5.1 More About Data Types

By the end of this chapter, you will be able to read the following Perl code:

```perl
use strict;
use warnings;
my @l = qw/a b c d a e b a b d e f/;
my %hash=();

foreach my $key (@l){
    $hash{$key} = $key;
}

print join(" ",sort keys %hash),"\n";
```

Again, please take note that each line of code, in most of the examples throughout this book, is numbered. The output and explanations are also numbered to match the numbers in the code. When copying examples into your text editor, don't include these numbers, or you will generate errors.

5.1.1 Basic Data Types (Scalar, Array, Hash)

In Chapter 3, “Perl Scripts,” we briefly discussed scalars. In this chapter, we will cover scalars in more depth, as well as arrays and hashes. It should be noted that Perl does not provide the traditional data types, such as `int`, `float`, `double`, `char`, and so on. It bundles all these types into one type, the scalar. A scalar can represent an integer, float, string, and so on, and can also be used to create aggregate or composite types, such as arrays and hashes.

Unlike C or Java, Perl variables don't have to be declared before being used, and you do not have to specify what kind data will be stored there. Variables spring to life just by
the mere mention of them. You can assign strings, numbers, or a combination of these to Perl variables and Perl will figure out what the type is. You may store a number or a list of numbers in a variable and then later change your mind and store a string there. Perl doesn’t care.

A scalar variable contains a single value (for example, one string or one number), an array variable contains an ordered list of values indexed by a positive number, and a hash contains an unordered set of key/value pairs indexed by a string (the key) that is associated with a corresponding value (see Figure 5.1). (See Section 5.2, “Scalars, Arrays, and Hashes.”)

![Figure 5.1 Namespaces for scalars, arrays, and hashes in package main.](image)

### 5.1.2 Package, Scope, Privacy, and Strictness

**Package and Scope.** The Perl sample programs you have seen in the previous chapters are compiled internally into what is called a `package`, which provides a `namespace` for variables.

An analogy often used to describe a package is the naming of a person. In the Johnson family, there is a boy named James. James is known to his family and does not have to qualify his name with a last name every time he is being called to dinner. “James, sit down at the table” is enough. However, in the school he attends there are several boys named James. The correct James is identified by his last name, for example, “James Johnson, go to the principal’s office.”

In a Perl program, “James” represents a variable and his family name, “Johnson,” a package. The default package is called `main`. If you create a variable, `$name`, for example, `$name` belongs to the `main` package and could be identified as `$main::name`, but qualifying the variable at this point is unnecessary as long as we are working in a single file and using the default package, `main`. Later when working with modules, we will step outside of the package `main`. This would be like James going to school. Then we could have a conflict if two variables from different packages had the same name and would have to qualify which package they belong to. For now, we will stay in the `main` package. When you see the word `main` in a warning or error message, just be aware that it is a reference to something going on in your `main` package.

The scope of a variable determines where it is visible in the program. In the Perl scripts you have seen so far, the variables live in the package `main` and are visible to the entire script file (that is, global in scope). Global variables, also called package variables, can be
changed anywhere within the current package (and other packages), and the change will permanently affect the variable. To keep variables totally hidden within their file, block, or subroutine programs, we can define lexical variables. One way Perl does this is with the `my` operator. An entire file can be thought of as a block, but we normally think of a block as a set of statements enclosed within curly braces. If a variable is declared as a `my` variable within a block, it is visible (that is, accessible within that block and any nested blocks). It is not visible outside the block. If a variable is declared with `my` at the file level, then the variable is visible throughout the file. See Example 5.1.

Example 5.1

```perl
# We are in package main
1  no warnings;  # warnings turned off so that output is not clouded with warning messages

2  my $family="Johnson";  # file scope
3  { my $mother="Mama";  # block scope
4     my $father="Papa";
4     my ($cousin, $sister, $brother);
5     my $family="McDonald";   # new variable
5     print "The $family family is visible here.\n";
5   }
6  print "$mother and $father are not visible here.\n";
7  print "The $family family is back.\n";

(Output)
5  The McDonald family is visible here.
6  and are not visible here.
7  The Johnson family is back.

Explanation

1  warnings are turned off so that you can see what's going on without being interrupted with warning messages. If warnings had been turned on, you would have seen the following:

Name "main::father" used only once: possible typo at my.plx line 10.
Name "main::mother" used only once: possible typo at my.plx line 10.
The McDonald family is visible here.
Use of uninitialized value $mother in concatenation (.) or string at my.plx line 10.
Use of uninitialized value $father in concatenation (.) or string at my.plx line 10.
And are not visible here.
The Johnson family is back.

The messages are telling you that for package main, the $mother and $father variables were used only once. That is because they are not visible outside of the block where they were defined, and by being mentioned outside the block, they are new uninitialized variables.


EXPLANATION (continued)

2 The $family variable is declared as a lexical my variable at the beginning of the program. The file is considered a block for this variable giving it file scope; that is, visible for the entire file, even within blocks. If changed within a block, it will be changed for the rest of the file.

3 We enter a block. The my variables within this block are private to this block, visible here and in any nested blocks, and will go out of scope (become invisible) when the block exits.

4 This is a brand new lexical $family variable (McDonald). It has nothing to do with the one created on line 2. The first one (Johnson) will be visible again after we exit this block.

6 The my variables defined within the block are not visible here; that is, they have gone out of scope. These are brand new variables, created on the fly, and have no value.

7 The Johnson family is back. It is visible in the outer scope.

The purpose in mentioning packages and scope now is to let you know that the default scope of variables in the default main package, your script, is global; that is, accessible throughout the script. To help avoid the future problems caused by global variables, it is a good habit (and often a required practice) to keep variables private by using the my operator. This is where the strict pragma comes in.

The strict pragma (a pragma is a compiler directive) is a special Perl module that directs the compiler to abort the program if certain conditions are not met. It targets barewords, symbolic references, and global variables. For small practice scripts within a single file, using strict isn't necessary, but it is a good, and often required, practice to use it (a topic you can expect to come up in a Perl job interview!).

In the following examples, we will use strict primarily to target global variables, causing your program to abort if you don't use the my operator when declaring them.

EXAMPLE 5.2

1 use strict;
2 use warnings;
3 $family="Johnson";  # Whoops! global scope
4 $mother="Mama";
5 $father="Papa";
6 print "$mother and $father are here.\n"; # global
7 print "The $family family is here.\n";

(Output)
Global symbol "$family" requires explicit package name at strictex.plx line 3.
Global symbol "$mother" requires explicit package name at strictex.plx line 4.
Global symbol "$father" requires explicit package name at strictex.plx line 5.
5.1 More About Data Types

EXAMPLE 5.2 (continued)

Global symbol "$mother" requires explicit package name at strictex.plx
line 6.
Global symbol "$father" requires explicit package name at strictex.plx
line 6.
Global symbol "$family" requires explicit package name at strictex.plx
line 7.
Execution of strictex.plx aborted due to compilation errors.

EXPLANATION

1. The strict pragma is being used to restrict all "unsafe constructs." To see all the restrictions, type the following at your command-line:

   perldoc strict

   If you just want to target global variables, you would use strict with an argument in your program, such as:

   use strict 'vars'

2. The warnings pragma is turned on, but will not issue warnings because strict will supersede it, causing the program to abort first.

3. This is a global variable in the program, but it sets off a plethora of complaints from strict everywhere it is used. By preceding $family and the variables $mother and $father with the my operator, all will go well. (You can also explicitly name the package and the variable, as $main::family to satisfy strict. But then, the warnings pragma will start complaining about other things, as discussed in the previous example.)

6, 7. Global variables again! strict complains, and the program is aborted.

The warnings and strict pragmas together are used to help you find typos, spelling errors, and global variables. Although using warnings will not cause your program to die, with strict turned on, it will, if you disobey its restrictions. With the small examples in this book, the warnings are always turned on, but we will not turn on strict until later.

5.1.3 Naming Conventions

Variables are identified by the “funny characters” that precede them. Scalar variables are preceded by a $ sign, array variables are preceded by an @ sign, and hash variables are preceded by a % sign. Since the “funny characters” (properly called sigils) indicate what type of variable you are using, you can use the same name for a scalar, array, or hash (or a function, filehandle, and so on) and not worry about a naming conflict. For example, $name, @name, and %name are all different variables; the first is a scalar, the second is an array, and the last is a hash.¹

¹ Using the same name is perfectly legal, but not recommended; it makes reading the program too confusing.
Since reserved words and filehandles are not preceded by a special character, variable names will not conflict with them. Names are case sensitive. The variables named $Num, $num, and $NUM are all different. If a variable starts with a letter, it may consist of any number of letters (an underscore counts as a letter) and/or digits. If the variable does not start with a letter, it must consist of only one character. Perl has a set of special variables (for example, $_, $^, $, $1, $2) that fall into this category. (See Section A.2, “Special Variables,” in Appendix A.) In special cases, variables may also be preceded with a single quote, but only when packages are used. An uninitialized variable will get a value of zero or undef, depending on whether its context is numeric or string.

5.1.4 Assignment Statements

The assignment operator, the equal sign (=), is used to assign the value on its right-hand side to a variable on its left-hand side. Any value that can be “assigned to” represents a named region of storage and is called an lvalue. Perl reports an error if the operand on the left-hand side of the assignment operator does not represent an lvalue.

When assigning a value or values to a variable, if the variable on the left-hand side of the equal sign is a scalar, Perl evaluates the expression on the right-hand side in a scalar context. If the variable on the left of the equal sign is an array, then Perl evaluates the expression on the right in an array or list context (see Section 5.2, “Scalars, Arrays, and Hashes”).

---

**EXAMPLE 5.3**

(The Script)

```perl
use warnings;
# Scalar, array, and hash assignment
1 my $salary=50000;  # Scalar assignment
2 my @months=('Mar', 'Apr', 'May');   # Array assignment
3 my %states= {
    CA => 'California',
    ME => 'Maine',
    MT => 'Montana',
    NM => 'New Mexico',
};
4 print "$salary\n";
5 print "@months\n";
6 print "$months[0], $months[1], $months[2]\n";
7 print "%states{"CA"}, %states{"NM\"}\n";
8 print $x + 3, "\n";  # $x just came to life!
9 print "****$name***\n";  # $name is born!
```

---

2. The value on the left-hand side of the equal sign is called an lvalue, and the value on the right-hand side is called an rvalue.
**Example 5.3 (continued)**

(Output)
4 50000
5 Mar Apr May
6 Mar, Apr, May
7 California, New Mexico
8 3
9 ******

**Explanation**
1. The scalar variable `$salary` is assigned the numeric literal `50000`.*
2. The array `@months` is assigned the comma-separated list, `'Mar', 'Apr', 'May'`. The list is enclosed in parentheses and each list item is quoted.
3. The hash, `%states`, is assigned a list consisting of a set of strings separated by either a digraph symbol (=>) or a comma. The string on the left is called the key and it is not required that you quote the key, unless it starts with a number. The string to the right is called the value. The key is associated with its value.
4. The `@months` array is printed. The double quotes preserve spaces between each element.
5. The individual elements of the array, `@months`, are scalars and are thus preceded by a dollar sign ($). The array index starts at zero.
6. The key elements of the hash, `%states`, are enclosed in curly braces ({}). The associated value is printed. Each value is a single value, a scalar. The value is preceded by a dollar sign ($).
7. The scalar variable, `$x`, is referenced for the first time with an initial value of undef. Because the number 3 is added to $x, the context is numeric. $x then gets an initial value of 0 in order to perform arithmetic. Initially $x is null.
8. The scalar variable, `$name`, is referenced for the first time with an undefined value. The context is string.

* The comma can be used in both Perl 4 and Perl 5. The => symbol was introduced in Perl 5.

### 5.2 Scalars, Arrays, and Hashes

Now that we have discussed the basics of Perl variables (types, visibility, funny characters, and so forth), we can look at them in more depth. Perhaps a review of the quoting rules detailed in Chapter 4, “Getting a Handle on Printing,” would be helpful at this time.
5.2.1 Scalar Variables

Scalar variables hold a single number or string and are preceded by a dollar sign ($). Perl scalars need a preceding dollar sign whenever the variable is referenced, even when the scalar is being assigned a value.

Assignment. When making an assignment, the value on the right-hand side of the equal sign is evaluated as a single value (that is, its context is scalar). A quoted string, then, is considered a single value even if it contains many words.

Example 5.4

```perl
1 $number = 150;  # Number
2 $name = "Jody Savage";  # String
3 $today = localtime();  # Function
```

Explanation

1 The numeric literal, 150, is assigned to the scalar variable $number.
2 The string literal "Jody Savage" is assigned to the scalar $name as a single string.
3 The output of Perl's localtime function will be assigned as a string to $today. (The return value of localtime is string context here and if assigned to an array its return value is an array of numbers. See perldoc -f localtime.)

Example 5.5

(The Script)
```perl
use warnings;

# Initializing scalars and printing their values
1 my $num = 5;
2 my $friend = "John Smith";
3 my $money = 125.75;
4 my $now = localtime;  # localtime is a Perl function
5 my $month="Jan";
6 print "$num\n";
7 print "$friend\n";
8 print "I need \$money.\n";  # Protecting our money
9 print qq/$friend gave me \$money./;
10 print qq/The time is $now\n/;
11 print "The month is ${month}uary.\n";  # Curly braces shield
12 print "The month is $month . "uary.\n";  # Concatenate
```

References are also stored as string variables.
5.2 Scalars, Arrays, and Hashes

### Example 5.5 (continued)

(Output)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>John Smith</td>
</tr>
<tr>
<td>8</td>
<td>I need $125.75.</td>
</tr>
<tr>
<td>9</td>
<td>John Smith gave me $125.75.</td>
</tr>
<tr>
<td>10</td>
<td>The time is Sat Jan 24 16:12:49 2014.</td>
</tr>
<tr>
<td>11</td>
<td>The month is January.</td>
</tr>
<tr>
<td>12</td>
<td>The month is January.</td>
</tr>
</tbody>
</table>

### Explanation

1. The scalar `$num` is assigned the numeric literal, 5.
2. The scalar `$friend` is assigned the string literal, `John Smith`.
3. The scalar `$money` is assigned the numeric floating point literal, 125.75.
4. The scalar `$now` is assigned the output of Perl's built-in `localtime` function.
5. The scalar `$month` is assigned `Jan`.
6. The quoted string is printed. The backslash allows the first dollar sign ($) to be printed literally; the value of `$money` is interpolated within double quotes, and its value printed.
7. The Perl `qq` construct replaces double quotes. The string to be quoted is enclosed in forward slashes. The value of the scalar `$friend` is interpolated; a literal dollar sign precedes the value of the scalar interpolated variable, `$money`.
8. The quoted string is printed as if in double quotes. The `$now` variable is interpolated.
9. Curly braces can be used to shield the variable from characters that are appended to it. `January` will be printed.
10. Normally, two strings or expressions are joined together with the dot operator (see Chapter 6, “Where’s the Operator?”), called the concatenation operator.

**The `defined` Function.** If a scalar has neither a valid string nor a valid numeric value, it is undefined. The `defined` function allows you to check for the validity of a variable's value. It returns 1 if the variable has a value (other than `undef`) and nothing if it does not.

### Example 5.6

```perl
$name = "Tommy";
print "OK \n" if defined $name;
```

**The `undef` Function.** When you define a variable without giving it a value, such as `my $name;` the initial value is `undef`. 
You can use the `undef` function to undefine an already defined variable. It releases whatever memory that was allocated for the variable. The function returns the undefined value. This function also releases storage associated with arrays and subroutines.

**Example 5.7**

```perl
undef $name;
```

**The `$_` Scalar Variable.** The `$_` (called a topic variable) is a ubiquitous little character. Although it is very useful in Perl scripts, it is often not seen, somewhat like your shadow—sometimes you see it; sometimes you don't. It is used as the default pattern space for searches, for functions that require a scalar argument, and to hold the current line when looping through a file. Once a value is assigned to `$_`, functions such as `chomp`, `split`, and `print` will use `$_` as an argument. You will learn more about functions and their arguments later, but for now, consider the following example.

**Example 5.8**

```perl
1  $_ = "Donald Duck\n";
2  chomp;   # The newline is removed from $_
3  print;   # The value of $_ is printed
```

**Explanation**

1. The `$_` scalar variable is assigned the string "Donald Duck\n". Now you see it!
2. The `chomp` function removes the newline from `$_`, the default scalar. Now you don't!
3. The `print` function has been given nothing to print, so it will print `$_`, the default scalar, without a trailing newline.

**The `$_` Scalar and Reading Input from Files**

When looping through a file, the `$_` is often used as a holding place for each line as it is read. In the following example, a text file called `datebook.txt` is opened for reading. The filehandle is `$fh`, a user-defined variable to represent the real file, `datebook.txt`. Each time the loop is entered, a line is read from the file. But where does the line go? It is implicitly assigned to the `$_` variable. The next time the loop is entered, a new line is read from the file and assigned to `$_`, overwriting the previous line stored there. The loop ends when the end of file is reached. The `print` function, although it appears to be printing nothing, will print the value of `$_` each time the loop block is entered.

---

4. A topic variable is a special variable with a very short name, which in many cases can be omitted.
5.2 Scalars, Arrays, and Hashes

### Example 5.9

*(The Script)*

```perl
use warnings;

# Reading input from a file
open(my $fh, "<", "datebook.txt") or die $!;
while(<$fh>){
    # loops through the file a line at a time storing
    # each line in
    print;
    # prints the value stored in
}
close $fh;
```

*(Output)*

Jon DeLoach:408-253-3122:123 Park St., San Jose, CA 04086:7/25/53:85100

### Explanation

1. A user-defined filehandle is a Perl way of associating a real file with an internal Perl structure by a name. In this example, `$fh` is a lexically scoped filehandle used to represent the real file, `datebook.txt`, which is opened for reading. If the file doesn’t exist or is unreadable, the program will “die” (exit) with the reason it died ($!).
2. The `while` loop is entered. Perl will read the first line from the file and implicitly assign its value to `$_`, and if successful enter the body of the loop. The angle brackets (`<>`) are used for reading, as we saw when reading from `STDIN`.
3. Every time the loop is entered, a new line from the file is stored in `$_`, overwriting the previous line that was stored there, and each time the current value of `$_` is printed.
4. This is the closing brace for the block of the loop. When the file has no more lines, the read will fail, and the loop will end.
5. Once finished with the file, it is closed via the filehandle. (See Chapter 10, “Getting a Handle on Files,” for a complete discussion on filehandles.)

### 5.2.2 Arrays

Let’s say when you moved into town, you made one friend. That friend can be stored in a scalar as `$friend = "John"`. Now let’s say a few months have gone by since you moved, and now you have a whole bunch of new friends. In that case, you could create a list of friends, give the list one name, and store your friends in a Perl array; for example, `@pals = ("John", "Mary", "Sanjay", "Archie")`. 
When you have a collection of similar data elements, it is easier to use an array than to create a separate variable for each of the elements. The array name allows you to associate a single variable name with a list of data elements. Each of the elements in the list is referenced by its name and a subscript (also called an index).

Perl, unlike C-like languages, doesn’t care whether the elements of an array are of the same data type. They can be a mix of numbers and strings. To Perl, an array is a list containing an ordered set of scalars. The name of the array starts with an @ sign and the list is enclosed in parentheses, each element assigned an index value starting at zero (see Figure 5.2).

**Assignment.** If the array is initialized, the elements are enclosed in parentheses, and each element is separated by a comma. The list is parenthesized due to the lower precedence of the comma operator over the assignment operator. Elements in an array are simply scalars.

The qw construct can also be used to quote words in a list (similar to qq, q, and qx). The items in the list are treated as singly quoted words and the comma is also provided.

