Introduction to Programming Using Perl

Don Colton
Brigham Young University Hawaii

June 10, 2009

Contents

Chapter 0

Preface

This book is available free online at http://ipup.doncolton.com/ in a PDF version.

0.1 Why This Book?

This book is different from many other books because of its focus on the question **why**. Many other books focus on the **how** of programming, expecting that the why will be obvious. For the authors themselves and for advanced readers, maybe the why **is** obvious. Fish are the last to discover water because they take it for granted.

If you already know why, you can skip ahead to Passing the Class (section 0.5, page 19) or chapter 1 (page 27) where we write our first program.

This book also explains how, but not totally. It explains enough of the how that you can do it, but it does not dig deeply into the intricate details of what you can do. Those details are an important study. There are almost endless websites and blogs where you can see examples of this or that. I find them very valuable. Use them. You will gain new insights. I know I do.

But the key question for this book is **why**. I illustrate with situations and examples that try to motivate different topics. I try to answer the question, "who cares?" In the end, I hope that you will come away with a better concept of what is behind the scenes that makes each topic interesting or useful, and not merely a skill at doing something that you are not yet sure you care to do.

This is probably your first programming class. You may be taking it simply because you have to. It's a required class. I hope to win you over into seeing how programming can be a fun and useful tool in your life.

0.2 Why Programming?

Computers are fast, powerful, and inexpensive. They make great slaves. The challenge is how to instruct them. Programming is the process of telling computers how to act.

This book teaches the fundamentals of programming. When you complete this course, you will be writing useful programs that run on the desktop or on the Web, and you will know whether programming is fun for you and worth pursuing with bigger projects and/or additional languages.

The book is designed to support an introductory (100-level) college course that meets for about forty hours during a semester or quarter.

We do not assume any prior programming knowledge or experience. We do assume you have math skills to do simple algebra (solve x - 7 = 3) and that you have access to a computer with Perl installed. Section 0.7 (page 24) explains how to install it.

0.3 Why Perl?

Many students will only learn one computer language. Perl seems like a good choice because it is well known, powerful, portable, typical, well supported, and freely available.

Well Known: By this I mean employers have heard of it. It has been in many top-ten lists of popular programming languages. It has been around since 1987 (four years before Linux).

Powerful: By this I mean useful programs can be written fairly easily. It has been called the Swiss Army Knife (or even Swiss Army Chainsaw) of programming languages.

Portable: By this I mean programs written for the Linux **platform** will also work on Microsoft or Macintosh and vice versa. In fact, Perl programs run on dozens of computer platforms.

Typical: By this I mean that skills in Perl transfer well to other programming languages such as PHP and Java. It is very similar to many of the languages that were invented after it.

Well Supported: By this I mean there is a large user community that is actively helping each other and there are large collections and archives of program libraries available to everyone.

Freely Available: By this I mean it can be easily found and downloaded from the Internet without paying any extra money or licensing fees.

0.4 Coverage

I wrote this book for two main purposes. (1) Cover the material I think is most important when learning to program. (2) Give students a free alternative to buying a dead-tree edition of a textbook.

This book covers the minimum information needed to pass the class (section 0.5, page 19), plus some introduction to other material frequently asked by students. This book is not intended to teach every topic in depth. Instead, it touches on the main points. The Internet of today is a huge and wonderful resource, powered by those very same computers we seek to tame through programming. We assume the students of today are skillful at using search engines and other tools to get in-depth answers on topics of interest to themselves, once they know the topics exist and what they are called.

Some students will find that having a dead-tree edition of some book on Perl is a great investment. You may not have the Internet while traveling. You may not want to curl up with a computer when you could curl up with a book. (Okay, maybe nobody would want to curl up with a book on computers.) If you are a book kind of person, you are encouraged to find one you like and read it. Or print this one out.

0.5 Passing the Class

This section is written mostly for my fellow teachers. Students can listen in.

What exactly is enough to pass the class? Each teacher seems to develop his or her own ideas about what is important. Is effort enough when performance is lacking? I also teach advanced classes and I have learned that

giving a C or higher for effort is a really bad idea. It just delays their failure by a semester. I waffle on D. But without the basic skills to go forward, you might as well give students an F.

Our department uses a philosophy that each course has an owner although other faculty may teach it. I am currently the owner for the class that uses this book (CIS 101, Beginning Programming) but I do not personally teach every section. The owner is responsible for establishing the performance levels required of students. The department approves those levels. Individual teachers can decide day to day what they will do, but they cannot compromise the performance targets.

0.5.1 Basic Expectations

Our standard is that basic material must be mastered so later courses can build on it. This is not a survey course or a high-level overview like, say, Art Appreciation. It is a "do it" class like, say, Drawing.

Mastery is divided into five topics: basics, decisions, loops, arrays, and subroutines. By the end of the semester, students must master the first two topics to pass the class (with a D). They must master the first three to get a C. They must master all five to get a B. They must also demonstrate ability with some advanced material to earn an A.

Basics: Students write correct programs that use standard input and output to get information into and out of the computer. Programs run from a Graphical User Interface (**GUI**) or from a Command Line Interface (**CLI**). Students demonstrate the ability to use normal (scalar) variables to do calculations such as inches to centimeters. Students use fundamental mathematical operators including add, subtract, multiply, divide, and parentheses. Students understand that statements are executed in order, one after an other, and that later statements can change the values of variables from what the earlier statements established. We introduce style rules including naming of variables and spacing of written programs.

Decisions (If/Elsif/Else): Students write correct programs that deal differently with alternate cases, such as whether to put AM or PM after the time, or whether a check will be honored or will bounce. This includes skill with **Boolean** operators (those yielding a **True** or **False** answer) such as comparatives (less than, greater than, equal to, not equal to) and conjunctives (and, or). This also includes following style rules of indentation and

spacing to make complex programs more readable.

Repeated Actions (Loops): Students write correct programs that deal with repetition of actions, such as filling out a table. Style is also emphasized. Operators like `++` and `+=` are mastered. `next`, `last`, and `redo` are introduced.

Repeated Data (Arrays): Students write correct programs that deal with lists of information. The `foreach` loop is mastered. `push`, `pop`, `shift`, `unshift`, and indexing (`[1]` and `[-1]`) are mastered.

Organizing (Subroutines): Students write correct subroutines to better organize and structure their code. Local and global scope is understood.

0.5.2 Proving Mastery

To prove mastery students are given a Final Exam that has five sections (one per topic area) each with several programs of varying difficulty. Students must correctly write those programs in a topic area before we consider it to be mastered.

Because students do not all learn at the same rate, we do not care when students demonstrate mastery as long as it is by the last day of class. The jury is still out on how much this simply invites students to procrastinate.

The course grade is based almost totally on the final exam, so little else really matters. This kind of takes the teeth out of midterms and homework assignments. What do we do about that? An all-or-nothing final can be pretty scary. So we compromise by giving ...

The Early Final

About once a week we offer an actual final exam. The questions are different each time, but are basically the same or of the same difficulty. The rule is that if a student passes any section of the exam, they don't have to take that section again. This gives them a reason to take the tests and to make progress. The entire final is probably too much to take on the last day of class. Knowing that part of the final is completed is a good motivator.

Because each Early Final is actually the real final, all the normal rules apply. The exams are closely proctored and performance must be at a final exam level. Toward the start of the semester very few students pass anything. Toward the end many students are passing things.

The unit tests throughout the book give actual test questions that have been used in the Early Finals to assess mastery of each topic. Students are also allowed to keep a copy of the exam and the work they did. One day later they can share their efforts with each other.

During the exams we allow the students to test their work by running it at their local machine. However, they are not allowed to use any notes or outside resources, including web pages. They are only allowed to test their programs by running them locally. If there are reference materials we wish to make available, we put them in the test itself.

Scoring: We grade programs "by hand," visually examining the student code. In addition to working, we expect student programs to demonstrate the requested programming style (indentation, spacing, comments, naming) to make the programs easy to read and understand.

We give ten points per problem for a perfect or near-perfect solution. By near-perfect we mean the program still performs its function but there is some trivial failing or variation, perhaps a minor style issue, that we overlook in the grading.

We rarely give eleven points for a surprisingly better than expected solution. Perhaps the solution shows a clever or highly efficient approach beyond what was covered in class. 11 is a serious atta-boy.

We give nine points for a minor mistake, possibly an obscure case that was not handled, or wording variations where the problem was specific but the student did something different. In all cases it must be clear to us that the student understands the material even though they may have misunderstood the question or not followed instructions exactly. Style violations can knock a 10 down to a 9 but not usually lower.

We require a score of 9 or higher on all three programs on a given topic at a given sitting before we recognize the student for mastery of that topic. 8 and below is encouragement but not completion.

Note: Partial topic scores are not cummulative. Passing two loop questions one week and another two the next week is not enough. Students must pass all three questions in one sitting. That is because the questions are of varying difficulty, including one that is hard. Passing the two easy questions twice is not as impressive as passing the hard question also.

We give eight points for a small but substantial mistake. Eight means the answer is very promising but not quite close enough to claim mastery. Typ-

ical errors might be using == when eq was needed, or reading in a value and then accidentally destroying it before using it.

From there, scores range down to zero, which means the program is really not even close to starting to solve the problem. At zero we might find programs that prompt for the inputs but do not do anything with them.

Review: Frequently I use the projector to review Early Final answers submitted by students. Answers are kept anonymous while they are reviewed by the whole class immediately after the test, which is given the first twenty minutes of class. This lets me see how the students are doing. It lets students see what good and bad answers look like. This also allows the faster students to keep moving and helps them understand how to tutor others.

0.5.3 Advanced Expectations

Some advanced skills must be demonstrated to earn an A. This is not mastery like the basics. Instead it involves creation of a working project. Ideally the project is something fun like a game or a service that students can share with their friends. Ideally students can inject their own creativity into their project.

I like to use the same project for all my classes of a given semester. This helps the end-of-semester lectures retain their relevance to the largest possible audience. I use something different the following semester. This reduces plagiarism from past students.

