"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. I still think it is the best Perl book on the market for anyone—beginner or seasoned programmer—who uses Perl daily."

—Bill Maples, Enterprise Network Support, Fidelity National Information Services

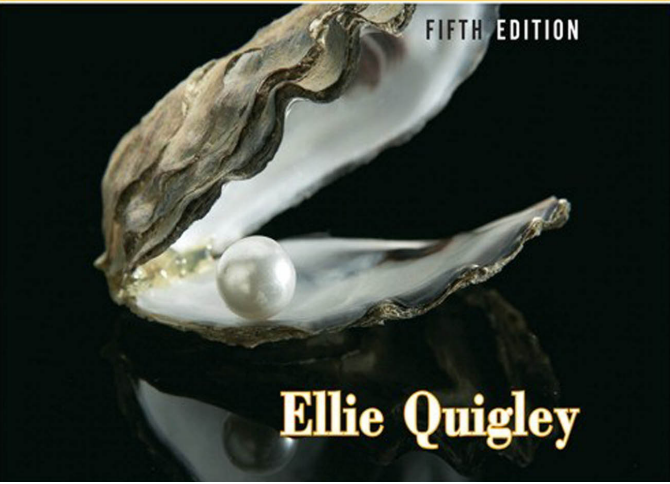
PRENTICE HALL

PERL

by

EXAMPLE

FIFTH EDITION



Ellie Quigley

Praise for Ellie Quigley's Books

"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

Perl by Example

Fifth Edition

Ellie Quigley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Quigley, Ellie.
Perl by example / Ellie Quigley.—Fifth edition.
pages cm
Includes index.
ISBN 978-0-13-376081-1 (pbk. : alk. paper)
1. Perl (Computer program language) I. Title.
QA76.73.P22Q53 2015
005.13'3—dc23
2014036613

ISBN-13: 978-0-13-376081-1
ISBN-10: 0-13-376081-2

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, December 2014

Editor-in-Chief
Mark L. Taub

Development Editors
Michael Thurston
Chris Zahn

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Project Manager
Moore Media, Inc.

Copy Editor
Moore Media, Inc.

Indexer
Larry Sweazy

Proofreader
Pam Palmer

Cover Designer
Chuti Prasertsith

Composition
Moore Media, Inc.

Contents

Preface

"You may wonder, why a new edition of *Perl by Example*?" That's how the preface for the fourth edition (2007) opened. So here we are again with a fifth edition and the twentieth anniversary since the first edition of *Perl by Example*, published in 1994. Same question: Why another edition? Perl 5 is still Perl 5.

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.

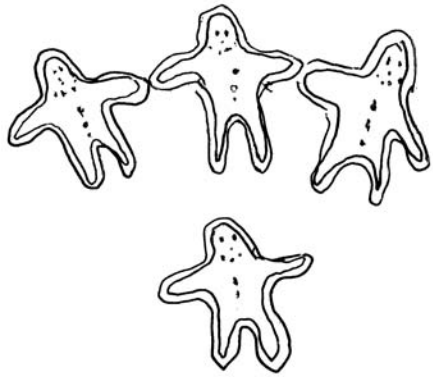
This page intentionally left blank

This page intentionally left blank

chapter

5

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:

```
use strict;
use warnings;
my @l = qw/a b c d 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.")

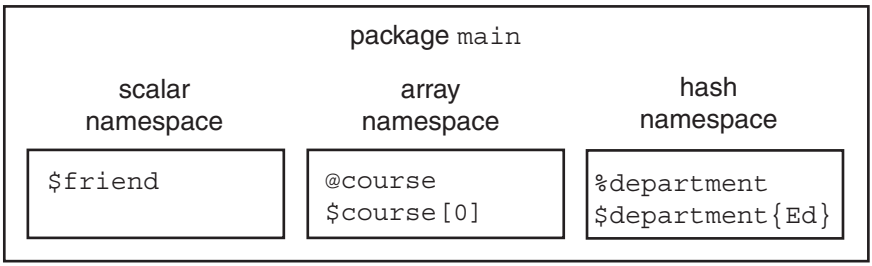


Figure 5.1 Namespaces for scalars, arrays, and hashes in package *main*.

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

```
   # 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
      my $father="Papa";
      my ($cousin, $sister, $brother);
4     my $family="McDonald";   # new variable
5     print "The $family family is visible here.\n";
   }
6  print "$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.
```

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*.[2] 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)
   use warnings;
   # Scalar, array, and hash assignment
1  my $salary=50000;                    # Scalar assignment
2  my @months=('Mar', 'Apr', 'May');    # Array assignment
3  my %states= (                        # Hash assignment
      CA => 'California',
      ME => 'Maine',
      MT => 'Montana',
      NM => 'New Mexico',
   );
4  print "$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.

5 The *@months* array is printed. The double quotes preserve spaces between each element.

6 The individual elements of the array, *@months*, are scalars and are thus preceded by a dollar sign ($). The array index starts at zero.

7 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 ($).

8 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.

9 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[3] 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