```perl
$pal = "John";  # Scalar holds one value
@pals = ("John", "Sam", "Nicky", "Jake");  # Array holds a list of values
@pals = qw(John Sam Nicky Jake);  # qw means quote word and include comma
```

![Figure 5.2](image_url) A scalar variable and an array variable.

**Example 5.10**

```perl
1  @name=("Guy", "Tom", "Dan", "Roy");
2  @list=(2..10);
3  @grades=(100, 90, 65, 96, 40, 75);
4  @items=($a, $b, $c);
5  @empty=();
6  $size=@items;
7  @mammals = qw/dogs cats cows/;
8  @fruit = qw/apples pears peaches/;
```

**Explanation**

1. The array @name is initialized with a list of four string literals.
2. The array @list is assigned numbers ranging from 2 through 10.
3. The array @grades is initialized with a list of six numeric literals.
### Explanation (continued)

4. The array `@items` is initialized with the values of three scalar variables.

5. The array `@empty` is assigned an empty list.

6. The array `@items` is assigned to the scalar variable `$size`. The value of the scalar is the number of elements in the array (in this example, 3).

7. The `qw` (quote word) construct is followed by a delimiter of your choice and a string. `qw()` extracts words out of your string using embedded whitespace as the delimiter and returns the words as a list. Variables are not interpolated. Each word in the list is treated as a singly quoted word. The list is terminated with a closing delimiter. This example could be written like so:

   ```perl
   @mammals = ('cats', 'dogs', 'cows');
   ```

8. The `qw` construct accepts paired characters `()`, `{}`, `< >`, and `[]`, as optional delimiters.

### Output and Input Special Variables ($, and "$")

The `$` is a special default global variable, called the **output field separator**. When used by the `print` function to print a list or an array (not enclosed in quotes), this variable separates the elements and is initially set to `undef`. For example, `print 1,2,3` would output `123`. Although you can assign a different value to the `$`, it’s not a good idea, as once changed, it will affect your whole program. (The `join` function would provide a better solution.)

### Example 5.11

```perl
use warnings;
my @pets = ("Smokey", "Fido", "Gills", "Skiddy");
print @pets, "\n";  # Output separator is undef

$,,="*****";  # Changes the output field separator
print @pets, "\n";  # no quotes; ***** replaces undef
print 1,2,3, "\n";
```

(Output)

```
SmokeyFidoGillsSkiddy
Smokey*****Fido*****Gills*****Skiddy*****
1*****2*****3*****
```

### Explanation

3. The array of pets is printed. The value of of `$`, is used to separate elements of an unquoted list for the `print` function and is initially set to `undef`.

4. The `$`, variable is reset to "*****".

5. Now, when the `print` function displays an unquoted list, the list items are separated by that string.

6. The comma evaluates to "*****" in the `print` function.

The "$" is a special scalar variable, called the **list separator**, used to separate the elements of a list in an array, and is by default a single space. For example, when you print an array enclosed in double quotes, the value of "$" will be preserved, and you will have a space between the elements.
**Example 5.12**

```perl
my @grocery_list = qw(meat potatoes rice beans spinach milk);
print "@grocery_list\n"; # The list separator is a space
$s" = "---"; # Change the list separator
print "@grocery_list\n"; # The list separator has been changed
$s", = "||"; # change print's separator
print @grocery_list, "\n"; # no quotes
```

**Output**

```
meat potatoes rice beans spinach milk
meat---potatoes---rice---beans---spinach---milk
meat||potatoes||rice||beans||spinach||milk
```

**Explanation**

The `$s" variable is called the list separator and is initially set to a space. Unless the array is enclosed in double quotes, the space is lost.

You can change the `$s" variable by assigning it a string.

Now you can see when we print the quoted array, the array separator between the elements has been changed.

Now the `print` separator is changed to "||". If the quotes are removed, the `print` function will display the list with the new separator.

**Array Size.** `$#arrayname` returns the largest index value in the array; that is, the index value of its last element. Since the array indices start at zero, this value is one less than the array size. The `$#arrayname` variable can also be used to shorten or truncate the size of the array.

To get the size of an array, you can assign it to a scalar or use the built-in `scalar` function which used with an array, forces scalar context. It returns the size of the array, one value. (This is defined as a unary operator. See perlop for more details.)

**Example 5.13**

```perl
use warnings;
my @grades = (90,89,78,100,87);
print "The original array is: @grades\n";
print "The number of the last index is $#grades\n";
print "The value of the last element in the array is $grades[$#grades]\n";

print "The size of the array is ", scalar @grades, "\n";
my $size = @grades; # Get the size of the array
@grades=();
print "The array is completely truncated: @grades\n";
```

**Output**

```
The original array is: 90 89 78 100 87
The number of the last index is 4
The value of the last element of the array is 87
The size of the array is 5
The array is completely truncated:
```
5.2 Scalars, Arrays, and Hashes

EXPLANATION

1. The array @grades is assigned a list of five numbers.
2. The $# construct gets the index value of the last element in the array.
3. By using $#grades as an index value, the expression would evaluate to $grades[4].
4. The built-in scalar function forces the array to be in scalar context and returns the number of elements in the array. You could also assign the array to a scalar variable, as in $size = @grades, to produce the same result as shown in line 6.
6. Using an empty list causes the array to be completely truncated to an empty list.

The Range Operator and Array Assignment. The .. operator, called the range operator, when used in a list context, returns a list of values starting from the left value to the right value, counting by ones.

EXAMPLE 5.14

```perl
use warnings;
1 my @digits=(0 .. 10);
2 my @letters=('A' .. 'Z');
3 my @alpha=('A' .. 'Z', 'a' .. 'z');
4 my @n=(-5 .. 20);
```

EXPLANATION

1. The array @digits is assigned a list of numbers, 0 incremented by 1 until 10 is reached.
2. The array @letters is assigned a list of capital letters, A through Z (ASCII values of A through Z).
3. The array @alpha is assigned a list of uppercase and lowercase letters.
4. The array @n is assigned a list of numbers, -5 through 20.

Accessing Elements. An array is an ordered list of scalars. To reference the individual elements in an array, each element (a scalar) is preceded by a dollar sign. The index starts at 0, followed by positive whole numbers. For example, in the array @colors, the first element in the array is $colors[0], the next element is $colors[1], and so forth. You can also access elements starting at the end of an array with the index value of -1 and continue downward; for example, -2, -3, and so forth.

1. To assign a list of values to an array:
   ```perl
   @colors = qw( green red blue yellow);
   ```
2. To print the whole array, use the @:
   ```perl
   print "@colors\n";
   ```
3. To print single elements of the array:
   ```perl
   print "$colors[0]  $colors[1]\n";
   ```
4. To print more than one element (meaning, a list):

   ```
   print "@colors[1,3]\n";  # Now the index values are in a list,
   # requiring the @ rather than the $ sign.
   ```

@colors

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;green&quot;</td>
<td>&quot;red&quot;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
|   "blue"  | "yellow"
|     2     | 3     |

Figure 5.3 Array elements.

**EXAMPLE 5.15**

(The Script)

```perl
use warnings;

# Populating an array and printing its values
my @names=('John', 'Joe', 'Jake');  # @names=qw/John Joe Jake/;
print @names, "\n";  # prints without the separator
print "Hi $names[0], $names[1], and $names[2]\n";
my $number=@names;  # The scalar is assigned the number
                   # of elements in the array
print "There are $number elements in the @names array.\n";
print "The last element of the array is $names[$number -1].\n";
print "The last element of the array is $names[$#names].\n";  # Remember, the array index starts at zero!
my @fruit = qw(apples pears peaches plums);
print "The first element of the @fruit array is $fruit[0];  
      the second element is $fruit[1].\n";
print "Starting at the end of the array; @fruit[-1, -3]\n";
```

(Output)

```plaintext
John Joe Jake
Hi John, Joe, and Jake!
There are 3 elements in the @names array.
The last element of the array is Jake.
The last element of the array is Jake.
The first element of the @fruit array is apples; the second element is pears.
Starting at the end of the array: plums pears
```
EXPLANATION

1. The @names array is initialized with three strings: John, Joe, and Jake.
2. The entire array is displayed without a space between the individual elements. The input field separator, a space, is preserved when the array is enclosed in double quotes: "@names".
3. Each element of the array is printed, starting with subscript number zero.
4. The scalar variable $number is assigned the array @names. The value assigned is the number of elements in the array @names. You can also use the built-in scalar function to get the size of an array; for example: $size = scalar @names;
5. The last element of the array is printed. Since index values start at zero, the number of elements in the array decremented by one evaluates to the number of the last subscript.
6. The last element of the array is printed. The $#names value evaluates to the number of the last subscript in the array. This value used as a subscript will retrieve the last element in the @names array.
7. The qw construct creates an array of singly quoted words from the string provided to it, using space as the word separator. (You don’t enclose the words in quotes or separate the words with commas.) The qw delimiter is any pair of nonalphanumeric characters.
8. The first two elements of the @fruit array are printed.
9. With a negative offset as an index value, the elements of the array are selected from the end of the array. The last element ($fruit[-1]) is plums, and the third element from the end ($fruit[-3]) is pears. Note that when both index values are within the same set of brackets, as in @fruit[-1,-3], the reference is to a list, not a scalar; that is why the @ symbol precedes the name of the array, rather than the $.

Looping Through an Array with the foreach Loop. One of the best ways to traverse the elements of an array is with Perl’s foreach loop. (See Chapter 7, “If Only, Unconditionally, Forever,” for a thorough discussion.)

This control structure steps through each element of a list (enclosed in parentheses) using a scalar variable as a loop variable. The loop variable references, one at a time, each element in the list, and for each element, the block of statements following the list is executed. When all of the list items have been processed, the loop ends. If the loop variable is missing, $_, the default scalar, is used. You can use a named array or create a list within parentheses.

You may also see code where the word for is used instead of foreach. This is because for and foreach are synonyms. In these examples, foreach is used simply to make it clear that we are going through a list, one element at a time; that is, “for each” element in the list.
EXAMPLE 5.16

(The Script)

```perl
use warnings;
# Array slices
1 my @names=('Tom', 'Dick', 'Harry', 'Pete');
2 foreach $pal (@names){
3    print "$pal
";
4}
5 foreach ("red", "green", "yellow", "blue"){
6    print "$_ 
";
7}
```

(Output)

3 Tom
Dick
Harry
Pete
5 red
   green
   Yellow
   blue

EXPLANATION

1 The array @names is assigned a list: 'Tom', 'Dick', 'Harry', 'Pete'.
2 The foreach loop is used to walk through the list, one word at a time.
3 The $pal scalar is used as a loop variable, called an iterator; that is, it points to each successive element of the list for each iteration of the loop. If you don't provide the iterator variable, Perl uses the topic variable $_ instead. For each iteration of the loop, the block of statements enclosed in curly braces is executed.
4 In this example, the foreach loop is not given an iterator variable, so Perl uses the $_ variable instead, even though you can't see it.
5 The value of $_ is printed each time through the loop. (This time we have to explicitly use $_ because we have added the \n to the string.)

Array Copy and Slices. When you assign one array to another array, a copy is made. It's that simple. Unlike many languages, you are not responsible for the type of data the new array will hold or how many elements it will need. Perl handles the memory allocation and the type of data that will be stored in each element of the new array.

A slice accesses several elements of a list, an array, or a hash simultaneously using a list of index values. You can use a slice to copy some elements of an array into another and also assign values to a slice. If the array on the right-hand side of the assignment operator is larger than the array on the left-hand side, the unused values are discarded. If it is
smaller, the values assigned are undefined. As indicated in the following example, the array
indices in the slice do not have to be consecutively numbered; each element is assigned
the corresponding value from the array on the right-hand side of the assignment operator.

**Example 5.17**

(The Script)

```perl
use warnings;
# Array copy and slice
my @names=('Tom', 'Dick', 'Harry', 'Pete');
@newnames = @names; # Array copy
print "@newnames\n";
@pal=@names[1,2,3]; # Array slice -- @names[1..3] also okay
print "@pal\n\n";

@friend[0,1,2], not $friend[0,1,2]; # Assign to an array slice
print "@friend\n";
```

(Output)

```
Tom Dick Harry Pete
Dick Harry Pete
Tom Dick Harry
```

**Explanation**

1. The array `@names` is assigned the elements 'Tom', 'Dick', 'Harry', and 'Pete'.
2. The array `@pal` is assigned the elements 1, 2, and 3 of the `@names` array. The elements
of the `@names` array are selected and copied in the `@pal` array.
3. The `@friend` array is created by copying all the values from the `@names` array and
assigning them to `@friend` elements 0, 1, and 2.

Multidimensional Arrays—Lists of Lists. Multidimensional arrays are sometimes
called **tables** or **matrices**. They consist of rows and columns and can be represented with
multiple subscripts. In a two-dimensional array, the first subscript represents the row, and
the second subscript represents the column.

Perl allows this type of array, but it requires an understanding of references. We will
cover this in detail in Chapter 12, “Does This Job Require a Reference?”

### 5.2.3 Hashes—Unordered Lists

A **hash** (in some languages called an associative array, map, table, or dictionary) is a
variable consisting of one or more pairs of scalars—either strings or numbers. Hashes are
often used to create tables, complex data structures, find duplicate entries in a file or array,
or to create Perl objects. We will cover objects in detail in Chapter 14, “Bless Those Things!
(Object-Oriented Perl).”
Hashes are defined as an unordered list of key/value pairs, similar to a table where the keys are on the left-hand side and the values associated with those keys are on the right-hand side. The name of the hash is preceded by the % and the keys and values are separated by a \ => , called the **fat comma** or **digraph** operator.

Whereas arrays are ordered lists with numeric indices starting at 0, hashes are unordered lists with string indices, called keys, stored randomly. (When you print out the hash, don’t expect to see the output ordered just as you typed it!)

To summarize, the keys in a hash must be unique. The keys need not be quoted unless they begin with a number or contain hyphens, spaces, or special characters. Since the keys are really just strings, to be safe, quoting the keys (either single or double quotes) can prevent unwanted side effects. It’s up to you. The values associated with the key can be much more complex than what we are showing here, and require an understanding of Perl references. These complex types are discussed in Chapter 12, “Does This Job Require a Reference?”

```perl
my %pet = ("Name"  => "Sneaky",
           "Type"  => "cat",
           "Owner" => "Carol",
           "Color" => "yellow",
           );
```

So for this example, the keys and values for the hash called %pet, are as follows:

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Name&quot;</td>
<td>&quot;Sneaky&quot;</td>
</tr>
<tr>
<td>&quot;Type&quot;</td>
<td>&quot;cat&quot;</td>
</tr>
<tr>
<td>&quot;Owner&quot;</td>
<td>&quot;Carol&quot;</td>
</tr>
<tr>
<td>&quot;Color&quot;</td>
<td>&quot;yellow&quot;</td>
</tr>
</tbody>
</table>

**Assignment.** As in scalars and arrays, a hash variable must be defined before its elements can be referenced. Since a hash consists of pairs of values, indexed by the first element of each pair, if one of the elements in a pair is missing, the association of the keys and their respective values will be affected. When assigning keys and values, make sure you have a key associated with its corresponding value. When indexing a hash, curly braces are used instead of square brackets.

```perl
my %seasons=({"Sp" => "Spring",
              "Su" => "Summer",
              "F"  => "Fall",
              "W"  => "Winter",
              });
```
5.2 Scalars, Arrays, and Hashes

**Example 5.18 (continued)**

```perl
my %days = (
    "Mon" => "Monday",
    "Tue" => "Tuesday",
    "Wed" => undef,
);
$days{"Wed"} = "Wednesday";
```

**Explanation**

1. The hash `%seasons` is assigned keys and values. Each key and value is separated by the fat comma, `=>`. The string "Sp" is the key with a corresponding value of "Spring", the string "Su" is the key for its corresponding value "Summer", and so on. It is not necessary to quote the key if it is a single word and does not begin with a number or contain spaces.
2. The hash `%days` is assigned keys and values. The third key, "Wed", is assigned `undef`. The `undef` function evaluates to an undefined value; in this example, it serves as a placeholder with an empty value to be filled in later.
3. Individual elements of a hash are scalars. The key "Wed" is assigned the string value "Wednesday". The index is enclosed in curly braces. Note: the keys do not have any consecutive numbering order and the pairs can consist of numbers and/or strings.

Accessing Hash Values. When accessing the values of a hash, the subscript or index consists of the key enclosed in curly braces. Perl provides a set of functions to list the keys, values, and each of the elements of the hash.

Due to the internal hashing techniques used to store the keys, Perl does not guarantee the order in which an entire hash is printed.

**Example 5.19**

(The Script)
```
use warnings;

# Assigning keys and values to a hash
my (%department, $department, $school); # Declare variables
%department = (
    "Eng" => "Engineering", # keys do not require quotes
    "M" => "Math",
    "S" => "Science",
    "CS" => "Computer Science",
    "Ed" => "Education",
);
$department = $department{'M'}; # Either single, double quotes
$school = $department{'Ed'};
print "I work in the $department section\n";
print "Funds in the $school department are being cut.\n";
print qq/I'm currently enrolled in a $department{'CS'} course.\n/;
print qq/The department hash looks like this:\n/;
```
### Example 5.19 (continued)

```perl
10  print $department, "\n";   # The printout is not in the expected
    # order due to internal hashing

(Output)
6  I work in the Math section

7  Funds in the Education department are being cut.
8  I'm currently enrolled in a Computer Science course.
9  The department hash looks like this:
10  SScienceCSComputer ScienceEdEducationMMathEngEngineering
```

### Explanation

1. The hash is called `%department`. It is assigned keys and values.
2. The first key is the string `Eng`, and the value associated with it is `Engineering`.
3. The closing parenthesis and semicolon end the assignment.
4. The scalar `$department` is assigned `Math`, the value associated with the `M` key. It's sometimes confusing to name different types of variables by the same name. In this example, it might be better to change `$department` to `$subject` or `$course`, for example.
5. The scalar `$school` is assigned `Education`, the value associated with the `Ed` key.
6. The quoted string is printed; the scalar `$department` is interpolated.
7. The quoted string is printed; the scalar `$school` is interpolated.
8. The quoted string and the value associated with the `CS` key are printed.
9, 10. The entire hash is printed, with keys and values packed together and not in any specific order. A key and its value, however, will always remain paired.

**Hash Slices.** A hash slice is a list of hash keys. The hash name is preceded by the `@` symbol and assigned a list of hash keys enclosed in curly braces. The hash slice lets you access one or more hash elements in one statement, rather than by going through a loop.

### Example 5.20

```perl
(The Script)
use warnings;
# Hash slices
1  my %officer= ("name" => "Tom Savage",
    "rank" => "Colonel",
    "dob"  => "05/19/66"
);
2  my @info=@officer{"name","rank","dob"};  # Hash slice
3  print "@info\n";
4  @officer{"phone","base"}=('730-123-4455','Camp Lejeune');
5  print %officer, "\n";