Tk: (advanced) Students write correct Tk-based event-based graphical programs that will run on their PC, Mac, or Linux machine.

CGI: (advanced) Students write correct web-based programs using the standard Common Gateway Interface (CGI) for event-based processing. (We do not introduce cookies. We keep state to a minimum.)

HTML: (advanced) Students use standards-compliant hyper-text markup language (HTML) and cascading style sheets (CSS) to present their web-based programs. This is done at a very introductory level and assumes no prior background in building web pages.

Database: (advanced) Students use structured query language (SQL) to store and retrieve data used by their programs.

0.6 Advanced Material*

I get questions in class. Most of the time they are about things everyone in a 100-level class should understand. Sometimes the questions are more advanced. I answer them but I do not expect everybody to understand. Some of that material has made its way into this book.

Advanced sections are marked in several ways. First, larger sections that stand alone are placed in the appendices.

Second, smaller sections that really belong with other things are given a star in their section title. Normal Distribution* would be an example of that. I may talk about it when I talk about random numbers, but I do not expect everybody to understand or even be interested.

Third, individual paragraphs may contain wording to let the reader know that the material is advanced and can be skipped.

0.7 Installing Perl on Your Personal Computer

Seach for Perl on the web. A good search string might be "install perl" or "download perl". You can find lots of good advice and tutorials on this simple process.

For Microsoft PCs, ActiveState.com has a nice and free Perl distribution. Download it to your PC and run the installer. Perl program file names must end in `.pl`.

For Macintosh and Linux, Perl is probably already installed. Get a terminal window and type `which perl` to see if anything is installed.

0.8 Suggestions?

Currently this book is a work in progress. If you find things I could improve (maybe typographical errors or things that are unclear to you), please let me know. Maybe I can rework a paragraph, page, section, or chapter so it will help you and others understand things better.

0.9 Exercises

Answers to starred exercises can be found on page 282.

Exercise 1:* Is Wikipedia a good source of information about programming?

Exercise 2:* Is Google a good source of information about programming?

Exercise 3:* What year was Perl invented?

Exercise 4:* What are the top ten programming languages?

Exercise 5:* What does a star mean in a chapter or section title?

Unit I

Basics

Chapter 1

Output: Hello, World!

Let's get started. We will write a program first and talk about it later.

1.1 Our First Program

Our first programming task is to get the computer to say hello.

Here is the first version of the program. On Microsoft Windows you can use a text editor like Microsoft Notepad (or Notepad++) to key in the program. Save it as "hello.pl".

```
print "Hello, World!";
```

That's it. One line. In section 1.5 (page 29) below we explain it in detail.

1.2 Making it Run

We want to run our program. Let's assume we are using a Microsoft Windows computer (operating system) and that Perl is installed. Then we can use a text editor like Microsoft Notepad to key in the program and save it. When we save it, the file name must end with `.pl` (dot pee ell). That ending indicates that the file is a Perl program. Some computers will also allow `.perl`.

Let's assume you save your program to your desktop. After saving your program you can run it by double-clicking it. The screen should flash briefly. What happened? The program ran. What did it do? It printed the words "Hello, World!". Or rather, it opened a window (probably black), ran the program, displayed the words on the window, and then closed the window, all so fast that you could barely tell anything was happening.

1.3 Making it Wait

Let's add another line to the program. This line is intended to make the computer wait before closing the window. We will do this by preventing the program from ending too quickly.

```
print "Hello, World!";
$wait = <STDIN>;
```

In this case our program has two statements. The first will print the words "Hello, World!". The second will request input. Since it is waiting for input we can see what it says.

Go ahead. Try it.

1.4 Making it Wait Temporarily

Let's try one more thing before we move on. We can make the computer wait a certain amount of time before closing the window. We will do this by delaying the program from continuing.

```
print "Hello, World!";
sleep 10;
```

The `sleep 10` statement causes the computer program to pause for ten seconds.

Save it and run the program by double-clicking it.

It should open a window, print the words "Hello, World!", and pause ten seconds, even if you press enter. (You can probably press control-C to break out of the program if you want to end it early.)

After ten seconds, the program will resume execution, finish running, and close the window.

1.5 The Language of Computers

The rest of this chapter is dedicated to helping you with a very important concept: why computers don't understand you. It is worth reading. Trust me.

Computers take your meaning from the exact words you write. In this case, the computer will recognize your first statement as having three parts. The first part is the word `print`. It recognizes this part, called a **token**, by the fact that it is an unbroken series of letters. Once it has identified the token it can look it up to find out what it means. The computer has a pre-defined meaning that goes with the word `print`.

The second part of the statement is `"Hello, World!"`, recognized in this case by the fact it starts and ends with a double quote mark. When a token starts with a quote mark (either single or double), it continues until the matching quote mark is found. The quote mark is called a **delimiter**. Everything between is the token. This special kind of token is called a **literal**, or a **string** of letters. Notice that this token has a space in it.

The third part of the statement is the semi-colon at the end. It has a pre-defined meaning of ending the statement, or more correctly, separating the statement from the next statement. In some languages semi-colons terminate statements. In Perl they separate statements. This means that the last statement does not need a semi-colon after it. (It is a good habit to put one there anyway, since if you add more lines later, it will need to be there.)

Let's look at the **wait** line from the example above.

```
$wait = <STDIN>;
```

`$wait` is the name of a variable. We will use many variables in the future. This is the first one we have seen. Variable names (in Perl, in the simple case) are constructed from a dollar sign, followed by a letter, followed by zero or more letters and digits. In this case, the variable name is dollar sign and the four letters w, a, i, and t.

`=` (the equals sign) is the assignment operator. It tells Perl to put something into the variable on the left (that is, `$wait`).

`<STDIN>`, pronounced "standard in," is the standard input channel. The `<>` part indicates an input channel and the `STDIN` part means standard input. Normally that would be the keyboard. "Standard out" is the screen.

When the program runs, this line will cause the computer to wait for the user (the human that is running the program) to type some input.

1.6 Narrow-minded Wording

Computers don't have common sense. They cannot guess your meaning. They have a very precise language. If you step outside their narrow abilities, your program will be rejected. More on that later.

Suppose I wrote this program. What would it do? What should it do?

```
Print "Hello, World!";
```

In this case the computer would parse three tokens just as before. However, the first token would be `Print` which is not on the computer's list of words it knows. You and I can look at it and decide it probably means `print` based on our own common sense. Someone could even program the computer to recognize `Print` as an alternate spelling for `print`. But at this point normally the computer would choke on our program and tell us that we have a syntax error. There is a near-infinite number of ways we could write things so other humans would understand but the computer would not.

What about this one?

```
print "Hello, World!".
```

The computer will not understand the dot (period, full stop) at the end of the line. That would be a syntax error. You or your friend or your tutor may look at the line and understand exactly what you mean. They can even tell you that you need to change the dot to a semi-colon. Why can't the computer do that? Computers don't have common sense.

What about this one?

```
print Hello, World!
```

The computer will not understand the word Hello. The quotes are missing. (Picky, picky, picky.)

1.7 Exercises

Important Vocabulary: Words worth memorizing.

Vocabulary words are sprinkled throughout each chapter. They are also indexed at the back of the book. Certain vocabulary words are so important that you should actually memorize them. This brief list identifies those words.

sleep: a command that causes your program to stop for some number of seconds before continuing.

Answers to starred exercises can be found on page 282.

Exercise 6:* What does the `sleep 5` command do?

Exercise 7:* What does the `$wait = <STDIN>` command do?

Exercise 8:* How does Microsoft Windows recognize a Perl program?

Exercise 9:* What does Print do?

Chapter 2

Syntax and Semantics

Programming is difficult for one main reason. Computers don't think the way people think. Therefore, computers cannot communicate the way people communicate. Although we can understand a process perfectly, and can do it ourselves, and can teach other people to do it, we still need a special approach to get the computer to do it. This chapter examines the question of why that is true and how to successfully write programs.

2.1 Communication

How do we learn to communicate?

Babies learn to talk. We have all done it. We were babies. Now we talk. How did it happen?

My theory is that basically we are able to learn because we can feel what we think other people are feeling. Hunger. Anger. Fear. Exhaustion. We map their words and gestures to those feelings to give them meaning. Whatever words our mothers say when they hold us and smile and smother us with kisses, those are good words. They must be. It is the very human-ness we share that gives us the ability to put ourselves in someone else's shoes and guess what's going on. Essentially, we are born with that foundation.

Scientists tell us that when one human becomes aware of another human a certain part of the brain jumps into action. It is a very reliable effect seen in FMRI scans (Functional Magnetic Resonance Imaging). The medial prefrontal cortex, known as the social cognition area of the brain, is strongly

connected to figuring out what other people are thinking and intending. Computers have no such response.

I tell a story about my son, now grown up with a daughter of his own. When he was small he approached me in the kitchen. He had an empty glass. He said: "I want taste." I knew immediately what he wanted based on things that recently happened. He wanted 7 Up™. Why did he call it taste? I think it is because someone asked him "do you want a taste" and then gave him a sip. Oh. That must be "taste." That would be a good word to remember. That is how his vocabulary was growing. Sure he got it wrong, but he was making a guess based on what he thought the other person meant. It was a guess. It was actually a pretty good guess.

Most of the guesses we make turn out to be right. Sometimes they are wrong. Sometimes they are so wrong it is funny. But most of the time they are right or at least close enough.

Another son, maybe about three years old, was standing on a chair at the kitchen table one day. A bug landed on the table. I turned to my son and said, "Don't eat the bug." He looked puzzled and said, "What?" I repeated, "Don't eat the bug." He thought a bit longer, apparently confused, and then repeated back, "What?"

He understood completely the words I had used. What he did not understand is why I would tell him that. He had no intention of eating the bug. So what I said did not make any sense to him. He decided he must not have heard me correctly. So he asked for clarification.

Have you ever taken something too literally and gotten the annoyed response, "oh, you know what I mean!"?

