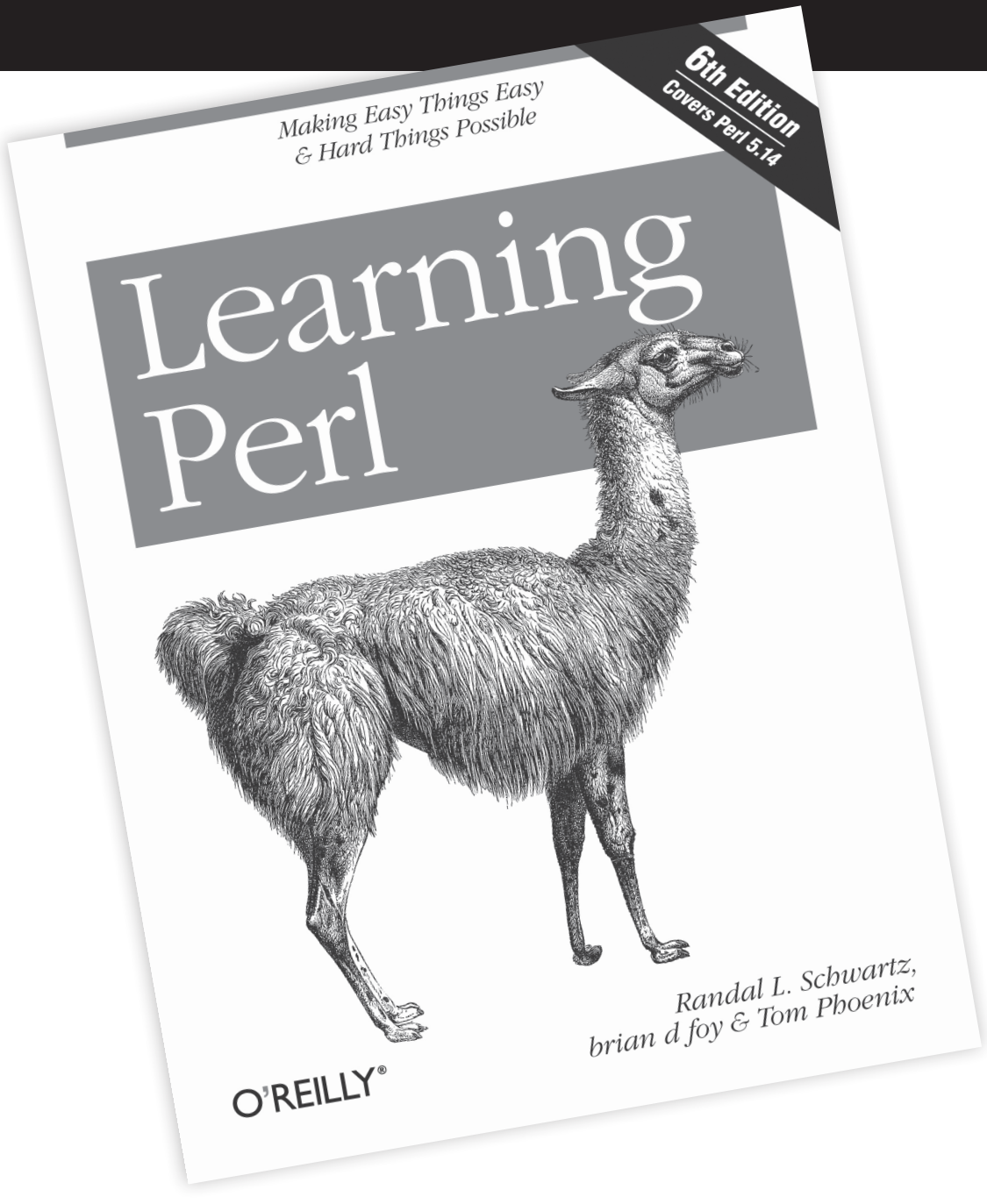
Student Workbook



O'REILLY®

brian d foy

SECOND EDITION

Student Workbook for Learning Perl

brian d foy

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Student Workbook for Learning Perl, Second Edition
by brian d foy

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Shawn Wallace
Production Editor: Melanie Yarbrough

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

January 2012: Second Edition.

Revision History for the Second Edition:
2012-01-25 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449328061* for release details.

ISBN: 978-1-449-32806-1

[LSI]

1327515583

Table of Contents

Preface

This workbook is an additional set of exercises to supplement those already in *Learning Perl, Sixth Edition*. The chapter and page references apply only to that edition. Additionally, unless I denote otherwise, *Learning Perl* without qualification means the sixth edition. That book already contains some exercises, and I try to cover the topics those exercises didn't.

I've been teaching Perl since 1998, and my beginner classes use *Learning Perl* as the course text. Along the way, I've posed a lot of additional problems to students so they could test their understanding of the topics I covered and some of the additional information I added in the lecture. In doing this, I pose two sorts of problems: one of simple knowledge where knowing the right fact or trick makes the problem easy, and the other somewhat clever where a Perl implementation of a particular technique solves the problem. I've tried to include both of those sorts of problems in this book.

You should find some exercises are easy, some require that you find nuggets of information you may have missed on a first reading of *Learning Perl*, and others require you know something about algorithms or programming that aren't simply a question of Perl knowledge and might require you to do some research on your own. As you learn about how to learn about Perl, whether through its documentation or online resources, I pose these challenging problems expecting that you'll use everything available to you. There's a lot more to programming than just the syntax.

This isn't a classroom and you aren't being graded, unless it is a classroom and you *are* being graded. I won't think you are cheating if you use Google, and you shouldn't feel that you have to come up with everything just by staring at a blank document. If someone is grading you, just show them this paragraph and tell them I say it's okay. If you're the instructor, fear not: no one reads the preface anyway, especially if it's not on the test.

You should be able to complete most of the exercises with Perl 5.8, and I expect that you might even be able to use Perl 5.004, a quite ancient version indeed. Some of the exercises require a minimum version of Perl because I want you to use a particular feature introduced in that version, and I'll note the minimum version in the exercise when it's appropriate. If you don't note a minimum version assume that it's at least

Perl 5.8. I'd prefer that you use a supported version of Perl, which at this writing is at least Perl 5.14. However, I know that some of you don't get to make that choice.

Perl's motto, for good or bad, is "There's More Than One Way To Do It", so don't take my solutions as gospel. Remember that the corollary to that is "But most of them are wrong", so don't go out of your way to avoid techniques that I show. I show you a way to do it, and often I give you more than one way to do it. However, if you accomplished the task and your solution doesn't look remotely similar to mine, that's okay (probably).

Some of my answers have extra material that show techniques you'll read about in later chapters. I don't expect you to use those techniques, but I show them because they are the techniques and features you'll use in everyday programming once you finish the book. They are just a taste of things to come.

PART I

Exercises

CHAPTER 1
Introduction

1.1. The *perldoc* command is the key to all of the Perl documentation, and *perl* comes with literally thousands of pages of printed documentation (should you print them all out, which would take quite a long time). Take yourself on a tour of the *perldoc* command starting with the *perl* documentation. Next, try the *perlfunc* documentation, which lists all of the built-in Perl functions. This information may also be available in other formats on your system, and on some systems, you may need to download a separate package to get the Perl documentation (which should be a crime, but is becoming more common):

```
$ perldoc perl
$ perldoc perlfunc
```

(Answer to Exercise 1.1 on page 45)

1.2. The *perlfunc* documentation is much too long to scroll through every time you want to read about a Perl function, so *perldoc* has a handy switch to look up functions directly. Can you figure out what it is? (Answer to Exercise 1.2 on page 45)

1.3. The *perldoc* command has another switch, `-q`, which also looks up answers in the *perlfaq* documents. Give it a search term and it looks for answers (or questions) that contain that word. Try it with a few words to see what comes up. If you can't think of one, try **`comma`**. (Answer to Exercise 1.3 on page 45)

1.4. You saw the backticks (`) operator in the second example in this chapter in *Learning Perl* (page 17), although we do not cover it until Chapter 16. You can use any command line you like in backticks to save its output in your Perl program. Take the backtick operator for a spin by inserting your own command in it. (Answer to Exercise 1.4 on page 46)

1.5. The *perldoc* command, starting with version 3.15, has a switch that allows you to look up Perl special variables in the *perlvar* documentation. Find out what it is and use it to look up the variable `$_`. You may need to use special quoting so the shell does not think it should interpret the special characters. Also, before you start, ensure you have a recent enough version of *perldoc*. (Answer to Exercise 1.5 on page 46)

CHAPTER 2
Scalar Data

2.1. In Perl versions 5.6 onward, you can also turn on warnings with a pragma instead of the *–w* command-line switch:

```
use warnings;
```

You may already have a program that outputs warnings, but if you don't, write one and add the warnings pragma to it. If you can't come up with one, try these programs, each of which outputs a different sort of warning. Which warning do you get for each?

Program 1:

```
print;
```

Program 2:

```
3 + 4;
```

Program 3:

```
print $n + 1;
```

(Answer to Exercise 2.1 on page 49)

2.2. We mention the *perldiag* on page 29 of *Learning Perl*. This pragma gives you longer versions of warnings you get from my program. Perl has a shortcut into that documentation through the `diagnostics` pragma. Change the programs from the Exercise 2.1 to look in *perldiag* instead by replacing the warnings line with this one:

```
use diagnostics;
```

What difference do you see? (Answer to Exercise 2.2 on page 49)

2.3. Write a program that asks the user for an integer then reports if that number is odd or even. Turn on warnings and try this program with non-numerical input. What warning do you get? (Answer to Exercise 2.3 on page 50)

2.4. In Exercise 2.3, you read some input from the user, and immediately used `chomp` on the input to remove the trailing newline. Would the program still work if you took out the `chomp`? Would you get a warning? (Answer to Exercise 2.4 on page 51)

2.5. Remembering that Perl operations of the same precedence use associativity to decide what happens first, what is the answer to these operations? Which operation happens first? Does it matter?

```
2 ** 3 ** 4
2 / 3 * 4
2 + 3 * 4 ** 5 - 6
```

(Answer to Exercise 2.5 on page 51)

2.6. Write a program that asks the user for two numbers and reports which one is larger. What happens if they are the same? What happens if you don't enter digits, but something like **`thirty-seven`**? (Answer to Exercise 2.6 on page 51)

2.7. Repeatedly prompt the user to enter numbers. If the user enters a number, add that number to the sum of all the previously entered numbers. If the user ends input (with Control D on Unix-like systems and Mac OS X or Control Z on Windows), stop the program and report the sum. What happens if the user enters nothing but a newline? (Answer to Exercise 2.7 on page 52)

2.8. Write a program that prompts the user for a code point and prints the character for that code point. If you can't think of any interesting code points, try U+26F3, U+267A, U+2126, and U+03B6. Which characters are those code points?

If you need help setting up your terminal to handle Unicode, see Appendix C in *Learning Perl*.

(Answer to Exercise 2.8 on page 53)

CHAPTER 3
Lists and Arrays

3.1. Write a program to read in a list of strings then report the second-to-last element. There are many ways to do this, but two of them are easy. Use an array to store the lines. (Answer to Exercise 3.1 on page 55)

3.2. In Exercise 3.1, you used array indices to get to the second-to-last line. Do the same thing but don't use an array this time. (Answer to Exercise 3.2 on page 56)

3.3. Print the list of numbers from 1 to 10 and separate them with commas by interpolating an array in a double quoted string. (Answer to Exercise 3.3 on page 56)

3.4. Using a `foreach` loop, go through the numbers from 1 to 10 and print their squares and cubes. Use the range operator to create the list of numbers. (Answer to Exercise 3.4 on page 56)

3.5. Starting with the array `@numbers` that contains a list of numbers, use a `while` loop and `shift` to print the squares and cubes of the numbers in the list. (Answer to Exercise 3.5 on page 57)

3.6. The range operator, `..`, works with more than just numbers. Use the range operator to create a list of only the lowercase letters then a list of only the uppercase letters. Can you create a list of only the upper and lowercase letters together, and can you do it with only one range operator? (Answer to Exercise 3.6 on page 57)

CHAPTER 4
Subroutines

In this chapter in *Learning Perl*, we introduced the `strict` pragma that enforces some good programming practices. For the rest of this workbook, I'm going to make "strict-clean" programs and I'll use the `strict` pragma at the top of my programs.

Remember, if you're using Perl 5.12 or later and you specify that version in your program, you'll automatically turn on `strict`.

4.1. Write a subroutine named `show_args` that simply prints its arguments. For instance, given "fred", "barney", and "betty", you would call the subroutine like this:

```
show_args( 'fred', 'barney', 'betty' );
```

And you would get this output:

```
The arguments are fred barney betty
```

(Answer to Exercise 4.1 on page 59)

4.2. Write another subroutine named `show_args_again` that does the same thing as the `show_args` that you wrote for the Exercise 4.1. From `show_args_again`, call that subroutine from `show_args`, using the `&` in front of the subroutine name. Try it without the `&`. What's the difference when you call `show_args_again` with the same arguments from the Exercise 4.1? (Answer to Exercise 4.2 on page 60)

4.3. Starting with the same program from the Exercise 4.2, try calling `&show_args` in two more ways: with the `&` and trailing parentheses, `&show_args()`, and with just the trailing parentheses, `show_args()`. What do you get? How is that different from the Exercise 4.2? (Answer to Exercise 4.3 on page 61)

4.4. Modify the `show_args_again` subroutine from your answer to Exercise 4.3 so the `show_args` subroutine prints the original argument list in reverse order. Do not change `show_args`! (Answer to Exercise 4.4 on page 61)

4.5. Write a subroutine that takes an operator and list of numbers and does one of two (rather silly) things. If the first argument is `+`, it adds the numbers and returns the sum. If the first argument is `*`, it multiplies the numbers and returns the product. The first argument is a string, not just the bare operator. (Answer to Exercise 4.5 on page 62)

4.6. Write a subroutine that takes a list of strings and separates the list just as you would in prose: with two items, put `" and "` between them. For three or more items, separate all but the last two with commas, and separate the last two with `" and "` (and perhaps a Harvard comma). (Answer to Exercise 4.6 on page 63)

CHAPTER 5
Input and Output

5.1. Write a program to read lines from all of the files on the command line and print out each line prefaced with its filename. (Answer to Exercise 5.1 on page 65)

5.2. Modify your answer for Exercise 5.1 so after it reads lines from the files you specify on the command line, it reads in lines from standard input and prefaces the line with `stdin`. (Answer to Exercise 5.2 on page 66)

5.3. Explain the output of this program then fix the bug. The line with `while` comes straight from the *perlfaq4* documentation. You can see the whole answer about adding commas to numbers using **`perldoc -q comma`**:

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

print format_number( 9999 ), "\n";

sub format_number {
    local $_ = shift;

    1 while s/^([-+]?\d+)(\d{3})/$1,$2/;

    print $_;
    }
```

(Answer to Exercise 5.3 on page 66)

5.4. Write a program that prompts the user for a whole number then prints that number in binary, octal, decimal, and hexadecimal notation. (Answer to Exercise 5.4 on page 67)

5.5. Write a program that takes two whole numbers from the command line then uses `printf` to report in columns their sum, product, and the percent fraction (to two decimal places) of the numbers (including a literal `%` in the output). For instance, given the numbers **`5`** and **`6`**, produce this output:

```
$ perl ex5-4.pl 5 6
  first  second     sum   product     %
      5       6      11        30   83.33%
```

(Answer to Exercise 5.5 on page 68)

5.6. Write a program that takes a list of whole numbers from the command line and prints them in a column. Before you print the column, ensure that all the numbers will line up correctly. The number of columns in the output should automatically choose their size based on the input:

```
$ perl ex5.6.pl 12466 5 67 984
12466
    5
   67
  984

$ perl ex5.6.pl 3478 48 120 7 36455
 3478
   48
  120
    7
36455
```

(Answer to Exercise 5.6 on page 69)

5.7. Write a program that goes through the list of files from the command line and reports their line count. Use `warn` to print a message for each file that does not exist. (Answer to Exercise 5.7 on page 70)

5.8. Modify your answer to Exercise 5.7 to re-open `STDERR` to a file in the current directory so you capture all of the error output in that file. Remember that `warn` outputs to `STDERR`. (Answer to Exercise 5.8 on page 71)

5.9. Write a program that takes a filename from the command line. If the file exists, open the file and print the first line. If you can't open the file, use `die` to report the reason. Try it with files that don't exist and files you don't have permission to read. (Answer to Exercise 5.9 on page 71)

5.10. Write a program that prompts the user for a message then appends the message to a log file. Add a timestamp to each entry such that the entry looks something like:

```
[Mon Aug 16 15:23:49 CDT 2010] this is a log message
```

(Answer to Exercise 5.10 on page 72)

5.11. Write a program that prints its command-line arguments to standard output using a carriage-return/line-feed pair as the line separator between each argument. Use `say` without specifying a line ending. Specify the line ending using an encoding with `open`. Save the file you create so you can use it with Exercise 5.12. (Answer to Exercise 5.11 on page 73)

5.12. Write a program that mimics the `dos2unix` program to convert a file with a carriage-return/newline-pair into just a newline. Do this by using Perl's encoding layers.

You can use the file you created in Exercise 5.11 as the input. (Answer to Exercise 5.12 on page 73)

5.13. Write a program that converts a file encoded as UTF-8 into UTF-16BE. Use Perl's encoding layers to read the data in the correct format and output the same data in the new encoding. Use command-line arguments to specify the input and output files. (Answer to Exercise 5.13 on page 73)

CHAPTER 6
Hashes

6.1. Given a list of names and birthdays, create a hash to store them. Print the contents of the hash. If you don't know anyone's birthday, you can use this list: Fred's birthday is April 5, Wilma's is October 26, and Pebbles is October 8. Check a *Flintstones* episode guide if you don't believe me! (Answer to Exercise 6.1 on page 75)

6.2. Modify the program from Exercise 6.1 so it prompts for the user for a name, and then reports the birthday for that name. (Answer to Exercise 6.2 on page 75)

6.3. What happens in your program from Exercise 6.2 if the user asks for the birthday for someone who is not in the hash? Modify the program you created for Exercise 6.2 to warn the user if the key is not in the hash. (Answer to Exercise 6.3 on page 76)

6.4. What happens in your program for Exercise 6.3 if the user-entered name is in the hash, but does not have a birthday? Add a couple of entries to the hash: one with a value of `undef`, and one with a value of the empty string. Modify your program for Exercise 6.3 to tell the user that you don't know the birthday if the key is in the hash, but does not have a value. (Answer to Exercise 6.4 on page 76)

6.5. Write a program to create a birthday report. Use the hash you created for Exercise 6.4 and output each entry that has a birthday. Go through each entry of the hash with a `foreach` loop. (Answer to Exercise 6.5 on page 77)

6.6. Modify your program from Exercise 6.5 to go through the hash with the `each` function instead of a `foreach` loop. (Answer to Exercise 6.6 on page 78)

6.7. In Exercise 6.1, you wrote a program to report the entries in the hash, and since then you've added keys that have values that are undefined or the empty string. Modify that first program to delete from the hash the keys that don't have true values then print the contents of the hash. Don't just skip the entries with false values—actually remove those entries from the hash. (Answer to Exercise 6.7 on page 79)

CHAPTER 7

In the World of Regular Expressions

7.1. Write a program that matches a string containing either "fred" or "barney" then reports the part of the string before the match, the part of the string after the match, and the name that it matched. (Answer to Exercise 7.1 on page 81)

7.2. Write a program to match a line that has an "a", a "b", and a "c", in that order. Any number of characters can be between those letters. (Answer to Exercise 7.2 on page 82)

7.3. Write a program to match a string that has an "a", a "b", and a "c", in that order, but must have at least one other character between each of those letters. (Answer to Exercise 7.3 on page 83)

7.4. Write a program to match a string that contains an "a", but doesn't have a "b" anywhere after the "a". (Answer to Exercise 7.4 on page 84)

7.5. [Perl 5.010] Use a relative backreference to construct a pattern to match a sequence of at least three characters that repeats itself at some point in the string. You should be able to match strings such as `abcdabc` since `abc` repeats itself. (Answer to Exercise 7.5 on page 85)

7.6. [Perl 5.010] Use relative backreferences to construct a pattern to match a sequence of at least three characters that reports itself, but interwoven with another sequence that repeats itself. You should be able to match strings such as `abc123abcd123` since `abc` and `123` each repeat themselves. (Answer to Exercise 7.6 on page 85)

7.7. [Perl 5.010] Use the Unicode properties to match lines that have numbers or whitespace. If the line has a character with the number property, output a message saying so. If the line has a character with whitespace, output a message saying so. If it has both, output two messages. (Answer to Exercise 7.7 on page 86)

CHAPTER 8

Matching with Regular Expressions

In Chapter 7, you used the same basic program to test all of the regular expressions. Instead of copying that into every answer for this chapter, I'll just give you the regular expressions and you can plug them into the program yourself.

8.1. Write a regular expression to match an "a","b", or "x" in a string. Use a character class. After you do that, make it case insensitive. (Answer to Exercise 8.1 on page 89)

8.2. Modify the regular expressions from Exercise 8.1 to match the "a", "b", or "x" at the beginning of the string. After that, modify it to match any of those characters at the end of the string. (Answer to Exercise 8.2 on page 89)

8.3. Modify your regular expression from Exercise 8.2 to match a non-empty string that doesn't have an "a", "b", or "x" at the beginning of the string. Change it again so the regular expression matches a non-empty string that doesn't have those same characters at the end of the string. (Answer to Exercise 8.3 on page 90)

8.4. Write a regular expression that matches a string that begins and ends with the same thing, without overlap between the two parts of the string. (Answer to Exercise 8.4 on page 90)

8.5. Write a program (not just a regular expression) that can tell the difference between scalar, array, and hash variables, and have it report what it finds. Try it with these names: `$fred`, `@barney`, `%betty`. Also try writing your pattern with the `/x` match modifier so you can comment each part of your pattern. How would you limit the variable names to just ASCII? How would you match variable names using non-ASCII characters? You don't have to handle any of the special variables such as `$1` or `${^PREMATCH}`. (Answer to Exercise 8.5 on page 90)

CHAPTER 9
Processing Text with Regular Expressions

9.1. Without escaping characters, write regular expressions that match these string descriptions.

1. The Unix path name */usr/bin/perl*.
2. A web address that has a named anchor fragment in it, such as *http://www.example.com/index.html#fragment*.
3. A single line C++ comment. That's a `//` until the end of the line.

(Answer to Exercise 9.1 on page 95)

9.2. In Chapter 8, you created some case-insensitive regular expressions, and if you only used the stuff from *Learning Perl* up to Chapter 8, you had to do a bit of work. Rewrite the regular expressions from Exercise 8.1 and Exercise 8.5 for what you know from this chapter. (Answer to Exercise 9.2 on page 95)

9.3. Prompt the user for three strings and a regular expression. Apply the regular expression to each string and report if it matches. (Answer to Exercise 9.3 on page 96)

9.4. Read lines from standard input and print them in four versions: the original string, all uppercase, all lowercase, and all lowercase except for the first character in the string. Use the case-shifting operators inside double-quoted strings. (Answer to Exercise 9.4 on page 97)

9.5. Modify your answer to Exercise 9.4 to also uppercase any letter after a word boundary. Try it with and without the `/r` modifier for the substitution operator. (Answer to Exercise 9.5 on page 98)

9.6. Create a file of tab-separated values, including a line that has the names of each column of values. Write a program that reads in the file and produces a report for each line.

If you don't have a tab-separated file, use this one. Note that the line with Dino has a blank field so it has two tabs next to each other:

```
First   Last Score
Fred    Flintstone 230
Barney  Rubble    195
Dino         30
```

The report for each line should look like this:

```
First: Fred
Last: Flintstone
Score: 230
```

(Answer to Exercise 9.6 on page 98)

9.7. Write a program to read the tab-separated file and output it as a pipe ("|") separated file. Assume that there are no | characters in the values. Use the `split` and `join` functions. (Answer to Exercise 9.7 on page 99)

CHAPTER 10
More Control Structures

10.1. Write a program that reads in lines of input and prints them unless they contain the words "ruby" or "python". (Answer to Exercise 10.1 on page 101)

10.2. Write a program that counts the number of lines of input until it finds the word "perl" then stops and reports the last line number. (Answer to Exercise 10.2 on page 101)

10.3. Write a program that reads lines of input and splits it into words. Use a hash to keep a count of the number of times you've seen each word then create a report of the count for each word. Does it count uppercase and lowercase versions of the same word? Does it handle punctuation and other "non-word" characters? (Answer to Exercise 10.3 on page 102)

10.4. Using a `for` loop, write a program to report a table of squares and cubes for the multiples of 3 between 3 and 99. (Answer to Exercise 10.4 on page 103)

10.5. Write a program to count the number of lines in a Perl program. Skip blank lines and lines that only have a comment. (Answer to Exercise 10.5 on page 103)

10.6. Perl has a special token, `__END__`, which marks the end of a program when it shows up on a line by itself. Modify your answer to Exercise 10.5 so that you stop reading lines when you encounter that token. (Answer to Exercise 10.6 on page 104)

10.7. Using `redo` and a naked block, continually prompt a user to guess a secret number between 1 and 10. (Answer to Exercise 10.7 on page 104)

CHAPTER 11
Perl Modules

11.1. Using the `File::Spec` module, write a program that takes a directory path from the command line and adds it to every file in the current working directory to create an absolute path. You shouldn't have to hardcode any directory separator characters in your program. (Answer to Exercise 11.1 on page 107)

11.2. Write a program to take a list of file paths and break them into filename and path portions using the `File::Basename` module. Report the results in two columns. (Answer to Exercise 11.2 on page 108)

11.3. The `File::Basename` and `File::Spec` commands come with the Perl Standard library so you don't need to install them yourself. Now it's time to install a module from CPAN. Use the *cpan* command (which also comes with Perl) to install the modules you'll need for the upcoming exercises: `DBD::SQLite`, `LWP::Simple`, `XML::Twig`. You might need to adjust the configuration before you start so you can install these modules in your own directories. If you have problems with this, you might want to try the alternate method I show in the Answer to Exercise 11.4. (Answer to Exercise 11.3 on page 108)

11.4. The *cpan* command uses `CPAN.pm`, a full-featured CPAN client, behind the scenes. However, since it handles almost situation, it can also be a pain to configure. There's also the cpanminus, or *cpanm*, command that reduces all of that to the most common configuration so you don't deal with any configuration. Find the `App::cpanminus` distribution on CPAN, install it, and use it to install the modules mentioned in the Exercise 11.3. (Answer to Exercise 11.4 on page 109)

11.5. Use the `LWP::Simple` module to download and store the HTML source for *http://perldoc.perl.org/*. (Answer to Exercise 11.5 on page 109)

11.6. Use the `DBI` module with the `DBD::SQLite` driver to create a small test database for SQLite. From your Perl program, create a table named "Characters" with columns for ID, First Name, and Last Name. Once you create the table, insert the names of the Flintstones characters we mention in *Learning Perl*. This exercise requires you to al-

ready know some SQL and how to interact with a relational database, so I'm adding some extra challenge on this one. (Answer to Exercise 11.6 on page 109)

11.7. Use the `XML::Twig` module to process this XML snippet to remove the `<bowling_score>` element from this snippet of XML data:

```
<?xml version="1.0"?>
<characters>
    <character>
        <name>Fred Flintstone</name>
        <score>250</score>
        <league>Adult</league>
    </character>
    <character>
        <name>Barney Rubble</name>
        <score>230</score>
        <league>Adult</league>
    </character>
    <character>
        <name>Dino</name>
        <score>30</score>
        <league>Pets</league>
    </character>
</characters>
```

(Answer to Exercise 11.7 on page 111)

CHAPTER 12
File Tests

12.1. Go through all the files on the command line and list them in the output. Put a / after the names of directories, an * after executables, and an @ after symbolic links. (Answer to Exercise 12.1 on page 113)

12.2. Using the `stat` function, go through all of the files on the command line and list the modification time of each file. (Answer to Exercise 12.2 on page 114)

12.3. [Perl 5.10] Use the stacked file test operators to report the filenames from the command line that are readable, writeable, and executable. (Answer to Exercise 12.3 on page 115)

12.4. Write a program that takes two numbers from the command line, prints their binary representation then prints their logical-ANDed value. Your report should look something like this:

```
$ perl ex12.4.pl  4113 27
        1000000010001
&               11011
  -------------------
                10001
```

(Answer to Exercise 12.4 on page 115)

12.5. Rework your answer to Exercise 12.1 such that for each symbolic link, instead of printing a / after the name, it prints the link target. Use `readlink`, which we didn't cover in *Learning Perl*. You'll have to read the documentation for that function yourself. (Answer to Exercise 12.5 on page 116)

12.6. Rework your answer to Exercise 12.2 such that for each symbolic link, instead of using `stat` on the symbolic link, you use `lstat` instead. If you are on a system that doesn't have symbolic links, you're already done since `stat` and `lstat` will do the same thing. (Answer to Exercise 12.6 on page 116)

12.7. Use `File::Find` to collect some statistics about the "bushiness" of a directory structure. After you traverse the directory, report the mean number of directories, files, and links per directory. The `Statistics::Descriptive` module is useful for this sort of

thing. This is a challenging problem for the brave student, and you might have to disable strictures for this one. (Answer to Exercise 12.7 on page 117)

CHAPTER 13
Directory Operations

13.1. Write a program that reports the number of files in the current directory. Don't count special files . or .., but count the "hidden" files. (Answer to Exercise 13.1 on page 121)

13.2. Modify your answer to Exercise 13.1 to also report the sum of all the file sizes (but don't try to get the sizes of subdirectories). (Answer to Exercise 13.2 on page 122)

13.3. Write a program that looks in the current working directory and creates a columnar report of all files, their sizes, and whether they are readable, writeable, and executable. Use `opendir` and a `while` loop. Your report might look like this:

```
Name                                  Size   R W E
-------------------------------------------------
.DS_Store                             6148   x x x
.Trashes                                68     x x
birthdays.db                         16384   x x
cores                                   68   x x x
Desktop                                102   x x x
```

(Answer to Exercise 13.3 on page 122)

13.4. Modify your answer to Exercise 13.3 to use `glob` and a `while` loop. (Answer to Exercise 13.4 on page 123)

13.5. Write a program that looks in the current working directory then lists all of the Perl programs and their size. Use `opendir` to find the files. You'll have to figure out on your own how to recognize a Perl program. (Answer to Exercise 13.5 on page 124)

13.6. Rewrite the previous program to use `glob` and `foreach`. (Answer to Exercise 13.6 on page 125)

13.7. Use the `utime` function to set the access time of the files on the command line to five minutes ago. Modify the program to set the modification time of the files to the current system time. You may want to use your solution for Exercise 12.2 to check the results. (Answer to Exercise 13.7 on page 125)

CHAPTER 14
Strings and Sorting

14.1. Write a program to read lines of input and report the position of the first mention of "perl" on each line. Use `index` to find the substring. (Answer to Exercise 14.1 on page 127)

14.2. Write a program to read lines of input and report the position of the last "e" in the line. Use `rindex`. (Answer to Exercise 14.2 on page 128)

14.3. Write a program that takes a starting column value, an ending column, and a filename. Go through the lines of the file and print only the portions of each line from the start column to the end column. Use `substr` to get the right part of the string. (Answer to Exercise 14.3 on page 128)

14.4. Write a program to print a list of the files in the current directory, sorted in order of their ascending file size. Change the program to make it list the files in order of descending file size. (Answer to Exercise 14.4 on page 129)

14.5. Write a program to print a list of the files in the current directory, sorted in order of their last modification line. Put the most recently modified file at the beginning of the list. (Answer to Exercise 14.5 on page 130)

14.6. [Perl 5.12] Create two arrays, one for the first names and one for the last names, for the Flintstones characters such that the elements in each array align with each other. Go through only one of the arrays with `each` and use the array index to access the other array. For each character, output their first and last names on the same line, with every character's full name on a separate line. Here are two arrays to get you started:

```
my @first = qw(Fred Barney Betty Wilma Larry);
my @last = qw(Flintstone Rubble Rubble Flintstone Slate);
```

(Answer to Exercise 14.6 on page 130)

CHAPTER 15
Smart Matching and given-when

When we first wrote *Learning Perl, Fifth Edition*, the Perl developers had just added the smart match feature to Perl 5.10, and the book covered up to 5.10.0. After we published our book, however, the developers had to change the smart match behavior because it was fundamentally flawed. In the book, we told you that the smart match operator is commutative, meaning that it doesn't matter which order you use the arguments. That's different in Perl 5.10.1 and later. Now the order of arguments matter. We updated this for *Learning Perl, Sixth Edition*, but you should be careful about your Perl versions and their documentation.

Additionally, some of the rules about how the smart match operator recognizes numbers may have changed. However, we also tell you to look at the table of smart match precedence in the *perlsyn* documentation, and that's still the best advice since that will always have the right answer for your version of Perl.

15.1. [Perl 5.10.1] Use the smart match operator in a program that prompts the user for a string and reports if that string is an element of an array you specify in your program. You only have to match exact elements.

If you can't think of a list of values to put into your array, use the *Flintstones* characters:

```
my @array = qw(Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm);
```

It might help you to remind yourself which values you have in your array. Here are some sample runs:

```
$ perl ex15.1.pl
The elements are (Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm)
Enter a name:
Fred
I found a matching name

$ perl ex15.1.pl
The elements are (Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm)
Enter a name:
Dino
I didn't find a matching name
```

(Answer to Exercise 15.1 on page 131)

15.2. [Perl 5.10.1] Modify your answer to Exercise 15.1 to prompt the user to enter several names, one line per name. Report success if at least one of the names is a key in a hash that you define in your program.

If you can't think of anything to put into your hash, use the *Flintstones* characters:

```
my %hash = qw(
    Fred      Flintstone
    Wilma     Flintstone
    Barney    Rubble
    Betty     Rubble
    Larry     Slate
    Pebbles   Flintstone
    Bamm-Bamm Rubble
    );
```

Here are some sample runs:

```
$ perl ex15.2.pl
The keys are [Pebbles Betty Wilma Barney Bamm-Bamm Larry Fred]
Enter some names then Control-D to stop:
Fred
Wilma
Foo
I found a matching name

$ perl ex15.2.pl
The keys are [Pebbles Betty Wilma Barney Bamm-Bamm Larry Fred]
Enter some names then Control-D to stop:
Joe
Buster
Mimi
I didn't find a matching name
```

(Answer to Exercise 15.2 on page 131)

15.3. [Perl 5.10.1] Write a program that prompts the user for a regular expression pattern and reports if any elements of an array match that pattern. Use the smart match operator to check if that pattern matches. You don't need to report which elements match. As a bonus, what do you do if the user enters an invalid regular expression (Hint: this actually shows up in Chapter 17).

If you can't think of a list of values to put into your array, use the *Flintstones* characters:

```
my @array = qw(Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm);
```

It might help you to remind yourself which value you have in your array. Here are some sample runs:

```
$ perl ex15.3.pl
The elements are (Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm)
Enter a pattern:
r.*y
At least one element matches
```

```
$ perl ex15.3.pl
The elements are (Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm)
Enter a pattern:
tt
At least one element matches

$ perl ex15.3.pl
The elements are (Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm)
Enter a pattern:
bb.*s
At least one element matches
```

(Answer to Exercise 15.3 on page 132)

15.4. [Perl 5.10.1] Write a program that prompts you for two lists of words. Use the smart match operator to report if these two lists are the same or not. To make things easier, enter each list on a single line then `split` that line on whitespace. Here are some sample runs:

```
$ perl ex15.4.pl
Enter the first list on a single line:
dog cat bird
Enter the second list on a single line:
bird cat dog
The lists are not the same

$ perl two_lists.pl
Enter the first list on a single line:
dog cat bird
Enter the second list on a single line:
dog cat bird
The lists are the same
```

As a bonus, can you explain why this program appears not to work (your answer might not have this issue):

```
$ perl ex15.4.pl
Enter the first list on a single line:
dog cat bird
Enter the second list on a single line:
 dog cat bird
The lists are not the same
```

(Answer to Exercise 15.4 on page 133)

15.5. [Perl 5.10.1] Use `given-when` and smart matching to implement a word guessing game. If the user enters a string that exactly matches the secret word, report success. If the string doesn't match, try that string as a regular expression pattern. If the user enters **`give up`**, stop the program (and, if you want to be nice, tell them the secret word).

Here is a sample run:

```
$ perl ex15.5.pl
Enter a string or pattern> Fred
[Fred] didn't help at all
```

```
Enter another string or pattern> Bar
The secret word matches the pattern [Bar]

Enter another string or pattern> ^.{5}$
[^.{5}$] didn't help at all

Enter another string or pattern> ^.{6}$
The secret word matches the pattern [^.{6}$]

Enter another string or pattern> ^Bar
The secret word matches the pattern [^Bar]

Enter another string or pattern> ney$
The secret word matches the pattern [ney$]

Enter another string or pattern> Barney
You guessed the secret word!
```

(Answer to Exercise 15.5 on page 134)

CHAPTER 16
Process Management

This is one of the odd chapters in *Learning Perl* because we've spent most of the book showing you how to write Perl that works on any of the systems that can run Perl. However, once you want to interact with external programs, you have to follow its rules for how things work.

Since Perl has its roots in the Unix toolbox and process model, you can still see vestiges of that. Many people have worked very hard to make this part of Perl as compatible as possible, but if you are using a non-Unix system, some of these exercises might not work for you. As a salve, I've also added some exercises that won't work on Unix so you can test your skills on Windows too.

16.1. Write a program that changes the time zone by using the `TZ` environment variable then uses `exec` to run the date command. Get the valid time zones from your system. (Answer to Exercise 16.1 on page 137)

16.2. Use the backticks operator to read the output of *ls –l* command then report which users and groups it finds. Run it in the directory that contains the user home directories. If you are on Windows, you probably don't have the notion of users and groups, so you can skip this one. (Answer to Exercise 16.2 on page 137)

16.3. Rewrite the program from the previous exercise, but use `IPC::System::Simple` instead of backticks. (Answer to Exercise 16.3 on page 138)

16.4. Write a program to print all of the environment variables, in alphabetical order, along with their values. Can you turn this into a CGI program? (Answer to Exercise 16.4 on page 139)

16.5. Modify your program from Exercise 16.1 to remove the value from `PATH` environment variable. What other changes do you have to make as a result? (Answer to Exercise 16.5 on page 139)

16.6. Modify your answer to Exercise 16.2 to execute the command using `open` and a pipe. Read the input one line at a time through a filehandle and report the same results. (Answer to Exercise 16.6 on page 140)

16.7. Write a program that reads in lines of input and counts the number of lines with of e's in them. Use your program from Exercise 16.4 as the source of input. (Answer to Exercise 16.7 on page 141)

16.8. Use the Windows *title* command to create a "progress bar" in the title bar of the *cmd* or *command* window. Add a * character once every second for a minute. If you aren't on Windows, you might be able to do this, but we don't know how to tell you to do it. (Answer to Exercise 16.8 on page 141)

CHAPTER 17
Some Advanced Perl Techniques

17.1. Write a program that repeatedly asks the user for a string and a regular expression then reports if the string matches the regular expression. What happens if you give your program an invalid regular expression? How can you fix that? If you can't think of an invalid regular expression, try one with an unmatched `(` or `[`. (Answer to Exercise 17.1 on page 143)

17.2. Rewrite your answer to Exercise 17.1 to use `Try::Tiny` instead of `eval` (or, if you used `Try::Tiny` in the previous exercise, try it with `eval`). (Answer to Exercise 17.2 on page 144)

17.3. Use the `grep` function to select the multiples of 3 between 1 and 1,000. (Answer to Exercise 17.3 on page 144)

17.4. Use the `map` function to create a list of the squares of the numbers from 1 to 10. (Answer to Exercise 17.4 on page 145)

17.5. Given this single string (you have to store the entire thing in one scalar even though it looks like multiple lines), transform it so that each of the names at the beginning of each line have an initial capital letter, and the rest of the name is in lowercase letters:

```
fRED has score 230
barney has score 190
DINO has score 30
```

Your program should report this output:

```
Fred has score 230
Barney has score 190
Dino has score 30
```

(Answer to Exercise 17.5 on page 145)

17.6. Given an array whose elements are the numbers from 0 to 1,000, use an array slice to a) select the first and last numbers (in that order), b) the last and first numbers (in that order), c) the first ten numbers, and d) the odd numbers. (Answer to Exercise 17.6 on page 145)

17.7. Given the start of a program that looks up the characters' bowling scores, add a line to create the hash using a hash slice. The names are the keys, and the values are their scores. Once you've done that, change the program to create the hash using a `map`. Replace the `...` (the "do what I mean" ellipses operator, valid in Perl 5.12) in this program with the right Perl expression:

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

my @names  = qw(Fred Barney Dino);
my @scores = qw(230   190    30 );

# CREATE THE HASH HERE...
...;

foreach my $name ( sort keys %scores ) {
    print "$name has score $scores{$name}\n";
    }
```

(Answer to Exercise 17.7 on page 146)

17.8. Add up the numbers from 1 to 1,000. Do it using the `sum` subroutine from `List::Util`, then redo it by using `reduce` from the same module. (Answer to Exercise 17.8 on page 147)

17.9. Use the `pairwise` subroutine from `List::MoreUtils` to add the corresponding elements in two arrays. For example, you have these two arrays:

```
my @m = ( 1, 2, 3 );
my @n = ( 4, 6, 8 );
```

You should create a third array with the sums of the two first elements, the two second elements, and so on:

```
my @o = ( 5, 8, 11 );
```

(Answer to Exercise 17.9 on page 148)

CHAPTER 18
Databases

Learning Perl, Third Edition was the last edition to contain the "Databases" chapter. Although this chapter is not in the current edition of *Learning Perl*, I leave it in this book as a bonus chapter in case you want to use the third edition or just want to practice some extra Perl.

As you might have read in *Programming Perl*, Perl's DBM support depends on the libraries that the *perl* binary links to. There are several implementations of the one thing we call "DBM", but as long as you stick with the same *perl*, you should be fine.