(Output)
2  Tom Savage Colonel 05/19/66
6  baseCamp Lejeunedob05/19/66nameTom Savagephone730-123-4455rankColonel
```
### Explanation

1. The hash `%officer` is assigned keys and values.
2. This is an example of a hash slice. The list of hash keys, "name", "rank", and "dob" are assigned to the `@info` array. The name of the hash is prepended with an `@` because this is a list of keys. The values corresponding to the list of keys are assigned to `@info`.
3. The keys and their corresponding values are printed. Using the slice is sometimes easier than using a loop to do the same thing.
4. Now using a slice in the assignment, we can create two new entries in the hash.

### Removing Duplicates from a List Using a Hash.

Because all keys in a hash must be unique, one way to remove duplicates from a list, whether an array or file, is to list items as keys in a hash. The values can be used to keep track of the number of duplicates or simply left undefined. The keys of the new hash will contain no duplicates. See the section, “The map Function,” later in this chapter, for more examples.

### Example 5.21

(The Script)
```
use warnings;
1  my %dup=();  # Create an empty hash.
2  my @colors=qw(red blue red green yellow green red orange);
3  foreach my $color (@colors){
    $dup{$color}++;  # Adds one to the value side of
    # the hash. May be written
    # $dup{$color}=$dup{$color}+1
}
4  printf "Color   Number of Occurrences
";
5  while((my $key, my $value)=each %dup){
    printf "%-12s%-s
",$key, $value;
}
6  @colors = sort keys %dup;
7  print "Duplicates removed: @colors\n";
```

(Output)
```
perl dup.plx
Color   Number of Occurrences
3 green       2
   blue        1
   orange      1
   red         3
   yellow      1
5 Duplicates removed: blue green orange red yellow
```
Chapter 5 • What’s in a Name?

EXPLANATION

1. This is the declaration for an empty hash called %dup().
2. The array of colors contains a number of duplicate entries, as shown in Figure 5.4.
3. For each item in the array of colors, a key and value are assigned to the %dup hash. The first time the color is seen, it is created as a key in the hash; its value is incremented by 1, starting at 0 (that is, the key is the color and the value is the number of times the color occurs). Because the key must be unique, if a second color occurs and is a duplicate, the first occurrence will be overwritten by the duplicate and the value associated with it will increase by one.
4. The built-in each function is used as an expression in the while loop. It will retrieve and assign each key and each value from the hash to $key and $value respectively, and a pair is printed each time through the loop.
5. The keys of %dup hash are a unique list of colors. They are sorted and assigned to the @colors array.

Figure 5.4 Removing duplicates with a hash.

5.2.4 Complex Data Structures

By combining arrays and hashes, you can make more complex data structures, such as arrays of hashes, hashes with nested hashes, arrays of arrays, and so on. Here is an example of an array of arrays requiring references.

my $matrix = [
    [ 0, 2, 4 ],
    [ 4, 1, 32 ],
    [ 12, 15, 17 ]
];

To create these structures, you should have an understanding of how Perl references and complex data structures are used. (See Chapter 12, “Does This Job Require a Reference?”)
5.3 Array Functions

Arrays can grow and shrink. The Perl array functions allow you to insert or delete elements of the array from the front, middle, or end of the list, to sort arrays, perform calculations on elements, to search for patterns, and more.

5.3.1 Adding Elements to an Array

The push Function. The push function pushes values onto the end of an array, thereby increasing the length of the array (see Figure 5.5).

**Format**

```plaintext
push(ARRAY, LIST)
```

**Example 5.22**

(In Script)
```
use warnings;
# Adding elements to the end of a list
my @names=("Bob", "Dan", "Tom", "Guy");
push(@names, "Jim", "Joseph", "Archie");
print "@names 
";
```

(Output)
```
Bob Dan Tom Guy Jim Joseph Archie
```

**Explanation**

1. The array @names is assigned list values.
2. The push function pushes three more elements onto the end of the array.
3. The new array has three more elements appended to it.

---

Figure 5.5 Adding elements to an array.
The **unshift** Function. The `unshift` function prepends `LIST` to the front of the array (see Figure 5.6).

**FORMAT**

```plaintext
unshift(ARRAY, LIST)
```

**EXAMPLE 5.23**

(In Script)

```perl
use warnings;
#
# Putting new elements at the front of a list
1 my @names = ("Jody", "Bert", "Tom") ;
2 unshift(@names, "Liz", "Daniel");
3 print ":@names\n";

(Output)
3 Liz Daniel Jody Bert Tom
```

**EXPLANATION**

1. The array `@names` is assigned three values, "Jody", "Bert", and "Tom".
2. The `unshift` function will prepend "Liz" and "Daniel" to the array.

![Figure 5.6](image)

*Figure 5.6 Using the unshift function to add elements to the beginning of an array.*

5.3.2 **Removing and Replacing Elements**

The **delete** Function. If you have a row of shoeboxes and take a pair of shoes from one of the boxes, the number of shoeboxes remains the same, but one of them is now empty. That is how `delete` works with arrays. The `delete` function allows you to remove a value from an element of an array, but not the element itself. The value deleted is simply undefined. (See Figure 5.7.) But if you find it in older programs, perldoc.perl.org warns not to use it for arrays, but rather for deleting elements from a hash. In fact, perldoc.perl.org warns that calling `delete` on array values is deprecated and likely to be removed in a future version of Perl.
Instead, use the `splice` function to delete and replace elements from an array, while at the same time renumbering the index values.

```perl
splice(ARRAY, OFFSET, LENGTH, LIST)
splice(ARRAY, OFFSET, LENGTH)
splice(ARRAY, OFFSET)
```

**EXAMPLE 5.24**

(The Script)

```perl
use warnings;
#
# Splicing out elements of a list
1 my @colors=("red", "green", "purple", "blue", "brown");
2 print "The original array is @colors\n";
3 my @discarded = splice(@colors, 2, 2);
4 print "The elements removed after the splice are: @discarded.\n";
5 print "The spliced array is now @colors.\n";
```
**Example 5.24 (continued)**

(Output)
2 The original array is red green purple blue brown.
4 The elements removed after the splice are: purple blue.
5 The spliced array is now red green brown.

**Explanation**
1 An array of five colors is created.
3 The `splice` function removes elements purple and blue from the array and returns them to `@discarded`, starting at index position two, `$colors[2]`, with a length of two elements.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;red&quot;</td>
<td>&quot;green&quot;</td>
<td>&quot;purple&quot;</td>
<td>&quot;blue&quot;</td>
<td>&quot;brown&quot;</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

`@colors`

Before splice

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;red&quot;</td>
<td>&quot;green&quot;</td>
<td>&quot;brown&quot;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

`@colors`

After splice

**Figure 5.8** Using the `splice` function to remove or replace elements in an array.

**Example 5.25**

(The Script)
```
use warnings;
#
Splicing and replacing elements of a list
1 my @colors=("red", "green", "purple", "blue", "brown");
2 print "The original array is @colors\n";
3 my @lostcolors=splice(@colors, 2, 3, "yellow", "orange");
4 print "The removed items are @lostcolors\n";
5 print "The spliced array is now @colors\n";
```

(Output)
2 The original array is red green purple blue brown
4 The removed items are purple blue brown
5 The spliced array is now red green yellow orange
5.3 Array Functions

EXPLANATION
1. An array of five colors is created.
2. The original array is printed.
3. The \texttt{splice} function will delete elements starting at \texttt{colors[2]} and remove the next three elements. The removed elements (\texttt{purple}, \texttt{blue}, and \texttt{brown}) are stored in \texttt{@lostcolors}. The colors \texttt{yellow} and \texttt{orange} will replace the ones that were removed.
4. The values that were removed are stored in \texttt{@lostcolors} and printed.
5. The new array, after the \texttt{splice}, is printed.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5_9.png}
\caption{Splicing and replacing elements in an array.}
\end{figure}

The \texttt{pop} Function. The \texttt{pop} function pops off the last element of an array and returns it. The array size is subsequently decreased by one. (See Figure 5.10.)

\begin{table}[h]
\centering
\begin{tabular}{l}
\texttt{FORMAT} \\
\hline
\texttt{pop(ARRAY)} \\
\texttt{pop ARRAY} \\
\hline
\end{tabular}
\caption{Syntax for the \texttt{pop} function.}
\end{table}

\begin{example}
(In Script)
\begin{verbatim}
use warnings;
# Removing an element from the end of a list
1 my @names=("Bob", "Dan", "Tom", "Guy");
2 print "@names\n";
3 my $got = pop @names;  # Pops off last element of the array
4 print "$got\n";
5 print "@names\n";
\end{verbatim}
\end{example}
**Example 5.26 (continued)**

(Output)

2 Bob Dan Tom Guy
4 Guy
5 Bob Dan Tom

**Explanation**

1. The @name array is assigned a list of elements.
2. The array is printed.
3. The pop function removes the last element of the array and returns the popped item.
4. The $got scalar contains the popped item, Guy.
5. The new array is printed.

![Diagram](image)

**Figure 5.10** Using the pop function to pop the last element off the array.

**The shift Function.** The shift function shifts off and returns the first element of an array, decreasing the size of the array by one element. (See Figure 5.11.) If ARRAY is omitted, then the @ARGV array is shifted. If in a subroutine, the argument list, stored in the @_ array is shifted.

**Format**

```
shift(ARRAY)
shift ARRAY
shift
```
5.3 Array Functions

**Example 5.27**

(In Script)

```perl
use warnings;

# Removing elements from front of a list
my @names=("Bob", "Dan", "Tom", "Guy");
my $ret = shift @names;
print "@names\n";
print "The item shifted is $ret.\n";
```

(Output)

```
3 Dan Tom Guy
4 The item shifted is Bob.
```

**Explanation**

1. The array `@names` is assigned list values.
2. The `shift` function removes the first element of the array and returns that element to the scalar `$ret`, which is `Bob`.
3. The new array has been shortened by one element.

![Diagram](image)

**Figure 5.11** Using the `shift` function to return the first element of an array.

### 5.3.3 Deleting Newlines

**The chop and chomp Functions (with Lists).** The `chop` function chops off the last character of a string and returns the chopped character, usually for removing the newline after input is assigned to a scalar variable. If a list is chopped, `chop` will remove the last letter of each string in the list.

The `chomp` function removes a newline character at the end of a string or for each element in a list.
5.3.4 Searching for Elements and Index Values

The `grep` Function. The `grep` function is similar to the UNIX `grep` command in that it searches for patterns of characters, called regular expressions. However, unlike the UNIX `grep`, it is not limited to using regular expressions. Perl’s `grep` evaluates the expression (EXPR) for each element of the array (LIST), locally setting $$_ to each element. The return value is another array consisting of those elements for which the expression evaluated as true. As a scalar value, the return value is the number of times the expression was true (that is, the number of times the pattern was found).
### Format

```
grep BLOCK LIST
grep(EXPR, LIST)
```

### Example 5.29

(The Script)

```perl
use warnings;
# Searching for patterns in a list
my @list = ("tomatoes", "tomorrow", "potatoes", "phantom", "Tommy");

my $count = grep($_ =~ /tom/i, @list);
# $count = grep(/tom/i, @list);
@items = grep(/tom/i, @list); # Could say: grep {/tom/i} @list;

print "Found items: @items
Number found: $count
";
```

(Output)

```
Found items: tomatoes tomorrow phantom Tommy
Number found: 4
```

### Explanation

1. The array `@list` is assigned a list of elements.
2. The `grep` function searches for the pattern (regular expression) `tom`. The `$_` scalar is used as a placeholder for each item in the iterator `@list`. (`$_` is also an alias to each of the list values, so it can modify the list values.) Although omitted in the next example, it is still being used. The `i` turns off case sensitivity. When the return value is assigned to a scalar, the result is the number of times the regular expression was matched.
3. `grep` again searches for `tom`. The `i` turns off case sensitivity. When the return value is assigned to an array, the result is a list of the matched items.

The next example shows you how to find the index value(s) for specific elements in an array using the built-in `grep` function. (If you have version 5.10+, you may want to use the more efficient `List::MoreUtils` module from the standard Perl library, or from CPAN.)

### Example 5.30

(The Script)

```perl
use warnings;
my (@colors, $index);
# Searching for the index value where a pattern is found.
@colors = qw(red green blue orange blueblack);
@index_vals = grep( $colors[$_] =~ /blue/, (0.. $#colors));
print "Found index values: @index_vals where blue was found.\n";
```

(Output)

```
Found index values: 2 4 where blue was found.
```
EXPLANATION
1 The array @colors is assigned a list of elements.
2 The grep function searches for the pattern blue in each element of @colors. (See Chapter 8, “Regular Expressions—Pattern Matching,” for a detailed discussion on pattern matching.) The list (0 .. $#colors) represents the index values of @colors. $_ holds one value at a time from the list starting with 0. If, for example, in the first iteration, grep searches for the pattern blue in $colors[0], and finds red, nothing is returned because it doesn’t match. (=~ is the bind operator.) Then, the next item is checked. Does the value $colors[1], green, match blue? No. Then, the next item is checked. Does $colors[2] match blue? Yes it does. 2 is returned and stored in @index_vals. Another match for blue is true when $colors[4], blueblack, is matched against blue. 4 is added to @index_vals.
3 When the grep function finishes iterating over the list of index values, the results stored in @index_vals are printed.

5.3.5 Creating a List from a Scalar

The split Function. The split function splits up a string (EXPR) by some delimiter (whitespace, by default) and returns a list. (See Figure 5.12.) The first argument is the delimiter, and the second is the string to be split. The Perl split function can be used to create fields when processing files, just as you would with the UNIX awk command. If a string is not supplied as the expression, the $_ string is split.

The DELIMITER statement matches the delimiters that are used to separate the fields. If DELIMITER is omitted, the delimiter defaults to whitespace (spaces, tabs, or newlines). If the DELIMITER doesn’t match a delimiter, split returns the original string. You can specify more than one delimiter, using the regular expression metacharacter [ ]. For example, [ +	:] represents zero or more spaces or a tab or a colon.

To split on a dot (.), use /\./ to escape the dot from its regular expression metacharacter.

LIMIT specifies the number of fields that can be split. If there are more than LIMIT fields, the remaining fields will all be part of the last one. If the LIMIT is omitted, the split function has its own LIMIT, which is one more than the number of fields in EXPR. (See the -a switch for autosplit mode, in Appendix A, “Perl Built-ins, Pragmas, Modules, and the Debugger.”)

FORMAT

split("DELIMITER", EXPR, LIMIT)
split(/DELIMITER/, EXPR, LIMIT)
split(/DELIMITER/, EXPR)
split("DELIMITER", EXPR)
split(/DELIMITER/)
split
EXAMPLE 5.31

(The Script)

use warnings;

# Splitting a scalar on whitespace and creating a list
1 my $line="a b c d e";
2 my @letter=split('  ',$line);
3 print "The first letter is $letter[0]\n"
4 print "The second letter is $letter[1]\n"

(Output)
3 The first letter is a
4 The second letter is b

EXPLANATION

1 The scalar variable $line is assigned the string a b c d e.
2 The value in $line (scalar) is a single string of letters. The split function will split
   the string, using whitespace as a delimiter. The @letter array will be assigned the
   individual elements a, b, c, d, and e. Using single quotes as the delimiter is not the
   same as using the regular expression /  /. The ' ' resembles awk in splitting lines on
   whitespace. Leading whitespace is ignored. The regular expression / / includes leading
   whitespace, creating as many null initial fields as there are whitespaces.
3 The first element of the @letter array is printed.
4 The second element of the @letter array is printed.

Figure 5.12 Using the split function to create an array from a scalar.
**Example 5.32**

(The Script)

```perl
use warnings;
# Splitting up $_
my @line;
while(<DATA>){
    @line=split("\:");
    # or split (/:/, $$_);
    print "$line[0]\n";
}
```

/Data_

Igor Chevsky:385-375-8395:3567 Populus Place, Caldwell, NJ
23875:6/18/68:23400
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI
23874:3/28/45:245700
Jennifer Cowan:548-834-2348:583 Laurel Ave., Kingsville, TX
83745:10/1/35:58900
Fred Fardbarkle:674-843-1385:20 Park Lane, Duluth, MN 23850:4/12/23:78900

(Output)

Betty Boop
Igor Chevsky
Norma Corder
Jennifer Cowan
Fred Fardbarkle

**Explanation**

1. The $$_$ variable holds each line of the file DATA filehandle; the data being processed is below the __DATA__ line. Each line is assigned to $$_$. $$_$ is also the default line for split.
2. The split function splits the line, ($$_$), using the : as a delimiter and returns the line to the array, @line.
3. The first element of the @line array, line[0], is printed.

**Example 5.33**

(The Script)

```perl
use warnings;
my($name, $phone, $address, $bd, $sal);
# Splitting up $$_$ and creating an unnamed list
while(<DATA>){
    ($name,$phone,$address,$bd,$sal)=split("\:");
    print "$name	$phone\n" ;
}
```
5.3 Array Functions

Example 5.33 (continued)

```plaintext
__DATA__
Igor Chevsky:385-375-8395:3567 Populus Place, Caldwell, NJ
23875:6/18/68:23400
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI
23874:3/28/45:245700
Jennifer Cowan:548-834-2348:583 Laurel Ave., Kingsville, TX
83745:10/1/35:58900
Fred Fardbarkle:674-843-1385:20 Park Lane, Duluth, MN 23850:4/12/23:78900

(Output)
Betty Boop 245-836-8357
Igor Chevsky 385-375-8395
Norma Corder 397-857-2735
Jennifer Cowan 548-834-2348
Fred Fardbarkle 674-843-1385
```

Explanation

1. Perl loops through the DATA filehandle one line at a time from `__DATA__`, storing each successive item in the `$_` variable, overwriting what was previously stored there. The `split` function splits each line in `$_` using the colon as a delimiter.
2. The returned list consists of five scalars, `$name`, `$phone`, `$address`, `$bd`, and `$sal`. The values of `$name` and `$phone` are printed.

Example 5.34

(The Script)
```perl
use warnings;
# Many ways to split a scalar to create a list
my $string= "Joe Blow:11/12/86:10 Main St.:Boston, MA:02530";
my @line=split( ":", $string ); # The string delimiter is a colon
print @line,

print "The guy's name is $line[0].\n";
print "The birthday is $line[1].\n";

@line=split( ":", $string, 2 );
print $line[0],"\n"; # The first element of the array
print $line[1],"\n"; # The rest of the array because limit is 2
print $line[2],"\n"; # Nothing is printed

($name, $birth, $address)=split(" ", $string);

print $name,"\n";
print $birth,"\n";
print $address,"\n";
```
Chapter 5 • What’s in a Name?

**Example 5.34 (continued)**

(Output)
3 Joe Blow 11/12/86 10 Main St. Boston, MA 02530
4 The guy’s name is Joe Blow.
5 The birthday is 11/12/86.

7 Joe Blow
8 11/12/86:10 Main St.:Boston, MA:02530
9
11 Joe Blow
12 11/12/86
13 10 Main St.

**Explanation**
1 The scalar $string is split at each colon.
2 The delimiter is a colon. The limit is 2.
6 The string is split by colons and given a limit of two, meaning that the text up to the first colon will become the first element of the array; in this case, $line[0] and the rest of the string will be assigned to $line[1]. LIMIT, if not stated, will be one more than the total number of fields.
10 The string is split by colons and returns a list of scalars. This may make the code easier to read.

5.3.6 Creating a Scalar from a List

The `join` Function. The `join` function joins the elements of an array into a single string and separates each element of the array with a given delimiter, sometimes called the “glue” character(s) since it glues together the items in a list (opposite of `split`). (See Figure 5.13.) The expression `DELIMITER` is the value of the string that will join the array elements in `LIST`.

**Format**

`join(DELIMITER, LIST)`

**Example 5.35**

(The Script)
use warnings;
my(@colors, $color_string);
# Joining each elements of a list with commas
1 @colors = qw( red green blue);

2 $color_string = join(", ", @colors); # Create a string from an array
3 print "The new string is: $color_string\n";

(Output)
3 The new string is: red, green, blue
EXPLANATION
1. An array is assigned three colors.
2. The `join` function joins the three elements of the `@colors` array, using a comma and space as the delimiter returning a string, which is then assigned to `$color_string`.
3. The new string with commas is printed.