```
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)
   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.\n/;
10 print qq/The time is $now\n/;
11 print "The month is ${month}uary.\n";    # Curly braces shield
                                            # the variable
12 print "The month is $month" . "uary.\n"; # Concatenate
```

3. References are also stored as string variables.

EXAMPLE 5.5 (CONTINUED)

```
(Output)
6  5
7  John Smith
8  I need $125.75.
9  John Smith gave me $125.75.
10 The time is Sat Jan 24 16:12:49 2014.
11 The month is January.
12 The month is January.
```

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 *$month* is assigned *Jan*.
8 The quoted string is printed. The backslash allows the first dollar sign ($) to be printed literally; the value of *$money* is interpolated within double quotes, and its value printed.
9 The Perl *qq* construct replaces double quotes. The string to be quoted is enclosed in forward slashes. The value of the scalar *$friend* is interpolated; a literal dollar sign precedes the value of the scalar interpolated variable, *$money*.
10 The quoted string is printed as if in double quotes. The *$now* variable is interpolated.
11 Curly braces can be used to shield the variable from characters that are appended to it. *January* will be printed.
12 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

```
.
$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

```
undef $name;
```

The $_ Scalar Variable. The $_ (called a **topic** variable[4]) 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

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

(Output)
Donald Duck
```

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.

EXAMPLE 5.9

```
(The Script)
   use warnings;
   # Reading input from a file
1  open(my $fh, "<", "datebook.txt") or die $!;
2  while(<$fh>){  # loops through the file a line at a time storing
                  # each line in $_
3     print;      # prints the value stored in $_
4  }
5  close $fh;
```

```
(Output)
Jon DeLoach:408-253-3122:123 Park St., San Jose, CA 04086:7/25/53:85100
Karen Evich:284-758-2857:23 Edgecliff Place, Lincoln, NB
92086:7/25/53:85100
Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB
92743:11/3/35:58200
Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB
92743:11/3/35:58200
Fred Fardbarkle:674-843-1385:20 Parak Lane, DeLuth, MN
23850:4/12/23:780900
```

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.

```
$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
```

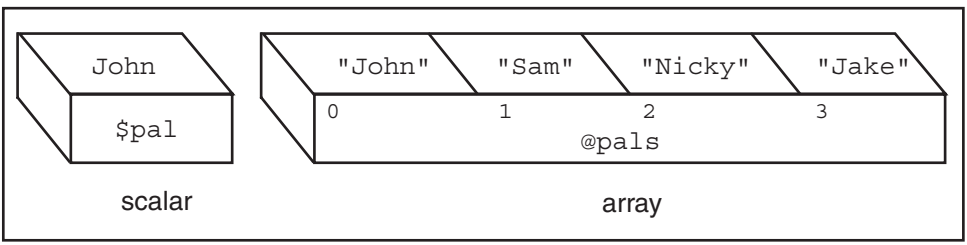


Figure 5.2 A scalar variable and an array variable.

EXAMPLE 5.10

```
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:

```
@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 ouput *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

```
1  use warnings;
2  my @pets=("Smokey", "Fido", "Gills", "Skiddy");
3  print @pets, "\n";  # Output separator is undef
4  $,="****";   # Changes the output field separator
5  print @pets,"\n";  # no quotes; ***** replaces undef
6  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

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

(Ouput)
2  meat potatoes rice beans spinach milk
4  meat---potatoes---rice---beans---spinach---milk
5  meat||potatotes||rice||beans||spinach||milk
```

EXPLANATION

2 The $" 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.

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

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

5 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

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

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

(Output)
2  The original array is: 90 89 78 100 87
3  The number of the last index is 4
4  The value of the last element of the array is 87
5  The size of the array is 5
6  The array is completely truncated:
```

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

```
   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:

```
@colors = qw( green red blue yellow);
```

2. To print the whole array, use the @:

```
print "@colors\n";
```

3. To print single elements of the array:

```
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.
```

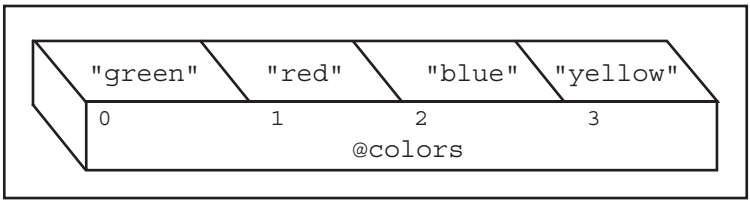


Figure 5.3 Array elements.

EXAMPLE 5.15