18.1. In the exercises for Chapter 6, you created a hash that contained names and birthdays. Write a program that creates the same hash, but as a DBM file using `dbmopen`. Write a second, separate program that reads the hash and reports the results. (Answer to Exercise 18.1 on page 149)

18.2. Using the same DBM file from your answer to Answer to Exercise 18.1, write a third program to add entries to the hash then run your hash reporter program again. You should see the new entries in the output. (Answer to Exercise 18.2 on page 150)

18.3. The `pack` function is one of the few places in Perl where the underlying architecture shows through. Pack the unsigned integer `0x12_34_56_78` using the `N` (network order), `V` (VAX order), and `I` (native integer), and then `unpack` each of those packed strings with each format (e.g. `unpack` the `V` packed string with the `V`, `N`, and `I` formats). Do you always get the right answer back? (Answer to Exercise 18.3 on page 151)

18.4. Write a program to report if the architecture you use is big or little endian. Your answer to Exercise 18.3 may give you a clue. (Answer to Exercise 18.4 on page 152)

18.5. Create a file that you won't mind losing. You probably want to create a special directory just for this exercise so you don't modify files you don't mean to. Using *perl*'s `-i` command-line switch, change all of the lowercase e's in the file to uppercase E's. (Answer to Exercise 18.5 on page 152)

PART II

Answers

APPENDIX A
Answers to Chapter 1 Exercises

Answer 1.1: This exercise doesn't really have an answer. Simply run the command:

```
$ perldoc perl
$ perldoc perlfunc
```

Answer 1.2: The trick is to run *perldoc* on itself:

```
$ perldoc perldoc
```

You can also simply run *perldoc* without giving it anything to read and it will print a summary of its use (which I've slightly edited and displayed here):

```
perldoc [options] PageName|ModuleName|ProgramName...
perldoc [options] -f BuiltinFunction
perldoc [options] -q FAQRegex

Options:
    -h   Display this help message

           [Perldoc v3.14_04]
```

It's the `-f` switch that you want to look up documentation for the `print` function, which you already saw in this chapter:

```
$ perldoc -f print
```

Answer 1.3:

```
$ perldoc -q comma
```

The command includes in its output (at the time of this writing) at least a section from *perlfaq5*, which shows you a couple of ways to add commas to numbers. Here's some of the output:

```
=head2 How can I output my numbers with commas added?

This one will do it for you:

    sub commify {
        local $_  = shift;
        1 while s/^(-?\d+)(\d{3})/$1,$2/;
```

```
    return $_;
}
```

Answer 1.4: Take the example program from page 17 of *Learning Perl* and insert your own command in the backticks. You can remove the line starting with `s/` to substitute your own. In this example, I've chosen the *ls* command since I'm using a Unix shell:

```
#!/usr/bin/perl
@lines = `ls`;
foreach (@lines) {
    print;
    }
```

My output is a bit different from *ls* on the command line because it doesn't create columns:

```
$ perl scratch/ex1.4.pl
bin
book.xml
bookinfo.xml
ch00.xml
ch01.xml
ch02.xml
ch03.xml
```

On Windows, the *dir* command should do the same sort of thing:

```
#!C:/strawberry/perl/bin/perl
@lines = `dir`;
foreach (@lines) {
    print;
    }
```

Now your output is a bit different when you run it from a *command* window using Cygwin's *perl*:

```
C:\>perl dir.pl
Directory of C:\CYGWIN\HOME\OWNER

08/18/2010  01:56 PM    <DIR>          .
08/18/2010  01:56 PM    <DIR>          ..
08/18/2010  12:47 PM             3,754 .bashrc
08/18/2010  12:47 PM             1,150 .bash_profile
08/18/2010  12:47 PM             1,461 .inputrc
08/18/2010  01:56 PM                 0 ch00.xml
08/18/2010  01:56 PM                 0 ch01.xml
08/18/2010  01:56 PM                82 dir.pl
               6 File(s)          6,447 bytes
               2 Dir(s)  16,110,485,504 bytes free
```

Answer 1.5: If you have to, have a recent version (3.15 or later) of the *perldoc* to use the `-v` switch to look up Perl special variables. You can use the `-V` switch to see which version you have (if you didn't do that already, keep reading to see how you could have discovered it yourself):

```
$ perldoc -V
Perldoc v3.15_02, under perl v5.012001 for darwin
```

Perl 5.12 and later comes with this version, but if you don't have a recent version, you can install the `Pod::Perldoc` module to get the latest version of *perldoc*. We don't expect you to do this for this exercise, though (we cover it in Chapter 11 in *Learning Perl*):

```
$ cpan Pod::Perldoc
```

When you have a current version, you can use *perldoc* on itself to find the command-line switch you need. At the end of this extract from that output, you see the `-v` command-line switch:

```
$ perldoc perldoc
NAME
       perldoc - Look up Perl documentation in Pod format.

SYNOPSIS
       perldoc [-h] [-D] [-t] [-u] [-m] [-l] [-F] [-i] [-V] [-T] [-r]
       [-ddestination_file] [-oformatname] [-MFormatterClassName]
       [-wformatteroption:value] [-nnroff-replacement] [-X] [-L lang]
       PageName|ModuleName|ProgramName

       perldoc -f BuiltinFunction

       perldoc -L it -f BuiltinFunction

       perldoc -q FAQ Keyword

       perldoc -L fr -q FAQ Keyword

       perldoc -v PerlVariable
```

Now that you know to use `-v`, you have one more issue to overcome. The characters in Perl special variable names are often special to the shell too (indeed, why do you think Perl has such weird variable names?). You have to quote the variable name so the shell does not interpret those characters. On Unix-like shells (bash and so on, even if you're using them on non-Unix-like systems), you need to use single quotes:

```
$ perldoc -v '$_'
$ARG
$_     The default input and pattern-searching space.  The following
       pairs are equivalent:

       while (<>) {...}    # equivalent only in while!
       while (defined($_ = <>)) {...}
```

From the Windows *command* window, you don't have to worry about the Unix shell escapes. If you're using Strawberry Perl:

```
C:\> perldoc -v $_
$ARG
$_     The default input and pattern-searching space.  The following
       pairs are equivalent:
```

```
while (<>) {...}    # equivalent only in while!
while (defined($_ = <>)) {...}
```

APPENDIX B
Answers to Chapter 2 Exercises

Answer 2.1: Although the exact warning message differs between versions of Perl, each of the programs should output a warning. To turn these into a full program, insert the given lines into a short program such as this one then run the program to see the warning:

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

Program 1 uses the `print` function, but doesn't print anything. You should get a warning that complains about an uninitialized value. The warning message also tells you the program name (I called mine *test*) and the line number of the warning:

```
Use of uninitialized value in print at test line 3.
```

Program 2 adds two numbers, but doesn't do anything with the result. That's a pretty silly thing. Why do the work if you don't use the result for anything?

```
Useless use of a constant in void context at test line 3.
```

Program 3 adds a number to a variable that has no value then prints the result. The warning message complains about the uninitialized value in the addition. You should also get the result of the `print` too; that's the `1` after the warning:

```
Use of uninitialized value in addition (+) at test line 3.
1
```

Answer 2.2: You can use the same program you created for the Answer to Exercise 2.1. Program 1 complained about an uninitialized value. With `diagnostics` on, you get the same warning, but also the entry from the *perldiag* for that warning. The longer warning tries to explain what you might have done to cause the warning. The entry may not always talk about your exact error, but it's usually close. Notice that the start of the longer warning has a `W`, meaning it's a warning (rather than a fatal error or something else), and that the class of the warning is `uninitialized`:

```
Use of uninitialized value in print at test line 3 (#1)
    (W uninitialized) An undefined value was used as if it were already
    defined.  It was interpreted as a "" or a 0, but maybe it was a mistake.
```

```
    To suppress this warning assign a defined value to your variables.

    To help you figure out what was undefined, perl tells you what operation
    you used the undefined value in.  Note, however, that perl optimizes your
    program and the operation displayed in the warning may not necessarily
    appear literally in your program.  For example, "that $foo" is
    usually optimized into "that " . $foo, and the warning will refer to
    the concatenation (.) operator, even though there is no . in your
    program.
```

In program 2, you added two numbers but discarded the result. Here's part of the warning you should get:

```
Useless use of a constant in void context at test line 3 (#1)
    (W void) You did something without a side effect in a context that does
    nothing with the return value, such as a statement that doesn't return a
    value from a block, or the left side of a scalar comma operator.  Very
    often this points not to stupidity on your part, but a failure of Perl
    to parse your program the way you thought it would.
```

In program 3, you added 1 to an undefined value, which causes a warning. The warning is the same as the first program, even though a different expression triggers it.

Answer 2.3: You can use the modulus operator, `%`, to determine if a number is odd or even. Odd numbers are divisible by 2 with remainder 1, so any odd number modulo 2 returns true, and that's the first part of your `if-else` structure. You have most of the output message outside of the `if-else` because you only need that to decide if you should use "odd" or "even", so you shouldn't have to type the entire message twice:

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

print "Enter a number: ";
chomp( $number =  <STDIN> );
print "The number $number is ";
if( $number % 2 ) { # true if the number is odd
    print "odd\n";
} else {
    print "even\n";
    }
```

If you give this non-numeric input, say **one**, you get a warning since you try to use a non-number with a mathematical operator. Not only that, since the string one converts to the number `0` your program reports that the "number" is even:

```
$ perl ex2.3.pl
Enter a number: one
Argument "one" isn't numeric in modulus (%) at ex2.3.pl line 6, <STDIN> line 1.
The number one is even
```

Although you haven't read Chapter 11 yet, once you know about Perl modules you can use the `looks_like_number` function from `Scalar::Util` to test a string to ensure it is a number.

Answer 2.4: The program would still work, and you don't get a warning. When you read the input, you have a string in `$number`. When you use that string in a mathematical operation, Perl converts the string to a number, and stops the conversion when it runs into a character that it can't translate into a number. So Perl just ignores the newline when it converts the string into a number.

The program's output is slightly different though, since `$number` still has a newline, so the `print` statement actually outputs two lines. The string form of the value in `$number` still has the newline even though Perl ignores it for numerical operations:

```
Enter a number: 2
The number 2
 is even
```

Answer 2.5: The first thing Perl looks at is precedence. In the first example, `2 ** 3 ** 4`, the two operators are the same, so they have the same precedence and Perl uses associativity to decide in which order to compute the answer. The complete details are in the *perlop* documentation. In this case it matters: `(2 ** 3) ** 4` is 4096, but `2 ** ( 3 ** 4 )` is 2.41785163922926e+24. The exponentiation operator is right-associative, so it does the operation on the right first. That's the `3 ** 4`. You can also write this as `2 ** ( 3 ** 4 )`.

In the second example, the division and multiplication operators have the same precedence, but both operators are left-associative. The first operation is the division, so the `2/3` happens first then Perl multiplies the result by 4 to get 2.66666666666667. I can also write this as `( 2 / 3 ) * 4`.

In the third example, you mixed operators of different precedences. You don't need to worry about associativity since you don't have operators of the same precedence next to each other. You can also write this as `2 + (3 * (4 ** 5)) - 6`.

If you can't remember the precedence or associativity, or you used to remember but have forgotten temporarily, or even if you think that your programmer teammates might be confused, you can always use the parentheses, which have the highest precedence, to denote the order of operations you intend.

Answer 2.6: Prompt for numbers as you did before, and chomp the input then compare the two numbers. If the first one, `$m`, is less than the second one, `$n`, output a message saying that it is smaller. If `$m` is larger, output a message saying that. Now, if neither of those is true, they must be equal, output a message that says that:

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

print "Enter a number: ";
chomp( $m = <STDIN> );

print "Enter another number: ";
chomp( $n = <STDIN> );

if(    $m < $n ) { print "$m is smaller than $n\n" }
```

```
elsif( $m > $n ) { print "$m is larger than $n\n"  }
else             { print "$m and $n are equal\n"   }
```

If you enter something that isn't digits, like **`thirty-seven`**, Perl tries to convert that to a number, and since it doesn't recognize anything, it uses the value 0 (that's a zero). Here are a few runs of your script:

```
$ perl ex2.6.pl
Enter a number: 5
Enter another number: 4
5 is larger than 4

$ perl ex2.6.pl
Enter a number: thirty-seven
Enter another number: 0
Argument "thirty-seven" isn't numeric in numeric lt (<)
at larger line 10, <STDIN> line 2.
thirty-seven and 0 are equal

$ perl ex2.6.pl
Enter a number: 5
Enter another number: 6
5 is smaller than 6
```

Answer 2.7: Here's one way to do it. Prompt for the first number then get the input, which you store in `$n`. Add `$n` to `$sum`, which doesn't exist until the first time you use and magically springs into existence without a value (so it's undefined). When you add `$n` to the undefined value, Perl converts `$sum` to 0, adds `$n` to it, and stores the result back in `$sum`. Having done that you're ready for the next number and prompt the user again:

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

print "Please enter the first number: ";

while( $n = <STDIN> ) {
    $sum += $n;
    print "Please enter another number: ";
    }
print "The sum is $sum\n";
```

The only way you can (legally) stop this program is to break out of the `while` loop. If the result of `<STDIN>` is false, the `while` loop ends. It can only be false by being undefined since it can't be the empty string (there will always be at least a newline), `0` (for the same reason: it has a trailing newline so it's a string), or the string `'0'`. The only way it can be undefined is at the end-of-input. You could write this to check for a lone newline in the input, but you've already seen that on page 39 of *Learning Perl*.

Here's a sample run of your program:

```
$ perl larger
Please enter the first number: 37
Please enter another number: 5
```

```
Please enter another number: 9
Please enter another number: 137
Please enter another number: 6
Please enter another number: The sum is 194
```

Answer 2.8: You can follow the examples in the chapter to get input from the user (page 39) and combine that with the examples to make characters by code point (page 34):

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

print 'Please enter the first code point: ';

while( $code_point = <STDIN> ) {
    chomp( $code_point );
    print 'The character is ', chr( hex( $code_point ) ), "\n";
    print 'Please enter another code point: ';
    }
```

Depending on how you've set up your environment, you might have to ensure that your `perl` sets up standard output correctly for wide characters. The `-CS` switch can do that:

```
$ perl -CS code_point.pl
Please enter the first code point: 26F3
The character is ⛳
Please enter another code point: 267A
The character is ♺
Please enter another code point: 2126
The character is Ω
Please enter another code point: 03B6
The character is ζ
Please enter another code point:^D
```

Now you can see that those characters are a flag in a hole, a recycling sign, the ohm symbol, and the Greek letter zeta.

If you had trouble with the Unicode aspects of this exercise, such as a "wide character in print" warning, you should review Appendix C of *Learning Perl*.

APPENDIX C
Answers to Chapter 3 Exercises

Answer 3.1: This program is a lot like the first exercise on page 60 of *Learning Perl*. Instead of printing all of the lines in reverse order, you only have to print one line. You don't know ahead of time how many elements the array will hold, so you can't use the exact index of the element that you want. You do know that `$#array_name` represents the last index of an array, so the second-to-last must be one less than that:

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

@lines = <STDIN>;
print "The second to last line is: $lines[$#lines-1]";
```

That one has a subtle special case though. When there is only one element in the array, the last index (the value in `$#lines`) is `0`, so `0 - 1` is `-1`, and instead of accessing a specific index, the `-1` makes Perl count from the other side of the array. The element at index `-1` is the last element, not the second-to-last element. You have to do this in a slightly different way:

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

@lines = <STDIN>;
if( $#lines > 0 ) {
    print "The second-to-last line is: $lines[$#lines-1]\n";
    }
else {
    print "There aren't enough lines for a second-to-last one!\n"
    }
```

You don't have to do so much typing though, because you can index arrays from the end. If you use a negative index, you can start with the last element and work your way backward. Thus, the index `-1` is the last element, and `-2` is the next-to-last element:

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

@lines = <STDIN>;
```

```
if( $#lines > 0 ) {
    print "The second-to-last line is: $lines[-2]\n";
    }
else {
    print "There aren't enough lines for a second-to-last one!\n"
    }
```

Answer 3.2: There is a more complicated way to do this, too. In your programs for the Answer to Exercise 3.1, you store the entire array in memory. I hope you didn't redirect a 1 TB log file into one of those programs! Since you need to remember only the second-to-last line, you only need to keep two lines in memory: the previous line, in case the current one is the last line, and the current one in case there are more lines (in which case it becomes the previous one). You start by reading in one line and assigning it to `$last_line`. In the `while` loop, if you can read another line of input, you assign that to `$next_line` then update the values of `$second_to_last` and `$last` inside the loop block. When there are no more lines to read, `$next_line` gets `undef` and the loop terminates. After that, you just have to print what is in `$second_to_last`:

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

$last_line = <STDIN>;
while( defined( $next_line = <STDIN> ) ) {
    ( $second_to_last, $last_line ) = ( $last_line, $next_line );
    }
print "The second to last line is: $second_to_last";
```

Answer 3.3: When you interpolate an array in a double-quoted string, Perl joins all of the array elements with spaces and puts that string into the array's place. This magic actually uses the `$"` special variable mentioned in the footnote on page 51 of *Learning Perl*. That variable is just a scalar, so you can assign to it any value you like. In this case, you can just assign a comma to `$"`. You interpolate the array in the double-quoted string and add a trailing newline for good measure:

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

@numbers = ( 1 .. 10 );
$" = ",";
print "@numbers\n";
```

When you get to Chapter 9, you'll see that `join` is a better way to do this:

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

@numbers = ( 1 .. 10 );
print join( ' ', @numbers ), "\n";
```

Answer 3.4: You can create a list of numbers with the range operator, `..`, and then create their squares and cubes with the exponentiation operator, **. You give `print` the list of things to output, including a space between each of the numbers:

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

foreach( 1..10 ) {
    print "$_: ", $_**2, " ", $_**3, "\n";
    }
```

Answer 3.5: You can modify your program from the Answer to Exercise 3.4. In this case, you create the array `@numbers` then go through that list one element at the time. You use the `shift` operator to remove the first element from the list then you assign that element to `$_`. Now the list has one less element, and the next time through the loop you get the next number, and so on until you have no more elements left in `@numbers`:

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

@nunbers = ( 1..10 );
while( $_ = shift @numbers ) {
    print "$_: ", $_**2, " ", $_**3, "\n";
    }
```

Answer 3.6: The range operator works for strings just like it does for numbers, although it might seem a little weird at first. To get all the lowercase letters, you use `'a'` on the left side and `'z'` on the right side:

```
@lowercase = ( 'a' .. 'z' );
```

To get all of the uppercase letters, I just use the uppercase versions:

```
@uppercase = ( 'A' .. 'Z' );
```

The third part is tricky. You can't just use `'a' .. 'Z'`. Did you try that? What happened? The range operator actually uses some magic to work with letters. If you start with a lowercase letter, you can only get lowercase letters, and if you start with an uppercase letter, you can only get uppercase letters. You can't do it with one range operator, but that's okay because you can use as many as you like without having to pay extra:

```
@letters = ( 'A' .. 'Z', 'a' .. 'z' );
```

This range operator can be a little bit odd when you first try it. What do you think this range produces?

```
@range = ( 'a1' .. 'z1' );
```

Try these ranges, too:

```
@range = ( 'a10' .. 'b10' );

@range = ( 'a1' .. 'b10' );
```

Don't keep yourself up all night trying to figure out the pattern Perl is guessing. You might want to use this to impress your friends, but don't rely on it for important algorithms.

APPENDIX D
Answers to Chapter 4 Exercises

In this chapter in *Learning Perl*, we introduced the `strict` pragma that enforces some good programming practices. For the rest of this workbook, I'm going to make "strict-clean" programs and I'll use the `strict` pragma at the top of my programs.

Remember, if you're using Perl 5.12 or later and you specify that version in your program, you'll automatically turn on `strict`.

Answer 4.1: The arguments to a subroutine are in the `@_` array, so you only need to interpolate that array in a double-quoted string:

```
#!/usr/bin/perl

show_args( 'fred', 'barney', 'betty' );

sub show_args {
    print "The arguments are @_\n";
    }
```

The output is just as I specified in the exercise:

```
$ perl ex4.1.pl
The arguments are fred barney betty
```

Usually, when you have to do this sort of thing, you put some characters around the variable that you are interpolating so you can separate it from the rest of the string. In this example, the stuff in the square brackets are the elements of `@_`:

```
#!/usr/bin/perl

show_args( 'fred', 'barney', 'betty' );

sub show_args {
    print "The arguments are [@_]\n";
    }
```

Now you can separate the arguments from the rest of the string:

```
$ perl ex4.1.pl
The arguments are [fred barney betty]
```

If you had leading or trailing whitespace in your arguments, you'd see them in the brackets. Try it with these arguments that add extra spaces:

```
show_args( ' fred', 'barney', 'betty ' );
```

The output shows the extra spaces in the brackets:

```
$ perl ex4.1.pl
The arguments are [ fred barney betty ]
```

Often, this technique is still insufficient because some of the interior elements may have extra spaces. Remember that `$"` special variable from the Answer to Exercise 3.3? You can use that here to great effect. You make it show a closing and opening square bracket around each element:

```
#!/usr/bin/perl

show_args( ' fred', ' barney', ' betty ' );

sub show_args {
    $" = "][";
    print "The arguments are [@_]\n";
    }
```

Now your output shows which whitespace belongs to which argument:

```
$ perl ex4.1.pl
The arguments are [ fred][ barney][ betty ]
```

When you get some more Perl under your belt, you'll start to use the `Data::Dumper` module to create this sort of output for you.

Answer 4.2: The `show_args_again` subroutine is similar to `show_args` subroutine from the Answer to Exercise 4.1, but it calls `show_args` in different ways. The first way prefaces the subroutine name with the `&`:

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

show_args_again( qw( fred barney betty ) );

sub show_args {
    print "The arguments are [@_]\n";
    }

sub show_args_again {
    print "The arguments are [@_]\n";
    &show_args;
    }
```

The output shows the same line twice. One line is from `&show_args`, and one is from `show_args_again`. The magic of `&` passes the value of `@_` to the second subroutine, so you don't have to specify an argument list:

```
The arguments are [fred barney betty]
The arguments are [fred barney betty]
```

With a slight change to `show_args_again`, you see different behavior:

```
sub show_args_again {
    print "The arguments are [@_]\n";
    show_args;
    }
```

This time, the arguments do not magically pass on to `show_args` because there is not an ampersand. The results can also vary based on which order you define the subroutines, since Perl has to guess what `show_args` actually means. When you have already defined `show_args`, Perl already knows it is a subroutine, so you see output from both subroutines:

```
The arguments are [fred barney betty]
The arguments are []
```

What happens if you define `show_args_again` first? I'll let you figure that one out. You may want to read the *perlsub* documentation, where you'll find all the details.

Answer 4.3: I start with the same program you had in the Answer to Exercise 4.2, so I'll only include the `&show_args_again` subroutine in this answer. In the first part, you want to use the `&` and the `()` with the subroutine name:

```
sub show_args_again {
    print "The arguments are [@_]\n";
    &show_args();
    }
```

The output is like the last part of your program in the Answer to Exercise 4.2: the value of `@_` is not passed to `&show_args`. The parentheses explicitly define the argument list, and you don't give it any arguments. The second subroutine doesn't have anything to interpolate into the string:

```
The arguments are [fred barney betty]
The arguments are []
```

In the second part, you want to lose the `&` and just use the `()`:

```
sub show_args_again {
    print "The arguments are [@_]\n";
    show_args();
    }
```

You get the same output for the same reason. The parentheses state the explicit value of the argument list, and the empty parentheses pass along no arguments:

```
The arguments are [fred barney betty]
The arguments are []
```

Answer 4.4: If you want `show_args` to print the original argument list in reverse order, you just need to give it the argument list in reverse order. You can flip around a list

with the built-in `reverse` function. Now your `show_args_again` subroutine looks like this:

```
sub show_args_again {
    print "The arguments are [@_]\n";
    show_args( reverse @_ );
    }
```

The output now has arguments in each subroutine's output:

```
The arguments are [fred barney betty]
The arguments are [betty barney fred]
```

In general, I like to use the `()` whenever I call a subroutine so I explicitly know which arguments I meant to pass to it, and I only use the `&` (with the `()`) when I'm doing something where I think I'm being clever. You've now seen the different ways to call a subroutine and can make up your mind on what you'd like to use. Good luck!

Answer 4.5: After you went through the Answer to Exercise 4.4 and made all that fuss about not using `&` in front of subroutines, you can do it here. In the `do_it` subroutine, you use an `if-elsif-else` construct to decide which second subroutine to call. You can use `shift` to remove the first element in the argument list (the default array `shift` uses if you don't say otherwise), so `@_` should contain just the numbers, and those are the things you want to pass on to the subroutines that will actually do the work. You use that first argument to decide which subroutine to call. If you get an argument you don't support, the final branch in the `if-elsif-else` prints a warning and returns nothing (but not the result of `print`!).

You want the arguments to `add` or `multiply` to be whatever is left in `@_`, so you preface their names with the `&` and leave off the parentheses when you call them. In this case, using the same `@_` is your intent.

The `add` and `multiply` subroutines are almost the same except for one character. They both look like accumulators. In the `add` subroutine, you can rely on the default starting value of `$n` to do the right thing since `0` is the additive identity. However, multiplying any number by `0` makes it `0`, so you start off `multiply` by setting `$n` to `1`, which is the multiplicative identity:

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

print "The sum is ",     do_it( "+", 1..10 ), "\n";
print "The product is ", do_it( "*", 1..10 ), "\n";

sub do_it {
    my $op = shift;

    if(    $op eq "+" ) { &add }
    elsif( $op eq "*" ) { &multiply }
    else                { print "I don't know $op!"; return }
    }
```

```
sub add      { my $n = 0; foreach ( @_ ) { $n += $_ }; $n }

sub multiply { my $n = 1; foreach ( @_ ) { $n *= $_ }; $n }
```

If you don't want to use the magic feature of `&`, you just have to pass the remaining elements of `@_` to `add` or `multiply` yourself:

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

print "The sum is ",     do_it( "+", 1..10 ), "\n";
print "The product is ", do_it( "*", 1..10 ), "\n";

sub do_it {
    my $op = shift;

    if(    $op eq "+" ) { add( @_ ) }
    elsif( $op eq "*" ) { multiply( @_ ) }
    else                { print "I don't know $op!"; return }
    }

sub add      { my $n = 0; foreach ( @_ ) { $n += $_ }; $n }

sub multiply { my $n = 1; foreach ( @_ ) { $n *= $_ }; $n }
```

Answer 4.6: You can solve this problem in several ways, and as with other solutions you have seen, you may have come up with something different than what you read here. In this case, you want to do different things for different numbers of arguments, so you use the `@_` array to figure out which thing to do. In scalar context, an array (just an array, not a list or anything else!) is the number of elements in the array. When you use it in a numerical comparison with `==` (and in the `if` condition, which is also a scalar context), `@_` is the number of elements.

When `@_` has one thing, you run the first `if` branch, which simply interpolates the array of one element into a string. You could have simply returned `$_[0]`, the first (and only) element of `@_`, too.

When `@_` has two things, you interpolate each thing in the double-quoted string through the single element accesses `$_[0]` and `$_[1]` with an `" and "` between them.

The hard part is the rest of the problem. The previous two cases were special. You can be a bit clever though: you only separate the last element with the `" and "`, so you take that off of the list right away and assign it to `$last`. Now you need to separate the rest of the elements with commas. You've already seen a trick in the program in the Answer to Exercise 3.3: use the `$"` special variable. When you interpolate `@_` in the double-quoted string, you get all but the last element. You add the last (serial) comma and `" and "` yourself before you interpolate the last value in `$last`:

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

```
print separate( qw(fred) ), "\n";
print separate( qw(fred barney) ), "\n";
print separate( qw(fred barney betty) ), "\n";
print separate( qw(fred barney betty wilma) ), "\n";

sub separate {
    if(    @_ == 1 ) { return "@_" }
    elsif( @_ == 2 ) { return "$_[0] and $_[1]" }
    elsif( @_ >  2 ) {
         my $last = pop;
         local $" = ", ";
         return "@_, and $last";
         }
    }
```

You get the lists separated like you would if you typed them yourself:

```
fred
fred and barney
fred, barney, and betty
fred, barney, betty, and wilma
```

What happens if there is nothing in the argument list, which is the only case you don't explicitly handle? As with any other subroutine, the last evaluated expression is the return value, and since you don't evaluate any expression in that case, you get back nothing, or in Perl-speak, the empty list, which is the right answer even though you don't see anything that represents it in the code.

APPENDIX E
Answers to Chapter 5 Exercises

Answer 5.1: This one is pretty easy once you know the trick. If you use the <> operator, you can look in `$ARGV` to discover the current filename. The <> operator will automatically move on to the next file when it finishes one, and it will update the filename in `$ARGV`. You simply tack on the filename in `$ARGV` to the current line in `$_` and print it. You don't need a newline at the end because `$_` already has it:

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

while( <> ) { print "$ARGV: $_" }
```

If you didn't know the trick, it's not so bad. You go through each of the command-line arguments in `@ARGV` and store each one in `$file` in turn. You open each file then go through its lines and print them as you did before:

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

foreach my $file ( @ARGV ) {
    open my( $fh ), '<', $file;

    while( <$fh> ) { print "$file: $_" }

    close $fh;
    }
```

A run might look like this:

```
$ perl ex5.1.pl cats dogs
cats: Buster
cats: Mimi
cats: Roscoe
dogs: Nicki
dogs: Addie
```

Answer 5.2: You can specify standard input using the virtual filename `-`. When the line input operator sees that filename, it starts reading from standard input:

```
$ perl ex5.2.pl *.pl -
```

You can do this in one `foreach` loop. You go through the files like you did in your program in Answer 5.1, but if the name of the file is `-`, you set the name to `stdin` instead of `-`:

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

foreach my $file ( @ARGV ) {
    open my( $fh ), '<', $file;

    $file = 'stdin' if $file eq '-';

    while( <$fh> ) { print "$file: $_" }

    close $fh;
    }
```

If you don't want to deal with `-` on the command line, you can do it in your program; you have to add it to the end of `@ARGV` yourself. You use `push` to add an element to the end of the list:

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

push @ARGV, '-';

foreach my $file ( @ARGV ) {
    open my( $fh ), '<', $file;

    $file = 'stdin' if $file eq '-';

    while( <$fh> ) { print "$file: $_" }

    close $fh;
    }
```

Now you don't need to add it on the command line yourself.

Either way, a run of your program looks a bit odd at first because the lines of input are intermingled with the lines of output:

```
$ perl ex5.2.pl -
This is a line of input
stdin: This is a line of input
This is another line
stdin: This is another line
```

Answer 5.3: You often see this problem when the programmer (or team of programmers) can't decide which parts of the program should be allowed to create output.

There's a lot of design issues to discuss and argue here, but for this problem I'll just say that subroutines should merely create the strings you print in the main part of the program.

The output you get from the unmodified program is close to what you want, but it's easy to miss the problem if you aren't looking closely:

```
9,9991
```

You just wanted to add commas in the right place. What's that extra `1`? It comes from the double `print`. You first `print` the formatted number in the subroutine; that's where you get the `9,999`. Once you do that, the subroutine returns the last evaluated expression. That's from the `print`, which returns `1` since it was able to do what it's supposed to do. The `print` in the main part of the program outputs what you give it, which is the return value of the subroutine. Thus, a `1` shows up after the `9,999`. If you see this sort of behavior in your programs (that extra `1`), look for something where `print` is actually printing the return value of `print`.

To fix the program, output in only one place by removing the `print` from the subroutine and leaving `$_` as the return value. You can't just stop after the `while` line because you still won't get the number as the last evaluated expression:

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

print format_number( 9999 ), "\n";

sub format_number {
    local $_ = shift;

    1 while s/^([-+]?\d+)(\d{3})/$1,$2/;

    $_;
    }
```

Answer 5.4: The `printf` function has format specifiers for each of the representations you want. Once you have the number, you simply need to plug it into the right places. Preface each format specifier with a `5` to set the number is a five character wide column so the numbers line up on the right hand side. You just have to use a number whose binary representation won't use more than five columns (but if you do, it just overflows to the right):

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

print "Please enter a number: ";
my $number = <STDIN>;

printf "binary:  %5b\noctal:   %5o\ndecimal: %5d\nhex:     %5x\n",
    $number, $number, $number, $number;
```

You could save yourself a bit of typing with a trick you didn't see in *Learning Perl*. You can actually use the replication operator, `x`, to replicate a list. It's not just for strings:

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

print "Please enter a number: ";
my $number = <STDIN>;

printf "binary:  %5b\noctal:   %5o\ndecimal: %5d\nhex:     %5x\n",
    ( $number ) x 4;
```

As a bonus, you could dynamically decide the field length. Once you get the number from the input, you just need to know the length of the longest representation of the number. The longest will be the binary representation, so you just need to know the lowest power of 2 that the number is not greater than (otherwise known as the "least upper bound"). Take the base-2 logarithm, use the next highest power, and store that number in `$width`. When you use it in the `printf` format, you need to limit the variable name so Perl doesn't think the format specifier is part of the name: just put braces around the name:

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

print "Please enter a number: ";
my $number = <STDIN>;

my $width = int( log($number) / log(2) ) + 1;

printf "binary:  %${width}b
octal:   %${width}o
decimal: %${width}d
hex:     %${width}x\n",
    ( $number ) x 4;
```

Note that Perl's `log` is a natural logarithm, so you have to divide by the natural log of 2 to convert the result to a power of 2.

Answer 5.5: You need to output two lines. The first is a line of column names, and the second is the computed numbers. The first four columns are easy: you just have to plug in the right numbers. For the percent fraction, first compute the quotient, but then multiply by `100` to get the percent. In the `printf` format string, use the `.2f` format specifer to denote a floating point number with only two decimal places, and follow it with a `%%` to get a literal `%` in the output:

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

print  "  first  second     sum   product     %\n";
```

```
printf "%7d %7d %7d   %7d   %.2f%%\n",
    $ARGV[0] , $ARGV[1],
    $ARGV[0] + $ARGV[1],
    $ARGV[0] * $ARGV[1],
    $ARGV[0] / $ARGV[1] * 100;
```

That's actually a simple-minded solution because it has a fatal flaw. If the second number you enter is 0 then you'll try to divide by zero when you compute the percent. You could do more work to get around that by making that a special case:

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

print  "  first  second     sum   product     %\n";

my $percent;
if( $ARGV[1] == 0 ) {
    $percent = 0;
    }
else {
    $percent = $ARGV[0] / $ARGV[1] * 100;
    }

printf "%7d %7d %7d   %7d   %.2f%%\n",
    $ARGV[0] , $ARGV[1],
    $ARGV[0] + $ARGV[1],
    $ARGV[0] * $ARGV[1],
    $percent;
```

Later, in Chapter 17, you'll learn about the `eval`, which is a more elegant way to handle this. With `eval`, you'll just let the error happen and the `eval` will catch it, giving undef as the result, which turns into the number 0:

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

print  "  first  second     sum   product     %\n";

printf "%7d %7d %7d   %7d   %.2f%%\n",
    $ARGV[0] , $ARGV[1],
    $ARGV[0] + $ARGV[1],
    $ARGV[0] * $ARGV[1],
    eval { $ARGV[0] / $ARGV[1] } * 100;
```

Answer 5.6: This is similar to Exercise 5.3 in *Learning Perl*. You solve this problem in two parts. First, go through all of the numbers from the command line and figure out the maximum length of the numbers. Since Perl has interchangeable numbers and strings, treat the number as a string to get its length. If the length of the current number is greater than the one you've stored in `$max`, update `$max`. In this short program, you don't need to pre-declare `$max`; Perl automatically creates it and gives it the `undef` value when you use for the first time:

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

my $max = 0;
foreach my $number ( @ARGV ) {
    my $l = length $number;
    $max = $l if $l > $max;
    }

foreach my $number ( @ARGV ) {
    printf "%${max}d\n", $number;
    }
```

Answer 5.7: For this answer, write a subroutine, `count_lines`, to count the lines so you can keep the `foreach` loop short and easy to read. In the `foreach`, go through each of the files on the command line and pass their names to `count_files`, which returns the number of lines in the file. If `count_lines` returns a defined value, output the number of lines that you found.

In your `count_lines` subroutine, take the name of the file off of the argument list and store it in `$file`. Initialize `$sum` to zero, and use this variable to collect the count of the lines. If you can open the file, use a `while` loop to read each line and add to `$sum`. When you finish with the `while` loop, return `$sum`. If you couldn't open the file, use `warn` to output a message that includes the Perl error variable `$!`. This message shows up on standard error. Since you don't include a newline at the end of the string you pass to `warn`, it will add the program name and line number to the end of the message. That way you know which `warn` issued the message. Finally, return nothing (which will be an undefined value when you store it in `$count`):

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

foreach my $file ( @ARGV ) {
    my $count = count_lines( $file );

    if( defined $count ) {
         print "There are $count lines in $file\n";
         }
    }

sub count_lines {
    my $file = shift;

    my $sum = 0;

    if( open my( $fh ), '<', $file ) {
         while( <$fh> ) { $sum++ }
         return $sum;
         }
    else {
         warn "Could not open file! $!";
```

```
            return;
            }
        }
```

Answer 5.8: This program is almost the same as the one you saw in Answer 5.7, but you add one line at the top. Re-open `STDERR` and connect it to the file *ex5.8.error.log*. Use the `>>` to open the file in append mode so you add to the file rather than overwrite it. All of the standard error output will now show up in that file:

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

open STDERR, '>>', 'ex5.8.error.log';

foreach my $file ( @ARGV ) {
    my $count = count_lines( $file );

    if( defined $count ) {
        print "There are $count lines in $file\n";
        }
    }

sub count_lines {
    my $file = shift;

    my $sum;

    if( open my( $fh ), '<', $file ) {
        while( <$fh> ) { $sum++ };
        return $sum;
        }
    else {
        warn "Could not open file! $!";
        return;
        }
    }
```

Answer 5.9: You can do this in a few ways (just like anything else in Perl). Attempt to `open` the file, and if you can't, stop the program with `die` and print an error message, including the Perl special variable `$!`, which tells the reason Perl could not `open` the file. You should always ensure you were actually able to `open` the files before you try to use their filehandles.

Once you have the opened filehandle, you need to get one line then print it. Use the `scalar` function to force `<$fh>` into scalar context so you get only one line from the filehandle. If you don't force the scalar context, the filehandle will be in list context and will return all of the lines, which you don't want:

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

```
open my( $fh ), '<', $ARGV[0] or die( "Could not open $ARGV[0]! $!" );

print scalar <$fh>;
```

You can also force scalar context by assigning the result of `<$fh>` to a scalar. The result of an assignment is the value that you assigned, and that's the value you give to `print`:

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

open my( $fh ), '<', $ARGV[0] or die( "Could not open $ARGV[0]! $!" );

print my $line = <$fh>;
```

If you didn't want to play either of those tricks, you can simply read a line of input and store it in a scalar. In this example, you assign the line to `$_` then call `print` without an argument since you know it uses `$_` by default:

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

open my( $fh ), '<', $ARGV[0] or die( "Could not open $ARGV[0]! $!" );

$_ = <$fh>

print;
```

Answer 5.10: First, open the log file where you want store the messages. Use `>>` for append mode in the `open` so you add to the file instead of overwriting it. If you can't open the file, call `die` with a message that includes the Perl special variable `$!`, which contains the reason you couldn't open the file.

Next, prompt for a message then read the answer from standard input and store it in `$message`. Call `localtime` and assign it to a scalar. Since this is scalar context, `localtime` returns a single string that represents the date instead of a list of date parts. That's just what you need for the timestamp.

Finally, output to the filehandle you opened earlier. Create a string that starts with the timestamp wrapped in `[]` to separate it from the rest of the string then the rest of the message:

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

my $log = 'ex5.10.log';
open my( $out ), '>>',  $log
    or die "Could not open $log! $!";

print "Enter a message>> ";

my $message = <STDIN>;
```

```
my $timestamp = localtime;

print $out "[$timestamp] $message";
```

Answer 5.11: You can complete this task very simply by using `binmode` to set the `:crlf` layer. This pseudolayer translates newlines to carraige-return/newline pairs. The `say` adds the newline for you, but the output layer takes care of the translation:

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

use 5.010;

binmode STDOUT, ':crlf';

foreach ( @ARGV ) {
    say $_;
    }
```

How can you tell that your output came out correctly, though? If you are on a Unix shell command line, you can pipe to the `hexdump` program to turn the output into a hexadecimal representation so you can see each byte's ordinal value. This lets you see the `0d 0a` that represents the carriage-return/line-feed pair:

```
$ perl ex5.11.pl cat dog bird | hexdump
0000000 63 61 74 0d 0a 64 6f 67 0d 0a 62 69 72 64 0d 0a
0000010
```

Answer 5.12: Your answer to this exercise is the inverse of the answer for Exercise 5.11. Instead of outputting information and adding the carriage return then, you need to input some information and remove the carriage return. The `:crlf` layer actually does both jobs. On reading, it does the conversion that you need. Open a file, setting the encoding with the `open` mode. When you read a line, `perl` converts it for you and you merely need to `print` the result:

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

open my $fh, '<:crlf', $ARGV[0];

while ( <$fh> ) {
    print $_;
    }
```

Answer 5.13: This task is one step beyond Exercise 5.12. You have to `open` the input file for reading using the UTF-8 encoding and `open` the output file for writing using the UTF-16BE encoding. Once you have those two filehandles, you read from the input and immediately print to the output. The Perl layers take care of the rest for you:

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

```
use warnings;

open my $in_fh,  '<:encoding(UTF-8)',    $ARGV[0];
open my $out_fh, '>:encoding(UTF-16BE)', $ARGV[1];

while ( <$in_fh> ) {
    print $out_fh $_;
    }
```

If you use the file you created in Exercise 5.11, a hex dump of your input file looks like compact and tidy:

```
$ perl ex5.11.pl cat dog bird > utf8.txt
0000000 63 61 74 0d 0a 64 6f 67 0d 0a 62 69 72 64 0d 0a
0000010
```

Your UTF-16BE-encoded data, however, take up much more space since every character takes up at least two bytes. In this case, many of those are `00`:

```
$ perl ex5.13.pl utf8.txt utf16be.txt
$ hexdump utf16be.txt
0000000 00 63 00 61 00 74 00 0d 00 0a 00 64 00 6f 00 67
0000010 00 0d 00 0a 00 62 00 69 00 72 00 64 00 0d 00 0a
0000020
```

APPENDIX F
Answers to Chapter 6 Exercises

Answer 6.1: Most of this problem is simply creating the hash. There is nothing fancy about it: you just have to type it in. In `%birthdays`, use a single line for each key-value pair and use the big arrow operator, `=>`, to separate the keys and values. The big arrow automatically quotes the string on its left. You can also line up the big arrows to create easy-to-read columns.

To report the contents of the hash, go through it with a `foreach` loop. With each of the keys, output a line that includes the name and the birthday and look up the value directly in the double-quoted string:

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

my %birthdays = (
    Fred    => 'April 5',
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    );

foreach my $name ( keys %birthdays ) {
    print "$name => $birthdays{$name}\n";
    }
```

Here's a sample run:

```
$ perl ex6.1.pl
Pebbles => October 8
Wilma => October 26
Fred => April 5
```

Answer 6.2: Take the hash that you created in the Answer to Exercise 6.1. Prompt the user for a name then read a line from standard input and store it in `$name`. You don't want the newline on the end of the input, so remove it with `chomp`.

Once you have the name, output a message. Look up the value and interpolate it directly in the double-quoted string:

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

my %birthdays = (
    Fred    => 'April 5',
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    );

print "Enter a name: ";
chomp( my $name = <STDIN> );

print "$name has a birthday on $birthdays{$name}\n";
```

Here's a sample run:

```
$ perl ex6.2.pl
Enter a name: Fred
Fred has a birthday on April 5
```

Answer 6.3: The `exists` function solves this problem. Before you output anything, check that the key is in the hash with `exists`. If `exists` returns true, meaning that key is in `%birthdays`, output the same thing you did in your program in the Answer to Exercise 6.2. If `exists` returns false, the key is not in `%birthdays`, so output a message telling the user that the name is not in `%birthdays`:

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

my %birthdays = (
    Fred    => 'April 5',
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    );

print "Enter a name: ";
chomp( my $name = <STDIN> );

if( exists $birthdays{$name} ) {
    print "${name}'s birthday is $birthdays{$name}\n";
    }
else {
    print "$name is not in the birthday registry\n";
    }
```

Here's a sample run:

```
$ perl ex6.3.pl
Enter a name: Fred
Fred's birthday is April 5
```

Answer 6.4: This answer can be a bit tricky. In the first case, you want to print the name and birthday if the name is in the hash and the key has a true value (the empty string is false). You just use the value itself as the condition in the first `if` branch. The

second case has a key in the hash, but not a true value. You still want to print something meaningful. If the key exists and doesn't have a value (you've already taken care of those that have values in the first branch), tell the user that the name is in the hash. The `exists` function doesn't care what the value is as long as the key is there, so even though you have the key, Barney, with the empty string as its value, you can still tell the user that Barney is in the hash. If neither of those works, catch it in the third branch that prints that the name is not in the hash:

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

my %birthdays = (
    Fred    => 'April 5',
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    Barney  => '',
    Slate   => undef,
    );

print "Enter a name: ";
chomp( my $name = <STDIN> );

if( $birthdays{$name} ) {
    print "${name}'s birthday is $birthdays{$name}\n";
    }
elsif( exists $birthdays{$name} ) {
    print "$name is in the hash, but I don't know the birthday\n";
    }
else {
    print "$name is not in the birthday registry\n";
    }
```

Here's a couple of sample runs:

```
$ perl ex6.4.pl
Enter a name: Fred
Fred's birthday is April 5

$ perl ex6.4.pl
Enter a name: Slate
Slate is in the hash, but I don't know the birthday
```

Answer 6.5: For this program, go through the keys of the hash and output a line for each key that has a true value. You get the list of keys with the `keys` function, and each key ends up in `$name` in turn. Inside the `foreach` loop, use the value for that key in the `if` condition. If the value is true, output the line. If it isn't, move on to the next key:

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

my %birthdays = (
    Fred    => 'April 5',
```

```
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    Barney  => '',
    Slate   => undef,
    );

foreach my $name ( keys %birthdays ) {
    if( $birthdays{$name} ) {
         print "$name -- $birthdays{$name}\n";
         }
    }
```

Here's a sample run:

```
$ perl ex6.5.pl
Pebbles -- October 8
Wilma -- October 26
Fred -- April 5
```

Answer 6.6: You have to change the program around a bit to use the each function. The each function thinks about one key-value pair at a time, which makes it really good for very large hashes where you don't want to create a list of all the keys at one time, potentially taking up a lot of memory. Instead of a `foreach` loop, use a `while`. When you've gone through all the key-value pairs, each returns `undef` and the `while` loop will stop. In turn, store each key and value in `$name` and `$birthday`.

Inside the `while` loop, test the value of `$birthday` to decide what you want to do. You don't bother with a hash lookup since you already have the value in `$birthday`:

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

my %birthdays = (
    Fred    => 'April 5',
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    Barney  => '',
    Slate   => undef,
    );

while( my( $name, $birthday ) = each %birthdays ) {
    if( $birthday ) {
         print "$name -- $birthday\n";
         }
    }
```

Here's a sample run:

```
$ perl ex6.6.pl
Pebbles -- October 8
Wilma -- October 26
Fred -- April 5
```

Answer 6.7: Most of this solution comes from the program you wrote for the Answer to Exercise 6.1. Add a couple of lines in the middle to cleanse the `%birthdays` hash before you output its contents. Go through all of the keys in a `foreach` loop, and unless the value is true (you don't really care about valid dates), use `delete` to remove the key from the hash.

Go through the keys with `foreach`. Test the value as you did in the Answer to Exercise 6.6, and if it is false, delete that key from the hash. At the end of this `foreach` loop, there are no false values in the hash.