Read some children's books on Amelia Bedelia by Peggy Parish for fun examples of a maid who takes things too literally.

2.2 Semantics is Meaning

The word **semantics** signifies the meaning of a thing. If I wave a stick at you and appear to be angry, you may take it as a threat. You are looking at the semantics: what did I probably mean. Or if I seem angry with you and make a hand gesture you have never seen before, you may interpret it as an insult. If I seem upset at you but I say, "I love you too," you may take it as sarcasm.

Meaning is very powerful. Somehow just by being human, we seem to get the meaning in lots of situations, even if a language is used that we were not familiar with.

I assert that humans understand by looking for a reasonable interpretation of what they see or hear. I assert that humans overlook the exact words if the meaning seems to be different.

2.3 Do What I Mean, Not What I Say

Humans can make up rules that are flawed.

In about 1987 I was driving in Massachusetts. The official driver's license study guide there said that when an emergency vehicle was approaching, you were supposed to pull over, off the main part of the road. I was stuck in traffic. An ambulance was coming toward me from behind. So I pulled off the road. Actually, it turns out the ambulance was driving on the edge of the road, and I probably knew it. (I can't remember.) My actions probably slowed them down. I was following the written rules as I had seen them in the drivers manual. But the intention of the law was, "get out of the way." And that is exactly what I did not do. It was stupid of me.

In 1997 I visited a website for a school that I was considering working at. At the bottom of the home page it had a link to some acceptable use rules. I was curious so I followed the link. In some legalistic terms it said, "you can visit our website, but you can't download anything." Well, duh. How exactly do you visit a website without downloading anything? Narrowly construed, just by looking at their rules I was already violating their rules. So I decided they must not mean precisely what they say. I had to make my best guess about what they had meant. I ended up assuming they had a different definition of download than me. I went ahead and looked at the rest of their website without feeling any guilt.

These examples show that humans can easily make up rules that are flawed if taken too literally. Fortunately, as humans, we naturally attempt to figure out what was meant by the rules. Blindly following the wording of the rules is not always the best approach, and is often considered to be silly.

2.4 Syntax is Wording

Legal contracts are not much fun to read unless you are a lawyer. Maybe not even then. I have carefully read a number of contracts including some of those "click to accept" licenses on software. The level of precision is annoyingly tedious. They are not much fun to read.

Why are they written that way? Because in a court of law **semantics** (meaning) depends on **syntax** (wording). If you are arguing about something, it is probably because you have different opinions about what the contract means. At that point we turn to a lawyer or judge to tell us both what it means. Get it in writing. The point is to protect yourself when the other person does not agree with your position.

Lots of communication is based on common sense. Maybe almost all of it. When it is based on exact wording, the going gets tough.

2.5 Computers Are Stupid (But Fast)

Well, here's the bottom line. You can write a program. It can be wrong. Still, I can understand it. The tutors can understand it. Your friends can understand it. You can understand it. But the computer cannot.

Why not? Because computers are stupid. They have no common sense. They are forced to give a precise meaning to each word you type. The meaning is not based on what they think you intended. It is based on exactly what you said.

This is the main thing that makes programming difficult. Computers do not understand us the way other humans do. I believe they never will, at least not by the end of this semester.

The whole semester will be consumed in me teaching and you learning how to communicate what is in your mind to a machine that does not understand what you want.

Because despite the fact that computers are stupid, they are also very fast, very reliable, and very cheap (compared to people). They don't call in sick or take vacation. And with some programming skill, many interesting tasks can be broken down into simple steps that computers can perform. For these reasons, even though computers are pretty stupid they are still very popular.

The art of programming is to convert useful activities into simple steps that a computer can perform.

2.6 Key Points

Semantics: Humans understand each other based on the most likely semantic interpretation of the words involved. If the words don't make sense, humans will search for alternate meanings until something does make sense.

Syntax: Computers understand us based on syntactic (grammatic) interpretation of the words involved. Computers have no idea whether the words make sense or not. Computers are just blindly obedient.

Legal **contracts** are an example of human communication that must be syntactically correct in case humans decide they disagree on the meaning. But even in the case of contracts, the syntax may be imperfect. In those cases lawyers and judges interpret the contract to tell us what it means.

2.7 Exercises

Important Vocabulary: Words worth memorizing.

semantics: the meaning or substance of things.

syntax: the expressed form of things. Especially the grammar that connects words together.

Answers to starred exercises can be found on page 283.

Exercise 10:* Why is it difficult to give good instructions to a computer?

Exercise 11:* What does **Syntax** mean?

Exercise 12:* What does **Semantics** mean?

Chapter 3

Input: Hello, Joe!

Our second task is to make the computer say hello, calling the user by name.

3.1 Standard In

Here is the first version of the program.

```
print "What is your name?";
$name = <STDIN>;
print "Hello, $name!";
$wait = <STDIN>;
```

We are familiar with most of this. The new part is the `"Hello, $name!"` part.

Inside double quotes the dollar sign introduces the variable `$name` which will contain whatever the user typed on the previous line.

Type it in and run it.

3.2 Line Breaks

Notice that when you run it, if you type in `Joe`, the output will be like this:

```
What is your name?
```

```
Joe
Hello, Joe
!
```

Strangely the Hello, and the name are on one line but the exclamation mark is on a different line. Why is that?

It turns out that the input, `<STDIN>`, returns everything that was typed, including the `enter` on the end of the line. When `$name` is printed as part of the literal, we get J, o, e, enter, in place of the `$name`.

We need to get rid of that pesky enter at the end of the line.

Fortunately Perl has a command for getting rid of enters at the ends of lines. It is called **chomp**. (Yeah, I think it sounds a little silly. These things happen.)

Here is our new program:

```
print "What is your name?";
$name = <STDIN>;
chomp ( $name );
print "Hello, $name!";
$wait = <STDIN>;
```

Here is the new result:

```
What is your name?
Joe
Hello, Joe!
```

What would happen if we try this?

```
print "What is your name?";
$name = <STDIN>;
chomp ( $name );
chomp ( $name );
chomp ( $name );
print "Hello, $name!";
$wait = <STDIN>;
```

Chomp only removes the enter if it is present. After the first chomp, the other chomps have no additional effect. The program will run the same as with one chomp.

There is a similar command called **chop**. It removes the last character, whether it was an enter or something else. After the first chop, the other chops continue to remove additional characters. A program with several chops will *not* run the same as with one chop (unless it runs out of data to chop).

Another thing: don't go overboard with **chomp**, throwing it everywhere in your program. The place to put it is at or near the place that input is received. Don't use it on variables that you know will not have a newline at the end.

3.3 NewLine

We have an annoyingly large amount of variation on what should be a simple subject. Life has such moments. The concept is called **newline**.

Originally two printer control codes were created. Originally **carriage return** (**CR**, hex **0D**, also `\r`, also the **Return** key) repositioned the cursor (print head) to the start of the line. Originally **line feed** (**LF**, hex **0A**, also `\n`, also **line break**, also the **Enter** key) moved the cursor down a line.

They were nearly always used together as **CRLF**, (**0D0A**). Some people found this to be wasteful and redundant. Apple went with one. Unix went with the other. Microsoft stayed with both. This has resulted in a world of minor confusion.

Perl uses `\n` to stand for newline, which on Apple equipment means what Apple wants, and on Unix it means what Unix wants.

You should be aware that these words are often used interchangably and usually mean the same thing. If you get into data communications, you may need to learn more.

3.4 Exercises

Important Vocabulary: Words worth memorizing.

newline: a special character that tells the computer to begin a new line on its output medium (usually a computer screen).

chomp: a command that removes the last character if it is a newline.

chop: a command that removes the last character of a string, no matter what it is.

Answers to starred exercises can be found on page 283.

Exercise 13:* What is the purpose of **chomp**?

Exercise 14:* What is the purpose of **chop**?

Exercise 15:* What is the difference between newline, line break, carriage return, line feed, CRLF, CR, LF, enter, return, and `\n`?

Chapter 4

Simple Variables

Compare watching a DVD to playing a computer game. Both are highly visual. Both have sound. Both can be done for hours at a time.

But watching a DVD is the same every time you do it. Nothing you do will change what happens next. Playing a computer game is all about you being in control. Your main goal is to change what happens next.

Within the computer, when we print "Hello, World", it's like watching a DVD. Every time we run the program, the same thing will happen. Maybe that's good. But maybe we need more than that.

In order to print "Hello Fred" when talking to Fred, and "Hello Joe" when talking to Joe, we generally use a variable to hold the name of the person we are talking to. Then we print "Hello $name" and the computer inserts the appropriate name.

This chapter provides the basics of working with simple variables. In Perl these are technically referred to as **scalar** variables. Later we look at lists (arrays), lookup tables (hashes), and databases.

4.1 What's Simple?

A simple variable holds one piece of information. Examples would be a number (3.1415) or a name (Fred). A series of characters is called a **string**. Strings can be very short (no characters at all) or very long (whole sentences, paragraphs, or books).

Note: many languages treat whole numbers, numbers with decimals, individual characters, and strings of characters as different types of data. They are. We will talk more about this in the database chapter. Perl and other scripting languages tend to simplify and use a single variable type for all of them.

4.2 Variable Names

In Perl, variable names are prefixed by a dollar sign ($) that identifies the variable as simple (or more precisely, as a scalar). The dollar sign can be thought of as part of the variable name or not. We will think of it as a prefix to the variable name.

The first real character of the variable name comes right after the dollar sign. It must be a letter, A through Z or a through z. As a special case, the **underscore** character (`_`) is considered to be a letter.

After the first character, digits are also allowed.

There is no specific maximum length for a variable name. You can make it as long as you like.

`$ABC` and `$abc` are not the same variable name. That is, the names are **case sensitive**. Letters like "ABC" are called **upper case** or **capital letters**. Letters like "abc" are called **lower case** or **small letters**. Apparently the upper and lower cases were used years ago by type setters. Small letters were more commonly used so they were placed in the lower case to be easily accessed. Capital letters were more rarely used so they were placed in the upper case which was more out of the way. When the case of the letters makes a difference, we say they are **case sensitive**. When they do not make a difference, we say they are **case insensitive**.