```
"red, green, blue" = "red" "green" "blue"
join("", ",", @colors)
```

**Figure 5.13** Using the `join` function to join elements of an array with a comma.

**EXAMPLE 5.36**

(The Script)
```
use warnings;
# Joining each element of a list with a newline
my @names = qw(Dan Dee Scotty Liz Tom);
@names = join("\n", sort(@names));
print @names,"\n";
```

(Output)
```
Dan
Dee
Liz
Scotty
Tom
```

EXPLANATION
1. The array `@names` is assigned a list of strings.
2. The `join` function will `join` each word in the list with a newline (`\n`) after the list has been sorted alphabetically.
3. The sorted list is printed with each element of the array on a line of its own.

### 5.3.7 Transforming an Array

**The map Function.** If you have an array and want to perform the same action on each element of the array without using a `for` loop, the `map` function may be an option. The `map` function maps each of the values in an array to an expression or block, returning another list with the results of the mapping. It lets you change the values of the original list.
Using *map* to Change All Elements of an Array
In the following example, the *chr* function is applied or mapped to each element of an array and returns a new array showing the results. (See Figure 5.14.)

<table>
<thead>
<tr>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 The array @list consists of six hexadecimal numbers and one octal number.</td>
</tr>
<tr>
<td>2 The <em>map</em> function maps each item in @list to its corresponding <em>chr</em> (character) value and returns a new list, assigned to @letters. (According to perldoc.perl.org, the <em>chr</em> function “returns the character represented by that NUMBER in the character set. For example, <em>chr</em>(65) is “A” in either ASCII or Unicode, and <em>chr</em>(0x263a) is a Unicode smiley face.”)</td>
</tr>
<tr>
<td>3 The new list is printed. Each numeric value was converted with the <em>chr</em> function to a character corresponding to its ASCII value; for example, <em>chr</em>(65) returns ASCII value “A”.</td>
</tr>
<tr>
<td>4 The array @n consists of a list of integers.</td>
</tr>
<tr>
<td>5 The <em>map</em> function evaluates the expression for each element in the @n array and returns the result to the new array @n.</td>
</tr>
<tr>
<td>6 The results of the mapping are printed, showing that the original list has been changed.</td>
</tr>
</tbody>
</table>
Using map to Remove Duplicates from an Array

The map function can be used to create a hash from an array. If you are using the array elements as keys for the new hash, any duplicates will be eliminated.

**Example 5.38**

(The Script)
```perl
use warnings;
my (@courses, %c);
@courses = qw( C++ C Perl Python French C C Perl);
%c = map { $_ => undef } @courses; # Create a unique list of keys
@courses = keys %c;
print "@courses\n";
```

(Output)
Python, French, Perl, C, C++

**Explanation**

1. The array of courses contains duplicates.
2. The map function is used to create a hash called %c. Each element in the array @courses is assigned in turn to $_. $_ serves as the key to the new %c hash. The value is left undefined since the keys are all we need to get a list of unique courses.
3. The keys in the %c hash are assigned to @courses, overwriting what was there. The new list will have no duplicate entries, although it will be unordered, as are all hashes.

### 5.3.8 Sorting an Array

**The sort Function.** The sort function sorts and returns a sorted list. Its default is to sort alphabetically, but you can define how you want to sort by using different comparison operators. If SUBROUTINE is specified, the first argument to sort is the name of the subroutine, followed by a list of values to be sorted. If the string cmp operator is used, the values in the list will be sorted alphabetically (ASCII sort), and if the <=> operator (called the space ship operator) is used, the values will be sorted numerically. The values are passed to the subroutine by reference and are received by the special Perl variables $a
and $b, not the normal @_ array. (See Chapter 11, “How Do Subroutines Function?” for further discussion.) Do not try to modify $a or $b, as they represent the values that are being sorted.

If you want Perl to sort your data according to a particular locale, your program should include the use locale pragma. For a complete discussion, see perldoc.perl.org/perllocale.

**FORMAT**

```perl
sort(SUBROUTINE LIST)
sort(LIST)
sort SUBROUTINE LIST
sort LIST
```

**EXAMPLE 5.39**

(The Script)
```
use warnings;
#
Simple alphabetic sort
my @list=("dog","cat","bird","snake" );
print "Original list: @list\n";
my @sorted = sort @list;
print "ASCII sort: @sorted\n";
#
Reversed alphabetic sort
@sorted = reverse sort @list;
print "Reversed ASCII sort: @sorted\n";
```

(Output)
```
Original list: dog cat bird snake
ASCII sort: bird cat dog snake
Reversed ASCII sort: snake dog cat bird
```

**EXPLANATION**

1. The @list array will contain a list of items to be sorted.
2. The sort function performs a string (lexographical for current locale) sort on the items. The sorted values must be assigned to another list or the same list. The sort function doesn’t change the original list.
3. The sorted string is printed.
4. This list is sorted alphabetically and then reversed.

**ASCII and Numeric Sort Using Subroutine**

You can either define a subroutine or use an inline function to perform customized sorting, as shown in the following examples. A note about $a and $b: they are special global Perl variables used by the sort function for comparing values. If you need more information on the operators used, see Chapter 6, “Where’s the Operator?”
EXAMPLE 5.40

(The Script)

use warnings;
my @list=("dog", "cat", "bird", "snake");
print "Original list: @list\n";
# ASCII sort using a subroutine
sub asc_sort{
    $a cmp $b;  # Sort ascending order
}
@sorted_list=sort asc_sort(@list);
print "ASCII sort: @sorted_list\n";

# Numeric sort using subroutine
sub numeric_sort {
    $a <=> $b;
}  # $a and $b are compared numerically
@number_sort=sort numeric_sort 10, 0, 5, 9.5, 10, 1000;
print "Numeric sort: @number_sort.\n";

(Output)
Original list: dog cat bird snake
ASCII sort: bird cat dog snake
Numeric sort: 0 5 9.5 10 10 1000.

EXPLANATION

1. The @list array will contain a list of items to be sorted.
2. The subroutine asc_sort() is sent a list of strings to be sorted.
3. The special global variables $a and $b are used when comparing the items to be sorted in ascending order. If $a and $b are reversed (for example, $b cmp $a), then the sort is done in descending order. The cmp operator is used when comparing strings.
4. The sort function sends a list to the asc_sort(), user-defined subroutine, where the sorting is done. The sorted list will be returned and stored in @sorted_list.
5. This is a user-defined subroutine, called numeric_sort(). The special variables $a and $b compare the items to be sorted numerically, in ascending order. If $a and $b are reversed (for example, $b <=> $a), then the sort is done in numeric descending order. The <=> operator is used when comparing numbers.
6. The sort function sends a list of numbers to the numeric_sort() function and gets back a list of sorted numbers, stored in the @number_sort array.
Example 5.41

(The Script)

```perl
use warnings;
# Sorting numbers with block
my @sorted_numbers = sort { $a <=> $b } (3, 4, 1, 2);
print "The sorted numbers are: @sorted_numbers", ".\n";
```

(Output)

```
The sorted numbers are: 1 2 3 4.
```

Explanation

1. The `sort` function is given a block, also called an **inline subroutine**, to sort a list of numbers passed as arguments. The `<>` operator is used with variables `$a` and `$b` to compare the numbers. The sorted numeric list is returned and stored in the array `@sorted_numbers`. (See [http://perldoc.perl.org/functions/sort.html](http://perldoc.perl.org/functions/sort.html) for more on the `sort` function.)

2. The sorted list is printed.

5.3.9 Checking the Existence of an Array Index Value

**The `exists` Function.** The `exists` function returns *true* if an array index (or hash key) has been defined, and *false* if it has not. It is most commonly used when testing a hash key's existence.

Format

```perl
exists $ARRAY[index];
```

Example 5.42

```perl
use warnings;
my @names = qw(Tom Raul Steve Jon);
print "Hello $names[1]\n", if exists $names[1];
print "Out of range!\n", if not exists $names[5];
```

(Output)

```
Hello Raul
Out of range!
```

Explanation

1. An array of names is assigned to `@names`.
2. If the index 1 is defined, the `exists` function returns true and the string is printed.
3. If the index 5 does not exist (and in this example it doesn't), then the string *Out of range!* is printed.
5.3.10 Reversing an Array

The reverse Function. The reverse function reverses the elements in a list, so that if the values appeared in descending order, now they are in ascending order, or vice versa. In scalar context, it concatenates the list elements and returns a string with all the characters reversed; for example, in scalar context Hello, there! reverses to !ereht ,olleH.

**FORMAT**

```plaintext
reverse(LIST)
reverse LIST
```

**EXAMPLE 5.43**

(In Script)

```perl
use warnings;
my(@names, @reversed);
# Reversing the elements of an array
1 @names=("Bob", "Dan", "Tom", "Guy");
2 print "@names \n";
3 @reversed=reverse @names;
4 print "@reversed\n";
```

(Output)

```plaintext
2  Bob Dan Tom Guy
4  Guy Tom Dan Bob
```

**EXPLANATION**

1 The array @names is assigned list values.
2 The original array is printed.
3 The reverse function reverses the elements in the list and returns the reversed list. It does not change the original array; that is, the array @names is not changed. The reversed items are stored in @reversed.
4 The reversed array is printed.

5.4 Hash (Associative Array) Functions

5.4.1 The keys Function

The keys function returns, in random order, an array whose elements are the keys of a hash (see also Section 5.4.2, “The values Function,” and Section 5.4.3, “The each Function”). Starting with Perl 5.12, keys also returns the index values of an array. In scalar context, it returns the number of keys (or indices).
FORMAT

keys (ASSOC_ARRAY)
keys ASSOC_ARRAY

EXAMPLE 5.44

(In Script)
use warnings;
my (%weekday, @daynumber, $key);
# The keys function returns the keys of a hash
1 %weekday = ( '1' => 'Monday',
                '2' => 'Tuesday',
                '3' => 'Wednesday',
                '4' => 'Thursday',
                '5' => 'Friday',
                '6' => 'Saturday',
                '7' => 'Sunday',
            );
2 @daynumber = keys (%weekday);
3 print '@daynumber
';
4 foreach $key ( keys (%weekday) ) { print "$key
";}
      print "\n";
5 foreach $key ( sort keys (%weekday) ) { print "$key
";}
      print "\n";

(Output)
6 4 1 3 7 2 5
6 4 1 3 7 2 5
1 2 3 4 5 6 7

EXPLANATION

1 The hash %weekday is assigned keys and values.
2 The keys function returns a list of all the keys in a hash. In this example, @daynumber
   is an unordered list of all the keys in the %weekday hash.
4 The keys function returns a list of keys. The foreach loop will traverse the list of keys,
   one at a time, printing the keys.
5 The keys function returns a list of keys in %weekday hash. The list will then be sorted,
   and finally the foreach loop will traverse the sorted list of keys, one at a time, printing
   each key.

5.4.2 The values Function

The values function returns, in random order, a list consisting of all the values of a named
hash. (After Perl 5.12, it will also return the values of an array.) In scalar context, it returns
the number of values.
5.4 Hash (Associative Array) Functions

**FORMAT**

values(ASSOC_ARRAY)
values ASSOC_ARRAY

**EXAMPLE 5.45**

(In Script)
use warnings;

# The values function returns the values in a hash
1 my $weekday= {
   '1' =>'Monday',
   '2' =>'Tuesday',
   '3' =>'Wednesday',
   '4' =>'Thursday',
   '5' =>'Friday',
   '6' =>'Saturday',
   '7' =>'Sunday',
};
2 foreach my $val ( values($weekday)) {print "$val ";}
print "\n";

(Output)
2 Saturday Thursday Monday Wednesday Sunday Tuesday Friday

**EXPLANATION**

1 The hash $weekday is assigned keys and values.
2 The values function returns a list of values from the hash $weekday. The foreach is used to loop through the list of values, one at a time, using $val as its loop variable.

Since hashes are stored in a random order, to get the hash values in the order in which they were assigned, you can use a hash slice as shown in the following example.

**EXAMPLE 5.46**

(In Script)
use warnings;

# Use a hash slice to get the values returned in order.
1 my $weekday= {
   '1' =>'Monday',
   '2' =>'Tuesday',
   '3' =>'Wednesday',
   '4' =>'Thursday',
   '5' =>'Friday',
   '6' =>'Saturday',
   '7' =>'Sunday',
};
**EXAMPLE 5.46 (continued)**

```perl
2  my @days = @weekday{1..7};
    print "@days\n";

(Output)
2  Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

**EXPLANATION**

1. The hash `%weekday` is assigned keys and values.
2. CA hash slice is a way of referring to one or more elements of the hash in one statement, to get a list of values, or to assign a list of values, and because it is using a list of keys, the list is preceded by the @ sign and the list is enclosed in curly braces to indicate that your are indexing a hash.*

* To preserve the insert order of hash keys, see Tie::InsertOrderHash at the Comprehensive Perl Archive Network—CPAN (http://search.cpan.org).

### 5.4.3 The each Function

The `each` function returns, in random order, a two-element list whose elements are the key and the corresponding value of a hash. It must be called multiple times to get each key/value pair, as it only returns one set each time it is called, somewhat like reading lines from a file, one at a time.

**FORMAT**

```perl
each(ASSOC_ARRAY)
each ASSOC_ARRAY
```

**EXAMPLE 5.47**

(In Script)
```perl
use warnings;
my(%weekday, $key, $value);
# The each function retrieves both keys and values from a hash
1  %weekday=
   'Mon' => 'Monday',
   'Tue' => 'Tuesday',
   'Wed' => 'Wednesday',
   'Thu' => 'Thursday',
   'Fri' => 'Friday',
   'Sat' => 'Saturday',
   'Sun' => 'Sunday',
);
2  while((($key,$value)=each(%weekday))){
3    print "$key = $value\n";
```

**Example 5.47 (continued)**

(Output)
3 Sat = Saturday
Fri = Friday
Sun = Sunday
Thu = Thursday
Wed = Wednesday
Tue = Tuesday
Mon = Monday

**Explanation**

1. The hash `%weekday` is assigned keys and values.
2. The `each` function returns a list consisting of each key and its associated value from the `%weekday` hash. They are assigned to the scalars `$key` and `$value`, respectively.
3. The keys and values are printed, but in an unordered way. You can order them as shown in Example 5.46 or use a `foreach` loop with an ordered list of keys:

```perl
foreach $key(1..7){
    print $weekday{$key},"\n";
}
```

### 5.4.4 Removing Duplicates from a List with a Hash

Earlier, we used a hash to remove duplicate entries in an array. In the following example, the built-in `map` function is used to map each element of an array into a hash to create unique hash keys.

**Example 5.48**

(The Script)
```perl
use warnings;
my(@list, @uniq);
# Using the map function with a hash
@list = qw/a b c d a e b a b d e f/;
1 @uniq = keys %{ map {$_ => 1 } @list };
2 print "@list\n@uniq\n";
```

(Output)
```
a b c d a e b a b d e f
e c a b d f
```

**Explanation**

1. The `map` function iterates through the values in the `@list` array to create a hash where each element in `@list` becomes a key, `$_`, to an unnamed hash with each key getting a corresponding value of 1. After the hash is created, the built-in `keys` function returns a list of the unique keys which are assigned to the array `@uniq`.
2. Both the original list, `@list`, and the new list, `@uniq`, are printed, showing that the duplicate values in the original list have been removed.
5.4.5 Sorting a Hash by Keys and Values

When sorting a hash, you can sort the keys alphabetically very easily by using the built-in sort command, as we did with arrays in the preceding section. But you may want to sort the keys numerically or sort the hash by its values. To do this requires a little more work.

You can define a subroutine to compare the keys or values. (See Chapter 11, “How Do Subroutines Function?”) The subroutine will be called by the built-in sort function. It will be sent a list of keys or values to be compared. The comparison is either an ASCII (alphabetic) or a numeric comparison, depending upon the operator used. The cmp operator is used for comparing strings, and the <=> operator is used for comparing numbers. The reserved global scalars $a, and $b are used in the subroutine to hold the values as they are being compared. The names of these scalars cannot be changed.

**Sort Hash by Keys in Ascending Order.** To perform an ASCII, or alphabetic, sort on the keys in a hash is relatively easy. Perl’s sort function is given a list of keys and returns them sorted in ascending order. A foreach loop is used to loop through the hash keys, one key at a time.

```
Example 5.49

(In Script)
  use warnings;
  my %wins = (   
    "Portland Panthers" => 10,  
    "Sunnyvale Sluggers" => 12,  
    "Chico Wildcats" => 5,  
    "Stevensville Tigers" => 6,  
    "Lewiston Blazers" => 11,  
    "Danville Terriors" => 8,  
  );
  print "\n\nSort Teams in Ascending Order:\n\n";
  foreach my $key(sort keys %wins) {
    printf "\t% -20s%5d\n", $key, $wins{$key};
  }

(Output)

Sort Teams in Ascending Order:

    Chico Wildcats          5
    Danville Terriors       8
    Lewiston Blazers       11
    Portland Panthers      10
    Stevensville Tigers     6
    Sunnyvale Sluggers     12
```
5.4 Hash (Associative Array) Functions

**EXPLANATION**

1. A hash called `%wins` is assigned key/value pairs.
2. The `foreach` loop will be used to iterate through each of an alphabetically sorted list of keys from a hash called `%wins`.
3. The `printf()` function formats and prints the sorted keys and its values.

**Sort Hash by Keys in Reverse Order.** To sort a hash by keys alphabetically and in descending order, just add the built-in `reverse` function to the previous example. The `foreach` loop is used to get each key from the hash, one at a time, after the reversed sort.

**EXAMPLE 5.50**

(In Script)
```
use warnings;
my %wins = (
    "Portland Panthers" => 10,
    "Sunnyvale Sluggers" => 12,
    "Chico Wildcats" => 5,
    "Stevensville Tigers" => 6,
    "Lewiston Blazers" => 11,
    "Danville Terriors" => 8,
);
print "\n\nSort Teams in Descending/Reverse Order: \n\n";
foreach my $key (reverse sort keys %wins) {
    printf "\n\t% -20s%5d\n", $key, $wins{$key};
}
```

(Output)
```
Sort Teams in Descending/Reverse Order:

Sunnyvale Sluggers 12
Stevensville Tigers 6
Portland Panthers 10
Lewiston Blazers 11
Danville Terriors 8
Chico Wildcats 5
```

**EXPLANATION**

1. A hash called `%wins` is assigned key/value pairs.
2. The `foreach` loop will be used to iterate through each of the elements in the hash. The `reverse` function takes the alphabetically sorted list returned from the `sort` function and reverses it.
3. The `printf()` function formats and prints the keys and sorted values.
Sort Hash by Keys Numerically. A user-defined subroutine is used to sort a hash by keys numerically. In the subroutine, Perl's special $a and $b variables are used to hold the value being compared with the appropriate operator. For numeric comparison, the <=> operator is used, and for string comparison, the cmp operator is used. The sort function will send a list of keys to the user-defined subroutine. The sorted list is returned.

**Example 5.51**

(In Script)
```
use warnings;
sub desc_sort_subject {
    $b <=> $a;    # Numeric sort descending
}
sub asc_sort_subject {
    $a <=> $b;    # Numeric sort ascending
}
my %courses = {
    "101" => "Intro to Computer Science",
    "221" => "Linguistics",
    "300" => "Astronomy",
    "102" => "Perl",
    "103" => "PHP",
    "200" => "Language arts",
};
print "\n\tCourses in Ascending Numeric Order:\n";
foreach my $key (sort asc_sort_subject(keys %courses)) {
    printf \"\t%-5d%s\n\", $key, $courses{"$key"};
}
print "\n\tCourses in Descending Numeric Order:\n";
foreach my $key (sort desc_sort_subject(keys %courses)) {
    printf \"\t%-5d%s\n\", $key, $courses{"$key"};
}
```

(Output)
Courses in Ascending Numeric Order:
101 Intro to Computer Science
102 Perl
103 PHP
200 Language arts
221 Linguistics
300 Astronomy

Courses in Descending Numeric Order:
300 Astronomy
221 Linguistics
200 Language arts
103 PHP
102 Perl
101 Intro to Computer Science
5.4 Hash (Associative Array) Functions

**EXPLANATION**

1. This is a user-defined subroutine called `desc_sort_subject`. When its name is given to the `sort` function, this function will be used to compare the keys passed to it. It will sort the keys numerically.

2. The special Perl variables `$a` and `$b` are used to compare the values of the keys from the hash called `%courses`. The `<=>` operator is a numeric comparison operator that will compare each of the keys to be sorted as numbers. In the previous examples, we sorted the keys alphabetically. Since `$b` precedes `$a`, the sort is descending.

3. This is also a user-defined subroutine called `asc_sort_subject`. This function is identical to the previous function on line 1, except it will sort the keys of the hash in ascending numeric order rather than descending.

4. In this function, the special variables `$a` and `$b` have been reversed, causing the sort after the comparison to be in ascending order.

5. The hash called `%courses` is defined with key/value pairs.

6. The `foreach` loop will be used to iterate through each of the keys in the hash. It receives its list from the output of the `sort` command.