```
(The Script)
   use warnings;

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

(Output)
2  JohnJoeJake
3  Hi John, Joe, and Jake!
5  There are 3 elements in the @names array.
6  The last element of the array is Jake.
7  The last element of the array is Jake.
9  The first element of the @fruit array is apples; the second element is
   pears.
10 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 *$#names* value evaluates to the number of the last subscript in the array. This value used as a subscript will retrieve the last element in the *@names* array.
8 The *qw* construct creates an array of **singly** quoted words from the string provided to it, using space as the word separator. (You don't enclose the words in quotes or separate the words with commas.) The *qw* delimiter is any pair of nonalphanumeric characters.
9 The first two elements of the *@fruit* array are printed.
10 With a negative offset as an index value, the elements of the array are selected from the end of the array. The last element (*$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)
   use warnings;
   # Array slices
1  my @names=('Tom', 'Dick', 'Harry', 'Pete' );
2  foreach $pal (@names){
3     print "$pal\n";
   }

4  foreach ("red", "green", "yellow", "blue"){
5     print "$_ \n";
   }

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

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

(Output)
3  Tom Dick Harry Pete
5  Dick Harry Pete
7  Tom Dick Harry
```

EXPLANATION

1 The array *@names* is assigned the elements *'Tom'*, *'Dick'*, *'Harry'*, and *'Pete'*.

4 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.

6 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 that 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?"

```
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:

Keys	Values
"Name"	"Sneaky"
"Type"	"cat"
"Owner"	"Carol"
"Color"	"yellow"

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.

EXAMPLE 5.18

```
1  my %seasons=("Sp" => "Spring",
                "Su" => "Summer",
                "F"  => "Fall",
                "W"  => "Winter",
               );
```

EXAMPLE 5.18 (CONTINUED)

```
2  my %days=("Mon" => "Monday",
             "Tue" => "Tuesday",
             "Wed" => undef,
            );
3  $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
1  %department = (
2     "Eng" => "Engineering",   # keys do not require quotes
      "M"   => "Math",
      "S"   => "Science",
      "CS"  => "Computer Science",
      "Ed"  => "Education",
3  );
4  $department = $department{'M'};  # Either single, double quotes
5  $school = $department{'Ed'};
6  print "I work in the $department section\n" ;
7  print "Funds in the $school department are being cut.\n";
8  print qq/I'm currently enrolled in a $department{'CS'} course.\n/;
9  print qq/The department hash looks like this:\n/;
```

EXAMPLE 5.19 (CONTINUED)

```
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

```
(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
   }
   printf"Color   Number of Occurrences\n";
4  while((my $key, my $value)=each %dup){
      printf"%-12s%-s\n",$key, $value;
   }
5  @colors = sort keys %dup;
   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
```

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.

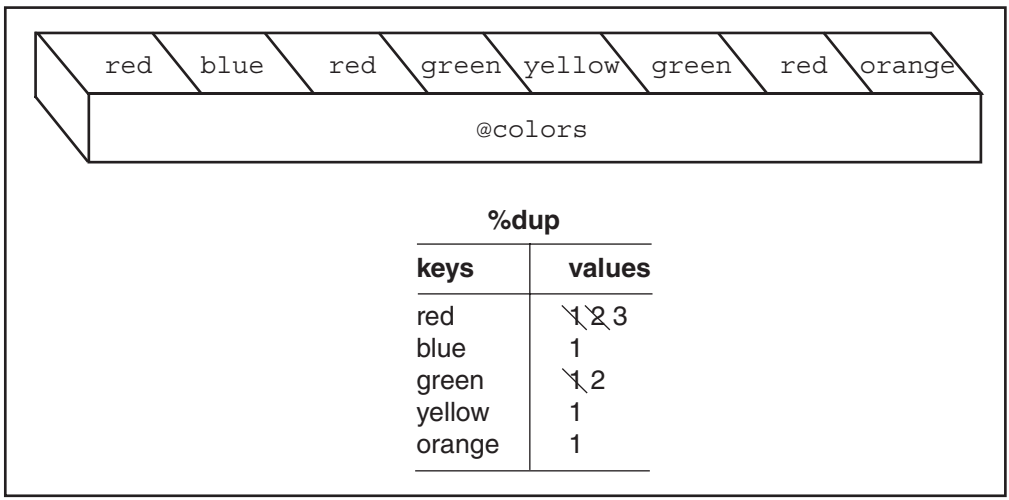


Figure 5.4 Removing duplicates with a hash.

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

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

EXAMPLE 5.22

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

(Output)
2  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.

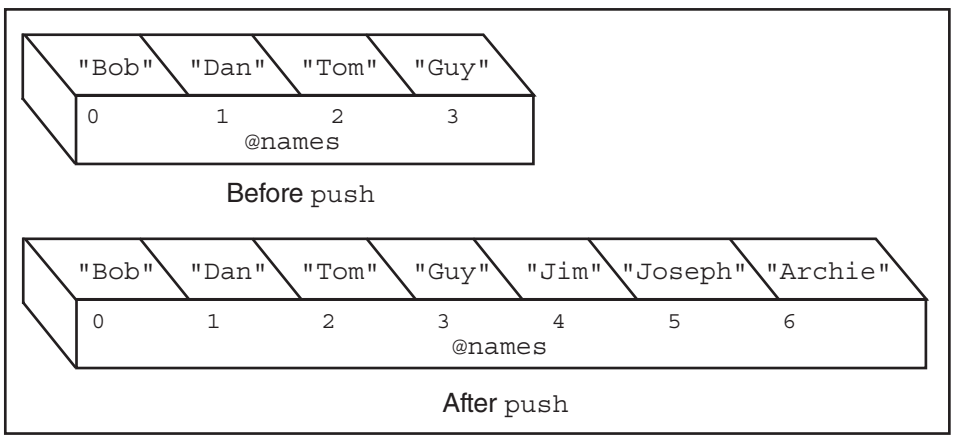


Figure 5.5 Adding elements to an array.

The *unshift* Function. The *unshift* function prepends *LIST* to the front of the array (see Figure 5.6).

FORMAT

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

EXAMPLE 5.23