Things other than letters, digits, and underscores are not allowed in variable names. Specifically, spaces are not allowed.

4.3 Putting Something Into a Variable

The main way to put information into a variable is to use the **assignment** operator which looks like an equals sign.

Here we have a variable named `$x` and we are copying the number `5` into

that variable.

```
$x = 5;
```

Note that we are not saying that `$x` is equal to `5`. We may read it that way when speaking aloud but in reality we are copying. It would be more accurate to say "x gets 5" instead of "x is equal to 5." This is computer science, not mathematics. It may look the same but the meaning is different.

The semantics of the assignment syntax is that the copying is done from right to left. The following example does not copy a `5` into `$x`. Instead, it tries to copy the value of `$x` into the number `5`, which does not make much sense.

```
5 = $x; # wrong
```

Next we have a variable named `$x` and we are copying the string `"5"` into that variable. A string is a series of zero or more characters.

```
$x = "5";
```

The string with zero characters, `""`, is also called the **empty string**.

Next we copy the string `"Hello, World!"` into a variable named `$x`.

```
$x = "Hello, World!";
```

Next we have a variable named `$x` and we are copying from the keyboard into that variable. When the program reaches this statement, it will wait for the user to type something and press Enter. The letters typed together with the Enter at the end will be copied into the variable.

```
$x = <STDIN>;
```

Next we have a variable named `$x` and we are copying from the keyboard into that variable. When the program reaches this statement, it will wait for the user to type something and press Enter. The letters typed but NOT the Enter at the end will be copied into the variable. The Enter is removed by the **chomp** command.

```
chomp ( $x = <STDIN> );
```

Note that each time you put something into a variable, the computer forgets what was there before. Simple variables only hold one thing at a time.

Also note that **chomp** should only be used when there is the chance that a **newline** is at the end of the variable.

```
chomp ( $x = "Hello" ); # wrong
```

4.4 Printing a Variable

The following command prints the words `"Hello, World!"` without using a variable.

```
print "Hello, World!";
```

We can directly print the contents of a variable. The following commands print the words `"Hello, World!"`.

```
$x = "Hello, World!";
print $x;
```

We can print the contents of a variable as part of a longer statement. The following commands print `"I shout Hello, World! every morning."`. Inserting a variable into the middle of a string is called **interpolation**.

```
$x = "Hello, World!";
print "I shout $x every morning.";
```

We can also print more than one thing at the same time by connecting them using the dot operator:

```
$x = "Hello, World!";
print "I shout " . $x . " every morning.";
```

Dot can be handy when we are trying to do a calculation and print at the same time. The **precedence** of dot is the same as plus, so to be safe we use parentheses to force all calculations to take place before stringing things together. See section 5.5 (page 49) and chapter 23 (page 117) for more on precedence.

```
$x = 5;
$y = 6;
print "$x + $y = " . ( $x + $y ) . "\n";
```

4.5 Calculating with a Variable

We will talk in detail about calculation and basic operators in chapter 5 (page 48). For now just understand that we can use things like `+` (add) and `*` (multiply) to calculate.

We can use a variable to hold a number for calculation. The following program asks the user for a distance in inches. It stores that value in a variable (`$in`). Then it uses the number in a calculation.

```
print "How many inches? ";
chomp ( $in = <STDIN> );
$cm = 2.54 * $in;
print "$in inches equals $cm cm.";
```

4.6 Use Meaningful Names

The computer does not recognize any particular meaning for your variables based on their names. You can call them whatever you want. If the variable is to hold a distance in inches, you could call it `$inches` or `$in` or `$aloha`. The computer does not care.

Humans (including programmers) naturally ascribe meaning to things they recognize. That makes this an important style issue. A human seeing a variable name of `$aloha` might expect it to hold the local greeting, such as "Hello" or "Hola" or "Bula" or "An nyoung" or "Guten Tag".

It is strongly recommended that you create variable names that describe the information stored inside. Thus, `$inches` would be a good name for a

variable that holds a distance in inches. `$aloha` would not be a good name for such a variable. When you create variables that are not meaningful, you increase the chances of creating programming errors in your code when you forget what they hold. This happens even to professional programmers.

Short variable names like `$x` are handy for localized scopes but you should not rely on the value in `$x` in far distant parts of your program. Think about how we use pronouns (he, she, it). They are useful in localized contexts. They are problematic when referring to distant objects because the context can be forgotten. Misunderstandings can easily happen. That would be like programming errors.

4.7 Summary

You should know the following things.

1. Simple variable names use a prefix of `$`, begin with a letter (or underscore), continue with zero or more letters, digits, or underscores, and can be as long as you like.

2. Upper case letters are different than lower case letters. `$abc` is not the same as `$ABC`.

3. Simple variables can hold a number or a string.

4. Simple variables only hold one thing at a time. When you put something in, the prior contents are simply lost.

5. Simple variables can be used in print statements.

6. Simple variables can be used in calculations.

7. Variable names should be meaningful to prevent programming bugs.

4.8 Exercises

Important Vocabulary: Words worth memorizing.

lower case: Small letters: abcdefg. Old-time typesetters kept these letters in the lower case on their workbench because they used them more often than the big letters.

upper case: Big letters: ABCDEFG. Also called capital letters. Old-time

typesetters kept these letters in the lower case on their workbench because they were less often used so it was okay to have to reach a bit to get them.

case sensitive: Situations where the difference between upper and lower case makes a difference. Is "hello" the same as "HELLO"? If so, we are not case sensitive. If they are different, then we **are** case sensitive.

Answers to starred exercises can be found on page 283.

Exercise 16:* What makes a "legal" variable name?

Exercise 17:* What makes a "good" variable name?

Exercise 18:* What does **case sensitive** mean?

Exercise 19:* What is the syntax for assignment?

Exercise 20:* What does "interpolation" mean?

Exercise 21:* What does "concatenation" mean?

Chapter 5

Basic Calculation

Perl and most other programming languages have easy ways to calculate with numbers. The calculations are written in fairly standard mathematical notation, like a+b. In this chapter we run through a list of basic operators and explain what each means.

5.1 Add

Addition is specified by using the `+` (plus) sign between two things. It's about what you would expect.

5.2 Subtract

Subtraction is specified by using the `-` (minus or dash or hyphen) sign between two things. It's about what you would expect.

5.3 Multiply

In computer programming, we often use the asterisk (`*`) to represent **multiplication**. In arithmetic we are accustomed to using an x ($\times$) to represent multiplication, but in computer programming we want the letter `x` to be a letter, so in many computer languages the asterisk has been assigned the meaning of multiply.

We say `3*5` is three times five, or 15.

5.4 Divide

Regular calculator division is called floating-point **division**. In this kind of division, `5/2` is `2.5`. **Floating point** means that the decimal can be in the middle of the number. Usually when we divide two numbers, we mean floating point divide.

There is another kind of division, integer division, that we will discuss later. In that kind of division, `5/2` is `2` with a remainder of `1`. With integer division, the decimal point is fixed at the end of the number, never in the middle.

5.5 Precedence and Parentheses

Precedence is covered in greater detail in chapter 23 (page 117).

When we say `3+2*5`, should the computer add first or multiply first?

We can control the order of calculation by using parentheses. We can say `(3+2)*5` or `3+(2*5)`, depending on what we mean.

When we leave out the parentheses, the computer follows rules of **precedence**. For convenience and simplicity, mathematicians have standardized on the precedence of multiplication before addition. It does not have to be that way, but by convention we have agreed as a group to follow that rule.

In the $3 + 2 \times 5$ example, this means we multiply 2 and 5 for a result of 10. We then add 3 for a result of 13.

Writing without parentheses is shorter. Shorter is more desirable because it is less work and also less cluttered. But it does require us to learn the rules of precedence so we will get the same answer as everyone else. These are covered in more detail in a future chapter.

5.6 Assignment

Assignment is represented by the `=` (equals) sign. It makes a copy of the value of the expression on the right and places it into the variable on the left. This will calculate 13 and put it into `$x`.

```
$x = 3+2*5;
```

Programming exercises can be found in chapter 13 (page 81).

Chapter 6

Names, Parsing, Scanning

I am spending a lot of time trying to help you understand the behavior and limitations that computers have. I do this because I have found students frequently imagine computers to be much more intelligent than they actually are. You have to come down to their level if you want to talk to computers and have them understand. Fortunately, it is worth the effort.

This seems like a good time to tell why we have such strict rules on variable names.

As you will recall, a variable name must start with a letter. (Letters include A through Z, a through z, and underscore.) It can continue with letters and digits. (Digits include 0 through 9.) It has a dollar sign as a prefix. It must not include special characters or spaces.

Why is that?

6.1 Computers Are Stupid

It's not polite to say things like that. Hopefully no computers are listening. In section 1.1 we established that computers are stupid. Humans understand things by looking for meaning. This allows us to overlook many kinds of mistakes that others might make.

For example, imagine a traffic light, red on top, yellow in the middle, green on the bottom. Red means stop. Yellow means stop if you can. Green means go. Now imagine the red glass lens covering the top light is broken

and missing. When the light should be red it is white. You drive up to the light. The white light (on top) is shining. What should you do?

Normally the human would conclude the light was supposed to be treated as though it were red. The human looks for the meaning and overlooks what might be seen as trivial variations.

Computers generally fail that kind of test. Computers generally are programmed to look for specific conditions and respond to them. Slightly different conditions can result in failure to properly recognize the situation.

6.2 Robust

When a program fails for apparently silly reasons, we say the program is **fragile** or **brittle**.

When a program does not fail for apparently silly reasons, we say the program is **robust** or **flexible** or **bullet proof**.

When small problems cause a small amount of breakdown we call it **graceful degradation**. Humans get old. Their bodies work less well. They walk slower. That is graceful degradation. Eventually the bones wear out and maybe one breaks a hip. They don't walk at all. That is **catastrophic failure**.