7, 8. The `printf` function formats and prints the keys and sorted values.

**Numerically Sort a Hash by Values in Ascending Order.** To sort a hash by its values, a user-defined function is also defined. The values of the hash are compared by the special variables `$a` and `$b`. If `$a` is on the left-hand side of the comparison operator, the sort is in ascending order, and if `$b` is on the left-hand side, then the sort is in descending order. The `<=>` operator compares its operands numerically.

**EXAMPLE 5.52**

(In Script)
```perl
use warnings;

sub asc_sort_wins {
    $wins{$a} <=> $wins{$b};
}

my %wins = (
    "Portland Panthers" => 10,
    "Sunnyvale Sluggers" => 12,
    "Chico Wildcats" => 5,
    "Stevensville Tigers" => 6,
    "Lewiston Blazers" => 11,
    "Danville Terriors" => 8,
);

print "\n\tWins in Ascending Numeric Order:\n\n";

foreach my $key (sort asc_sort_wins(keys %wins)) {
    printf "\t\% -20s\%5d\n", $key, $wins{$key};
}
```
Chapter 5  •  What’s in a Name?

EXAMPLE 5.52 (CONTINUED)

(Output)

Wins in Ascending Numeric Order:

<table>
<thead>
<tr>
<th>Team</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chico Wildcats</td>
<td>5</td>
</tr>
<tr>
<td>Stevensville Tigers</td>
<td>6</td>
</tr>
<tr>
<td>Danville Terriors</td>
<td>8</td>
</tr>
<tr>
<td>Portland Panthers</td>
<td>10</td>
</tr>
<tr>
<td>Lewiston Blazers</td>
<td>11</td>
</tr>
<tr>
<td>Sunnyvale Sluggers</td>
<td>12</td>
</tr>
</tbody>
</table>

EXPLANATION

1. This is a user-defined subroutine called `asc_sort_wins`. When its name is given to the `sort` function, this function will be used to compare the hash values passed to it. It will sort the values by value, numerically.

2. The special Perl variables `$a` and `$b` are used to compare the values of the hash called `$wins`. The `<=>` operator is a numeric comparison operator that will compare each of the values to be sorted. To compare strings, the `cmp` operator is used.

3. The hash called `%wins` is assigned key/value pairs.

4. The `foreach` loop iterates through each of the elements in the hash. It receives its list from what is returned from the `sort` function.

5. The `printf` function formats and prints the keys and sorted values.

Numerically Sort a Hash by Values in Descending Order. To sort a hash numerically and in descending order by its values, a user-defined function is created as in the previous example. However, this time the `$b` variable is on the left-hand side of the `<=>` numeric operator, and the `$a` variable is on the right-hand side. This causes the `sort` function to sort in descending order.

EXAMPLE 5.53

(In Script)

```perl
use warnings;

# Sorting a hash by value in descending order

sub desc_sort_wins {
    $wins{$b} <=> $wins{$a};  # Reverse $a and $b
}

my %wins = (  
    "Portland Panthers" => 10,  
    "Sunnyvale Sluggers" => 12,  
    "Chico Wildcats" => 5,  
    "Stevensville Tigers" => 6,  
    "Lewiston Blazers" => 11,  
    "Danville Terriors" => 8,  
);  
```
5.4 Hash (Associative Array) Functions

**Example 5.53 (continued)**

```
print "\n\nWins in Descending Numeric Order:\n\n";
4 foreach my $key (sort desc_sort_wins(keys %wins)){
5   printf "\t-20s%5d\n", $key, $wins{$key};
}
```

(Output)

**Wins in Descending Numeric Order:**

- Sunnyvale Sluggers 12
- Lewiston Blazers 11
- Portland Panthers 10
- Danville Terriors 8
- Stevensville Tigers 6
- Chico Wildcats 5

**Explanation**

1. This is a user-defined subroutine called `desc_sort_wins`. When its name is given to the `sort` function, this function will be used to compare the hash values passed to it. It will sort the values by value, numerically but in descending order.
2. The special Perl variables `$a` and `$b` are used to compare the values of the hash called `$wins`. The position of `$a` and `$b` determines whether the sort is in ascending or descending order. If `$a` is on the left-hand side of the `<>` operator, the sort is a numeric ascending sort; if `$b` is on the left-hand side of the `<>` operator, the sort is descending. To compare strings, the `cmp` operator is used.
3. The hash called `%wins` is assigned key/value pairs.
4. The `foreach` loop will be used to iterate through each of the keys in the hash. It receives its list from what is returned from the `sort` function.
5. The `printf` function formats and prints the keys and sorted values.

### 5.4.6 The delete Function

The `delete` function deletes a specified element from a hash. The deleted value is returned if successful.5

**Example 5.54**

```
(In Script)
use warnings;
1 my %employees=(
   "Nightwatchman" => "Joe Blow",
   "Janitor" => "Teddy Plunger",
   "Clerk" => "Sally Olivetti",
);
```

5. If a value in an `%ENV` hash is deleted, the environment is changed. (See “The `%ENV` Hash” on page 137.)
**Example 5.54 (continued)**

```perl
my $layoff = delete $employees{"Janitor"};
print "We had to let $layoff go.\n";
print "Our remaining staff includes: \n";
while((my $key, my $value)=each %employees) {
    print "$key: $value\n";
}
```

(Output)
We had to let Teddy Plunger go.
Our remaining staff includes:
Nightwatchman: Joe Blow
Clerk: Sally Olivetti

**Explanation**

1. A hash is defined with three key/value pairs.
2. The `delete` function deletes an element from the specified hash by specifying the key. `Janitor` is the key. Both key and value are removed. The hash value associated with the key `Janitor` is removed and returned. The value `Teddy Plunger` is returned and assigned to the scalar `$layoff`.

### 5.4.7 The `exists` Function

The `exists` function returns true if a hash key (or array index) exists, and false if not.

**Format**

```perl
exists $ASSOC_ARRAY{KEY}
```

**Example 5.55**

```perl
use warnings;

my %employees = (  
    "Nightwatchman" => "Joe Blow",  
    "Janitor" => "Teddy Plunger",  
    "Clerk" => "Sally Olivetti",  
);

print "The Nightwatchman exists.\n" if exists $employees{"Nightwatchman"};
print "The Clerk exists.\n" if exists $employees{"Clerk"};
print "The Boss does not exist.\n" if not exists $employees{"Boss"};
```

(Output)
2. The Nightwatchman exists.
3. The Clerk exists.
4. The Boss does not exist.
5.4 Hash (Associative Array) Functions

<table>
<thead>
<tr>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 A hash is defined with three key/value pairs.</td>
</tr>
<tr>
<td>2 If a key &quot;Nightwatchman&quot; exists, the exists function returns true.</td>
</tr>
<tr>
<td>3 If a key &quot;Clerk&quot; exists, the exists function returns true.</td>
</tr>
<tr>
<td>4 If the key &quot;Clerk&quot; does not exist, the inverted value of the exists function is false.</td>
</tr>
</tbody>
</table>

5.4.8 Special Hashes

The %ENV Hash. The %ENV hash contains the environment variables handed to Perl from the parent process; for example, a shell or a Web server. The key is the name of the environment variable, and the value is what was assigned to it. If you change the value of %ENV, you will alter the environment for your Perl script and any processes spawned from it, but not the parent process. Environment variables play a significant role in CGI Perl scripts.

<table>
<thead>
<tr>
<th>EXAMPLE 5.56</th>
</tr>
</thead>
<tbody>
<tr>
<td>(In Script)</td>
</tr>
<tr>
<td>use warnings;</td>
</tr>
<tr>
<td>1 foreach my $key (keys %ENV){</td>
</tr>
<tr>
<td>2 print &quot;$key\n&quot;;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>3 print &quot;\nYour login name $ENV{'LOGNAME'}\n&quot;;</td>
</tr>
<tr>
<td>4 my $pwd = $ENV{'PWD'};</td>
</tr>
<tr>
<td>5 print &quot;\n&quot;, $pwd, &quot;\n&quot;;</td>
</tr>
<tr>
<td>(Output)</td>
</tr>
<tr>
<td>2 OPENWINHOME</td>
</tr>
<tr>
<td>3 MANPATH</td>
</tr>
<tr>
<td>4 FONTPATH</td>
</tr>
<tr>
<td>5 LOGNAME</td>
</tr>
<tr>
<td>6 USER</td>
</tr>
<tr>
<td>7 TERM</td>
</tr>
<tr>
<td>8 TERMCAP</td>
</tr>
<tr>
<td>9 SHELL</td>
</tr>
<tr>
<td>10 PWD</td>
</tr>
<tr>
<td>11 HOME</td>
</tr>
<tr>
<td>12 PATH</td>
</tr>
<tr>
<td>13 WINDOW_PARENT</td>
</tr>
<tr>
<td>14 WMGR_ENV_PLACEHOLDER</td>
</tr>
<tr>
<td>3 Your login name is ellie</td>
</tr>
<tr>
<td>5 /home/jody/home</td>
</tr>
</tbody>
</table>
**EXPLANATION**

1. The `foreach` loop iterates through the keys of the `%ENV` hash.
2. Print the value of the key `LOGNAME`.
3. Assign the value of the key `PWD` to `$pwd`.
4. Print the value of `$pwd`, the present working directory.

---

**The %SIG Hash.** The %SIG hash allows you to set signal handlers for signals. If, for example, you press <CTRL>+C when your program is running, that is a signal, identified by the name `SIGINT`. (See UNIX manual pages for a complete list of signals.) The default action of `SIGINT` is to interrupt your process. The signal handler is a subroutine that is automatically called when a signal is sent to the process. Normally, the handler is used to perform a clean-up operation or to check some flag value before the script aborts. (All signal handlers are assumed to be set in the main package.)

The %SIG hash contains values only for signals set within the Perl script.

---

**EXAMPLE 5.57**

(In Script)
```
use warnings;
sub handler {
    local($sig) = @_;  # First argument is signal name
    print "Caught SIG$sig -- shutting down\n";
    exit(0);
}
$SIG{'INT'} = 'handler';  # Catch <CTRL>+C
print "Here I am!\n";
sleep(10);
$SIG{'INT'}='DEFAULT';
$SIG{'INT'}='IGNORE';
< Program continues here >
```

**EXPLANATION**

1. `handler` is the name of the subroutine. The subroutine is defined.
2. `$sig` is a local variable and will be assigned the signal name.
3. When the `SIGINT` signal arrives, this message will appear, and the script will exit.
4. The value assigned to the key `INT` is the name of the subroutine, `handler`. When the signal arrives, the handler is called.
5. The `sleep` function gives you 10 seconds to press <CTRL>+C to see what happens.
6. The default action is restored. The default action is to abort the process if the user presses <CTRL>+C.
7. If you assign the value `IGNORE` to the `$SIG` hash, then <CTRL>+C will be completely ignored and the program will continue.
The `%INC` Hash. The `%INC` hash contains the entries for each filename that has been included via the `use` or `require` functions. The key is the filename; the value is the location of the actual file found.

5.4.9 Context Revisited

In summary, the way Perl evaluates variables depends on how the variables are being used; they are evaluated by context, either scalar, list, or void.

If the value on the left-hand side of an assignment statement is a scalar, the expression on the right-hand side is evaluated in a scalar context; whereas if the value on the left-hand side is an array, the right-hand side is evaluated in a list context.

Void context is a special form of scalar context. It is defined by the Perl monks as a “context that doesn’t have an operator working on it. The value of a thing in void context is discarded, not used for anything...” An example of void context is when you assign a list to a scalar separating the elements with a comma. The comma operator evaluates its left argument in void context, throws it away, then evaluates the right argument, and so on, until it reaches the end of the list, discarding all but the last one.

```perl
$fruit = ("apple","pear","peach");  # $fruit is assigned "peach";
# "apple" and "pear" are discarded
# as useless use in void context
```

You’ll see examples throughout the rest of this book where context plays a major role.

---

**Example 5.58**

(The `perldoc` function describes how `reverse` works)

```bash
1  $ perldoc -f reverse
   reverse LIST
   In list context, returns a list value consisting of the elements of LIST in the opposite order. In scalar context, concatenates the elements of LIST and returns a string value with all characters in the opposite order.
   ....
```

**Example 5.59**

(The Perl Script)

```perl
use warnings;
1  my @list = (90,89,78,100,87);
2  my $str="Hello, world"
3  print "Original array: @list\n";
4  print "Original string: $str\n";
5  my @revlist = reverse @list;
```
**Example 5.59 (Continued)**

6  my $revstr = reverse $str;
7  print "Reversed array is: @revlist\n";
8  print "Reversed string is: $revstr\n";
9  my $newstring = reverse @list;
10 print "List reversed, context string: $newstring\n";
11 "Later, going into the Void!!!\n";  # Void context

(Output)
11 Useless use of a constant ("Later, going into the void\n")
   in void context at Example line 13.
3  Original array: 90 89 78 100 87
4  Original string: Hello, world
7  Reversed array is: 87 100 78 89 90
8  Reversed string is: dlrow ,olleH
10 List reversed, context string: 78001879809

**Explanation**

11 This is a case where you will see a warning message about using void context when you have a string constant that is not being used in assignment, print out, or doesn’t return anything, and appears to be doing nothing. It doesn’t have any side effects and doesn’t break the program, but demonstrates a case where Perl views void context.

5  Context is demonstrated in the documentation for Perl’s built-in `reverse` function.
6  The `reverse` function reverses the elements of an array and returns the reversed elements to another array. Context is list.
8  This time, the `reverse` function reverses the characters in a string. It returns the reverse string as a scalar. Context is scalar.
9  Here the `reverse` function reverses the array again, but the returned value will be assigned to a string. The context being scalar, the function will reverse the array elements and convert the list into a string of characters.

### 5.5 What You Should Know

1. If you don’t give a variable a value, what will Perl assign to it?
2. What are “funny characters”? What is a sigil?
3. What data types are interpreted within double quotes?
4. How many numbers or strings can you store in a scalar variable?
5. In a hash, can you have more than one key with the same name? What about more than one value with the same name?
6. What function would you use to find the index value of an array if you know the value of the data stored there?
7. How does the \textit{scalar} function evaluate an expression if it's an array?
8. How do you find the size of an array?
9. What does the $^n$ special variable do?
10. When are elements of an array or hash preceded by a $(dollar sign)$?
11. What is the difference between \textit{chop} and \textit{chomp}?
12. What is the difference between \textit{splice} and \textit{slice}?
13. What does the \textit{map} function do?
14. How do you sort a numeric array? How do you sort a hash by value?
15. What function extracts both keys and values from a hash?
16. How can you remove duplicates in an array?
17. What is meant by the term \textit{scope}?
18. What is “scalar” context, “list” context, “void” context? Would you be able to write an example to demonstrate how they differ?

\section*{5.6 What's Next?}

In the next chapter, we discuss the Perl operators. We will cover the different types of assignment operators, comparison and logical operators, arithmetic and bitwise operators, how Perl sees strings and numbers, how to create a range of numbers, how to generate random numbers, and some special string functions.

\begin{center}
\textbf{EXERCISE 5}
\textbf{The Funny Characters}
\end{center}

1. Write a script that will ask the user for his five favorite foods (read from \textit{STDIN}). The foods will be stored as a string in a scalar, each food separated by a comma.
   a. Split the scalar by the comma and create an array.
   b. Print the array.
   c. Print the first and last elements of the array.
   d. Print the number of elements in the array.
   e. Use an array slice of three elements in the \textit{food} array and assign those values to another array. Print the new array with spaces between each of the elements.
2. Given the array @names=qw(Nick Susan Chet Dolly Bill), write a statement that would do the following:
   a. Replace Susan and Chet with Ellie, Beatrice, and Charles.
   b. Remove Bill from the array.
   c. Add Lewis and Izzy to the end of the array.
   d. Remove Nick from the beginning of the array.
   e. Reverse the array.
   f. Add Archie to the beginning of the array.
   g. Sort the array.
   h. Remove Chet and Dolly and replace them with Christian and Daniel.

3. Write a script called elective that will contain a hash. The keys will be code numbers—2CPR2B, 1UNX1B, 3SH414, 4PL400. The values will be course names—C Language, Intro to UNIX, Shell Programming, Perl Programming.
   a. Sort the hash by values and print it.
   b. Ask the user to type the code number for the course he plans to take this semester and print a line resembling the following:

           You will be taking Shell Programming this semester.

4. Modify your elective script to produce output resembling the output below. The user will be asked to enter registration information and to select an EDP number from a menu. The course name will be printed. It doesn't matter if the user types in the EDP number with upper- or lowercase letters. A message will confirm the user's address and thank him for enrolling.

   Output should resemble the following:

   REGISTRATION INFORMATION FOR SPRING QUARTER
   Today's date is Wed Apr 19 17:40:19 PDT 2014
   Please enter the following information:
   Your full name: Fred Z. Stachelin
   What is your Social Security Number (xxx-xx-yyyy): 004-34-1234
   Your address:
       StreetHobartSt
       CityStateZipChicoCA
“EDP” NUMBERS AND ELECTIVES:

2CPR2B | C Programming

IUNIX1B | Intro to UNIX

4PL400 | Perl Programming

3SH414 | Shell Programming

What is the EDP number of the course you wish to take? 4pl400
The course you will be taking is “Perl Programming.”

Registration confirmation will be sent to your address at
1424 HOBART ST.
CHICO, CA 95926

Thank you, Fred, for enrolling.