```
(In Script)
   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.

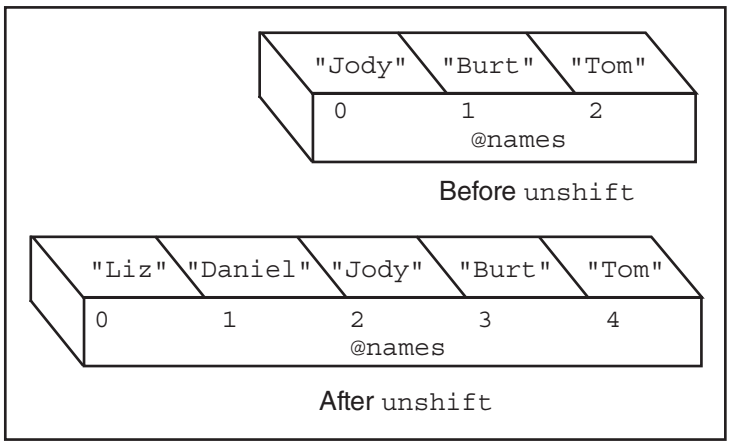


Figure 5.6 Using the *unshift* function to add elements to the beginning of an array.

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.

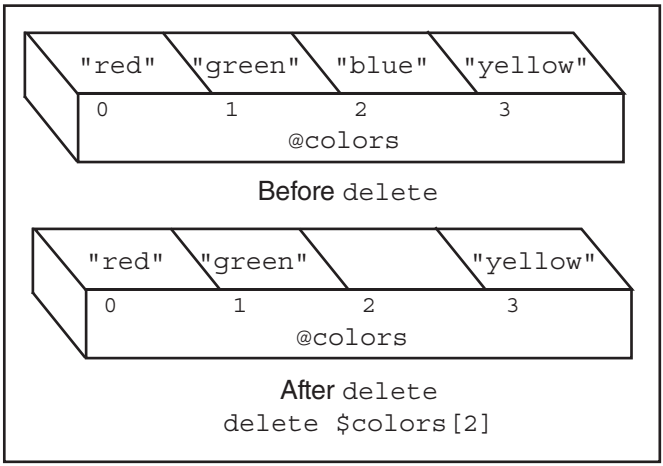


Figure 5.7 Using the *delete* function to remove elements from an array.

The *splice* Function. For the *delete* function, we described a row of shoeboxes in which a pair of shoes was removed from one of the boxes, but the box itself remained in the row. With *splice*, the box and its shoes can be removed and the remaining boxes pushed into place. (See Figure 5.8.) We could even take out a pair of shoes and replace them with a different pair (see Figure 5.9), or add a new box of shoes anywhere in the row. Put simply, the *splice* function removes and replaces elements in an array. The *OFFSET* is the starting position where elements are to be removed. The *LENGTH* is the number of items from the *OFFSET* position to be removed. The *LIST* consists of an optional new elements that are to replace the old ones. All index values are renumbered for the new array.