When you report the contents of the hash, you don't see the entries for Barney or Slate since `delete` removed them. Otherwise, you should get output that looks like that for the Answer to Exercise 6.1:

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

my %birthdays = (
    Fred    => 'April 5',
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    Barney  => '',
    Slate   => undef,
    );

foreach my $name ( keys %birthdays ) {
   delete $birthdays{$name} unless $birthdays{$name};
   }

foreach my $name ( keys %birthdays ) {
    print "$name -- $birthdays{$name}\n";
    }
```

Here's a sample run, where you output looks like that from your program in the Answer to Exercise 6.1, although for different reasons:

```
$ perl ex6.7.pl
Pebbles -- October 8
Wilma -- October 26
Fred -- April 5
```

APPENDIX G
Answers to Chapter 7 Exercises

Answer 7.1: The regular expression portion of this answer is a simple alternation. You want to match a string that contains either "fred" or "barney". Use alternation for that. This regular expression matches either word anywhere in the input string:

```
m/fred|barney/
```

Once you have matched the string, you can use the Perl special variables `$`` (that's a backtick) and `$'` (that's a single tick) to get the parts of the string before and after the parts that matched.

Around that, wrap a `while` loop to prompt the user for a string then try to match the string. Put the `m//` in the `if` condition. If the string matches, output the report for the problem. If the string does not match, output a different string:

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

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/fred|barney/ ) {
    print "Before: $`\nName:  $&\nAfter: $'\n";
    }
else {
    print "The string did not match\n";
    }
```

Here are some sample runs of the program:

```
$ perl ex7.1.pl
Enter a string>> fred
Before:
Name:  fred
After:

$ perl ex7.1.pl
Enter a string>> barney
Before:
```

```
Name:  barney
After:

$ perl ex7.1.pl
Enter a string>> aaaafredzzzz
Before: aaaa
Name:  fred
After: zzzz

$ perl ex7.1.pl
Enter a string>> xxxxbarney
Before: xxxx
Name:  barney
After:
```

If you write real programs using $`, $', or $&, other Perlers are likely to complain since those variables come with a performance problem as they slow down every match in your program. Perl 5.10 introduces a workaround for this by making the penalty per match. If you use the /p flag, you can use the ${^PREMATCH}, ${^POSTMATCH}, or ${^MATCH} variables instead:

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

use 5.010;

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/fred|barney/p ) {
    print "Before: ${^PREMATCH}\nName:  ${^MATCH}\nAfter: ${^POSTMATCH}\n";
    }
else {
    print "The string did not match\n";
    }
```

Answer 7.2: Use the same program that you wrote for Answer 7.1, but change the regular expression. You need to match an "a" followed by a "b" followed by a "c", so just put them in sequence. This regular expression matches when the characters are next to each other:

```
m/abc/
```

For this problem, though, you want to allow any number of characters between those letters. The regular expression meta-character for any character (except a newline) is the full-stop (.). That number could be zero, or one, or two, and so on, so after the ".", use the * quantifier:

```
m/a.*b.*c/
```

Now replace the regular expression from Answer 7.1 with your new one:

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

```
use warnings;

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/a.*b.*c/ ) {
    print "The string has an a, b, and c\n";
    }
else {
    print "String did not match\n";
    }
```

Here are some sample runs of the program:

```
$ perl ex7.2.pl
Enter a string>> abd
String did not match

$ perl ex7.2.pl
Enter a string>> abc
The string has an a, b, and c

$ perl ex7.2.pl
Enter a string>> axxxxxbyyc
The string has an a, b, and c
```

This solution has a slight problem. What if there are newlines between the letters? We don't tell you about the `/s` match flag until Chapter 8, so you could use a character class instead. If your character class is, for instance, all the digits and non-digits, it covers every character:

```
m/a[\d\D]*b[\d\D]*c/
```

Once you know about the `/s` flag, you can apply that to make the `.` match any character, including a newline:

```
m/a.*b.*c/s
```

Answer 7.3: This answer is a lot like the answer for your program for Answer 7.2 although a string like "abc" should not match because no other characters show up between the letters. Modify the regular expression from Answer 7.2 so it requires characters between those letters. The * quantifier becomes the + quantifier that requires one or more matches:

```
m/a.+b.+c/
```

The full program is otherwise the same:

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

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/a.+b.+c/ ) {
```

```
    print "The string has an a, b, and c\n";
    }
else {
    print "String did not match\n";
    }
```

Here are some sample runs of the program:

```
$ perl ex7.3.pl
Enter a string>> xyz
String did not match

$ perl ex7.3.pl
Enter a string>> abc
String did not match

$ perl ex7.3.pl
Enter a string>> axbyc
The string has an a, b, and c
```

Answer 7.4: This is a bit of a trick question since the best answer is not always a single regular expression. You could write such a beast, but *Learning Perl* hasn't shown you enough regular expression kung fu yet.

You already know how to match an "a". That's this regular expression:

```
m/a/
```

That gets you as far as a string that matches an "a". The second part of the problem states that there cannot be any "b" after that. You can check that by looking at `$'`, the part of the string after the portion you matched.

Learning Perl hasn't introduced the binding operator yet (wait until Chapter 8), so once you know you can match an "a", assign `$'` to `$_` and then try another match, `m/b/`. You don't want the program to find a "b", so negate the test in the second `if`. That `if` is true as long as `$'` does not have a "b":

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

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/a/ ) {
    $_ = $';
    if( ! m/b/ ) {
         print "The string has an a without a following b\n";
         }
    else {
         print "The string has an a, but a following b\n";
         }
    }
else {
    print "String did not match\n";
    }
```

Answer 7.5: Your program in this answer should look similar to your other answers in this chapter; it's just the regular expression that's interesting. To match at least three characters, use `(...+)`. To allow some additional characters between that sequence and its repetition, use `.*`. Finally, use the relative backreference `\g{-1}` to match the sequence again:

```
(...+).*\g{-1}
```

Here's the complete program:

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

use 5.010;

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/(...+).*\g{-1}/ ) {
    say "Found a repeated [$1]"
    }
else {
    say 'String did not match';
    }
```

Here's a sample run:

```
$ perl ex7.5.pl
Enter a string>> abcdabc
Found a repeated [abc]
```

If you couldn't use a relative backreference, you can still solve the exercise. Prior to Perl 5.10, just use `\1`:

```
m/(...+).*\1/
```

Perl has an easier way to handle patterns like "at least three", but we don't show you that until Chapter 8 of *Learning Perl*. You can also use the minimum-maximum quantifier with no maximum:

```
(.{3,}).*\g{-1}
```

Answer 7.6: Your program in this answer should look similar to your program in Answer 7.5 and the parts of your pattern are the same as in your answer to that exercise, just more so:

```
(...+).*(...+).*\g{-2}.*\g{-1}
```

Here's the complete program:

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

use 5.010;
```

```
print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/(...+).*(...+).*\g{-2}.*\g{-1}/ ) {
    say "Found a repeated [$1] and [$2]"
    }
else {
    say 'String did not match';
    }
```

Here's a sample run:

```
$ perl ex7.6.pl
Enter a string>> abc123abcd123
Found a repeated [abc] and [123]
```

If you couldn't use a relative backreference, you can still solve the exercise. Prior to Perl 5.10, just use `\1` and `\2`:

```
m/(...+).*(...+).*\1.*\2/
```

If you rewrite this with the minimum-maximum quantifier, which you don't actually read about until Chapter 8 of *Learning Perl*, you have:

```
m/(.{3,}).*(.{3,}).*\1.*\2/
```

Answer 7.7: To match a Unicode property, you use the `\p{}` sequence with the appropriate property name. To find the names, you can look at the list of properties on the Unicode.org website or look in perluniprops. Perl adds a few bonus properties, though, so checking Perl's list is better. The property name for numbers is `Number` and for whitespace is `Space`. You'll find many other properties that are variations on these and specify various types of numbers or whitespace, but these are probably the general ones that you want:

```
#!/usr/bin/perl

use 5.010;

use strict;
use warnings;

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/\p{Space}/ )  { say 'Found a type of space'; }
if( m/\p{Number}/ ) { say 'Found a type of number'; }
```

Here's a couple of sample runs:

```
$ perl5.14.1 properties.pl
Enter a string>> This is a line
Found a type of space

$ perl5.14.1 properties.pl
Enter a string>> This is 2 lines
```

```
Found a type of space
Found a type of number
```

APPENDIX H
Answers to Chapter 8 Exercises

In Chapter 7, you used the same basic program to test all of the regular expressions. Instead of copying that into every answer for this chapter, I'll just give you the regular expressions and you can plug them into the program yourself.

Answer 8.1: The regular expression is just the character class that contains those letters. Put the letters between the `[]`:

```
m/[abx]/
```

To make it case insensitive, add the uppercase characters, too. There is an easier way to do this, but *Learning Perl* doesn't show you that until Chapter 9:

```
m/[abxABX]/
```

You could have also done this problem with an alternation, although I told you to use a character class. The alternation works, but it's a lot more work, not only for you to type but for Perl to run it:

```
m/a|b|x|A|B|X/
```

Answer 8.2: You've already done most of the work for this problem. Now you just need to anchor the character class to the beginning of the string. The `\A` specifies that the regular expression has to start at the beginning of the string:

```
m/\A[abx]/
```

To match it at the end of the string, use the end-of-string anchor, `\z`, at the end of the regular expression:

```
m/[abx]\z/
```

If you are used to older versions of Perl, you might have used either the `^` or `$` anchors instead. Strictly speaking, these aren't anchors for the beginning or end of a string since you can modify them to anchor at the beginning or end of logical lines with the `/m` modifier. If you are looking strictly for the string anchors, you might be surprised by these. This is even more important as people can set default regular expression modifiers that you don't see with the operators. Review Anchors (page 138) in *Learning Perl* if you need to remind yourself of the details.

Answer 8.3: You can *not* match something in several ways, but when you just want to exclude a few characters, you can use a negated character class. Instead of specifying the characters that should match, specify the ones that shouldn't. The character class starts with an `^` that forms the complement of the set of characters you specify:

```
m/\A[^abx]/
```

It can get even more twisted. Suppose you don't want to match a literal `^` at the beginning of the string. That regular expression looks like, well, I don't know what it looks like but it isn't pretty. The `^` is only special at the beginning of the character class, so there are actually three different meanings of `^` in this pattern (see the Answer 8.2 for an explanation of the difference between `\A` and `^`):

```
m/^[^^]/
```

The end-of-string versions are the same, except you use the `\z` anchor at the end instead of the `\A` at the beginning:

```
m/[^abx]\z/
m/[^^]\z/
```

Answer 8.4: The trick here is the regular expression back references. You can reuse anything you capture in parentheses. You don't have to know how long that first thing is, or how many characters intervene between it and the end of the string.

You know that you need to anchor each part at the beginning and the end of the string, so the regular expression starts with `\A` and ends with `\z`. At the beginning of the string, match one or more characters except a newline, `(.+)`. Whatever you match there ends up in the back reference `\1`, which you put next to the end-of-string anchor. There might be characters between those parts, so put a `.*` in the middle. Putting it all together, you get this regular expression:

```
m/\A(.+).*\1\z/
```

You can also specify the backreference with `\g` (see Chapter 7):

```
m/\A(.+).*\g1\z/
```

You can also use a relative backreference:

```
m/\A(.+).*\g{-1}\z/
```

Answer 8.5: Bring back the regular expression testing program from Chapter 7. To report the variable type, create a hash that uses the sigils characters (`$`, `@`, `%`) as the keys, and their variable type as the value. You'll use that to lookup the variable type when you want to output it.

The regular expression to match the variable name is a bit tricky. You'll build it up gradually. In real life, complicated regular expression does not come to mind all at once (at least for me). Instead, write parts of it and test them as you go, add to it, test again, and so on.

First, you want to limit the string to exactly the part matched by the regular expression, so use the beginning-of-string and end-of-string anchors:

```
m/\A\z/
```

Next, you know that you need to start with one of the sigil characters, so you put that right after the beginning of string anchor as a character class (an alternation would also work). You have to escape the `$` and `@` characters because the match operator is actually a double-quoted string context and will try to interpolate variables. You don't have to worry about the `%` because hashes don't interpolate into strings:

```
m/\A[\$\@%]\z/
```

You want to remember the variable type after the match, so put that character class in parentheses to trigger the memory variables:

```
m/\A([\$\@%])\z/
```

Are you still with me? The next part is the first character of the variable name. For user-defined variable names, start with a letter or an underscore:

```
m/\A([\$\@%])[a-zA-Z_]\z/
```

Finally, the name can have any number of letters, underscores, or digits. I could make a character class for that. It's `[a-zA-Z0-9_]`, but Perl has a shortcut for that: `\w` (I'll say more about this in a moment). Since you can have zero or more of those characters, I use the * quantifier after it:

```
m/\A([\$\@%])[a-zA-Z_]\w*\z/
```

Now you plug that into my regular expression tester with a few changes. Once you match the string, tell the user which variable type it was. The sigil character is in the memory variable `$1`. In the string you output, use `$1` as the key for a hash lookup to get the variable type name:

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

my %sigils = qw( $ scalar @ array % hash );

print "Enter a string>> ";
chomp( $_ = <STDIN> );

if( m/\A([\$\@%])[a-zA-Z_]\w*\z/ ) {
    print "The variable type is $sigils{$1}\n";
    }
else {
    print "The string is not a valid Perl variable name\n";
    }
```

Since that pattern is so complicated, you can use the `/x` match modifier to add insignificant whitespace and comments to remind yourself what each part does:

```
m/
    \A          # start of the string
    (           # start of capture
        [\$\@%]  # sigil
    )
    [a-zA-Z_]    # first character of name
    \w*          # the rest of the characters
    \z           # end of the string
    /x
```

I mentioned earlier that I'd say more about the `\w` shortcut. Prior to Perl's Unicode support, the `\w` was strictly a subset of ASCII, the `[a-zA-Z0-9_]` you used so far. With Unicode support, the `\w` character class can match much more, meaning that patterns that you wrote a long time ago might now match more than you intended.

Perl 5.14 added the `/a` modifier to force regular expressions to use their old ASCII meanings:

```
m/
    # Perl 5.14 version
    \A           # start of the string
    (            # start of capture
        [\$\@%]   # sigil
    )
    [a-zA-Z_]     # first character of name
    \w*           # the rest of the characters, ASCII only now with /a
    \z            # end of the string
    /xa
```

If you don't have Perl 5.14 or later, you have to do the work yourself by avoiding the character class altogether:

```
m/
    \A             # start of the string
    (              # start of capture
        [\$\@%]    # sigil
    )
    [a-zA-Z_]      # first character of name
    [a-zA-Z0-9_]*  # the rest of the characters, ASCII only now with /a
    \z             # end of the string
    /x
```

With Perl 5.14, you can also explicitly tell Perl that you want to use the Unicode interpretation. Turn that on with the `/u` modifier:

```
m/
    # Perl 5.14 version
    \A           # start of the string
    (            # start of capture
        [\$\@%]   # sigil
    )
    [a-zA-Z_]     # first character of name
    \w+           # the rest of the characters, UCS now with /u
    \z            # end of the string
    /xu
```

This still has a problem because the first character can be more than just `[a-zA-Z_]`. It can't be a decimal digit still, but with a Unicode identifier, it can be much more. You probably didn't know this, but there are Unicode properties to handle this. Characters valid at the start of an identifier have the `ID_Start` property and the characters valid in the rest of the identifier have the `ID_Continue` property. This greatly simplifies your pattern:

```
m/
    \A            # start of the string
    \p{ID_Start}
    \p{ID_Continue}*
    \z            # end of the string
    /x
```

However, Unicode Technical Report 31 recommends the updated properties `XID_Start` and `XID_Continue` for a couple of picky details, which you can read about in that report:

```
m/
    \A            # start of the string
    \p{XID_Start}
    \p{XID_Continue}*
    \z            # end of the string
    /x
```

If you tried to fake it yourself with a combination of properties, you might have come up with something like:

```
m/
    \A            # start of the string
    [\p{Alphabetic}\p{Mark}\p{Connector_Punctuation}]
    [\p{Alphabetic}\p{Mark}\p{Connector_Punctuation}\p{Decimal_Number}]*
    \z            # end of the string
    /x
```

APPENDIX I
Answers to Chapter 9 Exercises

Answer 9.1: The first one is easy. You use an alternate delimiter for your match operator. Since you're not using the default // delimiters, you need the leading `m`. I like to use the pipe ("|") character, but you can use other characters, or even the paired characters like { and }. These are the same thing:

```
m|/usr/bin/perl|
m(/usr/bin/perl)
m{/usr/bin/perl}
m,/usr/bin/perl,
```

To match a web address with a fragment identifier, you can cheat a little. Rather than get into the edge and special cases, look for the beginning *http://* then non-whitespace characters (`\S`) followed by the `#` character which starts the fragment name (assuming any literal `#`'s are properly escaped). Using the paired curly braces as your delimiter, your pattern looks like:

```
m{http://(\S+)#}
```

Matching a real C++ comment is rather tricky. In the simple case, it's just the // until the end of the line. What if that // shows up in a literal string instead of a comment, though? Well, then, this won't work. I just want you to use an alternate delimiter, so here's the simple regular expression. If you really wanted to do this, you'd have to completely parse the C++ code, and luckily for you, *Learning Perl* didn't cover that:

```
m[//]
```

However, if you are curious, *perlfaq6* has a complicated regular expression that tries to do this for any situation.

Answer 9.2: The regular expression from Exercise 8.1 wanted to match a string with an "a", "b", or "x" of either case:

```
m/[abxABX]/;
```

Now you know that is too much work. You can rewrite that now with the `/i` modifier that turns off case-sensitivity:

```
m/[abx]/i
```

In the answer to Exercise 8.5, you needed to match a Perl variable name, and you used this regular expression:

```
m/^([\$\@%])[a-zA-Z_]\w*$/
```

You can write that now as a regular expression that is a bit shorter. If you decided, for some reason, to limit the match to the characters between "a" and "m", you only have to modify one range:

```
m/^([\$\@%])[a-z_]\w*$/i
```

However, using the fancier Unicode properties means you don't have to do that because the lowercase and uppercase characters are already in the properties. So, there is no change:

```
m/
    \A            # start of the string
    \p{ID_Start}
    \p{ID_Continue}*
    \z            # end of the string
    /x
```

Answer 9.3: First you have to get the input, but rather than type the same lines over and over again, make a loop to prompt the user for each line. Start with an array, `@orders`, where you put the strings "first", "second", and "third". Use those strings when you ask the user for a line, and you'll use its length to determine how many lines to read. If you want to read more or fewer lines, just change the number of things in `@orders`.

Go through `@orders` with a `foreach` loop, and use the value of the current element as part of the string. That gives the prompt you need so you know which line you're on. Read in the line, store it in `$line`, `chomp` off the newline, then push the line onto `@lines`, where you store all of the entered lines. You'll use that later when you want to check the matches.

Prompt for a regular expression and store it in `$regex`. Don't worry about invalid regular expressions in this program, although *Learning Perl* covers that sort of thing in Chapter 17 with `eval`.

Once you have the input you need so you're ready to start the task. For the output, print which line it was (using the elements of `@orders`) and if that line matched. Since you want to pull elements from two arrays at the same time, use `foreach` to go through the indices of one of them and pull out the two elements by their index. Since the arrays are the same length, you go through all the elements of each of them.

If the `if` condition is true, bind an element of `@lines` to the regular expression, which is the string in `$regex`. The match operator interpolates that value and treats it as if you had literally typed it right there. The binding operator, `=~`, lets you apply that regular expression to any scalar value or variable.

If the string matches, output a string that says which line it is and that it matched. If it doesn't match, output a string that says which line it is and that it didn't match:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

my @orders = qw( first second third );
foreach my $order ( @orders ) {
    print "Enter the $order string>> ";

    chomp( my $line = <> );

    push @lines, $line;
    }

print "Enter a regular expression>> ";

chomp( my $regex = <> );

foreach my $index ( 0 .. $#lines ) {
    if( $lines[$index] =~ m/$regex/ ) {
         say "The $orders[$index] line matches";
         }
    else {
         say "The $orders[$index] line does not match";
         }
    }
```

Answer 9.4: It only looks like you have a lot to do in this program, but you can use a single `print` statement and give it four strings to output. Don't `chomp` the string so the lines of output show up on separate lines without extra typing. There's no sense taking off a newline just to put it back on.

The first string is the original string, `$_`, which you don't modify. The next string is `$_` in all uppercase, which you get with the `\U` case-shifting operator. Then you have the all lowercase version with `\L`, and finally the initial upper case version using `\L\u`. You end the whole thing with an additional newline so the groups of successive lines have a blank line between them:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

while( <> ) {
    say $_, "\U$_", "\L$_", "\L\u$_";
    }
```

You might write that as one long string, too, although it's a bit harder for you to see the parts:

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

while( <> ) {
    print "$_\U$_\L$_\L\u$_";
    }
```

Answer 9.5: This one is not as simple. Start with the program from Answer 9.4. After you output the first part, store an all lowercased version of `$_` in `$last`. That gets you halfway there. After that, use the substitution operator with the global modifier (`/g`) to find all the word characters (`\w`) that follow a word boundary (`\b`). By enclosing the word character in parentheses, you capture it in `$1`. On the replacement side of the substitution, you use the all uppercased version of `$1`. Finally, output the result and a trailing newline for good measure:

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

while( <> ) {
    print $_, "\U$_", "\L$_", "\L\u$_";

    my $last = "\L$_";

    $last =~ s/\b(\w)/\U$1/g;

    print $last;
    }
```

That's the version without the `/r` modifier. You can get rid of a couple of lines that prepare the string you eventually bind to the substitution since the `/r` will return the result. You don't need the temporary variable at all:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.014;

while( <> ) {
    print $_, "\U$_", "\L$_", "\L\u$_";

    print "\L$_" =~ s/\b(\w)/\U$1/rg;
    }
```

Answer 9.6: For the report, you need the field names and their values. You get the field names from the first line of input, which you store in `$header`. You immediately `chomp` to get rid of the newline. Split up the value in `$header` by tabs and store the fields in `@field_names`. You'll use that later when you need to output the report.

Continue reading lines, but now each line is data. You can `split` those on tabs and store the fields in `@fields`. Now you have the two parts you need: the fields and the values.

To output a section of the report, pull data from two different arrays. In this situation, rather than go through an array, go through all the indices of one of the arrays. It doesn't matter which one you choose since they have the same number of elements. With each index, you can pull the right element from `@field_names` and `@fields` at the same time:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

chomp( my $header = <> );
my @field_names = split /\t/, $header;

while( <> ) {
    chomp;
    my @fields = split /\t/;

    foreach my $index ( 0 .. $#fields ) {
         say "$field_names[$index]: $fields[$index]"
         }
    }
```

Perl 5.12 offers another interesting way to handle this, although we didn't cover it in *Learning Perl*. Now you can use `each` on an array to get both the value and its index at the same time:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

chomp( my $header = <> );
my @field_names = split /\t/, $header;

while( <> ) {
    chomp;
    my @fields = split /\t/;

    while( my( $index, $value ) = each @fields ) {
         say "$field_names[$index]: $value"
         }
    }
```

Answer 9.7: Read lines of input and `split` them on tabs, assigning the result to `@fields`. You can immediately `join` the values with "|":

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

while( <> ) {
    my @fields = split /\t/;

    print join( "|", @fields );
    }
```

You can also do this without the intermediate variable:

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

while( <> ) {
    print join "|", split /\t/;
    }
```

You can also flip this around to put it all on one line, although *Learning Perl* hasn't shown you this yet:

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

print join "|", split /\t/ while( <> );
```

And, just as a bonus, you could do this directly on the command line although *Learning Perl* hasn't told you this yet, either:

```
$ perl -ne 'print join "|", split /\t/' filename
```

APPENDIX J
Answers to Chapter 10 Exercises

Answer 10.1: Use the line-input operator to get lines. Before you output the line, check if it has the word "ruby" or "python" by trying to match either of those words with a regular expression. If your regular expression succeeds, stop the loop iteration and move on to the next line. If the regular expression fails, continue with the iteration and execute the `print` statement, which without an argument outputs the value of `$_`. This is a common Perl idiom for processing lines of files: read in a line and decide if you want to process it, and if not, use `next` to move on to the next line:

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

while( <> ) {
    next if m/ruby|python/;
    print;
 }
```

Answer 10.2: Read lines of input, and for each line use the auto-increment operator to add one to the value of `$sum`. After you do that, check if the word "perl" is in the line, and if it is, use the `last` command to stop the `while` loop and move on with the program.

Once you break out of the `while` loop, the value in `$sum` is the number of lines of input you read until you found the word "perl":

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

my $sum = 0;
while( <> ) {
    $sum++;
    last if m/perl/;
    }

say "There were $sum lines until I found perl";
```

There's another way that you can do this because Perl automatically tracks the line number for you in the `$.` special variable:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

while( <> ) {
    last if m/perl/;
    }

say "There were $, lines until I found perl";
```

Answer 10.3: Read in lines of input and `chomp` off the newline. Immediately lowercase the entire string so you don't have to worry about uppercase and lowercase versions of the same word. Although *Learning Perl* didn't tell you about the `lc` function, you could have used that too. In the next line, you use the substitution operator with a negated character class to replace all of the non-alphabetic characters with a space. You don't care too much about spaces because you're going to use `split` to break up the lines on spaces to get the list of words you assign to `@words`.

Once you have the list of words, go through it with `foreach` to increment the value for that key in the `%count` hash. Each word becomes a key, and the value is the number of times that you have seen it. This sort of hash is sometimes known as an "accumulator" and is a very common Perl idiom.

When you finish with the `while` loop, you just need to report the results. Go through all of the keys in `%count` and print a columnar report with the keys and their values:

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

my $count;
while( <> ) {
    chomp;
    $_ = "\L$_";   # or $_ = lc $_

    s/[^a-z]/ /g;
    my @words = split;

    foreach my $word ( @words ) {
         $count{$word}++;
         }
    }

foreach my $word ( keys %count ) {
    printf "%-25s %3d\n", $word, $count{$word};
    }
```

You could have skipped the intermediate variable `@words` by doing more in the `foreach` loop. Put the `split` in there directly and save yourself a little bit of typing:

```
foreach my $word ( split ) {
     $count{$word}++;
     }
```

Answer 10.4: You need a way to make a list of the multiples of three then go through them, but there isn't an elegant way to do that in Perl. You could go through all of the numbers and just skip the ones that aren't divisible by three, but the `for` loop is a pretty good tool for this.