5. Write a script called findem that will do the following:
   a. Assign the contents of the datebook file to an array. (The datebook file is on the CD that accompanies this book.)
   b. Ask the user for the name of a person to find. Use the built-in grep function to find the elements of the array that contain the person and number of times that person is found in the array. The search will ignore case.
   c. Use the split function to get the current phone number.
   d. Use the splice function to replace the current phone number with the new phone number, or use any of the other built-in array functions to produce output that resembles the following:

   Who are you searching for? Karen

   What is the new phone number for Karen? 530-222-1255
   Karen’s phone number is currently 284-758-2857.
   Here is the line showing the new phone number:

   Karen Evich:530-222-1255:23 Edgecliff Place, Lincoln, NB 92086:7/25/53:85100\n   Karen was found in the array three times.
6. Write a script called *tellme* that will print out the names, phones, and salaries of all the people in the *datebook* file. To execute, type the following at the command line:

    tellme datebook

Output should resemble the following:

    Salary: 14500
    Name: Betty Boop
    Phone: 245-836-8357

7. The following array contains a list of values with duplicates.

    @animals=qw( cat dog bird cat bird monkey elephant cat elephant pig horse cat);

    a. Remove the duplicates with the built-in *map* function.
    b. Sort the list.
    c. Use the built-in *grep* function to get the index value for the *monkey*. 
Index

Symbols

`!` operator, 222
`/` operator, 163
`=` operator, 158
$_$ (topic variable) function, 90–91, 300
$_$ scalar, 223
$` operator, 163
!= operator, 158
$_$ (topic variable) function, 90–91, 300
$ perldoc DBI, 558
$_$ scalar, 223
$ sign, 52
$_$ variable, 240
$$ variables, 635–636
%ENV hash, 137–138
%INC hash, 139
%=` operator, 151
%(modulo) operator, 166
%SIG hash, 138, 669–673
% wildcard, 741–742
&(ampersands), 350
& operator, 163
=& operator, 152
() (parentheses), 92
* (asterisk), 262, 663
** operator, 151
*=` operator, 151
**(exponentiation) operator, 166
* (multiplication) operator, 166
+= operator, 151
+ (addition) operator, 166
-d switch, 718
-= operator, 151
- (subtraction) operator, 166
. (dot metacharacter, 251–252
.= operator, 151
/(forward slashes), 56, 597
/etc/passwd file, 638
/= operator, 151
/ (division) operator, 166
:: (double colons), 410
; (semicolons), 529, 726
< (less than) operator, 736, 739
<= operator, 152
=> (less than or equal) operator, 736
<> (not equal to) operator, 736
=> (space ship) operator, 121, 130, 158
= (equal sign), 86, 503
== operator, 158
=> operator, 222
= (equal) operator, 151, 736, 737
>= operator, 152
> (greater than) operator, 736, 739
>= (greater than or equal) operator, 736
? (question mark), 663
? placeholder, 571–578
@ARGV array, 333–338
@ array, passing arguments, 352–368
@INC array, 418–420, 797–802
@ISA array, 484–486
@ symbol, 52
[] (square brackets), 100, 663
\ (backslash), 52, 379, 597
^=` operator, 152
_ (underscore), 743
{ (curly braces), 100, 265
|=` operator, 152
|| operator, 163

831
A

abs function, 675
accept function, 675
accessing
databases, 521
directories, 608–612
elements
arrays, 95–97
slicing, 98–99
files, modifying, 620–621
hash values, 101–102
accounts, SAM (Security Accounts Manager), 639
ActivePerl, 8

adding
columns, 554
elements, arrays, 105
entries, 579
multiple records, 573
primary keys, 555
tables, primary keys, 543–544
addition (+) operator, 166

addresses
blessings, 455
memory, 380, 454

alarm function, 671, 672–673, 675

aliases
SQL (Structured Query Language), 758
typeglobs, references, 400–404

alphanumeric characters, 59
alternation of patterns, 273
alternative characters, 249
alternative quotes, 20, 55–59
ALTER TABLE command, 554, 748, 759
American National Standards Institute. See ANSI

ampersands (&), 350
anchored characters, 249, 269–271
AND operator, 736, 740
anonymous arrays, 382
anonymous hashes, 383
anonymous pipes, 326–333
anonymous subroutines, 393–394, 478. See also closures
anonymous variables, 382–383

ANSI (American National Standards Institute), 723

APIs (application programming interfaces), 530
appending files, 316
application programming interfaces. See APIs
applications (Dancer), 808–830
applying
CPAN Minus, 441–444
DBI (Database Independent Interface), 560–561
modules, 431–436, 798–799
multiple placeholders, 572
Perlbre, 441–444
PFM (Perl Program Manager), 439–441
quotes, 737
architecture, client/server, 521
ARCHIVE attribute, 600
arguments
command-line, passing at, 29
methods, passing, 466
passing, 333–341
subroutines, passing, 352–368
arithmetic functions, 167–171
arithmetic operators, 166–167
arrays, 17, 81–82, 91–99
@_, passing arguments, 352–368
@ARGV, 333–338
@INC, 418–420, 797–802
@ISA, 484–486
anonymous, 382
assigning, 92–93
copying, 98–99
elements
adding, 105
modifying, 120
referencing, 95–97
removing, 106–107
replacing, 106–107
files, slurping, 302

functions, 105–125
chomp function, 111–112
chop function, 111–112
delete function, 106–107
exists function, 124
grep function, 112–114
join function, 118–119
map function, 119–121
pop function, 109–110
push function, 105
reverse function, 125
shift function, 110–111
sort function, 121–124
splice function, 107–109
split function, 114–118
unshift function, 106

hashes, 104, 387
indexes, checking values, 124
input, assigning, 311–312
lists, 385, 386
looping, 97–98
multidimensional, 99
naming, 92
output field separators, 93–94
range operators, 95
reversing, 125
Index

rows, fetching, 564
sizing, 94–95
slicing, 98–99
sorting, 121–124
times function, 645
transforming, 119–121
variables, 92
arrow (±) operator, 382
ascending order, 130, 550
ASCII, 122, 159, 290
assigning
arrays, 92–93
hashes, 100–101
input
arrays, 311–312
hashes, 312–313
scalar variables, 307–308
numbers, 82
range operators, 95
scalar variables, 88
strings, 82
typeglobs, 412
values, 353–355
assignment operators, 151–153
assignment statements, 86–87
associativity, operators, 149–151
asterisk (*), 262, 663
atan2 function, 675
attributes, 448, 525
directories, 599–602
files, 599–602, 613
Moose, 776–7.95
PrintError, 567
RaiseError, 567
autodecrement operators, 172–173
autoincrement operators, 172–173
AUTOLOAD function, 369–370, 484
SAUTOLOAD function, 486–489
automatic error handling, 567
autovivification, 297
awk command, 114

B
backquotes, 52, 55, 659–660
backslash (\), 52, 379, 597
barewords, 44, 58
base classes, 484, 489. See also classes
BEGIN block, 371
BETWEEN operator, 736
BETWEEN statement, 743
binary operators, 147. See also operators
bind function, 676

binding
columns, 569
parameters, 571–578
runtime, 472–476
bind_param() method, 574
bin folders, 532
binmode function, 676
bits, 173–174
bitwise logical operators, 173–175
bitwise operators, 174–175
black boxes, 348
blank lines, formatting, 503
bles function, 453, 676
bless function, 454, 676
blessings, 454
blocks, 182–187
BEGIN, 371
END, 371
Boolean context, 38
Boolean types, 153
bracket expressions (POSIX), 257–258
break statements, 204
build() method, 459
built-in functions, 3, 596
arithmetic, 168
scripts, 39–40
bytecode, 2

C
C, 3
C++, 3
caches, queries, 577–578
call-by-references, 353
caller function, 676
calling
functions, 473
methods, 473, 484–486
processes, 629
subroutines, 349–352, 410
system calls, 595–629. See also system calls
capturing
patterns, 276–279
turning off, 281
Carp module, 665–666
Carp.pm module, 428–430
case sensitivity, 86
databases, 529
SQL (Structured Query Language), 727
catching signals, 669
categories (Perl), 11
categoryID key, 756
C dependencies, 805–806
CGIs (Common Gateway Interfaces), 522, 585, 807
here documents, 67
CGIs (Common Gateway Interfaces) (continued)

modules, 711
characters
  alphanumeric, 59
  classes, 253–256
  conversion, 69
delimiters, 220
  globbing, 663–664
  metacharacters, 220, 245–296. See also metacharacters
  sigils, 85
  special, 53
  whitespace, 249
char data type, 81
charts, flow, 162
chdir function, 607–608, 676
checkers, data, 469
checking syntax, 46
child processes, 629, 649
chmod command, 43
chmod function, 614–615, 676
chomp function, 43, 111–112, 308–309, 676
chop function, 111–112, 308–309, 677
chown function, 615, 677
chr function, 120, 677
Christianson, Tom, 449
chroot function, 677
classes, 450, 453–454, 459
  base, 489
  characters, 253–256
  creating, 30
  DBI (Database Independent Interface), 558–560
  defining, 448–449
  derived, 489–496
  methods, 457. See also methods
  parent, 489
  SUPER pseudo, 499–501
  UNIVERSAL, 484
clauses
  FROM, 546
  GROUP BY, 763
  JOIN, 551–552
  LIMIT, 550, 734
  ORDER BY, 550, 744
  WHERE, 548–550, 736
clients
  databases, 521–522
  MySQL, 532
closedir function, 610, 677
close function, 677
closing filehandles, 299
closures
  defining, 478–480
  objects, 481–484
clustering patterns, 273–275
cmp operator, 132, 159
Cobb, E. F. "Ted," 723
code, threaded, 2
coevolution, 148
columns, 524, 525
  adding, 554
  binding, 569
  dropping, 555
  selecting by, 546, 732
  combining arrays and hashes, 104
command-lines
  arguments, passing at, 29
  MS-DOS, 605. See also Windows
  mysql, 724
  switches, 44–47, 716–717
  testing, 45
  UNIX, 41
commands. See also functions
  ALTER TABLE, 554
  awk, 114
  chmod, 43
  cpan, 802–803
  CREATE DATABASE, 540–541
  CREATE TABLE statement, 541–543
  date, 57
  debugging, 720–722
  DELETE, 552–553
  DESCRIBE, 543, 730–731
  DROP DATABASE, 555
  drop database, 761
  EXTRACT, 769
  INSERT, 745–746
  INSERT statement, 544–546
  interpreters, 45
  LIKE, 530
  ls, 509
  net.exe, 639
  NOT LIKE, 530
  pod, 504–505
  pwd, 55
  QUIT, 529
  SELECT, 546–547, 731–745
  SHOW, 543, 730–731
  show, 537
  show database, 538
  show databases, 728
  SQL (Structured Query Language), 539–540, 725–728
  start, 654–655
  substitution, 53, 659–660
  system calls, 595
Index

touch, 620
UPDATE, 553–554, 746–747
USE, 529, 728
WHERE clause, 548–550
comments, 16
scripts, 38–39
commit() method, 583–585
Common Gateway Interfaces. See CGIs
comparing operands, 154
compiler directives, 84. See also pragmas
compiling programs, 412, 421
complex data structures, 104
components of relational databases, 522–527
compound statements, 182–187
conditional operators, 156–157
conditionals, 21
operators, 22
configuring passwords (MySQL), 533
correct function, 677
connecting
   databases, 521, 561–563. See also databases
   MySQL, 532–533
correct() method, 560, 561–562
consoles
   mysql, 724
   MySQL, editing keys, 533
constants, 18, 408. See also literals
constructors, 450, 457, 459
constructors, creating with objects, 458
constructs, 15–27
decision-making, 183–187
   if, 183–184
   if/else, 156, 184–185
   if/else/else, 185–186
quotes, 55
qw, 92
unless, 186–187
contents, viewing modules, 428–430
context
   hashes, 139–140
   operators, 145–147
   scripts, 38
   subroutines, 366–368
continue statements, 210–212
control
   loops, 25, 204–212
   structures, 182–187
controlling terminals, 630
conventions
case sensitivity, 529, 727
naming, 85–86
databases, 529
SQL (Structured Query Language), 727
UNC (universal naming convention), 597
conversion characters, 69
converting strings/numbers, 148
Coordinated Universal Time (UTC), 643
c (complement) option, 289
copying arrays, 98–99
CORE namespace, 215
cos function, 677
CPAN (Comprehensive Perl Archive Network), 6–7, 408
   @INC, 797–802
   DBDs (database driver modules), 558
   modules, 436–441
cpan command, 802–803
CPAN Minus, applying, 441–444
CPAN.pm module, 437
cpan shells, 438
CPU time, 643, 645. See also time
CREATE DATABASE command, 540–541
CREATE INDEX statement, 748
create() method, 459
CREATE TABLE statement, 541–543, 748, 751–753
cross joins, 756
crypt function, 677
   -c switches, 46
curly braces ({}), 100, 265
customizing sorting, 122
D
Dancer, 585–590, 807–808
   applications, 808–830
   exercises, 829–830
   parameters, 818–826
   POST requests, 826–828
   resources, 811
   templates, 814–818
data, packing/unpacking, 624–629
database driver modules. See DBDs
Database Independent Interface. See DBI
databases
   ? placeholder, 571–578
case sensitivity, 529
commands
   ALTER TABLE command, 554
   CREATE TABLE statement, 541–543
   DELETE command, 552–553
   DROP DATABASE command, 555
   INSERT statement, 544–546
   JOIN clause, 551–552
   SELECT command, 546–547
   UPDATE command, 553–554
   WHERE clause, 548–550
databases (continued)
  connecting, 561–563
  disconnecting, 561–563
  dropping, 538, 555
  error messages, 567–570
  formatting, 538, 748–749
  interfaces, modules, 713
  MySQL, 519–594. See also MySQL
  naming, 529
  schemas, 527
  searching, 537–538
  servers, 523
SQL (Structured Query Language). See also SQL
  navigating, 728–729
  tables, 729–731
  statements, 579–582
  syntax, 528–530
  tables, 523–524
    adding, 543–544
    sorting, 550–551
    transactions, 583–590
USE statements, 541
Databases Demystified, 520
data checkers, 469
Data Definition Language. See DDL
Data: Dumper module, 384
data encapsulation, 448, 450
DATA filehandles, 223–225
_DATA_ _literal, 63, 64
Data Manipulation Language. See DML
data structures, inodes, 599, 621
data types, 81–87
  arrays, 91–99
    assignment statements, 86–87
    complex data structures, 104
    hashes, 99–104
    naming conventions, 85–86
    packages, 82–85
    scalar variables, 87–91
  scope, 82–85
  SQL (Structured Query Language), 749–750
date and time functions, 766–770
date command, 57
DBDs (database driver modules), 556
  installing, 556–558
DBI (Database Independent Interface), 556–578
  applying, 560–561
  class methods, 558–560
dbmclose function, 678
dbopen function, 678
DDL (Data Definition Language), 748–761
debbugging, 718–722
commands, 720–722
  exiting, 719–720
  script errors, 43–44
  starting, 719–720
decision-making constructs, 183–187
declaring
  forward declarations, 351
  packages, 410
  subroutines, 349
default databases, 534. See also databases
defined function, 89, 349, 678
defining
  classes, 448–449
  closures, 478–480
  lexical variables, 83
  methods, 456
  objects, 447–448
  subroutines, 122, 349–352
DELETE command, 552–553
delete function, 17, 18, 106–107, 135–136, 678
DELETE statement, 560, 747–748
deleting
  directories, 607
  duplicates
    arrays, 121
    hashes, 103–104
  entries, 580
  newlines, 111–112
delimitters, 220
global change, 232
  substitution, modifying, 234
DELIMITER statement, 114, 118
deposit() method, 448
dereferencing pointers, 379
derived classes, 489–496
descendants, 629
descending order, 134, 550
describe command, 543, 730–731
destroy method, 476
destructors, 450, 476–478
diagnostics, 31
  errors, 567
diagnostics pragma, 76–77
die function, 299–300, 665, 678
digits, metacharacters, 248
digraph operators, 100
directives, compilers, 84. See also pragmas
directories, 597–612
  accessing, 608–612
  attributes, 599–602
  creating, 605–607
deleting, 607
modifying, 607–608
passwords, 638–639
searching, 603–605
UNIX, 609
DIRECTORY attribute, 600
disconnecting databases, 561–563
disconnect() method, 561, 563
DISTINCT keyword, 733
distributions (Perl), 6–9
division (/) operator, 166
DML (Data Manipulation Language), 731–748
documentation
   modules, 501–508, 596
   MySQL, 531, 539
   online, 12
   Perl, 9–12
   text, translating pod, 506–508
documents, here, 19, 66–68
do function, 678
do() method, 579
d (delete) option, 288
dot (.) metacharacter, 251–252
double colons (:), 410
double data type, 81
double quotes, 52, 53–54
do/until loops, 194–196
do/while loops, 24
do/while loops, 194–196
downloading Perl, 6–9
DROP DATABASE command, 555
drop database command, 761
DROP INDEX statement, 748
dropping
columns, 555
databases, 538, 555	tables, 555
DROP TABLE statement, 748, 761
dump function, 679
duplicates
   arrays, removing, 121
   hashes, removing, 103–104, 129
effective uids. See uids
elements
   arrays
      adding, 105
      modifying, 120
      referencing, 95–97
      removing, 106–107
      replacing, 106–107
      values, searching, 112–114
   e modifier, 238
encapsulating, data, 448, 450
END block, 371
   _ _END_ _ literal, 63, 64
entries
   adding, 579
   deleting, 580
   updating, 581
environments, processes, 632–633
eof function, 338–340, 679
eq operator, 159
equality operators, 157–160
equal sign (=), 86, 503
equal to (=) operator, 736, 737
error handling, 664–669, 711
error messages
   HTTP (Hypertext Transfer Protocol), 585
   SQL (Structured Query Language), 567–570
errors
   scripts, 43–44
   spelling, 85
   syntax, 2
escape sequences, 57
   string literals, 61–63
-e switches, 45
euids (effective uids), 631
eval function, 666–669, 679
evaluating expressions, 147, 150, 238
examples (Moose), 778–781
   extensions, 785–791
   inheritance, 791–795
exclusive or (xor) operator, 164
exec function, 652, 679
execute() method, 560
execute statement, 571
executing
   hashes, 566
   last statements, 357
   loops, 204
   rows, 564
   scripts, 40–42
SQL (Structured Query Language) statements, 724–725
E
   each function, 18, 128–129, 679
   editing, 85
      files, 340–341
      keys, 533
   editors
      text, selecting, 34–35
      third-party, 34
      types of, 35
   effective guids. See guids
exercises (Dancer), 829–830
exists function, 18, 124, 136–137, 679
exit function, 654, 679
exiting debugging, 719–720
exp function, 679
exponentiation (** operator), 166
Exporter module, 489
Exporter.pm module, 424–426, 435
exporting modules, 424–426
expressions, 147
  bracket (POSIX), 257–258
  evaluating, 147, 150, 238
  regular, 28, 112, 219–244. See also regular expressions
extensions
  languages, modules, 715
  .LNK, 617
  Moose examples, 785–791
  passwords, 641
  Win32::NetAdmin, 640
EXTRACT command, 769

F
fat comma operators, 100
fcntl function, 680
feature pragma, 74
features, state, 363
fetch_array() method, 564
fetching
  results, 563–566
  values, 569
fields, 524, 525
  map function, creating, 303
  output field separators, 93–94
File::Find module, 603
filehandles. See also files
  @ARGV arrays, 333–338
  closing, 299
  DATA, 223–225
  printing, 49–50
  processes, 634–636
  references, typeglobs, 402–404
  scripts, 37–42
  special variables, 705
  STDERR, 402
  STDIN, 307–333, 402
  STDOUT, 402
underscore, 622
  user-defined, 297–307
__FILE__, __ literal, 63, 64
filenames, globbing, 663–664
fileno function, 680
files, 3, 26–27, 297–346, 597–612
/etc/passwd, 638
  accessing, modifying, 620–621
  arguments, passing, 333–341
  attributes, 599–602, 613
  editing, 340–341
  handling, modules, 711–712
  hard/soft links, 616–620
  hashes, loading, 306–307
  House.pm, 465
  input from, reading, 90–91
  locking, 317–319
  opening, 297–298
  appending, 316
  reading, 324–325
  writing, 313–314
  packing/unpacking, 624–629
  passwords, 638–639
  pattern matching, 241
  permissions, 605, 606, 612–616
  .pm packages, 420–423
  pod, 502–504
  reading
    opening, 298
    scalar assignments, 300–305
    renaming, 620
    scripts, 16
    searching, 603–605
    slurping
      arrays, 302
      into strings with read() function, 304
    statistics, 621–623
    testing, 342–343
    Win32 binary, 315
File::spec module, 598
file systems, ReFS (Resilient File System), 597
filters, 326. See also pipes
  input, 330–333
  output, 327–329
find() function, 603
finish() method, 561
flags, modifiers, 70
float data type, 81
fork function, 317–319, 680
flow
  charts, 162
  loops, 204
  folders, bin, 532
foreach loops, 24, 97–98, 130, 198–202
foreach modifiers, 203–204
foreign keys, 755
fork function, 649–651, 680
forks, 649
for loops, 24, 196–198
Index