FORMAT

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

EXAMPLE 5.24

```
(The Script)
   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.

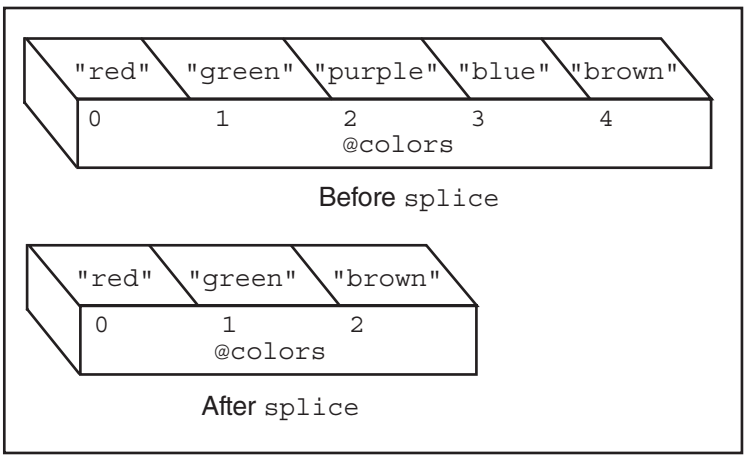


Figure 5.8 Using the *splice* function to remove or replace elements in an array.

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
```

EXPLANATION

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

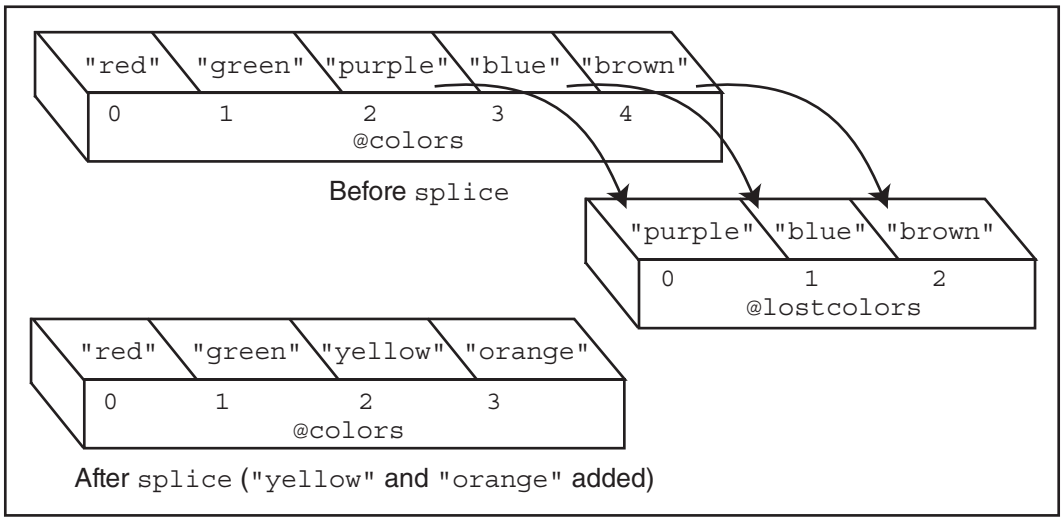


Figure 5.9 Splicing and replacing elements in an array.

The *pop* Function. The *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.)

FORMAT

```
pop(ARRAY)
pop ARRAY
```

EXAMPLE 5.26

```
(In Script)
   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";
```

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.

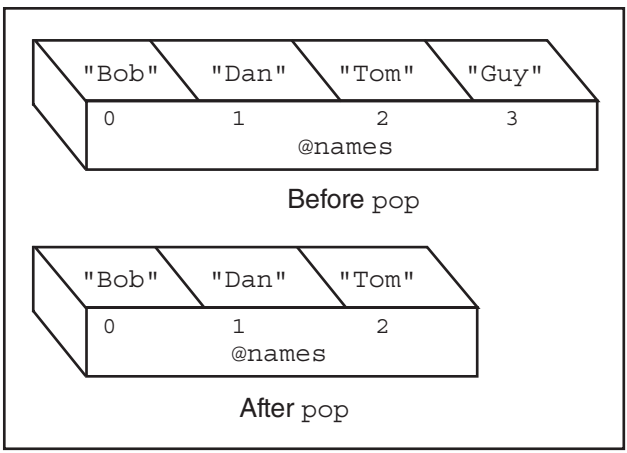


Figure 5.10 Using the *pop* function to pop the last element off the array.