Programmers can spend lots of time making their programs less brittle, but it is a difficult task. There are too many things that can go wrong. It is virtually impossible to catch all of them and respond properly.

6.3 Parsing

Vocabulary warning: this section introduces even more new words, many of which are only used in this section.

Computer programs are typically written in text. They are then converted into a runnable program through a process called compiling. Compiling is done by a **compiler**.

Parsing is the activity your compiler does when it reads in your program and tries to understand it. The first step in parsing is called **lexical scanning**. In this step, the text of your computer program is divided up into

individual words, called **tokens**.

In the next step, the tokens are organized according to the **grammar** of your programming language.

Let's look at an example.

```
print "Hello, World!";
```

In this example, the lexical scanner would break up the program into tokens. It would find the following tokens:

```
print
"Hello, World!"
;
```

Tokens are identified by the spaces between them and by the **punctuation** around them. The same tokens would be found if the program looked like this:

```
print"Hello, World!"   ;
```

The spacing between tokens is usually not significant. If you can have one space, you can have several. It's all the same to the compiler. Spaces, tabs, and newlines are collectively called **white space** because they are invisible, or white, as opposed to other things—like dashes—that are sometimes used for space. Where you can have one white space of any type, you are allowed to have many of any type. This comes in handy when we start working with programming **style**.

6.4 Summary

For the parser to work efficiently, variable names and other tokens must follow simple rules. The simple rule as stated in chapter 4 (page 41) is that viable names are prefixed with a dollar sign, start with a letter (including underscore), and continue with letters and digits.

It is possible to have much more flexible rules for variables, and indeed for programming in general, but that makes it much harder for computers to do their jobs. There is a whole field of study called Natural Language Processing. It is a difficult task.

6.5 Exercises

Important Vocabulary: Words worth memorizing.

white space: Characters that are invisible. Examples include spacebar, tab, and newline. They are called "white" because they do not use any ink.

Answers to starred exercises can be found on page 284.

Exercise 22:* Identify the errors in the following code.

```
Print "Enter a length in inches: ";
chomp $inches <STDIN>;
$cm = 2.54 $inches
Print "$inches inches equals $cm centimeters.\n;
```

Chapter 7

Style

Semantics is what you mean. Syntax and style are how you say it.

Our goal in this chapter is to set some standards for how programs are expressed. We know that the computer is equally comfortable no matter what variable names you select or how neat and orderly your code appears. The key question with style is how easily other programmers can read your code and fix it when necessary. In later chapters we will return to issues of style and complexity. Right now we will introduce the subject and set forward a few rules.

7.1 Whitespace

Whitespace is the invisible parts of what you write. Every time you press the space bar you are adding whitespace to your program. Every time you press Enter you are adding whitespace.

The main purpose of whitespace is to keep things (tokens) from crashing into each other. Our goal is to make them easy to read. It is also a syntactic crutch to tell us where to divide the words. Consider "together" and "to get her". Same letters. Different meaning. Spaces can clarify the meaning.

A secondary but important purpose of whitespace is to show relationships. Starting on a fresh line shows that the old thought is ended and something new is starting. Leaving a blank line shows that the old thought has Really ended and something new is Really starting. Lines in a "paragraph" cluster together to show that they are related. These are semantic crutches to help

make the meaning more clear.

7.2 Spaces

Put a single space between each token in the program, except before the semicolon at the end of each line. This improves readability. It does not change the meaning of any of the tokens. It does not change the meaning of the program. Examples:

```
$x=1; # bad
$x = 1; # good
while($x<=100){ # bad
while ( $x <= 100 ) { # good
```

Exception: if putting spaces in makes the line too long, you may leave some or all of them out. This should be rare.

Putting spaces between tokens is requested. Putting extra spaces **inside** tokens is forbidden. It changes the meaning of the token. It changes the meaning of the program. It breaks the program. Examples:

```
$x="Hello"; # original
$x = "Hello"; # good
$x = " Hello "; # BAD
```

In the lines above, extra spaces were inserted inside the string "Hello". This is not allowed as it changes the meaning of the program. Instead of `$x` being the five-character string "Hello" it becomes the seven-character string " Hello ". Sometimes that may not matter, but as a programmer you are not allowed to change the strings that have been specified.

```
chomp($name=<STDIN>); # original
chomp ( $name = <STDIN> ); # good
chomp ( $ name = < STDIN > ); # BAD
```

It is not permitted to put a space between the `$` and the variable name. It is not permitted to insert spaces inside the `<STDIN>`. These things change the meaning of the program and probably break it.

7.3 Separate Lines

Generally each statement should be on its own line. This is especially true for longer statements. Exception: Short statements that are closely related may be combined on the same line.

7.4 Blank Lines

If a block of code is many lines long, it is recommended that blank lines be inserted in appropriate places to divide up the code into "paragraphs" (shorter visual pieces). The places chosen should be meaningful, such that related code is kept together.

Excessive blank lines should not be used.

7.5 Names

Pick names that are meaningful and helpful.

Variables, subroutines, files, and other named objects should be named in useful and helpful ways. Your program may work just fine with variables like `$frog` and `$cracker`, but unless the variables are related to actual frogs and crackers, you will create a nightmare for the next programmer that has to read your code.

By convention, variables start with lower-case letters (a-z). Long variable names can be built up from several words connected by underscores. For example:

```
$miles_per_gallon
$address_current
```

Another common option for long variable names is to use what is called **camel case** (as compared to **upper case** and **lower case**). For example:

```
$milesPerGallon
$addressCurrent
```

Variables that start with upper-case letters are generally special in some important way. They may be constants or global variables.

```
$SpeedLimit = 55;
```

Variables that start with an underscore are generally intended to be private. They may be in library subroutines. Libraries use the underscore to avoid picking a variable name that you might also pick.

Avoid starting common variable names with upper-case letters or with an underscore. Start with a lower-case letter unless you have a good reason to do something else.

Read **Hungarian notation** on Wikipedia for a very helpful discussion of variable naming for real projects.

http://en.wikipedia.org/wiki/Hungarian_notation

7.6 Comments

Code should include an appropriate degree of commenting.

Comments help the programmer organize his/her thinking while writing the program. Write the main comments first before you start writing code. When comments are written after the program is complete, frequently they are of the unhelpful type. It's almost not worth commenting when the program is done.

Write comments as you go along. Each time you stop to ponder how to do something, think about whether a comment should be included to explain your thinking to others.

Comments help the programmer remember the organization of the program when it is necessary to fix bugs or add new functionality.

Comments really help other programmers that may have to work on your code after you hand it off and move to another project.

Comments should primarily describe the purpose of the code. It is much less important to state exactly what is happening, since that is usually obvious.

```
$x = $x + 1; # add 1 to x (bad)
$Ct = $Ct + 1; # count the items processed (good)
```

At the start of a subroutine, it is good to write a block of comments telling what the subroutine does as a whole, what the inputs are, what the outputs are, whether it is of limited or general use, and if limited where it is called from. Focus first on why it was written and what it should accomplish. Then if necessary tell how it actually works.

Exception: if your code uses an algorithm that may be a bit tricky to understand, it would be good to explain in detail how it actually works, maybe even giving references to the source of the algorithm if you got it from a book or website.

Chapter 8

Sequence

Mathematics and programming can look a lot alike. This leads to confusion at times. Sequence is an important area where this can be a problem.

8.1 Equations

In mathematics, a system of equations is viewed to be a set of facts that are true no matter what the order is. Consider these equations.

```
x + y = 7
y = 3
```

Using normal rules of algebra, you can easily see that `x` must be equal to `4`. We could figure it out like this:

```
(1) x + y = 7
(2) y = 3
(3) x + 3 = 7 (substitute line 2 into line 1)
(4) x + 3 - 3 = 7 - 3 (subtract 3 from both sides)
(5) x = 4 (simplify 3-3 and 7-3)
```

It would not matter if we started out with this instead:

```
y = 3
x + y = 7
```

The reason is that mathematical equations are supposed to be statements of truth that have no particular order to them. They are just all true at the same time.

8.2 Steps

In computing, each statement is not a statement of truth. It is a command to be carried out. It is an instruction.

```
y = 3
x + y = 7
```

In programming, this would take the value 3 and copy it into the variable y. Then the program would die because `x + y = 7` cannot be carried out. The `=` sign means **assignment**, or "copy" instead of "it is equal". We can copy into a variable. We cannot copy into an expression like `x + y`. We would probably have to rewrite the program as follows.

```
y = 3
x = 7 - y
```

Now we can calculate the value of x. It will be seven minus the value currently stored in y. Since y has a three in it, we calculate that x should receive the value of four.

Reversing things, we have the following:

```
x = 7 - y
y = 3
```

Here the first step is to calculate x. We have seven. We look at variable y. Since y has not yet received its value of three, it does not have a value, or it has a value of **garbage**. Since the steps are happening in order, x receives a value of garbage.

Then y receives its value of three. But this does not help x because that step is now ancient history. Computers do not remember what they just did. All they remember is what the variables are right now and what step they are working on.

(For **uninitialized variables** instead of using garbage, Perl uses `""` in a string context and `0` in mathematical contexts. Other languages are not always so helpful. You should always **initialize**, that is, give a value to each of your variables before using them.)

8.3 Sequence

Sequence is the most important method we have of controlling our programs. In nearly every programming language, steps are written one after the other. The computer is required to execute one step completely and then go on to the next step.

(There is an exception called **pipelining** in which the computer starts working on the next instruction before finishing the first instruction. If done properly, it can result in much faster programs. There is a whole field of study centered around pipelining. For this course we will ignore it. You can learn about it in a class on computer organization or computer architecture.)

8.4 Later

Later we will look at ways to change the sequence of operations. Steps never run backwards, but we can make programs jump from one part of the program to another. This is called `goto`. Programs can be made to choose between two paths based on the results of a calculation. This is called `if/else` or conditional execution. Programs can be made to repeat instructions over and over. This is called `iteration` or `looping`. There are other things including subroutines and parallel programming.