format function, 680
format specifiers, 69–70
formatting
databases, 538, 748–749
date and time, 767
directories, 605–607
fields, map function, 303
instance methods, 460–461
instructions, 503
keys, 753–755
lists from scalar variables, 114–118
MySQL passwords, 533
objects with constructors, 458
OOP (Object-Oriented Perl), 450–451, 464–472
printing
printf function, 69–74
say function, 73–74
sprintf function, 73
processes
UNIX, 649–654
Win32, 654–657
scripts, 33–37, 42–44
filehandles, 37–42
linebreaks, 35–36
numbers, 36–37
statements, 35–36, 39
strings, 36–37
switches, 44–47
whitespace, 35–36
SQL (Structured Query Language) statements,
  528, 725
tables, 751–753
formline function, 680
forward declarations, 351
forward slashes (/), 56, 597
frameworks, Dancer, 585–590. See also Dancer
free-form languages, 16
FROM clause, 546
full joins, 756
functions, 25–26, 347, 675–704. See also
  subroutines
$_ (topic variable), 90–91
abs, 675
accept, 675
alarm, 671, 672–673, 675
arithmetic, 167–171
arrays, 105–125
  chomp function, 111–112
  chop function, 111–112
delete function, 106–107
exists function, 124
grep function, 112–114
join function, 118–119
map function, 119–121
pop function, 109–110
push function, 105
reverse function, 125
shift function, 110–111
sort function, 121–124
splice function, 107–109
split function, 114–118
unshift function, 106
atan2, 675
AUTOLOAD, 369–370, 484
SAUTOLOAD, 486–489
bind, 676
binmode, 676
call, 455, 676
built-in, 3, 39–40, 596
caller, 676
calling, 473
cdir, 607–608, 676
cmod, 614–615, 676
crimp, 43, 308–309, 676
crop, 308–309, 677
crown, 615, 677
csh, 120, 677
croot, 677
close, 677
closedir, 610, 677
connect, 677
context, 38
cos, 677
crypt, 677
d, 679
dbmclose, 678
dbmapen, 678
defined, 89, 349, 678
delete, 17, 18, 678
die, 299–300, 665, 678
do, 678
dump, 679
each, 18, 679
eof, 338–340, 679
eval, 666–669, 679
exec, 652, 679
exists, 18, 679
exit, 654, 679
exp, 679
fcntl, 680
fileno, 680
File::spec module, 598
find(), 603
flock, 317–319, 680
fork, 649–651, 680
format, 680
functions (continued)
formline, 680
getc, 311, 680
getgrent, 681
getgrent(), 681
getgrent(), 681
getgrgid, 681
getgrgrgidname(), 681
gethostent(), 681
gethostent(), 681
gethostbyname(), 681
gethostent(), 681
getlogin(), 635, 681
getlogin(), 635
getnetbyaddr(), 681
getnetbyname(), 682
getnetent(), 682
getpeername(), 682
getpgrp(), 682
getppid(), 635–636, 682
getpriority(), 637, 682
getprotobyname(), 682
getprotobynumber(), 683
getprotoent, 683
getpwent, 641, 683
getpwent(), 641
getpwnam, 642, 683
getpwnam(), 642
getpwuid, 643, 683
getpwuid(), 643
getservbyname(), 683
getservbyname(), 684
getservbyport, 684
getservbyport(), 684
getservent, 684
getsockname, 684
getsockopt, 684
glob(), 663–664, 684
gmtime(), 646, 684
goto, 684
grep, 685
has, 777–778
hashes, 125–140
delete function, 135–136
each function, 128–129
exists function, 136–137
map function, 129
values function, 126–128
hex, 685
import, 685
index, 685
int, 685
ioctl, 685
join, 685
key, 685
keys, 18
kill, 670–671, 685
last, 686
lc, 686
lcfirst, 686
length, 686
link, 618, 686
listen, 686
local, 686
localtime, 648, 686
localtime(), 40, 43, 88
lock, 686
log, 687
lstat(), 600, 621–623, 687
m, 687
map, 303, 687
mkdir(), 605–607, 687
msgctl, 687
msgget(), 688
msgset(), 688
msgsnd, 688
msgsnd(), 688
my, 688
new, 688
next, 688
no, 688
not, 688
oct, 689
open, 297–298, 689
opendir(), 609, 689
ord, 689
our, 689
pack, 624–629, 690
package, 690
pgrp, 636
pipe, 690
pop, 17, 690
pos, 691
print, 43, 50, 51–52, 691
printf, 16, 50, 69–74, 691
prototype, 691
push, 17, 691
q, 691
qq, 691
quotemeta, 691
qw, 691
qx, 691
rand, 168, 692
read, 692
read(), 304, 310
readdir, 609, 692
readlink, 619
readline, 692
readlink, 692
readpipe, 692
recv, 692
redo, 692
ref, 396, 693
readdir, 607
rename, 620, 693
require, 421, 693
reset, 693
return, 349, 693
reverse, 693
rewinddir, 611, 693
rindex, 693
rmdir, 693
s, 694
say, 16, 73–74
scalar, 694
seek, 319–322, 694
seekdir, 611, 694
select, 317, 694
semctl, 694
semget, 694
semop, 695
send, 695
setpriority, 637–638, 695
setsockopt, 695
shift, 17, 695
shmctl, 695
shmget, 695
shmread, 696
shmwrite, 696
shutdown, 696
sin, 696
sleep, 672, 696
socket, 696
socketpair, 696
sort, 17, 132, 697
splice, 17, 697
split, 697
sprintf, 73, 697
SQL (Structured Query Language), 761–770
date and time, 766–770
numeric, 762–764
string, 765
sqrt, 697
srand, 168, 697
stat, 599, 621–623, 698
string operators, 175–178
study, 698
sub, 698
sub SAUTOLOAD, 486–489
subs, 371–372
substr, 699
symlink, 619, 699
syscall, 658–659, 699
sysopen, 699
sysread, 699
sysseek, 699
system, 661–662, 700
syswrite, 700
tell, 322–324, 700
telldir, 611, 700
tie, 701
tied, 701
time, 702
times, 645, 702
topic variable ($_), 300
tr, 222, 702
truncate, 702
uc, 702
ucfirst, 702
umask, 616, 702
unlink, 618, 703
unpack, 624–629, 703
unshift, 17, 703
untie, 703
use, 421, 703
utime, 620–621, 703
values, 18, 703
vec, 704
wait, 653, 704
waitpid, 653, 704
wantarray, 367–368, 704
wanted0), 603
warn, 666, 704
Win32::Spawn, 655–656
write, 704
y, 704
funny characters. See sigils

garbage collection, 476–478
generating random numbers, 168
ge operator, 155
getc function, 311, 680
getgrent function, 681
getgrgid function, 681
getgrnam function, 681
gethostbyaddr function, 681
gethostbyname function, 681
gethostent function, 681
getlogin function, 635, 681
getnetbyaddr function, 681
getnetbyname function, 681
getnetent function, 682
getnetbyname function, 682
getpeername function, 682
getprotobyname function, 682
getpriority function, 637, 682
getprotobynumber function, 683
getprotoent function, 683
getpwnam function, 642, 683
getpwuid function, 643, 683
GET requests, 811, 812
getservbyname function, 683
getservbyport function, 684
getservent function, 684
getsockname function, 684
getsocket function, 684
GET strings, 818
getters, 450
global change, 232
global match modifiers, 229
global special variables, 706–708
global variables, 349
globbing, 663–664
glob function, 663–664, 684
g modifier, 229, 236
GMT (Greenwich Mean Time), 643
gmtime function, 646, 684
Goldberg, Ian, 168
goto function, 684
goto statements, 205–109
grant tables, 536
graphical user tools (MySQL), 534–537
greater than (>) operator, 736, 739
greater than or equal (>=) operator, 736
greedy metacharacters, 261, 267–268, 280
Greenwich Mean Time. See GMT
grep function, 112–114, 685
GROUP BY clause, 763
groups
patterns, 273–275
processes, 630
gt operator, 155
guids (effective guids), 631

H
h2ph scripts, 658–659
handlers, verbs, 812
handles, 558
statements, 563–566
handling
errors, 664–669, 711
files, modules, 711–712
quotes, 576–577
hard references, 378–380
hard/soft links, files, 616–620
has function, 777–778
hashes, 18, 81–82, 99–104
%SIG, 669–673
anonymous, 383
arrays, 104, 387
assigning, 100–101
context, 139–140
duplicates, removing, 103–104, 129
files, loading, 306–307
functions, 125–140
delete function, 135–136
each function, 128–129
exists function, 136–137
map function, 129
values function, 126–128
hash of, 389
indexes, 100
input, assigning, 312–313
references, 603
rows, fetching, 566
slicing, 102–103
sorting, 130–135
special, 137–139
subroutines, passing, 355
values, accessing, 101–102
HEAD requests, 812
here documents, 19, 66–68
hex function, 685
HIDDEN attribute, 600
House.pm file, 465
HTTP (Hypertext Transfer Protocol) error messages, 385

I
IBM, SQL. See SQL
identifiers, 408
identifying versions, 9
IDEs (Integrated Development Environments), 34
if constructs, 183–184
if/else constructs, 156, 184–185
if/else/else constructs, 185–186
if/else/else statements, 22
if/else statements, 21
if modifiers, 188–189
if statements, 21
i modifier, 230, 237
import function, 685
importing
methods, creating, 435
modules, 424–426
Importing module, 426
indexes, 91, 526. See also lists
arrays, checking values, 124
hashes, 100
resource representation, 826
values, searching, 112–114
index function, 685
inheritance, 449, 450, 484–501
length function, 686
le operator, 155
less than (<) operator, 736, 739
less than or equal (<=) operator, 736
lexographical ordering, 155
lexical variables, defining, 83
lib pragma, 420
libraries, 31
   modules, applying, 431–436
   objects, applying, 508–512
   RegExLib.com, 245–247
   standard Perl 5.18 library, 417–436
LIKE command, 530
LIKE operator, 736, 741
LIMIT clause, 550, 734
limiting number of lines, 734
linebreaks
   scripts, 35–36
   _ _LINE_ _ literal, 63, 64
   lines, limiting number of, 734
link function, 618, 686
links
   hard/soft, files, 616–620
   symbolic, 617
Linux
   PPM (Perl Program Manager), 558
   system calls, 595
list context, 366
listen function, 686
lists, 91. See also arrays
   arrays, 385
   of lists, 99, 386
   scalar variables, creating, 114–118
   separators, 93
   unordered, 99–104. See also hashes
   values, returning, 126–128
literals, 18
   numeric, 60–61
   printing, 59–66
   special, 63–66
   strings, 61–63
LNK extensions, 617
loading files, hashes, 306–307
locales, modules, 714
local function, 686
localhost, 523
local operator, 358
local Perl, 801
localtime function, 648, 686
localtime() function, 40, 43, 88
local to block special variables, 705
lock function, 686
locking files, 317–319
log function, 687
logical operators, 162–164
logical word operators, 164–166
login information, 635
look around assertions, 282–285
looping
   arrays, 97–98
   modifiers, 202–216
loops, 23
   for, 24, 196–198
   control, 25, 204–212
   do/until, 194–196
   do-while, 24
   do/while, 194–196
   foreach, 24, 97–98, 130, 198–202
   nested and labels, 208–210
   repetition, 190–202
   until, 23, 192–194
   while, 23, 190–192, 223
ls command, 599
lstat function, 600, 621–623, 687
lt operator, 155
M
main package, 82
main packages, 348
management
   RDBMS (relational database management systems), 521
   SAM (Security Accounts Manager), 639
managers, package, 800–801
man pages, 10
manual error handling, 567
map function, 119–121, 129, 303, 687
masks, system, 616
matching
   modifiers, 226
   patterns, 219–244, 261–286
   % wildcard, 741–742
   m operator, 225–229
quotes, 53, 58
math modules, 713
memory addresses, 380, 454
messages, error, 43–44
HTTP (Hypertext Transfer Protocol), 585
SQL (Structured Query Language), 567–570
metacharacters, 220, 245–296
   alternative characters, 249
   anchored characters, 249, 269–271
   digits, 248
do ( ), 251–252
look around assertions, 282–285
miscellaneous characters, 250
m modifier, 271–272
RegExLib.com library, 245–247
remembered characters, 250, 276–279
repeated characters, 249, 261–286
single characters, 248, 251–258
s modifier, 252
substitution, 285–290
Unicode, 290–294
whitespace characters, 249, 258–261
metasymbols, 248, 253
methods, 347, 448, 450. See also subroutines
arguments, passing, 466
bind_param(), 574
build(), 459
calling, 473, 484–486
commit(), 583–585
connect(), 560, 561–562
constructors, 459
create(), 459
DBI (Database Independent Interface), 558–560
defining, 456
deposit(), 448
DESTROY, 476
disconnect(), 561, 563
do(), 579
execute(), 560
fetch_array(), 564
finish(), 561
importing, creating, 435
init(), 459
instance, 457
formatting, 460–461
invoking, 458
passing parameters, 467–469
invoking, 457, 462–464
new(), 456
overriding, 499–501
prepare(), 560
rollback(), 583–585
set_color(), 460
set_owner(), 456, 460
set_price(), 460
shoot(), 473
speak, 457
startup(), 459
subroutines, 456–464, 459
types of, 457
view(), 448
withdraw(), 448
m function, 687
miscellaneous characters, 250
mixing types, 148–149
mkdir function, 605–607, 687
m modifier, 271–272
models, client/server, 521
modes, 606
modifiers
e, 238
flags, 70
foreach, 203–204
g, 229, 236
j, 230, 237
if, 188–189
looping, 202–216
m, 271–272
matching, 226
regular expressions, 221–225
s, 252
statements, 188–190
substitution, 235
tr, 287
unless, 189–190
while, 202–203
x, 231
modifying
directories, 607–608
elements, arrays, 120
expressions, 221
files, accessing, 620–621
global change, 232
substitution delimiters, 234
modules, 31, 407–446, 710–715
applying, 431–436, 798–799
Carp, 665–666
Carp.pm, 428–430
C dependencies, 805–806
CGIs (Common Gateway Interfaces), 711
contents, viewing, 428–430
CPAN (Comprehensive Perl Archive Network), 436–441
cpan command, 802–803
CPAN Minus, applying, 441–444
CPAN.pm, 437
Dancer. See Dancer
database interfaces, 713
Data::Dumper, 384
documentation, 501–508, 596
error handling, 711
Exporter, 489
Exporter.pm, 424–426, 435
exporting/importing, 424–426
File::Find, 603
file handling, 711–712
File::spec, 598
Importing, 426
installing manually, 801–802
modules (continued)
language extensions, 715
locales, 714
math, 713
networks, 713–714
OOP (Object-Oriented Perl), 464–472, 714
overview of, 407–417
package managers, 800–801
Perlbrew, 441–444
programming, 710
retrieving, 438
searching, 798
Shell.pm, 660–661
SomeModule.pm, 426
standard Perl 5.18 library, 417–436
terminals, 714
text processing, 712
time, 714
Time::Piece, 644
Win32::File, 600–602, 613
Win32::NetAdmin, 640
Win32::Process, 656–657
Windows, 806
modulo (%) operator, 166
Moose, 775–796
attributes, 776–795
examples, 778–781
extensions, 785–791
inheritance, 791–795
has function, 777–778
Moo (2/3 Moose), 795
types, 781–785
m operator, 225–229
MS-DOS command line, 605. See also Windows
msgctl function, 687
msgget function, 688
msgsnd function, 688
msgrcv function, 688
multidimensional arrays, 99
multiple inheritance, 489, 496–499
multiple placeholders, 572
multiple records, adding, 573
multiplication (*) operator, 166
my function, 688
my operator, 84, 358–361
MySQL, 519–594
? placeholder, 571–578
commands, 539–540
ALTER TABLE command, 554
CREATE DATABASE command, 540–541
CREATE TABLE statement, 541–543
DELETE command, 552–553
DROP DATABASE command, 555
INDEX

N
named parameters, 469
namespaces, 82
CORE, 215
packages, 412
variables, 82
naming
arrays, 92
case sensitivity, 86
databases, 529
modules, 408
scripts, 35
UNC (universal naming convention), 597
naming conventions, 85–86, 727
navigating
databases, 728–729
directories/files, 597–612
error handling, 664–669
MySQL, 530–555
operating systems, 658–664
Perl, 595–674
processes, 629–657
system calls, 595–629
negative look behinds, 282
ne operator, 159
nested data structures, 383–393
nested loops and labels, 208–210
net.exe command, 639
network modules, 713–714
new function, 688
newlines
  deleting, 111–112
  $ modifier, 252
new() method, 456
next function, 688
NICEVALUE value, 638
no function, 688
northwind databases, 524. See also databases, relational databases
not equal to (<> operator, 736, 737
not function, 688
NOT LIKE command, 530
NOT LIKE operator, 736, 741
NOT NULL, defining as, 543
NOT operator, 736
NULL, 737–739
numbers, 19
  assigning, 82
  inodes, 600
  random, generating, 168
  scripts, 36–37
  strings, converting, 148
numeric equality operators, 157–158
numeric functions, 762–764
numeric literals, 60–61
numeric values, relational operators and, 154

Object-Oriented Perl. See OOP
objects, 30, 450
  closures, 481–484
  constructors, creating with, 458
  defining, 447–448
  libraries, applying, 508–512
  references, 454, 460
oct function, 689
online documentation, 12
OOP (Object-Oriented Perl), 447–518
  classes, defining, 448–449
  closures
    defining, 478–480
    objects, 481–484
destructors, 476–478
formats, 450–451
garbage collection, 476–478
inheritance, 484–501
methods, subroutines, 456–464
modules, 714
  creating, 464–472
documentation, 501–508
objects
  applying from Perl libraries, 508–512
  defining, 447–448
  polymorphism, 472–476
  programs, 451–454
  runtime binding, 472–476
terminology, 449–450
.opendir function, 609, 689
open function, 297–298, 689
opening
  anonymous pipes, 326–333
  files, 297–298
  appending, 316
  reading, 298, 324–325
  writing, 313–314
operands, 147
  comparing, 154
  smartmatch operators, 160–162
operators, 20, 145–180. See also specific operators
  AND, 736, 740
  BETWEEN, 736
  arithmetic, 166–167
  arrow (±), 382
  assignment, 86, 151–153
  associativity, 149–151
  autodecrement, 172–173
  autoincrement, 172–173
  backlash (\), 379
  bitwise, 174–175
  bitwise logical, 173–175
  Boolean types, 153
cmp, 132
  conditional, 156–157
  conditionals, 22
  context, 38, 145–147
digraph, 100
equality, 157–160
equal to (=), 736, 737
expressions, evaluating, 147, 150
fat comma, 100
operators (continued)

file testing, 342–343
greater than (>), 736, 739
greater than or equal (>=), 736
IS [NOT] NULL, 736
less than (<), 736, 739
less than or equal (<=), 736
LIKE, 736, 741
local, 358
logical, 162–164
logical word, 164–166
my, 84, 358–361
NOT, 736
not equal to (<>), 736, 737
NOT LIKE, 736, 741
OR, 736, 740
pattern binding, 222–223
precedence, 149–151
range, 95, 175
regular expressions, 225–242
    g modifier, 229
    i modifier, 230
    m operator, 225–229
    pattern binding with substitution, 232–242
    s operator, 232
    x modifier, 231
relational, 154–155
s, 232
scope, 357–361
smartmatch, 160–162
SQL (Structured Query Language), 736
state, 358–361
strings, 175–178
tr, 285–290
types, mixing, 148–149
XOR, 736
y, 285–290