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
```

EXAMPLE 5.27

```
(In Script)
   use warnings;
   # Removing elements from front of a list
1  my @names=("Bob", "Dan", "Tom", "Guy");
2  my $ret = shift @names;
3  print "@names\n";
4  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.

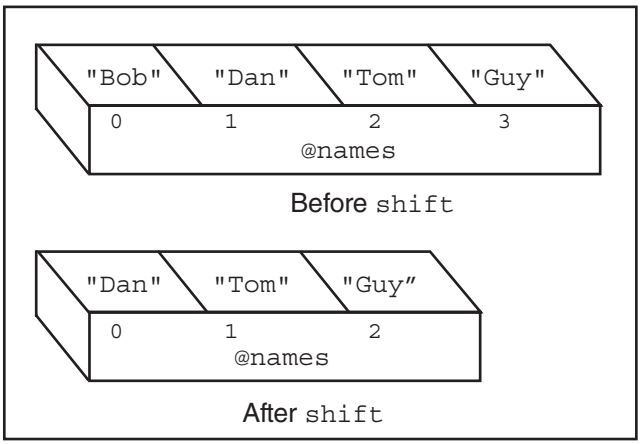


Figure 5.11 Using the *shift* function to return the first element of an array.

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.

FORMAT

```
chop(LIST)
chomp(LIST)
```

EXAMPLE 5.28

```
(In the Script)
   use warnings;
   # Chopping and chomping a list
1  my @line=("red", "green", "orange");
2  chop(@line);   # Chops the last character off each
                  # string in the list
3  print "@line";
4  @line=( "red\n", "green\n", "orange\n");
5  chomp(@line);  # Chomps the newline off each string in the list
6  print "@line";

(Output)
3  re gree orang
6  red green orange
```

EXPLANATION

1 The array *@line* is assigned a list of elements.
2 The array is chopped. The *chop* function chops the last character from each element of the array.
3 The chopped array is printed.
4 The array *@line* is assigned a list of elements.
5 The *chomp* function will chop off the newline character from each word in the array. This is a safer function than *chop*.
6 If there are no newlines on the end of the words in the array, *chomp* will not do anything.

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

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

4  print "Found items: @items\nNumber found: $count\n";

(Output)
4  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 libaray, or from CPAN.)

EXAMPLE 5.30

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

(Output)
3  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, *[+\t:]* 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.

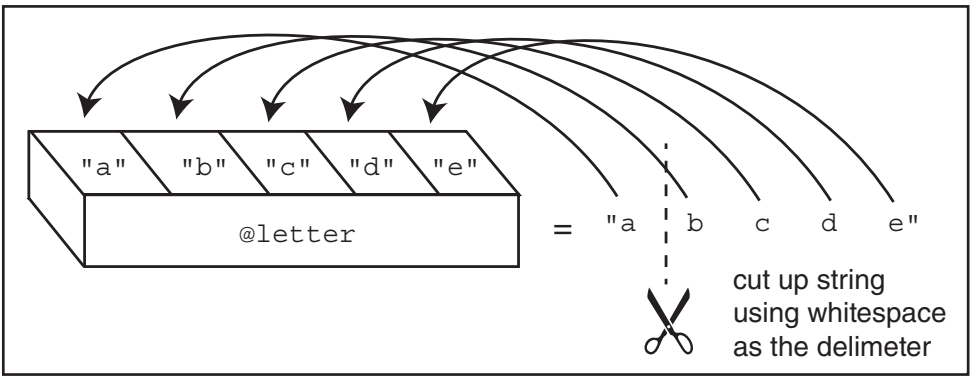


Figure 5.12 Using the *split* function to create an array from a scalar.

EXAMPLE 5.32

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

__DATA__
Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500
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)
   use warnings;
   my($name, $phone, $address, $bd, $sal);
   # Splitting up $_ and creating an unnamed list
   while(<DATA>){
1     ($name,$phone,$address,$bd,$sal)=split(":");
2     print "$name\t $phone\n" ;
   }
```

EXAMPLE 5.33 (CONTINUED)

```
__DATA__
Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500
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)
2  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)
   use warnings;
   # Many ways to split a scalar to create a list
1  my $string= "Joe Blow:11/12/86:10 Main St.:Boston, MA:02530";
2  my @line=split(":", $string);   # The string delimiter is a colon
3  print @line,"\n";
4  print "The guy's name is $line[0].\n";
5  print "The birthday is $line[1].\n\n";

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

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

11 print $name,"\n";
12 print $birth,"\n";
13 print $address,"\n";
```

EXAMPLE 5.34 (CONTINUED)

```
(Output)
3  Joe Blow11/12/8610 Main St.Boston, MA02530
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.