For now, the important thing to remember is that programs run sequentially, one step after another. The order of the steps is usually very important. Sequence is the foundation of programming.

8.5 Exercises

Answers to starred exercises can be found on page 284.

Exercise 23:* What will the following program print?

```
$a = 5;
$b = 6;
$a = 10;
$b = $b + $a;
$c = $a - 3;
print $a;
print $b;
print $c;
```

Chapter 9

Crafting Formulas

The task of programming frequently involves the creation of formulas. We have the real world in front of us. We know how it operates. We need to explain it to the computer.

But formulas can be scary. It is easy to get Math Anxiety at this point. Relax. Take a deep breath. Let's work through a few examples.

9.1 Coin Purse

Imagine I hand you a coin purse. It contains two quarters, three dimes, and two pennies. What is the total value of the money in the coin purse?

If you are not familiar with US coinage, I may need to tell you that a quarter is worth 25 cents, a dime is worth 10 cents, a nickel is worth 5 cents, and a penny is worth 1 cent. Now what is the total value of the money in the coin purse?

Most people have little or no trouble in answering 82 cents. (You did get 82 cents, right?)

9.2 Rough Formula

Let's put that in a formula. I will tell you `$Q`, `$D`, `$N`, and `$P`, the number of quarters, dimes, nickels, and pennies respectively. I want a formula that converts those four numbers into a total value for the purse.

When I give this problem in my introductory programming class, about half of the students respond with this answer, or something like it:

```
$val = $Q + $D + $N + $P
```

The other half respond with this answer, or something like it:

```
$val = 25 * $Q + 10 * $D + 5 * $N + 1 * $P
```

Both answers are basically right. The first answer is correct if you intended to add up the total **value** based on the **value** of the quarters, dimes, nickels, and pennies. The second answer is correct if you intended to add up the total **value** based on the **number** of quarters, dimes, nickels, and pennies. Both are correct.

Go back and look at the problem. It says the `$Q` is the number of quarters, not the value of the quarters.

9.3 Improved Formula

Starting with a rough formula, even an incorrect one, we can test it and debug it. Substitute in the numbers that we know should work. That would be called a test case. Look at the answer.

```
82 = 2 + 3 + 0 + 2
```

Nope. I don't think that is right. Hopefully we can see our mistake and say, "Hey, wait a minute, I did not mean to add 2 for quarters. I meant to add 50."

But you've got 2 as your input. If you want to use 50 in your formula, how do you convert it from what you have to what you want? In this case, multipy it by 25 (the value of each quarter).

```
82 = 25 * 2 + 10 * 3 + 5 * 0 + 1 * 2
82 =   50   +   30   +   0   +   2
```

That's looking correct.

9.4 Test Your Formula

Are we done yet? Maybe. But generally it is a good idea to work a couple of sample problems by hand and make sure the answers come out right before we decide we are done. Invent some data. Make sure you know what the correct answer should be. Then run your program (or formula) on your data and see if the answer comes out right.

Most of the time, if you came up with the data yourself and you know how you got it, you will be able to quickly see whether your formula is right or not.

Chapter 10

Debugging

Your Perl program looks right to you, but it does not work. What you (are supposed to) do next is called "debugging." Where do you start in finding the problem and correcting it?

For our examples, we will assume you have written a program called `myprog`.

10.1 Syntax Errors

There are two major kinds of errors: syntax errors and run-time errors.

Syntax errors are flaws in the wording of your program. They include such things as forgetting to end a statement with a semi-colon, or spelling "print" with a capital P as in "Print". Generally computers are very stupid about figuring out what you mean. They are very good at doing what you say if it is within their power, but you must say it exactly the right way. If you say it wrong, the program will not be able to run.

Syntax errors are caught by the compiler or interpreter, usually before the program even starts to run. They are reported by identifying the line at which the compiler decided that it was hopelessly confused. The actual error is usually just before that point, but it could be many lines earlier if the mistake is a missing quote mark, for instance.

Many compilers will try to report as many syntax errors as they can find. This is a throw-back behavior more suited to an earlier time in the history of computing, when programs were submitted through a real window using

a deck of punched cards and outputs were retrieved four hours later. It was desirable to see as many errors as possible all at once. But those days are gone for most of us.

Focus your attention on only the first error. Later errors could well be part of a domino effect much like taking the wrong turn with driving directions. Once you are off course, the other errors may simply reflect the fact that you are already off course, and may not be actual errors themselves. Focus your attention on only the first error (or other things you know are errors). Fix it and then try running your program again.

10.2 Run-Time Errors

Run-time errors, sometimes called **semantic errors** or **logic errors**, are flaws in the logic of your program. They include things like forgetting to initialize a variable before adding to it, or doing things in the wrong order. The resulting program will still run, but may do the wrong thing.

Run-time errors cannot be reliably caught by the compiler. Probably the best and easiest way of figuring them out is by lacing your program with **print** statements to dump the values of various variables to show you how the program is acting. This effect can also be achieved by using a good debugger using features like single-step and breakpoint. Most languages have debuggers available. Learning to use a debugger can take time and effort. It is usually easier to rely on print statements and to graduate to debuggers after you have developed some skill.

Sometimes it is good to put **assert** statements into your program. In Perl, these are **if** statements that check for things you are sure must be true, and if they are not true, you want your program to stop so you can fix the error. For example:

```
if ( $a > 1 ) { print "assert failed, a>1: a is ($a)"; die }
```

Later, if your program is working and the assert makes your program run too slowly, you can comment it out or delete it.

Print statements and assert tests are sometimes called **scaffolding**. During building construction it is common to set up metal or wooden frames around the building to allow construction workers a place to stand. These frameworks are called scaffolding. When the building is completed, the scaffolding is removed. When your program is debugged, the scaffolding is either

deleted or commented out.

In Perl, when you **comment out** something you simply put a `#` (pound sign) symbol at the start of each line to convert it into a comment. You do not delete the lines so that later if you need to debug that section again, you can just un-comment those lines and use them. Deciding whether to delete or comment out the scaffolding is largely left up to the programmer.

Another phrase is **test bed**. This name is used when your program must be tested in the context of some larger system. For example, your program may be rely on other programs or subroutines to create inputs and display outputs. In that case, to test your part of the program you would develop or **stub** out other parts of the program. The stubs are generally highly simplified. Instead of actually asking for input, for example, they might always provide the same input.

Important Vocabulary: Words worth memorizing.

syntax error: A mistake in the grammar of a program. Syntax errors are detected when the program is first read, before any attempt is made to carry out the instructions.

logic error: A mistake in the semantics of a program. Logic errors are detected when the program runs but does not do what the programmer wanted.

Chapter 11

Going Online

Programs at your desktop are useful and fun but in today's world the Internet connects computers everywhere. We can write our programs to be used by people all around the world. In this chapter we take the first steps.

This chapter shows how to put simple programs online. By "simple" we mean something that does not work with user input. In chapter 24 (page 122) we build on this foundation by showing how to receive simple (closed set) user input, specifically the pressing of buttons, into an online program. In chapter 51 (page 228) we show how to receive multiple and arbitrary (open set) user inputs.

You will need a web account to do the activities of this chapter. You will need to know (a) your domain name, (b) your login name, (c) your password, and (d) how to get help.

Important Vocabulary: These are words worth memorizing. There are more at the end of the chapter.

local: is a program running at the computer where you are sitting. It is not running on the web.

offline: means a program that is not running on the web. It is another word for local.

online: means a program that *is* running on the web.

web server: is a computer that hosts web pages and web-based programs.

static: is something that does not change, as though it were frozen in time.

dynamic: is something that can change in response to things around it.

11.1 Using a Browser

With a program at your desktop, you can run it by double-clicking the icon or by opening a command-line window and typing the command itself.

Online we use a browser such as Firefox(tm) or Microsoft Internet Explorer(tm) to run the program. We see the results locally but the program itself runs remotely.

Specifically, the browser allows a human, called the **user**, to click on a **link** or type in a **URL** (universal resource locator, more properly called a **URI** for universal resource identifier).

Through a chain of events, the URL causes the server to find or create the web page content that the user will see. If the content is a static page, the server will simply find it and send a copy, plus some header information, to the browser.

If the content is a dynamic page, the server will start a program. The program will create the web page and give it to the server. The server will send the newly created web page back to the user.

In either case, the browser then renders it as a web page for the human that made the original request.

11.2 Static Web Page

Let's assume your domain name is `doncolton.com`. Where you see it mentioned below, substitute your own information.

We will assume you are using cPanel(R) to access your account. If not, find the appropriate documentation.

Log in to your web host. Navigate to your `public_html` directory. You can use the cPanel File Manager.

We will create a folder for our work. We will call it `ipup`.

In your browser, go to `http://doncolton.com/ipup/`. You should see a 404 error because no web content exists yet.

In cPanel File Manager, click on Create Folder and key in `ipup`.

In your browser, go to `http://doncolton.com/ipup/`. You should see an empty directory. We are making progress.

Inside the ipup folder, create a file named **index.html** and consisting of wording like "Hi, I'm (Your Name Here). Welcome to my website."

In your browser, go to `http://doncolton.com/ipup/`. You should see the web page that you just created.

Congratulations. You have created a static web page. Each time it is requested by a browser, the web server will find the page and transmit it to the browser.

Our next step is to create a dynamic web page. Rather than saving the page on our web server, we will write a program that creates the page each time it is requested.

11.3 Hippopotamus Offline, Online

Our basic example is a program that prints the word "Hippopotamus" five times.

The offline version of the program is fairly simple. Write it and save it to your desktop as `hippo.pl`.

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

You can test it offline by doing a `start` / `run` / `cmd` to open a command line window. In that window, type `cd desktop` and then `hippo.pl` to run your program.