Start by initializing `$i` to `3`, which is your starting value. Before you enter each iteration of the loop, Perl checks the condition in the second part of the `for`. In this case, as long as the value of `$i` is less than or equal to 99, you'll keep going. Inside the loop, output a line that has the number, its square, and its cube. Before you do the next iteration, do the last part of the `for`, which increments the value of `$i` by three. You don't have to skip any numbers. Just get the values you want:

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

for( my $i = 3; $i <= 99; $i += 3 ) {
    printf "%2d   %4d   %6d\n", $i, $i**2, $i**3;
    }
```

If you really wanted to use a `foreach`, you don't have to iterate from 3 to 99. Just go from 1 to 33 and multiply each by 3:

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

foreach my $i ( 1 .. 33 ) {
    $i *= 3;
    printf "%2d   %4d   %6d\n", $i, $i**2, $i**3;
    }
```

Later, after you know about the stuff in Chapter 17, you could move that multiplication into the `foreach` with a `map`:

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

foreach my $i ( map { $_ * 3 } 1 .. 33 ) {
    printf "%2d   %4d   %6d\n", $i, $i**2, $i**3;
    }
```

Answer 10.5: Read in lines of input, and if the line is blank, or just whitespace, use `next` to move to the next line. If the line starts with optional whitespace then a comment character, assume the line is a comment and move on to the next line (although in real life, that line might be in the middle of a big string, but I'll gloss over that for now and let you find the Perl parsers on CPAN). If you get past both of those `next`s, increment the value of `$sum`.

After you've gone through all of the lines and either counted them or skipped them, `$sum` has the total and you output its value:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

my $sum = 0;
while( <> ) {
    next if /^\s*$/;
    next if /^\s*#/;
    $sum++;
    }

say "I counted $sum lines";
```

Answer 10.6: This answer is mostly the program from the Answer to Exercise 10.5. After you count the line, use a regular expression to check if the line was `__END__`, and if it is, use `last` to break out of the loop. Continue the program after the `while` loop and print the results:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

my $sum = 0;
while( <> ) {
    next if /^\s*$/;
    next if /^\s*#/;
    $sum++;
    last if /^__END__$/;
    }

say "I counted $sum lines\n";

__END__
Don't count this line.
```

The `__END__` token can be a pretty handy debugging tool. If you all of a sudden run into a weird error and can't track it down, such as an unpaired brace or quote, you can put an `__END__` somewhere in the program. If the problem is before the end of the halved program, you should get the same error. If it is after, the error should disappear. Then, at least, you should have an idea which side of that token the error appears.

Answer 10.7: This problem is similar to the exercise at the end of this chapter in *Learning Perl*, but you don't care to tell the user if the guess is too high or too low. Why give them any help?

First, you need a secret number, so create it just as in the answer in *Learning Perl*. The `rand(10)` gives you fractional numbers between 0 and 9.9999, and the `int` portion of

that gives you whole numbers between 0 and 9. Add 1 to that to shift the range to 1 to 10.

Set up the naked block as your looping structure, and decide what to do in the `if` structure. If you guess the secret number, use `last` to break out of the loop. If you didn't guess the number, use `redo` to go back to the top of the loop and start again. This loop will keep going until you guess the number:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

my $number = int( rand( 10 ) + 1 );

{
print 'Guess the number>> ';
my $guess = <STDIN>;

if( $guess == $number ) {
    say 'You guessed it!';
    last;
    }
else {
    redo;
    }
}
```

The interesting part of this answer is that you don't have to write special code to handle the first time that you prompt the user.

You can also label the block, if it helps you see what's happening:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

my $number = int( rand( 10 ) + 1 );

LOOP: {
print "Guess the number>> ";
my $guess = <STDIN>;

if( $guess == $number ) {
    say 'You guessed it!';
    last LOOP;
    }
else {
    redo LOOP;
    }
}
```

APPENDIX K
Answers to Chapter 11 Exercises

Answer 11.1: You don't have to know how modules work on the inside as long as you follow their interfaces (but since this is Perl, you can always look under the hood). To use the `File::Spec` module, which comes with Perl, follow the examples in its documentation (or *Learning Perl*). You can view the module documentation with **`perldoc File::Spec`**, although in real life I look at either *http://metacpan.org/* or *http://perldoc.perl.org/*.

To create a path name, use the `File::Spec->catfile` method. You don't have to understand it, you just have to use it correctly. Behind the scenes, `File::Spec` figures out which platform it is on (Unix, Mac Classic, VMS, Windows, and so on) then does the right thing for that system. This way, you can create native path names without programming all of that logic yourself. Your program can move from Unix to VMS without a change. This sort of code portability is a big advantage of Perl, and you can read more tricks such as this one in the *perlport* documentation:

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

use File::Spec;

@ARGV = "/Users/brian";

my $dir = $ARGV[0];

foreach my $file ( glob( ".* *") ) {
    print File::Spec->catfile( $dir, $file ), "\n";
    }
```

There's an alternate, function-oriented interface for `File::Spec` called `File::Spec::Functions`. Instead of calling methods (which you won't read about until *Intermediate Perl*), you can deal with them as normal subroutines:

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

```
use File::Spec::Functions qw(catfile);

@ARGV = "/Users/brian";

my $dir = $ARGV[0];

foreach my $file ( glob( ".* *") ) {
    print catfile( $dir, $file ), "\n";
    }
```

Answer 11.2: Just like you used the `File::Spec` module in your program for the Answer to Exercise 11.1, you simply have to follow the examples in the `File::Basename` documentation. The `use File::Basename` statement adds two functions to your program: `dirname`, to get the directory portion of the path, and `basename`, to get the filename portion. *Learning Perl* doesn't talk about how this works, but *Intermediate Perl* does.

Use the `dirname` and `basename` directly in the `printf` statement. You actually have two `printf` statements: the first prints column names. You want to use the same format string for each `printf`, so store that in `$format` and use that in place of a string literal as the first argument. If you want to change the format, you only have to change it in one place:

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

use File::Basename;

my $format = "%-70s %-10s\n";

printf $format, "Directory", "Name";

while( <> ) {
    chomp;

    printf $format, dirname( $_ ), basename( $_ );
    }
```

Answer 11.3: Once you have everything set up with the *cpan*, you simply tell it which modules you want to install:

```
$ cpan DBD::SQLite LWP::Simple XML::Twig
```

I chose these particuar modules for a reason. The `DBD::SQLite` provides a handy, lightweight, single-file-based database server, and it requires a C compiler to install. Some Perl modules are actually just wrappers around C libraries, although you can't tell which ones just by looking at the distribution names. If you don't have a C compiler, you'll have to get one. If you are on Windows, Strawberry Perl comes with everything you need to use the *cpan*. If you have a Unix-like system, either check your `PATH` settings, install a compiler through your operating system's packaging system, or download and

install *gcc*. If you don't control your system, you might have to make friends with the system administrator.

`LWP::Simple` provides a quick and easy interface to downloading almost anything that you can find on the Internet. You'll use it in upcoming exercises. It's so popular and widely used that it should be in the Perl Standard Library, but it's not.

`XML::Twig` takes a little bit of work to start to use effectively, but once you know it, it's very handy for the light or occassional XML tasks. Unless your main goal is to work with XML, consider using `XML::Twig` to handle your XML work. You'll need it for an upcoming exercise, too.

You might have run into problems installing these modules because you don't have permission to install them in the main Perl library directories. Instead of going through the whole explanation here, I'll point you to How do I keep my own module/library directory? and How do I add the directory my program lives in to the module/library search path? in *perlfaq8*, as well as the instructions we gave you in this chapter.

Answer 11.4: *cpanm* is an interesting case because it provides a way for it to install itself. According to its documentation, you can download it, pipe the downloaded data to *perl*, and let it do it's work:

```
$ curl -L http://cpanmin.us | perl - --sudo App::cpanminus
```

If you don't have the *curl* program, you can still install *cpanm* with *cpan*:

```
$ cpan App::cpanminus
```

Once you have *cpanm*, you can use it to install other modules.

Answer 11.5: This one is very simple because the `LWP::Simple` module does everything for you. The `getstore` function does it all:

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

use LWP::Simple;

getstore( 'http://perldoc.perl.org', 'perldoc.html' );
```

Answer 11.6: This is one of the challenging questions because you have to know quite a bit more than just Perl, and if you haven't used the `DBI` module before you might have to take some time to figure out that module as well. However, that's mostly the point of this chapter in *Learning Perl*. There's the basic syntax of Perl, which is easy to teach, but even when you know the syntax, you have only scratched the surface since a good programmer, no matter the language, has experience with the libraries. That's where all the good stuff is.

Here's a simple `DBI` program that uses some of the most common `DBI` constructs. First, create a DBI object with `connect`. Once you have that, use `do` to run a query from which you won't read the results; that's the statement to create the table. SQLite that allows

you to only create the table if it doesn't already exist, which lets you run this program repeatedly without a SQL error telling you that the table already exists. Next, `prepare` a SQL statement that you will run later: you're just getting it ready. In that `INSERT` statement, you should use `?` as a placeholder to show where you will fill in later. This is a major feature of `DBI`, which will auto-quote values for you to prevent SQL injection. Once you have your prepared statement, `execute` it for each of the key-value pairs in `%characters`. The arguments to `execute` fill in the placeholders from the `prepare`:

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

my $dbh = DBI->connect( 'dbi:SQLite:dbname=flintstones.db', '', '' );

$dbh->do( '
    CREATE TABLE IF NOT EXISTS Characters (
        id    integer,
        first    text,
        last    text
        )
    ');

my %characters = qw(
    Fred Flintstone
    Wilma Flintstone
    Betty Rubble
    Barney Rubble
    Larry  Slate
    );

my $sth = $dbh->prepare( 'INSERT INTO Characters VALUES ( ?, ?, ? )' );

my $id = 0;
foreach my $first_name ( keys %characters ) {
    $sth->execute( $id++, $first_name, $characters{$first_name} );
    }
```

To make sure that you've created your database with the right fields, you should look at the database to ensure that you have the right values. In this case, I like to use a database browser application, such as SQLite Database Browser. However, if you don't want to use one of those, you can easily see the results with another `DBI` program. Prepare a `SELECT` statement, `execute` it (without arguments because you don't use placeholders), and read one record at a time with `fetchrow_array`:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

use DBI;

my $dbh = DBI->connect( 'dbi:SQLite:dbname=flintstones.db','','' );
```

```
my $sth = $dbh->prepare( 'SELECT * FROM Characters' );
$sth->execute;

while( my @record = $sth->fetchrow_array ) {
    say join( '|', @record );
    }
```

The output shows that you have the characters in your database:

```
0|Betty|Rubble
1|Wilma|Flintstone
2|Barney|Rubble
3|Larry|Slate
4|Fred|Flintstone
```

If you had trouble with this exercise, don't fret. It's a pretty tough one for the Perl beginner. If you want to learn more about `DBI`, you might want to read *Programming the Perl DBI*; it's a bit dated, but the basics are mostly the same.

Answer 11.7: This is another challenging problem because it's more about using the modules and their interface instead of using basic Perl syntax. Although `XML::Twig` is an object-oriented module, you don't have to know much about object-oriented programming to follow the examples you see in the documentation. The trick with this module is figuring out which methods you need to use to get the job done (and there are *many* methods).

This short program gives you the flavor of a twig program. After you create the object in `$twig` and pull in my XML file with `parsefile`, you have to figure out how to traverse the XML structure to complete your task. In this case, you want to remove the `<score>` from each of the `<character>` elements.

To traverse the XML structure, first get its top-level element by calling `root`. You really want `<character>` elements though, and you get those by calling `children`. That returns a list of nodes in which each element represents a single `<character>` element. From there, the task is quite simple: all `cut_children` to remove the `<score>`:

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

use XML::Twig;

my $twig = XML::Twig->new;
$twig->parsefile( 'ex11.7.xml' );
my $root = $twig->root;

foreach my $char ( $root->children( 'character' ) ) {
    $char->cut_children( 'score' );
    }

$twig->print( pretty_print => 'indented' );
```

Output the final twig using the `indented` setting, which produces easy-to-read to output:

```
<?xml version="1.0"?>
<characters>
  <character>
    <name>Fred Flintstone</name>
    <league>Adult</league>
  </character>
  <character>
    <name>Barney Rubble</name>
    <league>Adult</league>
  </character>
  <character>
    <name>Dino</name>
    <league>Pets</league>
  </character>
</characters>
```

APPENDIX L
Answers to Chapter 12 Exercises

Answer 12.1: Go through each of the files you specified on the command line and store each name in `$file` in turn. Start by printing the file name, and use the next part of the program to decide what to print after it. This way there is only one statement that prints the filename and if you decide to change it, you only have to change it in one place.

Next, try some file test operators to decide if you need to print anything after the filename. If `-d` returns true, it's a directory and you output a `/` after the name. If it's not a directory, check the `-x` file test operator. If `-x` returns true, it's an executable file and you output an `*` after it. If that doesn't work, check `-l`, and if that returns true, the file is a symbolic link so output an `@` after the name. If none of those were true, don't output anything after the name.

No matter what you put after the filename, end the line with a newline so the next filename shows up on the next line. You get this for free with `say`. Again, you only have to print this in one place:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

foreach my $file ( @ARGV ) {
    print $file;

       if( -d $file ) { say '/' }
    elsif( -x $file ) { say '*' }
    elsif( -l $file ) { say '@' }
    }
```

Your output looks like:

```
$ perl ex12.1.pl subdir* fred*
subdirA/
subdirB/
fred1.txt*
```

```
fred2.txt*
fred_link.txt@
```

Answer 12.2: This program looks much like the one for the previous exercise. As you process each file, you use `stat` to get the access and modification times then output a line showing those. Since you only want the last two return values, you can use `undef` as placeholders for the values you don't want:

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

foreach my $file ( @ARGV ) {
    my( undef, undef, undef, undef, undef, undef, undef, undef,
         $atime, $mtime ) = stat $file;

    printf "%-15s  %24s   %24s\n",
         $file,
         scalar localtime( $atime ),
         scalar localtime( $mtime );

    }
```

Your output looks something like:

```
$ perl ~/Desktop/*
cats.webloc    Sun Aug 15 11:27:56 2010   Thu Aug 12 07:30:08 2010
Fridge.html    Sun Aug 15 11:30:11 2010   Sat Aug 14 18:33:37 2010
Moose_article Sat Aug 14 23:15:09 2010   Wed Jun  2 07:37:40 2010
bad_dist       Sat Aug 14 23:15:08 2010   Fri Jul  9 12:12:02 2010
extra_reports Sat Aug 14 23:15:09 2010   Thu Nov 19 01:59:09 2009
perldoc-1.00   Sat Aug 14 23:15:09 2010   Tue Apr 28 01:58:51 2009
```

We haven't told you about slices yet, but you can use them to simplify your code when you only want a few of the list elements that a function might return. Instead of using `undef` as a placeholder, an intermediate Perl programmer would just use the indices he wanted:

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

foreach my $file ( @ARGV ) {
    my( $atime, $mtime ) = ( stat $file )[8,9];

    printf "%-15s  %24s   %24s\n",
         $file,
         scalar localtime( $atime ),
         scalar localtime( $mtime );

    }
```

Don't bother to count the elements yourself to get the right indices. The perlfunc entry for `stat` tells you the indices.

Answer 12.3: First, without the stacked file test operators, you might write this program that tests each attribute individually, using the virtual _ filehandle to reuse the results of the previous file lookup:

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

foreach my $file ( @ARGV ) {
    if( -r $file and -w _ and -x _ ) {
        print "$file\n"
        }
    }
```

That's a bit wordy and you have to remember details about _, the virtual filehandle. With the stacked file test operators, you can condense your program a bit. You have to ensure you're using at least Perl 5.10 since you can't stack file test operators in earlier versions:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

foreach my $file ( @ARGV ) {
    print "$file\n" if -r -w -x $file;
    }
```

If you use the implicit `$_` and `say`, since you're already using Perl 5.10, your program is shorter because you can eliminate the `$file` variable:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

foreach ( @ARGV ) {
    say if -r -w -x;
    }
```

Answer 12.4: This exercise is mostly `printf` statements. For each of the command-line arguments, use the `%b` format in `printf` to create the binary string. Create the line to separate the operands and the result with the string replication operator, `x`. In the last line, you use the same `printf` statement you used first, but give it the ANDed result:

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

printf "  %20b\n", $ARGV[0];
printf "& %20b\n", $ARGV[1];
printf "  %20s\n", "-" x 20;
printf "  %20b\n", $ARGV[0] & $ARGV[1];
```

Answer 12.5: Yor solution to this exercise should be very close to your solution for the Answer to Exercise 12.1. This time, you test for a symbolic link with `readlink`. Since we didn't tell you about `readlink`, you'll have to look it up in the documentation:

```
$ perldoc -f readlink
```

You store the value `readlink` returns in `$target` then test for definedness (just in case the filename of the target is *0*). If `$target` has a defined value, output an arrow and the value of `$target`:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

foreach my $file ( @ARGV ) {
    print $file;

       if( -d $file ) { say '/' }
    elsif( -x $file ) { say '*' }
    elsif( defined( my $target = readlink( $file ) ) ) {
        say " --> $target";
        }
    }
```

Answer 12.6: This answer is very similar to the Answer to Exercise 12.2. Change it around slightly so you can select which built-in program that you want to run. Define `@stat` where you'll store the return values from `stat` or `lstat`, depending on which one you choose. Next, test the value in `$file` to check if it is a link. If it's a link, use `lstat`. Otherwise, use `stat`. Once you have the result in `@stat`, the test of the problem is the same as the Answer to Exercise 12.2:

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

foreach my $file ( @ARGV ) {
    my @stat;

    if( -l $file ) { @stat = lstat $file }
    else           { @stat =  stat $file }

    my ( $atime, $mtime ) = ( $stat[-2], $stat[-1] );

    printf "%-15s  %24s   %24s\n",
         $file,
         scalar localtime( $atime ),
         scalar localtime( $mtime );

    }
```

With a slice, you could write that line to get `$atime` and `$ctime` as:

```
    my ( $atime, $mtime ) = @stat[-2, -1];
```

You could also use the conditional operator instead of the `if-else`:

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

foreach my $file ( @ARGV ) {
    my @stat = -l $file ? lstat $file : stat $file;

    my ( $atime, $mtime ) = ( $stat[-2], $stat[-1] );

    printf "%-15s  %24s   %24s\n",
         $file,
         scalar localtime( $atime ),
         scalar localtime( $mtime );

    }
```

If you combine the conditional operator with a slice, you get a simpler program since you can remove the intermediate `@stat` variable:

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

foreach my $file ( @ARGV ) {
    my( $atime, $mtime ) =
        ( -l $file ? lstat $file : stat $file )[-2,-1];

    printf "%-15s  %24s   %24s\n",
         $file,
         scalar localtime( $atime ),
         scalar localtime( $mtime );

    }
```

This last version is more like what you would expect from a practicing Perl programmer.

Answer 12.7: This is a challenging problem for the beginning Perl programmer, so you might have taken quite a bit of time to complete it (or try to complete it). Don't worry too much about that.