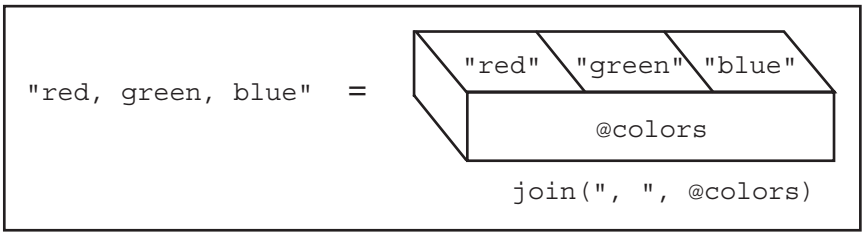


Figure 5.13 Using the *join* function to join elements of an array with a comma.

EXAMPLE 5.36

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

(Output)
3  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.

FORMAT

```
map EXPR, LIST;
map {BLOCK} 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.)

EXAMPLE 5.37

```
(The Script)
   use warnings;
   my(@list, @words, @n);
   # Mapping a list to an expression
1  @list=(0x53,0x77,0x65,0x64,0x65,0x6e,012);
2  my @letters = map chr $_, @list;
3  print @letters;
4  my @n = (2, 4, 6, 8);
5  @n = map $_ * 2 + 6, @n;
6  print "@n\n";

(Output)
3  Sweden
6  10 14 18 22
```

EXPLANATION

1 The array *@list* consists of six hexadecimal numbers and one octal number.

2 The *map* function maps each item in *@list* to its corresponding *chr* (character) value and returns a new list, assigned to *@letters*. (According to *perldoc.perl.org*, the *chr* function "returns the character represented by that NUMBER in the character set. For example, chr(65) is "A" in either ASCII or Unicode, and chr(0x263a) is a Unicode smiley face.")

3 The new list is printed. Each numeric value was converted with the *chr* function to a character corresponding to its ASCII value; for example, *chr(65)* returns ASCII value *"A"*.

4 The array *@n* consists of a list of integers.

5 The *map* function evaluates the expression for each element in the *@n* array and returns the result to the new array *@n*.

6 The results of the mapping are printed, showing that the original list has been changed.

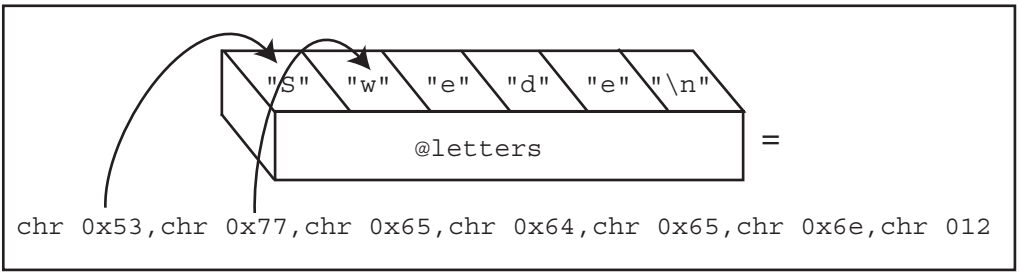


Figure 5.14 Using the *map* function to change elements in an array.

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)
   use warnings;
   my(@courses, %c);
1  @courses=qw( C++ C Perl Python French C C Perl);
2  %c = map { $_ => undef } @courses;  # Create a unique list of keys
3  @courses = keys %c;
4  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

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

EXAMPLE 5.39

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

   # Reversed alphabetic sort
4  @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;
1  my @list=("dog","cat", "bird","snake" );
   print "Original list: @list\n";
   # ASCII sort using a subroutine
2  sub asc_sort{
3     $a cmp $b;  # Sort ascending order
   }
4  @sorted_list=sort asc_sort(@list);
   print "ASCII sort: @sorted_list\n";

   # Numeric sort using subroutine
5  sub numeric_sort {
      $a <=> $b ;
   }  # $a and $b are compared numerically

6  @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)
   use warnings;
   # Sorting numbers with block
1  my @sorted_numbers = sort {$a <=> $b} (3,4,1,2);
2  print "The sorted numbers are: @sorted_numbers", ".\n";

(Output)
2  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* 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

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

EXAMPLE 5.42

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

(Output)
2  Hello Raul
3  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

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

EXAMPLE 5.43

```
(In Script)
   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)
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\n";

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.