Oppel, Andy, 520
options
    c (complement), 289
    command-line, 44–47
    d (delete), 288
    s (squeeze), 290
Oracle, 723
ORDER BY clause, 550, 744
ordered lists, 92. See also lists
ord function, 689
OR operator, 736, 740
our function, 689
output
    filters, 327–329
    of filters to files, sending to, 329–330
output field separators, 93–94

overriding methods, 499–501
ownership of files, 612–616

P

package function, 690
__PACKAGE__ literal, 64
packages, 82–85, 408–411, 453. See also classes
declaring, 410
main, 348
managers, 800–801
namespaces, 412
.pm files, 420–423
references, 409–411
variables, 349, 416
pack function, 624–629, 690
packing data, 624–629
pages, man, 10
parameters
    binding, 571–578
    Dancer, 818–826
    instance methods, passing, 467–469
    named, 469
parent classes, 484, 489. See also classes
parentheses (()), 92
parent methods, overriding, 499–501
parent process ids. See ppids
Parrot, 4–6
passing
    arguments, 333–341
    command-line, 29
    methods, 466
    subroutines, 352–368
    parameters, instance methods, 467–469
    references, 394
passwords
    extensions, 641
    files, 638–639
    getpwent function, 641
MySQL, 533
PATHEXT environment variables, 41
pathnames, 417
pattern binding operators, 222–223
patterns
    alternation, 273
    capturing, 276–279
    clustering, 273–275
    groups, 273–275
    matching, 219–244, 261–286
    % wildcard, 741–742
    m operator, 225–229
    saving, 230–231
Perl
    categories, 11
documentation, 9–12
functions, 675–704
local, 801
modules, 710–715
navigating, 595–674
overview of, 1–2
pragmas, 708–710
Quick Start, 15–32
Strawberry, 806
users of, 3
versions, 4
PERL5LIB environment variable, 419–420
Perl 6, 4–6
Perlbrew, 441–444, 803–805
permissions, files, 603, 606, 612–616
pgids (process group ids), 636
pgrp function, 636
phpMyAdmin tool, 535–536
pids (positive integers), 629
pipe function, 690
pipes, 27, 326–333
placeholders
?, 571–578
multiple, 572
multiple records, adding, 573
.pm files, packages, 420–423
pod (Plain Old Documentation), 501–508
pointers, 29–30, 377, 379. See also references
polymorphism, 450, 472–476
pop function, 17, 109–110, 690
Portable Operating System Interface. See POSIX
pos function, 691
positive integers. See pids
positive look behinds, 282
POSIX (Portable Operating System Interface), 257–258
POST requests, 812, 826–828
ppids (parent process ids), 635–636
PPM (Perl Program Manager), 408
applying, 439–441
DBDs (database driver modules), installing, 556–558
Linux, 558
pragmas, 74–78, 417, 422, 708–710. See also modules
diagnostics, 76–77
feature, 74
lib, 420
strict, 77–78, 84, 361–364, 400
use locale, 122
warning, 75–76
warnings, 85
precedence, operators, 149–151, 164
predefined variables, 18
prepare() method, 560
primary keys, 526, 753–754
adding, 555
tables, 543–544
PrintError attribute, 567
printf function, 16, 50, 691
formatting, 69–74
print function, 43, 50, 51–52, 691
printing, 16, 49–79
filehandles, 49–50
here documents, 66–68
literals, 59–66
numeric, 60–61
special, 63–66
strings, 61–63
pragmas, 74–78
diagnostics, 76–77
feature, 74
strict, 77–78
warning, 75–76
printf function, 69–74
print function, 51–52
quotes, 52–59
say function, 73–74
sprintf function, 73
words, 51
print statements, 44
priorities of processes, 637–638
privacy, 82–85
private objects, 448. See also objects
privileges (MySQL), 536
procedures, 347. See also subroutines
processes, 3, 629–657
calling, 629
child, 629, 649
environments, 632–633
filehandles, 634–636
groups, 630
priorities, 637–638
servers, 523
signals, sending, 670
text modules, 712
time, 643–649
UNIX, 629–631, 649–654
Win32, 631–632, 654–657
process group ids. See pgids
programming modules, 710
programs
compiling, 412, 421
methods, calling, 473
Moose. See Moose
programs (continued)
  OOP (Object-Oriented Perl), 451–454
set user ID, 631
properties, 448
prototype function, 691
prototypes, 365–366
pseudo classes, SUPER, 499–501
pseudo-random numbers, 168
public objects, 448. See also objects
push function, 17, 105, 691
PUT requests, 812
pwd command, 55
Python, 2

Q
q function, 691
qq function, 691
quantifiers, 261
queries, 521, 723. See also databases; MySQL; SQL
caches, 577–578
MySQL Query Browser, 534
SQL (Structured Query Language), 725–728
Query Browser (MySQL), 725
question mark (?), 663
Quick Start (Perl), 15–32
QUIT command, 529
quotemeta function, 691
quotes, 19
  alternative, 20, 55–59
  applying, 737
  backquotes, 55
  constructs, 55
  double, 53–54
  handling, 576–577
  here documents, 66–68
  matching, 53, 58
  printing, 52–59
  rules, 57
  single, 54
qw construct, 92
qw function, 691
qx function, 691

R
RaiseError attribute, 567
Rakudo Perl, 4–6
rand function, 168, 692
random numbers, generating, 168
range operators, 95, 175
RDBMS (relational database management systems), 521, 522, 530. See also MySQL
readdir function, 609, 692
read function, 692
read() function, 304, 310
reading
  files
    opening, 298, 324–325
    scalar assignments, 300–305
    input from files, 90–91
    STDIN filehandle, 307–333
readlink function, 619
readline function, 692
readlink function, 692
READONLY attribute, 600
readdir function, 692
records, 524, 526
  multiple, adding, 573
recv function, 692
redo function, 692
redo statements, 204, 205–209
references, 29–30, 377–405
  anonymous variables, 382–383
  call-by-references, 353
  elements, arrays, 95–97
  filehandles, typeglobs, 402–404
  hard, 378–380
  hashes, 603
  memory addresses, 454
  nested data structures, 383–393
  objects, 454, 460
  overview of, 377–378
  strict pragma, 400
  subroutines, 393–396
  symbolic, 398–400
  typeglobs, 400–404
  variables, packages, 409–411
  refers, 454, 460
ref function, 396, 693
ReFS (Resilient File System), 597
RegExLib.com library, 245–247
regular expressions, 28, 112, 219–244
  metacharacters. See metacharacters
  modifiers, 221–225
  need for, 220–221
  operators, 225–242
    g modifier, 229
    i modifier, 230
    m operator, 225–229
    pattern binding with substitution, 232–242
    s operator, 232
    x modifier, 231
  overview of, 219–220
relational database management systems. See RDBMS
relational databases, 520–530, 723
Index

client/server databases, 521–522
components, 522–527
relational operators, 154–155
relations, 756
remdir function, 607
remembered characters, 250, 276–279
removing
directories, 607
duplicates
arrays, 121
hashes, 103–104, 129
elements, 106–107
newlines, 111–112
rename function, 620, 693
removing files, 620
repeated characters, 249
metacharacters, 261–286
repeating patterns, matching, 261–286
repetition, loops, 190–202
replacing elements, arrays, 106–107
representation, index resources, 826
requests, 723. See also queries
GET, 811
HEAD, 812
POST, 812, 826–828
PUT, 812
require function, 421, 693
reserved words, 453, 529, 727
reset function, 693
Resilient File System. See ReFS
resources, 825
Dancer, 811
index representation, 826
for Perl, 7–8
results
fetching, 563–566
sorting, ORDER BY clauses, 744
result sets, 525, 530
number of lines, limiting, 734
SQL (Structured Query Language), 728
retrieving modules, 438
return function, 349, 693
returning
lists, 126–128
values, 356–357
return values, 647. See also values
reverse function, 125, 693
reversing
arrays, 125
hashes, sorting, 131
rewinddir function, 611, 693
rindex function, 693
rmdir function, 693
roles, multiple inheritance, 496–499
rollback() method, 583–585
roots, 631
routines, 347. See also subroutines
rows, 526
rules, quotes, 57
runtime
binding, 472–476
modules as, 421
S
SAM (Security Accounts Manager), 639
saving
formatting, sprintf function, 73
patterns, 230–231
say function, 16, 73–74
scalar context, 139–140, 366
scalar function, 694
scalar variables, 17, 29, 81–82, 87–91, 92
input, assigning, 307–308
lists, creating, 114–118
scripts, 37–38
schemas, 527, 534
Schwartz, Randal, 2
scope, 82–85
operators, 357–361
of variables, 351–352
scripts, 2, 33–48
built-in functions, 39–40
comments, 38–39
context, 38
creating, 33–37
errors, 43–44
executing, 40–42
filehandles, 37–42
files, 16
formatting, 42–44
h2ph, 658–659
linebreaks, 35–36
Moose, 776
naming, 35
numbers, 36–37
scalar variables, 37–38
statements, 33–36, 39
strings, 36–37
switches, 44–47
system calls, 595
text editors, selecting, 34–35
whitespace, 35–36
searching
databases, 537–538
directories, 603–605
files, 603–605
searching (continued)
  modules, 798
Security Accounts Manager. See SAM
seekdir function, 611, 694
seek function, 319–322, 694
SELECT command, 546–547, 731–745
SELECT DISTINCT statement, 733
select function, 317, 694
selecting
columns, 732
by columns, 546
databases, USE statements, 541
hashes, 566
rows, 564
text editors, 34–35
semctl function, 694
semget function, 694
semicslons (;), 529, 726
semop function, 695
send function, 695
sending
output of filters to files, 329–330
signals, processes, 670
values to subroutines, 352
separate resources, 825
separators
lists, 93
output field, 93–94
SEQUEL (Structured English Query Language), 723
sequences
escape, 57
operators, associativity, 149–151
string literals, 61–63
subroutines. See subroutines
server databases, 521–522, 523
set_color() method, 460
set_owner() method, 456, 460
set_price() method, 460
setpriority function, 637–638, 695
sets, result, 525, 728
setsockopt function, 695
setters, 450
set user ID programs, 631
s function, 694
shebang lines, 41
Shell.pm module, 660–661
shells
  cpan, 438
  CPAN (Comprehensive Perl Archive Network), 442
  metacharacters, globbing, 663–664
shift function, 17, 110–111, 695
shmctl function, 695
shmget function, 695
shmread function, 696
shmwrite function, 696
shoot() method, 473
short-circuit operators, 162–164
shortcuts, 617
SHOW command, 543, 730–731
show command, 537
show database command, 538
show databases command, 728
shutdown function, 696
sigils, 85
signals
%SIG hash, 669–673
catching, 669
processes, sending, 670
simple statements, 188–190, 221–225
sin function, 696
single characters, metacharacters, 248, 251–258
single inheritance, 489
single quotes, 52, 54
single statements, 182
sizing arrays, 94–95
sleep function, 672, 696
slicing
arrays, 98–99
hashes, 102–103
slurping files
into arrays, 302
into strings with read() function, 304
smartmatch operators, 160–162
s modifier, 252
socket function, 696
socketpair function, 696
soft links, files, 616–620
SomeModule.pm module, 426
s operator, 232
s (squeeze) option, 290
sort function, 17, 121–124, 132, 697
sorting
arrays, 121–124
hashes, 130–135
results, ORDER BY clauses, 744
tables, 550–551
space ship (<=>) operators, 121, 130, 158
speak method, 457
special characters, 53
special hashes, 137–139. See also hashes
%ENV hash, 137–138
%INC hash, 139
%SIG hash, 138
special literals, 63–66
special process variables, 635
special variables
   $&, 240
   filehandles, 705
   global, 706–708
   local to block, 705
   specifiers, format, 69–70
   spelling errors, 85
   splice function, 17, 107–109, 697
   split function, 114–118, 697
   sprintf function, 73, 697
   SQL (Structured Query Language), 520. See also
   MySQL
   CREATE DATABASE command, 540–541
   DDL. See DDL
   DML. See DML
   error messages, 567–570
   functions, 761–770
   operators, 175–178
   relational operators, 155
   scripts, 36–37
   state operator, 358–361
   square brackets ([]), 100, 663
   std function, 599, 621–623, 698
   standards, 724
   state feature, 363
   statements
   executing, 724–725
   formatting, 725
   sqrt function, 697
   square brackets ([]), 100, 663
   srand function, 168, 697
   standard Perl 5.18 library, 417–436
   standards
   ANSI (American National Standards Institute), 723
   SQL (Structured Query Language), 724
   Unicode, 290–294
   start command, 654–655
   starting debugging, 719–720
   startup() method, 459
   strictness, 82–85
   strict pragmas, 84, 400
   strings, 19
   assigning, 82
   binding, 222–223
   equality operators, 159
   files, slurping, 304
   functions, 765
   GET, 818
   literals, 61–63
   numbers, converting, 148
   operators, 175–178
   execution, 724–725
   formatting, 725
   switch, 212–216
   UPDATE, 560
   USE, 541
   STATE operator, 358–361
   stat function, 599, 621–623, 698
   statistics, files, 621–623
   stat structure, 342
   STDERR filehandle, 49–50, 402
   STDIN filehandle, 49–50, 307–333, 402
   STDOUT filehandle, 49–50, 402
   Strawberry Perl, 6, 806
   streams, 49–50
   Structured English Query Language. See SEQUEL
Structured Query Language. See SQL structures
control, 182–187
inodes, 599, 621
nested data, 383–393
stat, 342
study function, 698
sub $AUTOLOAD function, 486–489
sub function, 698
subprograms, 347, 409–411. See also subroutines
subroutines, 25–26, 347–375, 408
anonymous, 478. See also closures
arguments, passing, 352–368
calling, 349–352, 410
context, 366–368
declaring, 349
defining, 122, 349–352
inline, 124
methods, 456–464, 459
overview of, 348–352
references, 393–396, 394
subs function, 371–372
substitution, 232
commands, 53, 659–660
delimiters, modifying, 234
metacharacters, 285–290
modifiers, 235
pattern binding with, 232–242
substr function, 699
subtraction (-) operator, 166
superclasses, 489
SUPER pseudo classes, 499–501
superusers, 536, 631
switches, 44–47
-d, 718
-c, 46
command line, 716–717
-e, 45
-i, 340–341
-w, 46–47
switch feature, 214–216
switch statements, 212–216
symbolic links, 617
symbolic references, 378, 398–400
symbols, 408
exporting, 425
metasymbols, 248, 253
tables, 412–417
symlink function, 619, 699
syntax, 15–27
errors, 2
MySQL, 528–530
shebang lines, 41
syscall function, 658–659, 699
sysopen function, 699
sysread function, 699
syssek function, 699
SYSTEM attribute, 600
system calls, 595–629
system function, 661–662, 700
system masks, 616
syswrite function, 700
T

tables, 99, 523–524
databases, 520. See also databases; MySQL
dropping, 555
formatting, 751–753
grant, 536
JOIN clause, 551–552
joins, 757
primary keys, adding, 543–544
sorting, 550–551
SQL (Structured Query Language)
databases, 729–731
symbols, 412–417
telldir function, 611, 700
tell function, 322–324, 700
templates
Dancer, 814–818
pack/unpack functions, 624–629
terminals
controlling, 630
modules, 714
terminating SQL statements, 529, 726
terminology
MySQL, 531
OOP (Object-Oriented Perl), 449–450
ternary conditional operators, 156–157
ternary operators, 147. See also operators
testing
command-lines, 45
files, 342–343
text
comments. See comments
editors, selecting, 34–35
processes, modules, 712
third-party editors, 34
threaded code, 2
tied function, 701
tie function, 701
time
data and time functions, 766–770
files, modifying, 620–621
modules, 714
processes, 643–649
### Index

**time function**, 702  
*Time::Piece* module, 644  
**times function**, 645, 702  
**tools**  
  *dancer*, 808  
  *MySQL*, 534–537  
  *phpMyAdmin*, 535–536  
**topic variable** ($_*) function, 300  
**topic variables**, 90–91  
**touch** command, 620  
**transactions**, 583–590  
**transforming arrays**, 119–121  
**translating pod documentation into text**, 506–508  
**tr function**, 222  
**tr operator**, 285–290  
**troubleshooting script errors**, 43–44. See also **error handling**  
**truncate** function, 702  
**turning off**  
  *capturing*, 281  
  *greedy metacharacters*, 267–268, 280  
**typeglobs**  
  *assigning*, 412  
  *references*, 400–404  
**types**  
  *Boolean*, 153  
  *of context*, 38  
  *data*, 81–87. See also **data types**  
  *of editors*, 35  
  *of methods*, 457  
  *mixing*, 148–149  
  *Moose*, 781–785  
  *of references*, 378  
  *of time values*, 643  
**typos**, 85

### U

**ucfirst** function, 702  
**uc** function, 702  
**umask** function, 616, 702  
**unary operators**, 147. See also **operators**  
**UNC** (universal naming convention), 597  
**undef** function, 89–90, 702  
**underscore** filehandle, 622  
**Unicode**, metacharacters, 290–294  
**Uniform Resource Locators**. See **URLs**  
**UNIVERSAL** class, 484  
**UNIVERSAL** function, 486–489  
**universal naming convention**. See **UNC**  
**UNIX**, 2  
  *command-lines*, 41  
  *commands*. See also commands  
    *ls*, 599  
  *touch*, 620  
  *directories*, 609  
    *attributes*, 599–600  
    *creating*, 605  
  *files*  
    *attributes*, 599–600  
    *hard/soft links*, 616–617  
    *ownership/permissions*, 612  
    *passwords*, 638–639, 641  
    *rehashing*, 620  
  **functions**  
    *chmod*, 614  
    *chown*, 615  
    *link*, 618  
    *readlink*, 619  
    *symlink*, 619  
    *umask*, 616  
    *unlink*, 618  
  **processes**, 629–631  
    *creating*, 649–654  
    *environments*, 632–633  
    *filehandles*, 634–636  
    *system calls*, 395  
  *times function*, 646  
  **unless constructs**, 186–187  
  **unless modifiers**, 189–190  
  *unlink function*, 618, 703  
  **unordered lists**, 99–104. See also **hashes**  
  *unpack function*, 624–629, 703  
  *unpacking data*, 624–629  
  *unshift function*, 17, 106, 703  
  *untie function*, 703  
  **until** loops, 23, 192–194  
  **UPDATE** command, 553–554, 746–747  
  **UPDATE statement**, 560  
  **updating entries**, 581  
  **URLs** (Uniform Resource Locators), 811  
  **USE** command, 529, 728  
  **use** function, 421, 703  
  **use locale** pragma, 122  
  **user-defined filehandles**, 297–307  
  **user interaction**, invoking methods, 462–464  
  **USE** statement, 541  
  **UTC** (Coordinated Universal Time), 643  
  **utime** function, 620–621, 703

### V

**values**  
  *ASCII*, 159  
  *assigning*, 353–355  
  *elements*, searching, 112–114  
  *fetching*, 569  
  *hashes*, accessing, 101–102
values (continued)
  indexes
    checking arrays, 124
    searching, 112–114
  indexes, searching, 112–114
  lists, returning, 126–128
  logical operators, 162
  numeric, relational operators and, 154
  return, 356–357, 647
  subroutines, sending, 352
  time, 643. See also time
values function, 18, 126–128, 703
variables, 17, 408
  $$, 635–636
  $\&$, 240
  anonymous references, 382–383
  arrays, 92
  environments, 632–633
  error diagnostics, 567
  global, 349
  hashes. See hashes
    instance, 466
    namespaces, 82
  packages, 349, 409–411, 416
  PATH environment, 41
  PERL5LIB environment, 419–420
  predefined, 18
  scalar, 17, 29, 81–82, 87–91, 92
    creating lists, 114–118
    scripts, 37–38
  scope of, 351–352
  special
    filehandles, 705
    global, 706–708
    local to block, 705
  special process, 635
  topic, 90–91
vec function, 704
verbs (Dancer), 811
versions
  identifying, 9
  MySQL, 530
  Perl, 4
viewing module contents, 428–430
view() method, 448
visibility, 409. See also scope
void context, 38

W
Wagner, David, 168
wait function, 653, 704
waitpid function, 653, 704
Wall, Larry, 1, 2, 3, 478
WAMP, 531
wantarray function, 367–368, 704
wanted() function, 603
warn function, 666, 704
warning pragma, 75–76
warnings, 46–47
warnings pragma, 85
Web servers, 522. See also servers
WHERE clause, 548–550, 736
WHICH value, 638
while loops, 23, 190–192, 223
while modifiers, 202–203
whitespace
  characters, 249
  metacharacters, 258–261
  scripts, 35–36
WHO value, 638
wildcards
  %, 741–742
  _ (underscore), 743
Win32
  binary files, 315
  password extensions, 641
  processes, 631–632, 654–657
Win32::File module, 600–602, 613
Win32::NetAdmin module, 640
Win32::Process module, 656–657
Win32::Spawn function, 655–656
Windows
  alarm function, 672–673
  directories
    attributes, 600–602
    creating, 605–607
  files
    attributes, 600–602, 613
    hard/soft links, 617–620
    ownership/permissions, 612–616
    passwords, 638–639
    renaming, 620
  functions, chmod, 614–615
  modules, 806
  processes, environments, 632–633
  times function, 646
withdraw() method, 448
words. See also text
  logical word operators, 164–166
  printing, 51
  reserved, 529
  strict pragma, 77–78
write function, 704
writing, 313–314. See also reading
  -w switches, 46–47
Index

X
XAMPP, 531
x modifier, 231
x= operator, 152
XOR operator, 736
xor (exclusive or) operator, 164

Y
y function, 704
y operator, 285–290

Z
zeroes, 631