Having said that, there are many ways that you might have completed this exercise. This solution sticks to what I can justify from what we have shown you in *Learning Perl* along with a couple of modules you saw in the exercise. Here's a solution:

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

use File::Basename;
use File::Find;
use Statistics::Descriptive;

sub wanted {
    my $full_path = $File::Find::name;
    my $dirname = dirname( $full_path );
```

```
        $file_count{ $dirname }++      if -f $full_path;
        $directory_count{ $dirname }++ if -d $full_path;
        $link_count{ $dirname }++      if -l $full_path;
        }

    find( \&wanted, @ARGV );

    report_stats( 'file',      %file_count      );
    report_stats( 'directory', %directory_count );
    report_stats( 'link',      %link_count      );

    sub report_stats {
        my( $label, %hash ) = @_;

        my $stats = Statistics::Descriptive::Full->new;
        $stats->add_data( values %hash );

        printf "Average $label count: %.2f\nStandard deviation: %.2f\n\n",
            $stats->mean || 0,
            $stats->standard_deviation || 0
            ;
        }
```

The `File::Find` module searches a directory structure using a subroutine that you define. It will handle all of the traversal details for you and call that subroutine once per file it finds. It puts the path to that file in the variable `$File::Find::name`. In your `wanted` subroutine, you use the `dirname` function from `File::Basename` to get the directory name. Since you're only looking to collect statistics per directory, that directory name is the key that you'll use in your acculumator hashes. Increment each hash key's value when you run into a file type that it's counting.

Once `File::Find` has visited each of the files and you've acculumated the counts in your hashes, it's time for you to report the statistics. You could write your own code to figure the mean, but it's much more fun to use `Statistics::Descriptive` since you can get much more than just the mean out of it. Just follow the examples in the documentation to create `report_stats`, which takes a label and a hash as an argument. You add all the values of the hash to the `$stats` object. From there, it's merely a matter of calling the right methods to get the statistics you want:

```
$ perl ex12.7.pl /some/directory
Average file count: 9.69
Standard deviation: 39.33

Average directory count: 3.77
Standard deviation: 10.49

Average link count: 0.00
Standard deviation: 0.00
```

When you continue on to *Intermediate Perl*, you'll learn about references, which makes this problem much easier. You can get rid of the global hashes and the repeated typing

you do to call `report_stats`. The program looks more complicated, but with that comes a lot of flexibility in the data structure when you decide to add to the program. Here's what that might look like, just to pique your interest:

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

use File::Basename;
use File::Find;
use Statistics::Descriptive;

my( $wanted_sub, $hashes_sub ) = do {

sub wanted {
    my( $file_count, $directory_count, $link_count );

    $file_count->{label}      = 'file';
    $directory_count->{label} = 'directory';
    $link_count->{label}      = 'link';

    sub {
        my $full_path = $File::Find::name;
        my $dirname = dirname( $full_path );

        $file_count->{dirs}{ $dirname }++      if -f $full_path;
        $directory_count->{dirs}{ $dirname }++ if -d $full_path;
        $link_count->{dirs}{ $dirname }++      if -l $full_path;
        },
    sub { ( $file_count, $directory_count, $link_count ) };
    }

find( $wanted_sub, @ARGV );

foreach my $hash ( $hashes_sub->() ) {
    report_stats( $hash );
    }

sub report_stats {
    my( $hash ) = @_;

    my $stats = Statistics::Descriptive::Full->new;
    $stats->add_data( values %{ $hash->{dirs} } );

    printf "Average $hash->{label} count: %.2f\nStandard deviation: %.2f\n\n",
        $stats->mean || 0,
        $stats->standard_deviation || 0
        ;
    }
```

APPENDIX M
Answers to Chapter 13 Exercises

Answer 13.1: You can use either a `glob` or `opendir` to get the number of files. In this case, you only want to report the number of files, so you can cheat a little. Use `glob` with two patterns at the same time: `.*` to find the hidden files and `*` that finds the rest. Store those in `@files`, and mixed in that list are the virtual files, . and .., that you don't want to count. To get the total, subtract two from the number of elements in `@files`:

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

my @files = glob( '.* *' );

# don't count . or ..
my $count = @files - 2;

print "I found $count files\n";
```

If you didn't use that trick, you really don't have that much more work to do:

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

my @files = glob( '.* *' );

# don't count . or ..
foreach my $file ( @files ) {
    next if $file =~ /^\.{1,2}$/;
    $count++;
    }

print "I found $count files\n";
```

You can do the same thing with `opendir`, although you need to do a little more work:

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

```
opendir my( $dh ), '.' or die "Could not open directory! $!";
my @files = readdir( $dh );
closedir $dh;

# don't count . or ..
my $count = @files - 2;

print "I found $count files\n";
```

Answer 13.2: Use your program from the Answer to Exercise 13.1 to solve this one. In this case, you use `next` to skip any file that is a directory (some operating systems use regular files as directories). Since you have to acculumate a sum of file sizes, you might as well count the files at the same time:

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

my @files = glob( '.* *' );

my( $count, $size ) = ( 0, 0 );
foreach my $file ( @files ) {
    next if -d $file;
    $count++;
    $size += -s $file;
    }

print "I found $count files with total size $size bytes\n";
```

You can also use `opendir`. In this case, you use `readdir` in a while loop so you get back one file at a time. Otherwise, this is the same thing as the previous program:

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

opendir my( $dh ), "." or die "Could not open directory! $!";

my( $count, $size ) = ( 0, 0 );
while( my $file = readdir( $dh ) ) {
    next if -d $file;
    $count++;
    $size += -s $file;
    }

closedir $dh;

print "I found $count files with total size $size bytes\n";
```

Answer 13.3: This program has the same sort of structure as the previous answers in this chapter. You start with `opendir` like you have in the other programs. Before you do anything else, you need to output a header to make things look pretty.

In the `while` loop, you go through each file. In scalar context, `readdir` returns the next file. You skip the . and .. virtual directories then use `printf` to create the report line. You do all of the work in the `printf`. Set up the format string so everything lines up under the headings you have already created (although, honestly, you make the `printf` format string first then make the header line up with the columns).

The first column is just the filename. The second column is the file size, so you use the file test operator `-s` to get the size in bytes, and that number ends up in the string. The last three fields look the same: you use the conditional operator. The file test operator returns true or false, and if it returns true, you want an `x` in the string. If it returns false, you want to put nothing into the string. If you just use the result of the file test operator, you get `1` or the empty string, which doesn't look as nice:

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

opendir my( $dh ), "." or die "Could not open directory! $!";

print "Name                                Size   R W E\n";
print "-------------------------------------------------\n";

while( my $file = readdir( $dh ) ) {
    next if $file =~ /^\.{1,2}$/;
    printf "%-30s %10d   %1s %1s %1s\n",
         $file,
         -s $file,
         -r $file ? 'x' : '',
         -w $file ? 'x' : '',
         -x $file ? 'x' : '';

    }

closedir $dh;
```

Answer 13.4: If you use `glob`, you just get rid of the `opendir` machinery. The rest of the program is the same. The `glob` function in scalar context returns the next filename, just as `readdir` does:

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

print "Name                                Size   R W E\n";
print "-------------------------------------------------\n";

while( my $file = glob( ".* *" ) ) {
    next if $file =~ /^\.{1,2}$/;
    printf "%-30s %10d   %1s %1s %1s\n",
         $file,
         -s $file,
         -r $file ? 'x' : '',
         -w $file ? 'x' : '',
```

```
            -x $file ? 'x' : '';

    }
```

Answer 13.5: Your solution is like your previous answers in this chapter, but with a few changes. After you get a filename from `readdir`, you try to match the file extensions *.pl* or *.plx*. Unless the name matches either of those extensions, you use `next` to move onto the next file. You output each line of the report like you've done previously:

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

opendir my( $dh ), "." or die "Could not open directory! $!";

print "Name                                    Size\n";
print "----------------------------------------\n";

while( my $file = readdir( $dh ) ) {
    next unless $file =~ /\.plx?$/i;
    printf "%-30s %10d\n", $file, -s $file;
    }

closedir $dh;
```

If you don't want to do this by checking the file extension, you can do a bit of work to look inside the file. With each file, try to open the file. If you can't, just move on to the next one. Otherwise, get the first line from the file and store it in `$shebang`. If you have a Perl program, you expect the first line to be `#!/usr/bin/perl` or `#!perl` or something like that. Look for the word "perl" (surrounded by word boundaries) in `$shebang`. If that doesn't match, move on to the next file. If that did match, output a report line like you've done before:

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

chdir "/Users/Buster/Tidbits";

opendir my( $dh ), "." or die "Could not open directory! $!";

print "Name                                    Size\n";
print "----------------------------------------\n";

while( my $file = readdir( $dh ) ) {
    open my( $fh ), "$file" or next;

    my $shebang = <$fh>;  #!/usr/bin/perl

    next unless $shebang =~ m/\bperl\b/;

    close $fh;

    printf "%-30s %10d\n", $file, -s $file;
```

```
    }

closedir $dh;
```

Answer 13.6: Your rewrite makes the program smaller. That's not necessarily the right reason to do anything, but sometimes it works for the situation. You don't need to open the directory and go through each file. You use the `glob` to create the list in `foreach`. The rest of the program works the same way as before:

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

print "Name                                 Size\n";
print "-----------------------------------------\n";

foreach my $file ( glob( ".* *" ) ) {
    next unless $file =~ /\.plx?$/i;
    printf "%-30s %10d\n", $file, -s $file;
    }
```

Answer 13.7: Start this program by computing the time you want to use for the access time. The `time` function returns the current time in seconds since the start of the epoch (the time when your computer starts counting time: on Unix that's January 1, 1970 00:00 GMT). Subtract 5x60 seconds to get the time 5 minutes ago. Use that time for all of the files so they will all have the same last access time when you are done.

Get the modification time from `stat`. You know that's in index 9 because that's what the *perlfunc* documentation tells me. Check it out yourself! You don't want to change the modification time, but `utime` is going to affect it. You'll just "change" it to its current value.

Finally, you use `utime` to affect the file. The first argument is the access time (five minutes ago) and the second argument is the mod time you just saved. The third argument is the filename. If you can't change the times for some reason (you don't own the files, for instance), you output an error message with `warn`:

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

my $atime = time - 5 * 60;

foreach my $file ( @ARGV ) {
    my @stat  = stat( $file );
    my $mtime = $stat[9];

    utime $atime, $mtime, $file
         or warn( "Could not set access time: $!" );
    }
```

APPENDIX N
Answers to Chapter 14 Exercises

Answer 14.1: Start the program by printing a short message reminding you to enter some lines of input. If you don't give the program any command-line arguments, the line input operator <> will wait for standard input, so you don't stare at the terminal waiting for something to happen as your program is staring at you waiting for you to type.

Use `index` on each line to look for the string "perl". If the line doesn't have it, `index` returns -1 (0 means it was the first character). Assign the value to `$pos` then check if its value was not -1, meaning you found "perl". If you found "perl", output a message that shows its position. If `index` returned -1, simply output a string that says you didn't find it:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

print "Enter some strings, and use Control-D to stop input\n";

while( <> ) {
    my $pos = index $_, 'perl';

    if( $pos != -1 ) {
         say qq(\tI found 'perl' at position $pos);
         }
    else {
         say qq(\tI didn't find 'perl');
         }
    }
```

Here's some sample output:

```
$ perl ex14.1.pl
Enter some strings, and use Control-D to stop input
This is a line without perl
    I didn't find 'perl'
This is a line with perl
    I found 'perl' at position 20
```

```
perl is at the beginning
    I found 'perl' at position 0
```

Answer 14.2: This answer is the same as your program in the Answer to Exercise 14.1 with only a change in one line. Instead of `index`, use `rindex`, and instead of "perl", look for "e". The `rindex` function starts looking for the string on the right hand side of the string, so it finds the last "e" in the string:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

say 'Enter some strings, and use ^D to stop input';

while( <> ) {
    my $pos = rindex $_, 'e';

    if( $pos != -1 ) {
         say "\tI found an e at position $pos";
         }
    else {
         say "\tI didn't find an e";
         }
    }
```

Answer 14.3: This program reimplements part of the *cut* command. By the way, the Perl Power Tools project reimplements many of the Unix utilities as Perl programs so you can use them wherever you have Perl, including on Windows. You might look at its *cut* implementation to compare it with your solution.

Start by taking the first two arguments and storing them in `$start` and `$end`. The remaining elements should be filenames, and since they are in `@ARGV`, you can use the line input operator `<>` to do the dirty work of opening them and reading lines from them.

To get the right part of the string, you have to use `substr`, and for that you need the starting position and a length, not the ending position. You get the length by subtracting the starting position from the ending one, but add 1 because the range is inclusive. For instance, if you want the characters between 5 and 10, that's 6 columns, although 10 - 5 is 5. You have to include the starting column in your count.

Once you have that, you just go through the lines. You `chomp` each line to remove all the lines in case you select part of the string that includes the end. You'll add the newline on the way out. In your `print` statement, put the substring between | characters to make it easy to see whitespace on either end of the string:

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

my $start = shift @ARGV;
my $end   = shift @ARGV;
```

```
my $length = $end - $start + 1;
print "Length is $length\n";

while( <> ) {
    chomp;
    print "|", substr( $_, $start, $length ), "|\n";
    }
```

Answer 14.4: This is a short program if you avoid using temporary variables. Start with a `glob` to get all of the filenames. Immediately send that list to the `sort`. You give the `sort` an inline code block to do the ordering. To sort by the file size, compare the result from the `-s` file test operator for the elements in `$a` and `$b`. Instead of sorting by the values themselves, you `sort` by a computed value. The output list of `sort` is still the list of files, but in ascending order of size:

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

foreach ( sort { -s $a <=> -s $b } glob( ".* *" ) ) {
    printf "%10d %-20s\n", (-s $_ ), $_;
    }
```

The first program ordered the list by ascending file size—that's the default order. To flip that around and get a descending sort, you change the order of comparison by putting `$b` on the left side of the `<=>` operator:

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

foreach ( sort { -s $b <=> -s $a } glob( ".* *" ) ) {
    printf "%10d %-20s\n", (-s $_ ), $_;
    }
```

This really isn't the best way to do this because you have to test the file size twice every time you want to compare two elements. You should only have to test any file once, but *Learning Perl* hasn't told you about the Schwartzian Transform yet. You can read about that in *Intermediate Perl*. Here's a modified version of that Transform that doesn't discard the file size so you can use it later:

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

my @sorted =
    sort { $b->[0] <=> $a->[0] }
    map  { [ -s, $_ ] }
    glob( '.* *' );

foreach my $tuple ( @sorted ) {
    printf "%10d %-20s\n", @$tuple;
    }
```

Answer 14.5: This one is a bit tricky, at least if you want to do it without much typing. You sort the files like you did in my program for the Answer to Exercise 14.4, but you use the `-M` file test operator, which gives you the file modification time in days since the start of the program. Yes, that's weird, but it means that files that were modified sooner have a lower value from `-M`. Since the default sort order is ascending, ordering by the value of `-M` means that the first element in the ordered list will be the file that was most recently modified. You don't have to change the order of `$a` and `$b` to change the direction of the sort; `-M` does it for you:

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

foreach ( sort { -M $a <=> -M $b } glob( ".* *" ) ) {
    my @stat = stat;

    printf "%24s   %-20s\n", scalar localtime $stat[9], $_;
    }
```

Answer 14.6: This exercise requires at least Perl 5.12, the version of Perl that added the ability of each to iterate through arrays. On each iteration, you get the index of the element and the value at that index. This makes it slightly easier to iterate through multiple parallel arrays. You get the value from the array that you give to each, and use the index to get the values from the other arrays. You typically use each in a `while` loop:

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

use 5.012;

my @first = qw(Fred Barney Betty Wilma Larry);
my @last = qw(Flintstone Rubble Rubble Flintstone Slate);

while( my( $index, $value ) = each @first ) {
    say "$value $last[$index]";
    }
```

APPENDIX O
Answers to Chapter 15 Exercises

Answer 15.1: This answer is going to look a lot like your answers for many of the other exercises in this chapter. Most of the code isn't about the smart match operator because that operator hides so much of the work that you're actually doing, which is what makes it so attractive for some problems.

Since the smart match operator is a Perl 5.10 feature, you have to enable that feature. One way to do that is `use v5.10.1`. This uses the major, minor, and point release numbers because you want to exclude Perl 5.10.0.

The smart match part of this problem is easy. Once you have the name that you want to match in a scalar variable, you smart match that to the array you want to search: `$name ~~ @array`. Wasn't that easy? The rest of the problem is just a simple matter of programming.

You can reuse this for the other exercises in this chapter. There's one tricky part. When you take the name from `<STDIN>`, you need to remember to remove the newline:

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

use v5.10.1;

my @array = qw(Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm);
say "The elements are (@array)";
say 'Enter a name: ';
chomp( my $name = <STDIN> );

if( $name ~~ @array ) {
    say 'I found a matching name';
    }
else {
    say "I didn't find a matching name";
    }
```

Answer 15.2: This answer is structurally similar to your program in Answer 15.1. Once you get the names in `@names`, it's a simple matter of using the smart match operator to

compare that array to your hash: `@names ~~ %hash`. If at least one of the elements of `@names` is a key in `%hash` then the smart match returns true:

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

use v5.10.1;

my %hash = qw(
    Fred      Flintstone
    Wilma     Flintstone
    Barney    Rubble
    Betty     Rubble
    Larry     Slate
    Pebbles   Flintstone
    Bamm-Bamm Rubble
    );

say 'The keys are [', join( ' ', keys %hash ), ']';
say 'Enter some names then Control-D to stop:';
chomp( my @names = <STDIN> );

if( @names ~~ %hash ) {
    say 'I found a matching name';
    }
else {
    say "I didn't find a matching name";
    }
```

There are a few other tricks in this answer. They don't really contribute to the overall task, but they are idiomatic Perl. First, the `qw()` operator automatically quotes and separates the list of elements you assign to `%hash`. When you assign a list to a hash, Perl takes the elements in key-value pairs. You can format the code to show the keys and values in columns. Next, you read in the list of names from `<STDIN>` and `chomp` the entire `@names`. We introduced both of these in Chapter 3.

Answer 15.3: As with most examples of the smart match operator, this answer is mostly not about the smart match. That's really the easy part since it's just `/$pattern/ ~~ @array`. The rest of the problem is just getting the pattern from the user.

There's a trick here. Remember that the value you get from the `<STDIN>` operator includes the trailing newline. That newline will be part of your pattern (so almost nothing will match) unless you get rid of it. Simply `chomp` the value:

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

use v5.10.1;

my @array = qw(Fred Wilma Barney Betty Larry Pebbles Bamm-Bamm);
say "The elements are (@array)";
say 'Enter a pattern:';
```

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

if( /$pattern/ ~~ @array ) {
    say 'At least one element matches';
    }
else {
    say 'No elements match';
    }
```

By this chapter, we haven't yet shown you an important Perl feature that you'll want to use when you take a pattern from the user. Since you don't know if that pattern is valid before you use it, you can wrap it in the `eval {}` that we show in Chapter 17:

```
if( eval { /$pattern/ ~~ @array } ) {
```

You don't really care if there's an error there. If it's a bad pattern, that still means that no elements match so you get the answer that you should get. If you were able to handle this without the hint, give yourself a pat on the back.

Answer 15.4: The smart match operator is a very small part of this answer, which is the point of that operator. It takes a task that used to require a lot of fiddly work and condenses it into one operation. To see how much work you save, you might want to review the non-smart match version we showed on page 252 of *Learning Perl*.

You solve the main part of this exercise with just `@first ~~ @second`. The rest of the work is what you already know from the previous chapters:

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

use v5.10.1;

say 'Enter the first list on a single line:';
my $first = <STDIN>;
my @first = split /\s+/, $first;

say 'Enter the second list on a single line:';
my $second = <STDIN>;
my @second = split /\s+/, $second;

if( @first ~~ @second ) {
    say 'The lists are the same';
    }
else {
    say 'The lists are not the same';
    }
```

In the exercise, I showed you an example run where it appears that two lists should be the same, but they aren't:

```
$ perl ex15.4.pl
Enter the first list on a single line:
dog cat bird
Enter the second list on a single line:
```

```
 dog cat bird
The lists are not the same
```

What's going on there? It's really a subtlety of `split`. Remember that `split` keeps leading empty fields, so there is really an extra element in the list you get from the string `' dog cat bird'`, which has leading whitespace. The first field is really the stuff before that first group of whitespace; that is, it's the empty string.

It's not very hard to fix this. You just have to adjust the input lines before you `split` them. There are a variety of ways to do this, but you can just remove any leading whitespace with a substitution:

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

use v5.10.1;

say 'Enter the first list on a single line:';
my $first = <STDIN>;
$first =~ s/^\s+//g;
my @first = split /\s+/, $first;

say 'Enter the second list on a single line:';
my $second = <STDIN>;
$second =~ s/^\s+//g;
my @second = split /\s+/, $second;

if( @first ~~ @second ) {
    say 'The lists are the same';
    }
else {
    say 'The lists are not the same';
    }
```

Answer 15.5: You know that you can set a topic with `given`, and you have two options. It can be the string the user entered or the secret word. Since you want to possibly use the string as a pattern, that means you need to use the secret word in `given` so you can use the string in the various `when` clauses. That's a bit of a twist because you're using the same thing in `given` each time instead of using a new element.

Once you decide to use the secret word as the topic, `given` puts its value in `$_` and you can smart match against that value. Now you just have to put your checks in the right order. First, you check if they want to give up. In that `when`, you use `last` to stop the `while`.

If the user wants to keep going, use the value in `$line` for the smart match, which does an exact match. If they guess the word exactly, you again use `last` to stop the `while`.

If the string is not an exact match, try the value in `$line` as a regular expression pattern in the next `when`. If that pattern matches, the user isn't done, but you tell them that the pattern matched and prompt them to keep going. The `while` will try another iteration.

If the pattern does not match, pick up the slack in the `default` block, telling the user that the guess did not help and prompting for another try:

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

use 5.010;

my $secret_word = 'Barney';

print 'Enter a string or pattern> ';
LINE: while( defined( my $line = <STDIN> ) ) {
    chomp( $line );

    given( $secret_word ) {
        when( $line eq 'give up' ) {
            say 'Too hard? Better luck next time!';
            last LINE;
            }
        when( $line ) {    # $_ eq $line
            say 'You guessed the secret word!';
            last LINE;
            }
        when( /$line/ ) {  # $_ =~ /$line/
            say "The secret word matches the pattern [$line]";
            print "\nEnter another string or pattern> ";
            }
        default {
            say "[$line] didn't help at all";
            print "\nEnter another string or pattern> ";
            }
        };
    }
```

If you couldn't use Perl 5.10 for this answer, the program is structurally the same but a bit more verbose per line, although a couple lines shorter, too:

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

my $secret_word = 'Barney';

print "Enter a string or pattern> ";
LINE: while( defined( my $line = <STDIN> ) ) {
    chomp( $line );

    if( $line eq 'give up' ) {
        print "Too hard? Better luck next time!\n";
        last LINE;
        }
    elsif( $line eq $secret_word ) {
        print "You guessed the secret word!\n";
        last LINE;
        }
```

```
elsif( $secret_word =~ /$line/ ) {
    print "The secret word matches the pattern [$line]\n";
    print "\nEnter another string or pattern> ";
    }
else {
    print "[$line] didn't help at all\n";
    print "\nEnter another string or pattern> ";
    }
}
```

APPENDIX P
Answers to Chapter 16 Exercises

Answer 16.1: Set the environment variable `TZ` by replacing its value in the `%ENV` hash. The value you need to set varies from system to system and, at least on a Unix machine, you use the names of files in */usr/share/zoneinfo*. For the Pacific time zone, some systems use `PST8PDT`. After that, use `exec` to turn your *perl* process into the *date* command, which has the same environment as the *perl* process. You've effectively created a wrapper that sets up the environment then executes the desired command:

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

$ENV{TZ} = 'PST8PDT';

exec 'date';
```

On a Mac OS X system, you have to use `US/Pacific`:

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

$ENV{TZ} = 'US/Pacific';

exec 'date';
```

You might try this with a time zone that doesn't exist. On some systems, you end up with GMT: the same time you get without a time zone.