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)

```
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

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

EXAMPLE 5.47

```
(In Script)
   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:

```
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)
   use warnings;
   my(@list, @uniq);
   # Using the map function with a hash
   @list = qw/a b c d 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 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;
1  my %wins = (
      "Portland Panthers"   => 10,
      "Sunnyvale Sluggers"  => 12,
      "Chico Wildcats"      => 5,
      "Stevensville Tigers" => 6,
      "Lewiston Blazers"    => 11,
      "Danville Terriors"   => 8,
   );
   print "\n\tSort Teams in Ascending Order:\n\n";
2  foreach my $key(sort keys %wins) {
3     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
```

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;
1  my %wins = (
      "Portland Panthers"   => 10,
      "Sunnyvale Sluggers"  => 12,
      "Chico Wildcats"      => 5,
      "Stevensville Tigers" => 6,
      "Lewiston Blazers"    => 11,
      "Danville Terriors"   => 8,
   );
   print "\n\tSort Teams in Descending/Reverse Order:\n\n";
2  foreach my $key (reverse sort keys %wins) {
3     printf "\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;
1  sub desc_sort_subject {
2     $b <=> $a;            # Numeric sort descending
   }
3  sub asc_sort_subject{
4     $a <=> $b;            # Numeric sort ascending
   }

5  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";
6  foreach my $key (sort asc_sort_subject(keys %courses)) {
7     printf "\t%-5d%s\n", $key, $courses{"$key"};
   }
8  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
```

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)
   use warnings;
1  sub asc_sort_wins {
2     $wins{$a} <=> $wins{$b};
   }
3  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";
4  foreach my $key (sort asc_sort_wins(keys %wins)) {
5     printf "\t% -20s%5d\n", $key, $wins{$key};
   }
```

EXAMPLE 5.52 (CONTINUED)

```
(Output)

Wins in Ascending Numeric Order:

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

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)
   use warnings;
   # Sorting a hash by value in descending order

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

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

EXAMPLE 5.53 (CONTINUED)

```
   print "\n\tWins 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)

```
2  my $layoff=delete $employees{"Janitor"};
   print "We had to let $layoff go.\n";
   print "Our remaining staff includes: ";
   print "\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

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

EXAMPLE 5.55

```
   use warnings;

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

2  print "The Nightwatchman exists.\n" if exists
      $employees{"Nightwatchman"};
3  print "The Clerk exists.\n" if exists $employees{"Clerk"};
4  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.
```

EXPLANATION

1 A hash is defined with three key/value pairs.
2 If a key *"Nightwatchman"* exists, the *exists* function returns true.
3 If a key *"Clerk"* exists, the exists function returns true.
4 If the key *"Clerk"* does **not** exist, the inverted value of the *exists* function is false.

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 roll in CGI Perl scripts.

EXAMPLE 5.56

```
(In Script)
   use warnings;
1  foreach my $key (keys %ENV){
2     print "$key\n";
   }
3  print "\nYour login name $ENV{'LOGNAME'}\n";
4  my $pwd = $ENV{'PWD'};
5  print "\n", $pwd, "\n";

(Output)
2  OPENWINHOME
   MANPATH
   FONTPATH
   LOGNAME
   USER
   TERMCAP
   TERM
   SHELL
   PWD
   HOME
   PATH
   WINDOW_PARENT
   WMGR_ENV_PLACEHOLDER

3  Your login name is ellie

5  /home/jody/home
```

EXPLANATION

1 The *foreach* loop iterates through the keys of the *%ENV* hash.
3 Print the value of the key *LOGNAME*.
4 Assign the value of the key *PWD* to *$pwd*.
5 Print the value of *$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;
1  sub handler{
2     local($sig) = @_;  # First argument is signal name
3     print "Caught SIG$sig -- shutting down\n";
      exit(0);
   }
4  $SIG{'INT'} = 'handler';  # Catch <CTRL>+C
   print "Here I am!\n";
5  sleep(10);
6  $SIG{'INT'}='DEFAULT';
7  $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.

```
$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)
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)
   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 *scalar* function evaluate an expression if it's an array?
8. How do you find the size of an array?
9. What does the *$"* special variable do?
10. When are elements of an array or hash preceded by a $ (dollar sign)?
11. What is the difference between *chop* and *chomp?*
12. What is the difference between *splice* and *slice*?
13. What does the *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 **scope**?
18. What is "scalar" context, "list" context, "void" context? Would you be able to write an example to demonstrate how they differ?

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.

EXERCISE 5
The Funny Characters

1. Write a script that will ask the user for his five favorite foods (read from *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 *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-xxxx): 004-34-1234
   Your address:
   StreetHobartSt
   CityStateZipChicoCA

"EDP" NUMBERS AND ELECTIVES:

2CPR2B | C Programming

1UNX1B | 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
      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

A

B

C

D

G

H

I

M

N

Q

R

S

T

W

X

Y

Z