By adding two lines, we can convert our program to run online.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";
```

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

The first line tells the server how to run the program. It specifies where `perl` is to be found on the server.

The second line has the server tell the browser that the information it is about to receive is text, possibly including html markup.

I should stress that, as you might guess, these lines need to be typed pretty much exactly as shown.

I should stress that, as you might guess, these lines need to be typed pretty much exactly as shown. Yes, I said that twice on purpose. Pay close attention to detail.

In your `ipup` folder on your web host, create a sub-folder (directory) and name it `hippo`.

Inside the new `hippo` folder, upload your program.

Rename your program to **index.cgi**.

Set the permissions to `0755`.

You should now be able to run the program by using the following URL. Of course, substitute your own domain name and directory names.

`http://doncolton.com/ipup/hippo`

It should display the word "Hippopotamus" five times, just as if it were running on your local computer.

11.4 All on One Line? Adding Markup

Notice that the word "Hippopotamus" appears five times on the same line, not on separate lines like the offline program did. That is because in HTML, "\n" is treated like all other forms of whitespace by the browser.

Back in section 3.3 (page 39) we talked about newlines and how they differ from one operating platform to another. HTML adopts the rule that when you want to go to a new line, you need to say `<br>` to cause the current line

to break. Let's add that to our program.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";

print "Hippopotamus<br>\n";
print "Hippopotamus<br>\n";
print "Hippopotamus<br>\n";
print "Hippopotamus<br>\n";
print "Hippopotamus<br>\n";
```

The `<br>` notation is called **markup**. Markup is the "M" in HTML, which stands for Hyper Text Markup Language. Markup tells the browser how to style your web page for the person who is viewing it.

Chapter 50 (page 224) gives a bigger introduction to HTML. Because HTML controls the appearance of web pages, and there is a huge variation in what web pages need to be, many books have been written to help web designers achieve the look and feel they want. In this book we will only barely scratch the surface of this huge topic area.

You can put `<br>` before some text to cause it to start on a new line.

You can put `<h1>` and `</h1>` around some text to cause it to be rendered as a level 1 (primary) heading.

You can put `<h2>` and `</h2>` around some text to cause it to be rendered as a level 2 (secondary) heading. Heading levels range from 1 to 6.

You can put `<b>` and `</b>` around some text to cause it to be rendered as BOLD.

11.5 Errors: What Could Go Wrong?

There are several common errors that programmers may encounter. If your page is not working, here are some things to check.

404 (File Not Found): If you get a 404 error, it means your path is not correct. Did you create a folder named `hippo`? Is it inside your `public_html` folder? Did you call your program **index.cgi**? Did you accidentally use capital letters when you were supposed to use small ones?

Forbidden: If you get a message like this, make sure your file permissions and directory permissions are set properly. The right setting for a text file is usually 0644, which normally happens automatically without any extra effort on your part. The right setting for a program file is usually 0755, which normally requires intervention on your part.

500 (Internal Server Error): This usually means the first two lines are wrong somehow. Did you type `user` instead of `usr`? Did you type `perl--` without a space instead of `perl --` with a space? Are the `#!` characters the absolutely first characters in your file? Did you put `content_type` or `content-type` or `context-type` or `content=type`? (Only one of them is correct.)

It could also mean that you have some other error in your program, such as a missing semi-colon or quote marks.

Try running your program from the command line. See if it shows any error messages.

Verify that your `content-type` line is exactly right.

Verify that your permissions are set to `0755`.

11.6 Roll The Dice

We can make a much more useful program. Let's roll some dice. This requires a new capability: random numbers.

Chapter E (page 319) introduces random numbers at a greater level of detail. At this point, just trust me on it. You can get random numbers from 1 to 6 (inclusive) using a formula like this.

```
$dice = 1 + int ( rand(6) );
```

The `rand(6)` part creates a number between zero and 5.999. (Technically it is between zero and 6, not inclusive.) The `int` part trims of the digits after the decimal leaving us with a whole number, either 0, 1, 2, 3, 4, or 5. The `1+` part adds one to the result, giving us either 1, 2, 3, 4, 5, or 6.

Here is a command-line version of the program.

```
$dice1 = 1 + int ( rand(6) );
```

```
$dice2 = 1 + int ( rand(6) );
print "You have rolled $dice1 and $dice2.\n";
```

Try it and see that it works.

Here is the online version of the program.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";

$dice1 = 1 + int ( rand(6) );
$dice2 = 1 + int ( rand(6) );
print "You have rolled $dice1 and $dice2.\n";
```

In your web account's ipup folder, create another folder (directory) and name it `dice`.

Inside the new `dice` folder, upload your program.

Rename your program to **index.cgi**,

Set the permissions to `0755`.

You should now be able to run the program by using the following URL.

`http://doncolton.com/ipup/dice`

11.7 Putting Images on Web Pages

Putting up a couple of numbers is fun but we can do better. Much better. Let's put up actual pictures of **dice**.

To do this, the first problem is to acquire pictures of the six possible outcomes. I did this by using a digital camera and an actual physical die. (Dice is plural, die is singular.) I took pictures of each of the six faces. Then I used **gimp** (you could use **PhotoShop(tm)**) to trim the images down and resize them. Make the resulting images about 100 pixels on each side. Save them as 1.jpg, 2.jpg, 3.jpg, 4.jpg, 5.jpg, and 6.jpg.

Another option is to simply find images already on the web. You are free to make copies of my dice faces. Drag them off my web page onto your desktop. You can find them here:

http://ipup.doncolton.com/dice/

Upload the six jpg files into the same directory where your dice program is located.

In order to display the actual dice pictures instead of the numbers, we need to modify our program.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";

$dice1 = 1 + int ( rand(6) );
$dice2 = 1 + int ( rand(6) );

$dice1 = "<img src=\"$dice1.jpg\" alt=\"$dice1\">";
$dice2 = "<img src=\"$dice2.jpg\" alt=\"$dice2\">";

print "You have rolled $dice1 and $dice2.\n";
```

We have added two lines. Each line replaces a dice roll number with a dice roll image. The rest of the program is just as it was before.

Hit your browser's "reload" button to run your online program again. Did the dice change?

11.8 Summary

We have demonstrated how to move a program from your desktop to the web. The program itself is modified by the addition of two new lines at the top. The program is copied to a web server and proper permissions established. Lots of small things can go wrong. These need to be corrected. Finally, we can add actual images to our computing experience.

There are two big advantages to being online. (1) Anyone in the world can use your program. (2) It is fairly easy to put graphics into your program.

There is one big disadvantage to being online. Inputs. It is more difficult write a conversational program where you prompt for and receive inputs. You will notice that the programs in this chapter do not use any inputs. For now, we just wanted to get your feet wet. We will go swimming later.

Important Vocabulary: These are words worth memorizing. There are more at the start of the chapter.

rand: a command that returns a random number between zero and one, or between zero and the limit specified.

int: a command that turns a non-integer into an integer by dropping all parts after the decimal. It turns 3.1415 into 3.

index.html: a standard file name for a web page. When a URL refers to a directory (folder), the web server looks for an index file first. If it finds one, it will display it. Otherwise, it will generate and index and display it.

index.cgi: a standard file name for online programs. A few other names are possible but this is the one we will use throughout this course.

Chapter 12

Games and Projects

At the end of each unit we look at some simple games and projects that use our knowledge of the material just learned. These may make good in-class activities or homework assignments.

At this point you should be able to do programs involving output (chapter 1, page 27), input (chapter 3, page 37), simple variables (chapter 4, page 41), operators (chapter 5, page 48), sequence (chapter 8, page 60), online without inputs (chapter 11, page 70), and random numbers (section 11.6, page 75).

Mad Libs (Offline)

http://www.eduplace.com/tales/ has a collection of stories, sometimes called Mad Libs. The user is asked for a number of words or phrases, such as "best friend's name" or "favorite food". These are then used in a fictional story. Your task is to invent and program a **Mad Lib** using at least five different words or phrases provided by the user. Have fun and be creative!

Slot Machine (Online)

Instead of rolling dice, simulate a slot machine program that displays three pictures, such as lemons, oranges, cherries, the digit seven, or other lucky or unlucky symbols. Find images on the web or create your own.

Magic Eight Ball (Online)

http://en.wikipedia.org/wiki/Magic_8-Ball

The user thinks of a question and visits your website. You randomly answer yes, no, or maybe. Use a variety of wordings, such as "Definitely" or "I doubt it". Could use graphics.

Chapter 13

Unit Test: Basics

To demonstrate mastery of this unit, you should be able to write each of these programs (basic string, basic numeric) in 5 to 10 minutes, including testing. As you develop this speed you will be better prepared to master the content in future units and you will be better able to apply your knowledge to opportunities around you.

At this point you should be able to do programs involving output (chapter 1, page 27), input (chapter 3, page 37), simple variables (chapter 4, page 41), operators (chapter 5, page 48), and sequence (chapter 8, page 60).

13.1 Vocabulary

The following words are worth memorizing. They are introduced in earlier chapters of this unit. It is assumed that you know them. They may appear in test questions without further explanation.

sleep: a command that causes your program to stop for some number of seconds before continuing.

semantics: the meaning or substance of things.

syntax: the expressed form of things. Especially the grammar that connects words together.

newline: a special character that tells the computer to begin a new line on its output medium (usually a computer screen).

chomp: a command that removes the last character if it is a newline.

chop: a command that removes the last character of a string, no matter what it is.

lower case: Small letters: abcdefg. Old-time typesetters kept these letters in the lower case on their workbench because they used them more often than the big letters.

upper case: Big letters: ABCDEFG. Also called capital letters. Old-time typesetters kept these letters in the lower case on their workbench because they were less often used so it was okay to have to reach a bit to get them.

case sensitive: Situations where the difference between upper and lower case makes a difference. Is "hello" the same as "HELLO"? If so, we are not case sensitive. If they are different, then we **are** case sensitive.

white space: Characters that are invisible. Examples include spacebar, tab, and newline. They are called "white" because they do not use any ink.

syntax error: A mistake in the grammar of a program. Syntax errors are detected when the program is first read, before any attempt is made to carry out the instructions.

logic error: A mistake in the semantics of a program. Logic errors are detected when the program runs but does not do what the programmer wanted.

local: is a program running at the computer where you are sitting. It is not running on the web.

offline: means a program that is not running on the web.

online: means a program that *is* running on the web.

web server: is a computer that hosts web pages and web-based programs.

rand: a command that returns a random number between zero and one, or between zero and the limit specified.

int: a command that turns a non-integer into an integer by dropping all parts after the decimal. It turns 3.1415 into 3.

index.cgi: a standard file name for online programs. A few other names are possible but this is the one we will use throughout this course.