Answer 16.2: When you use the backticks in a list context, you get a list of lines. Go through these lines with `foreach`. For the *ls* on my system, the first line is a header and starts with "total", so skip that one. You might need to do something different. You can `split` the line on whitespace, the default separator, and assign the list to your list of variables. The first column is the file permissions string, which you don't care about so you assign it to `undef`, which is really just a placeholder. The next column is the number of links to that file, but you don't need that either, so you assign it to `undef`,

which is just another placeholder. The next two columns are the user and group names (or numbers, in some cases), and you store those in `$user` and `$group`.

Once you have the user and group, you use those as keys in the `%users` and `%groups` hash and increment their values. These are just two accumulator hashes. Once you've gone through all the lines in the loop, report the results by going through each key, although you use `sort` to put them in "ASCII-betical" order before you let `foreach` iterate through them:

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

my( %users, %groups );
foreach ( `ls -l` ) {
    next if /^total/;   # skip the header

    my( undef, undef, $user, $group ) = split;

    $users{$user}++;
    $groups{$group}++;
    }

foreach my $user ( sort keys %users ) {
    printf "%-8s %3d\n", $user, $users{$user};
    }

foreach my $group ( sort keys %groups ) {
    printf "%-8s %3d\n", $group, $groups{$group};
    }
```

Answer 16.3: Instead of using backticks, you use the `IPC::System::Simple` module. This module does not come in the Perl Standard Library (yet), so you might have to install it yourself.

The `capturex` routine can run an external command and return a list of lines just like backticks, so that's what you use. For `capturex`, you have to separate the command name from its arguments because the first argument is the complete command name. You probably don't have a command named exactly *ls -l*, so you use two arguments:

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

use IPC::System::Simple qw(capturex);

my( %users, %groups );
foreach ( capturex( 'ls', '-l' ) ) {
    next if /^total/;   # skip the header

    my( undef, undef, $user, $group ) = split;

    $users{$user}++;
    $groups{$group}++;
```

```
    }

foreach my $user ( sort keys %users ) {
    printf "%-8s %3d\n", $user, $users{$user};
    }

foreach my $group ( sort keys %groups ) {
    printf "%-8s %3d\n", $group, $groups{$group};
    }
```

Answer 16.4: This is the program I have written the most in my career as a Perl programmer. A lot of people write programs once and forget about it, but whenever I start working with a new web server, I type in this little program as not only a test, but to see how the web server is set up and which environment variables my CGI program can use.

To see the environment variables, go through the keys in `%ENV`, and `sort` them to put them in order. You output the key and its value:

```
#!/usr/bin/perl

foreach my $key ( sort keys %ENV ) {
    printf "%-30s %s\n", $key, $ENV{$key};
    }
```

To turn this into a CGI program, you just need to print a CGI header. You can use `text/plain` so you don't need to add a bunch of HTML mumbo jumbo. The plain text version is just as easy to read:

```
#!/usr/bin/perl

print "Content-type: text/plain\n\n";

foreach my $key ( sort keys %ENV ) {
    printf "%-30s %s\n", $key, $ENV{$key};
    }
```

Give yourself a nice pat on the back if you used `CGI.pm`, but take it back for doing too much work for such a simple task.

Answer 16.5: To remove the value from the `PATH` environment variable, you just set its value on `%ENV` to the empty string. When you want to use `system`, backticks, `exec`, or anything else that executes an external command, you need to give it the full path. Perl doesn't have any directories in `PATH` to look in to find the command:

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

$ENV{PATH} = '';
$ENV{TZ}   = 'US/Pacific';

exec '/bin/date';
```

Changing the PATH environment variable ensures that you run the command that you want. Otherwise, someone can change the PATH and Perl will execute the first program it finds with the same name. They don't have to "crack" your program because they change the behavior by changing the environment. I've called this the "loaded path attack". If you worry about this sort of thing, you can check out Perl's taint mode, which offers more sophisticated protection.

Answer 16.6: This program is mostly the same as your program from the Answer to Exercise 16.2 except for the first couple of lines. I use open with the command as the filename, and you follow the filename with the pipe "|". Since the pipe comes after the command, I'll be able to read from this filehandle. I use that filehandle in a while loop to read one line at a time. The rest of the program is the same.

You could have used the foreach loop, but that would have read all the lines before it started to loop through them. The while loop deals with one line at a time.

You also don't check the return value of open, which is almost always true for a pipe open. The return value is the result of Perl forking the process, not running the command. Perl can almost always run another copy of itself since it is already running, so the open almost always succeeds (unless you've reached a process or resource limit, for instance). If you want to check the result of running the command, you need to check the return value of close and look in the Perl special variable $? for the error. You don't show all that here, though, because this program is not that important:

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

open my $ls, '-|', 'ls', '-l'; # three+ argument form

my( %users, %groups );
while( <$ls> ) {
    next if /^total/;

    my( $perm, $n, $user, $group ) = split;

    $users{$user}++;
    $groups{$group}++;
    }

foreach my $user ( sort keys %users ) {
    printf "%-8s %3d\n", $user, $users{$user};
    }

foreach my $group ( sort keys %groups ) {
    printf "%-8s %3d\n", $group, $groups{$group};
    }
```

You can write the open is a couple of other ways. You might have combined all of the parts of the external command into a single string:

```
open my $ls, '-|', 'ls -l';
```

You can also write this in the two argument form, although like most two argument forms, we discourage this:

```
open my $ls, 'ls -l |'; # two argument form
```

Answer 16.7: Open the filehandle `$input` to read from your answer for Exercise 16.4. With a `while` loop, read through lines of input. Check for the "e" character in the line with a regular expression. If the line does not match, skip it. If it does match, continue with the loop and increment the value in `$sum`. After you've gone through all of the lines, output a report of the number of lines you counted:

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

open my $input, '-|', 'perl', '16.4.pl';

my $sum = 0;
while( <$input> ) {
    next unless /e/;
    $sum++;
    }

print "The number of lines with an 'e' is $sum\n";
```

You can also write this without the `next`, which in this case is a lot simpler, but then you wouldn't get to practice using `next`:

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

open my $input, '-|', 'perl', '16.4.pl';

my $sum = 0;
while( <$input> ) {
    $sum++ unless /e/;
    }

print "The number of lines with an 'e' is $sum\n";
```

You don't have to explicitly use the *perl* in that `open`. Although we didn't cover this in the book, the Perl special variable `$^X` holds the path to the *perl* interpreter running the current program. You could have written:

```
open my $input, '-|', $^X, '16.3.pl';
```

Answer 16.8: To start a command shell, use the `Run...` from the Start menu. Run either the *cmd* or *command* program. From there, it's a simple matter of calling the *title* with `system`. Each time you call *title*, you give it a string of the right length that you construct with the string replication operator, `x`:

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

```
foreach ( 1 .. 60 ) {
    system 'title', '*' x $_;
    sleep 1;
    }
```

APPENDIX Q
Answers to Chapter 17 Exercises

Answer 17.1: To repeatedly ask the user for input, wrap my program in a naked block and use `redo` to start another go-around when you finish one round. To break out of the block, you end input with Control D (or Control Z on Windows), in which case `$string` gets `undef` and triggers the `last` function to end the loop.

There are a couple of ways you could handle the possible error from the user input, and in this answer you'll use an `eval {}` block (Answer 17.2 is the same problem with a different method). The `eval` will catch the error from an invalid regular expression. You need a semicolon after the `eval {}` block because this is a statement, not a control structure like `foreach`. After the `eval` block, output an error message if there is something in the `eval` error variable `$@`:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

{
print 'Enter a string: ';
chomp( my $string = <STDIN> );
last unless defined $string;

print 'Enter a regex: ';
chomp( my $regex = <STDIN> );

eval {
    if( $string =~ /$regex/ ) {
         say 'The string matches!';
         }
    else {
         say 'The string does not match.';
         }
    };
say "Could not use regular expression: $@" if $@;

redo;
}
```

Answer 17.2: This answer is almost the same as the one for Answer 17.1, but you have to install this module on your own since it doesn't come with Perl.

You wrap the problematic code in a `try` block. Instead of checking the result by examining the `$@` variable, you put a `catch` block right after the `try`. Notice that there is no semicolon between the end of the `try` block and the start of the `catch`; these are part of a single statement. You do need a semicolon after the `catch` block though. If you forget that final semicolon, weird things may happen because the `Try::Tiny` syntax actually allows for more stuff to follow:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

use Try::Tiny;

{
print 'Enter a string: ';
chomp( my $string = <STDIN> );
last unless defined $string;

print 'Enter a regex: ';
chomp( my $regex = <STDIN> );

try {
    if( $string =~ /$regex/ ) {
         say 'The string matches!';
         }
    else {
         say 'The string does not match.';
         }
    }
catch {
    say "Could not use regular expression: $@";
    };

redo;
}
```

Answer 17.3: This answer is similar to the example on page 289 of *Learning Perl*, although you need a different expression to filter the input list. In this case, you want the numbers that are divisible by three, so you want the elements for which the modulus operator returns 0 (because there is no remainder). The `grep` function only passes through the elements for which the expression evaluates to a true value, so you use the logical negation operator (`!`, from Chapter 2) to flip the value from false to true (and true to false). You have to put the modulus operator in parentheses because it has a lower precedence than the negation operator, but you want it to happen first. Only the multiples of three show up in `@threes`:

```
my @threes = grep { ! ( $_ % 3 ) } 1..1000;
```

Answer 17.4: The `map` function has two forms, and you can use either one of them for my solution. The first way uses a single expression to create the element that goes into the output. You put a comma between the expression and the input list:

```
my @squares = map $_**2, ( 1..10);
```

The other way uses a block of code. You don't put a comma after the inline code block:

```
my @squares = map { $_ ** 2 } ( 1..10 );
```

Answer 17.5: You could do this in a lot of different ways, but given that everything is already in a single string, leave it like that and use one regular expression to pull everything out at once.

Create a regular expression to match non-whitespace characters at the beginning of each line. At the end of the `s///` operator, add the `/m` flag so that the beginning of string anchor, `^`, can match either at the true beginning of the string, or immediately after any newline. And since you want to affect each line beginning, add the `/g` flag to do the global match-and-replace.

For the replacement, you can use the value in `$1`, the match memory variable, but preface it with the double-quoted string sequence `\u\L`, which lowercases everything after it and then makes the first letter uppercase. This is like the example on page 158 ("Case Shifting") of *Learning Perl*:

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

my $string = "fRED has score 230
barney has score 190
DINO has score 30";

my @words = $string =~ s/^(\S+)/\u\L$1/mg;

print $string;
```

Answer 17.6: In each case, you use an array slice to select multiple elements of the array simultaneously. To select the first and last elements, you can use the subscripts 0 and -1 (or `$#numbers`). It doesn't matter which order you extract the elements, so to select the last and first element, just reverse the order of the subscripts and use -1 (or `$#numbers`) and 0.

To select the first 10 elements, create the list of indices using the range operator, `0..9`: remember that the first index is `0`! That last one is there to remind you that you can use an expression to create the indices.

To select the odd numbers, you could have simply used a `grep` just as you did earlier in this section, but now you have to use an array slice. There are probably some interesting ways to do this, but you simply move the `grep` inside the array subscript and pick the indices where you know the odd numbers will be (which are just the odd indices).

You probably shouldn't do that in a real-world Perl program though, unless you really want to baffle someone:

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

my @numbers = 0 .. 1000;

my ( $first, $last ) = @numbers[0,-1];
print "first is $first and last is $last\n";

   ( $last, $first ) = @numbers[-1,0];
print "first is $first and last is $last\n";

my @first_ten        = @numbers[ 0 .. 9 ];
print "The first ten numbers are @first_ten\n";

my @odd_numbers      = @numbers[ ( grep { $_ % 2 } 0 .. $#numbers ) ];
print "The odd numbers are @odd_numbers\n";
```

Answer 17.7: You only have to change one line in the program. First, create the hash using a hash slice. You already have the keys in `@names` and the values in `@scores`. In the hash slice, use those arrays in the right places. The array with the keys goes in the `{ }`, and you assign to the hash slice the array holding the values. The first value goes with the first key, the second value with the second key, and so on:

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

my @names  = qw(Fred Barney Dino);
my @scores = qw(230   190    30 );

my %scores;
@scores{ @names } = @scores;

foreach my $name ( sort keys %scores ) {
    print "$name has score $scores{$name}\n";
    }
```

You can also create the hash with a `map`, and you will often see this technique in real-world Perl code since the `map` can return more than one value (say, a key and value pair!) for each input element. In this case, you use the list of array indices for `@names` as the input list, and for each index, you return a list of two elements: an element from `@names` and an element from `@scores`. Taken all together, you end up with an output list of key-value pairs, just like you saw in Chapter 6 of *Learning Perl*. The `map` function is an everyday tool of the expert Perl programmer, and you can read more about it in *Intermediate Perl*:

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

```
my @names  = qw(Fred Barney Dino);
my @scores = qw(230   190    30 );

# CREATE THE HASH!
# %scores = (); #fill in here

my %scores = map { ( $names[$_], $scores[$_] ) } 0 .. $#names;

foreach my $name ( sort keys %scores ) {
    print "$name has score $scores{$name}\n";
    }
```

Answer 17.8: The `List::Util` module provides convenience subroutines for frequent list tasks. These subroutines also run a little bit faster than the equivalent Perl code that you might write since they are implemented at a lower level. The actual code is thus very easy. For `sum`, you just give it the list of numbers, in this case a range, and it returns the number you need:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

use List::Util qw(sum);

say 'The sum of 1 to 1000 is ', sum( 1 .. 1000 );
```

The `reduce` example is a little more complex. It's an iterating block like `map` or `grep`, but it takes the first two items from the input list, giving them to you as `$a` and `$b`. At the end of the block, unlike `map` or `grep`, `reduce` puts its result back onto the list. That means the result of the previous operation is also the first item for the next one! It continues to do this until it has reduced the list to one item, which it uses as the final result:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

use List::Util qw(reduce);

say 'The sum of 1 to 1000 is ', reduce { $a + $b } 1 .. 1000;
```

When you run this, you're probably going to get some warnings about `$a` and `$b`:

```
Name "main::b" used only once: possible typo at reduce.pl line 8.
Name "main::a" used only once: possible typo at reduce.pl line 8.
The sum of 1 to 1000 is 500500
```

You have to use those two variable names, and you can't make them lexical variables. These are really the same `$a` and `$b` that you saw in `sort`. One trick is to simply declare them as package variables so they show up in the source code twice:

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

```
use warnings;
use 5.010;

use List::Util qw(reduce);

our( $a, $b );
say 'The sum of 1 to 1000 is ', reduce { $a + $b } 1 .. 1000;
```

Answer 17.9: This answer is similar to the one for `reduce` in Answer 17.8. The `pairwise` subroutine takes two arrays, though, and puts the next element from the first array in `$a` and the next element from the second array in `$b`. It returns the result in the output list:

```
#!/usr/bin/perl
use strict;
use warnings;
use 5.010;

use List::MoreUtils qw(pairwise);

my @m = ( 1, 2, 3 );
my @n = ( 4, 6, 8 );

our( $a, $b );
my @c = pairwise { $a + $b } @m, @n;

say "The sums are ( @c )";
```

Many of the subroutines in `List::MoreUtils` have these fancy tricks with multiple arrays. Just don't ask how they do it!

APPENDIX R
Answers to Chapter 18 Exercises

Answer 18.1: The first program just needs to create the hash. Define the hash in the `dbmopen` function. Give the DBM file the name "birthdays" but don't use an absolute path so `dbmopen` creates the file in the current directory. You may see a file extension added to the filename, and that extension depends on the particular DBM implementation on your system. In some implementations, you may even see a second, index file. As long as you use the same filename in every program (in this case, "birthdays"), you don't need to think about how the file lives in the filesystem.

For the third parameter to `dbmopen`, give it a file permissions mask to use if `dbmopen` needs to create a file that doesn't already exist. This number must be an octal number, so start it with a leading `0` to tell Perl that it's a base-8 number. If you can't open the DBM hash, use `die` to output an error message. You might not be able to create new files in the current working directory, for instance.

After you define the hash, add keys and values to it just like any other hash. You can treat a DBM hash just like a normal hash. Except for the `dbmopen`, you shouldn't see any differences in the code. Traditionally, Perl programmers uppercase the names of the variable they create with `dbmopen`:

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

dbmopen my %BIRTHDAYS, "birthdays", 0644
    or die "Could not open birthdays! $!";

%BIRTHDAYS = (
    Fred    => 'April 5',
    Wilma   => 'October 26',
    Pebbles => 'October 8',
    Barney  => '',
    Slate   => undef,
    );

dbmclose %BIRTHDAYS;
```

End with dbmclose to ensure that your program writes to disk any pending changes to %BIRTHDAYS. It's good to be safe, sometimes.

For the second program, start just as you did in the first. You want to open the same DBM hash. This time, you don't want to create the file if it doesn't already exist, so you use undef as the third parameter. Perl will not create the file if it does not exist, so you'll have to deal with that situation on your own.

To go through the hash, use each. You could use foreach, but you typically expect a DBM hash to be large and you don't want to wait for Perl to load the entire disk file into memory before it does anything. Through the DBM implementation, you can get one key-value pair at a time. You don't have to eat up a lot of memory to go through a large hash because you deal with one pair at a time:

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

dbmopen my %CHECK, "birthdays", undef
    or die "Could not open birthdays! $!";

while( my( $name, $birthday ) = each %CHECK ) {
    print "$name ==> $birthday\n";
    }

dbmclose %CHECK;
```

Answer 18.2: Start your program by creating the DBM hash like you did in Exercise 18.1. In this case, you use 0644 as the third parameter to dbmopen in case the file doesn't already exist and you are adding my first element.

Prompt for the name and birthday and chomp each answer to remove the trailing newlines. Once you have the chomp-ed versions, add it to the hash like any other single element hash assignment:

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

dbmopen my %BIRTHDAYS, "birthdays", 0644
    or die "Could not open birthdays! $!\n";

print "Enter a new name: ";
chomp( my $name = <STDIN> );

print "Enter the birthday for $name: ";
chomp( my $birthday = <STDIN> );

$BIRTHDAYS{$name} = $birthday;

dbmclose %BIRTHDAYS;
```

Answer 18.3: The results of your program will give different results based on the architecture. It's not the operating system, but the actual processor that matters. First, here's the program. You use the `N` (network order), `V` (VAX order), and `I` (integer) formats. Use the uppercase versions because you want unsigned integers. Also use the hexadecimal representation for `$i` and add some underscores so you can easily see where the bytes end up:

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

my $i = 0x12_34_56_78;

my $packed_n = pack( "N", $i );
my $packed_v = pack( "V", $i );
my $packed_i = pack( "I", $i );

printf "unpacked N -> V %X\n", unpack( "V", $packed_n );
printf "unpacked N -> N %X\n", unpack( "N", $packed_n );
printf "unpacked N -> I %X\n", unpack( "I", $packed_n );

printf "unpacked V -> N %X\n", unpack( "N", $packed_v );
printf "unpacked V -> V %X\n", unpack( "V", $packed_v );
printf "unpacked V -> I %X\n", unpack( "I", $packed_v );

printf "unpacked I -> N %X\n", unpack( "N", $packed_i );
printf "unpacked I -> V %X\n", unpack( "V", $packed_i );
printf "unpacked I -> I %X\n", unpack( "I", $packed_i );
```

On my Mac OS X PowerBook that has a PowerPC processor (I wrote this a long time ago!), I see big-endian results. As long as I `unpack` the number the same way that I `pack`ed it, I get the same number back. On the PowerBook, when I unpack the `I` format with the `N` (network, or big-endian) format I get the right number back, but when I `unpack` it with the `V` (VAX, or little-endian) format, the bytes get flipped around. Look closely—it's not just in reverse:

```
unpacked N -> V 78563412
unpacked N -> N 12345678
unpacked N -> I 12345678
unpacked V -> N 78563412
unpacked V -> V 12345678
unpacked V -> I 78563412
unpacked I -> N 12345678
unpacked I -> V 78563412
unpacked I -> I 12345678
```

My FreeBSD box, which has an Intel processor, shows the results on a little-endian machine. In this case, I get the wrong answer when I `unpack` the `I` format with the `N` format:

```
unpacked N -> V 78563412
unpacked N -> N 12345678
unpacked N -> I 78563412
```

```
unpacked V -> N 78563412
unpacked V -> V 12345678
unpacked V -> I 12345678
unpacked I -> N 78563412
unpacked I -> V 12345678
unpacked I -> I 12345678
```

I don't know what you are using, so I can't tell you what output you'll get.

Answer 18.4: Your program for the Answer to Exercise 18.3 has most of the stuff you'll need to know to figure out this problem. If you spent several hours creating a hash table of processor types and their characteristics, you can still upload it to CPAN. For this problem, you can `pack` an integer then `unpack` it with the `N` format. If you get the same number back, you are on a big-endian machine. Everything happens in the `print` statement. The second line of the `print` uses the conditional `?:` operator. If the part before the `?` evaluates to true, it chooses the first option (the thing between the `?` and `:`) and if it's false, it chooses the other option:

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

my $i = 0x12_34_56_78;

print "This machine is ",
    $i == unpack( 'N', pack( 'I', $i ) ) ? 'big' : 'little',
    " endian\n";
```

You could have done it the other way, using the `V` format, too. You just flip around the strings you output. You could have also used the `!=` operator instead of the `==` if you were really lazy:

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

my $i = 0x12_34_56_78;

print "This machine is ",
    $i == unpack( 'V', pack( 'I', $i ) ) ? 'little' : 'big',
    " endian\n";
```

Answer 18.5: Use Perl's in-place editing facility to do this. The command line actually has quite a bit going on:

```
perl -i.bak -pe 's/e/E/g' filename
```

The `-i` switch takes an optional file extension. When you use the file extension, *perl* creates a backup file with that extension so you can recover the original data when (not if!) you mess up. The `-e` switch tells *perl* that you've typed out the program on the command line, and that's the `s/e/E/g`. The `-p` flag tells *perl* to wrap `while( <> ) { …; print;}` around the program. Finally, you give *perl* the file you want to modify.

The program on the command line is the same as this short program:

```
#!/usr/bin/perl
$^I = ".bak";
@ARGV = qw( filename );

while( <> ) {
    s/e/E/g;
    print;
    }
```