13.2 Exercises: Strings

Answers to starred exercises can be found on page 284.

Things to watch out for: (a) `\n` and `/n` are not the same. (b) `/` and `\` are not the same.

Pure Output

Exercise 24: Write a program to do the following. Print out the word "Hippopotamus" five times. Don't worry about spacing or newlines.

Basic Strings: Input / chomp

Exercise 25:* Prompt for and read in the name of an animal. Print on one line: "I think (animal) would make a nice pet." where (animal) is replaced by the name that was just read.

Exercise 26:* Prompt for and read in the name of a zoo animal. Print on one line: "I love to see the (animal) at the zoo." where (animal) is replaced by the zoo animal that was just read.

Exercise 27: Write a program to do the following. Prompt for and read in a name. On one line print "Hello, X, how are you?" where the letter X is replaced by the name that was read in.

Exercise 28: Write a program to do the following. Prompt for and read in a name. On one line, print "I think (name) is a lovely name!" where (name) is replaced by the name that was just read.

Exercise 29: Write a program to do the following. Prompt for and read in the name of a person. Print on one line: "I would say that (name) is my best friend." where (name) is replaced by the name that was just read.

Exercise 30:* Prompt for and read in a kind of animal. Prompt for and read in a name. Print out this story (or very similar). "Yesterday I took my pet (animal) to school. My friend (name) laughed all day."

13.3 Exercises: Numeric

Things to watch out for: (a) `\n` and `/n` are not the same. (b) `/` and `\` are not the same. (c) `"$x+$y"` and `$x+$y` are not the same.

Basic Numeric: Input and Simple Calculation

Exercise 31:* Prompt for and read in a number. Add 5. Print the result.

Exercise 32:* Prompt for and read in a number. Subtract 10. Print the result.

Exercise 33:* Prompt for and read in a number. Multiply by 2. Print the result.

Exercise 34:* Prompt for and read in two numbers. Add them. Print the result.

Exercise 35:* Prompt for and read in a number. Double it. Print the result.

Exercise 36:* Prompt for and read in a number. Tell what number comes before it. Tell what number comes after it.

Exercise 37:* Write a program to do the following. Prompt for and read in a number. Do it again. Do it again. Add the three numbers together. Print out the answer.

Exercise 38: Write a program to do the following. Prompt for and read in a number. Multipy it by itself and print the result.

13.4 Exercises: Numeric Story Problems

Exercise 39:* Write a program to do the following. An airplane has 240 seats. Ask how many passengers are on board. Read in the number. Calculate how many seats are empty. Report the answer.

Exercise 40:* Convert miles to kilometers. Assume that one mile is 1.6 kilometers. Prompt for and read in miles. Calculate the answer in kilometers. Print the result in a helpful way.

Exercise 41:* A family plans to save ten percent of its income. Prompt for and read in numeric values for two paychecks (husband's and wife's). Print the total savings in a helpful way.

Exercise 42:* Phonecard. Prompt for and read in values for Connect Fee, Per-Minute Rate, and Minutes Connected. Calculate and print the cost of the call. Example: connect is 0.50, per-minute is 0.05, minutes is 20, the total cost would be 1.50. Don't worry about formatting the number. If 1.50 comes out looking like 1.5, that's okay.

Exercise 43: Write a program to do the following. Prompt for and read in values for Starting Balance, Deposits, and Withdrawals. Calculate and print the Ending Balance.

Exercise 44: Temperature. Write a program to do the following. Prompt for and read in a temperature in Celsius. Convert it to Fahrenheit and print the result in a friendly and helpful way. The formula to get Fahrenheit from Celsius is: F=32+9/5*C.

Exercise 45: Knots. Write a program to do the following. Prompt for and read in a speed in miles per hour. Convert it to knots and print the result. Assume that one mile per hour is equal to 0.868976 knots.

Exercise 46: Prompt for and read in the score for Alvin. Prompt for and read in a score for Betty. Prompt for and read in a score for Charles. Report the average score.

Exercise 47: GPA. Prompt for and read in numeric values for three grades. Calculate the average and print it. Include helpful prompts and descriptive text.

Exercise 48: Write a program to do the following. Amy, Bob, and Cathy are having a party for the children of an orphanage. They each bring cookies to divide evenly among the children. Find out how many cookies each person brought. Report the total. Find out how many children are at the orphanage. Report how many whole cookies each child can receive. Tell how many cookies are left over. (Assume all the cookies are identical.)

Unit II

Making Decisions

Chapter 14

The If Statement

It is important that computers be able to respond to changing needs. This can be done by using an **if** statement. The result is that the computer program chooses to do something or to skip it. The special code is followed if the condition turns out to be **True**. Otherwise it is simply skipped.

14.1 Syntax

The syntax of the `if` statement is as follows:

```
if ( condition ) { special }
```

`if` is the literal word "if".

Parentheses surround the condition. The condition is a mathematical expression that evaluates to either **True** or **False**. Chapter 15 (page 89) discusses these expressions in detail. Here are some examples.

```
if ( $count < 100 ) ...
if ( $name eq "Joe" ) ...
if ( $speed > 55 && $speed < 70 ) ...
```

The special code must be enclosed in curly braces. The special code is a block of instructions to be carried out if the condition is true.

14.2 Coordinated Alternatives

We can handle several alternatives by the coordinated use of several **if** statements.

```
if ( $x == 1 ) { print "x is one" }
if ( $x != 1 ) { print "x is not one" }
```

14.3 Exercises

Answers to starred exercises can be found on page 286.

Exercise 49:* What will the following program print?

```
$x = 9; $y = 3;
if ( $x < $y ) { print "$x is less than $y." }
if ( $x > $y ) { print "$x is greater than $y." }
if ( $x == $y ) { print "$x is equal to $y." }
```

Exercise 50:* What will the following program print?

```
$x = "hello"; $y = "world";
if ( $x < $y ) { print "$x is less than $y." }
if ( $x > $y ) { print "$x is greater than $y." }
if ( $x == $y ) { print "$x is equal to $y." }
```

Exercise 51:* What will the following program print?

```
$x = 9; $y = 3;
if ( $x < $y ) { print "$x is less than $y." }
if ( $x > $y ) { print "$x is greater than $y." }
if ( $x = $y ) { print "$x is equal to $y." }
```

Programming exercises can be found in chapter 25 (page 128).

Chapter 15

Numeric Comparison

We just spent some time looking at decisions that computers can make. In their typical form, decisions either select one of several alternative blocks or decisions cause some block of actions to happen repeatedly. The current chapter focuses on the decisions themselves. Decisions are made using **Boolean** operators.

Mathematical calculations generally result a numeric answers, like `3+2*5` is 13. But some calculations result in answers that are True or False. For example, `$x<5` is True if `$x` is less than 5, and False otherwise.

Calculations that result in True and False are called **Boolean**, after George Boole, a pioneer in computer logic. Their primary use is in things like **if** statements.

Perl tends to output a `1` to indicate True and a `""` (the empty string) to indicate false. Many other languages use `0` to indicate false.

Perl regards these things as False: the empty string, the zero string, and all numeric zeros. Specifically, `""`, `"0"`, `0`, `0.`, `0.0`, `0.00`, etc.

Perl regards these things as True: all other strings and all non-zero numbers.

15.1 Numeric Operators

We can compare two things to see which is less or greater, or whether they are equal. For these operators, the answer is always True or False.

For numbers we use the following symbols.

numeric	what it means
`$A < $B`	is A less than B ?
`$A <= $B`	is A less than or equal to B ?
`$A > $B`	is A greater than B ?
`$A >= $B`	is A greater than or equal to B ?
`$A == $B`	is A equal to B ?
`$A = $B`	copy B into A
`$A != $B`	is A different than B ?

If we say `a<b`, we mean "is `a` less than `b`?"

Thus, `3<5` is `1`, because three is less than five, true.

We use `<=` for "less than or equal to" because the $\leq$ symbol is not found on most computer keyboards.

We use `>=` for "greater than or equal to" because the $\geq$ symbol is not found on most computer keyboards.

Note: `=>` is tempting to write, but means something different.

We use `==` for "equal to" because the `=` symbol is already being used for another purpose (assignment). `$a=$b` means to copy the value from `$b` and store it into variable `$a`. `$a==$b` means to compare `$a` and `$b` and answer with True if they are equal and False if they are different.

We use `!=` for "not equal to" because the $\neq$ symbol is not found on most computer keyboards. Some computer languages use `<>` (less than or greater than) to mean not equal, but `!=` will be used in this course.

When comparing strings (as opposed to numbers), Perl uses `lt`, `le`, `gt`, `ge`, `eq`, and `ne` instead of the mathematical symbols. We will consider this in chapter 21 (page 108).

15.2 Exercises

Important Vocabulary: Words worth memorizing.

True: All non-zero numbers and non-zero strings, including negative ones, are regarded as being true.

False: Zero and the empty string are regarded as being false.

Answers to starred exercises can be found on page 287.

Exercise 52:* What will the following program print?

```
$x = 9;
$y = 3;
print "a" . ( $x < $y ) . "\n";
print "b" . ( $x > $y ) . "\n";
print "c" . ( $x == $y ) . "\n";
```

Chapter 16

Two Alternatives: The Else Statement

In chapter 14 (page 87) we learned that the **if** statement can be used to make special-case behaviors part of the program. We learned that we can say:

```
if ( $x == 1 ) { print "x is one" }
if ( $x != 1 ) { print "x is not one" }
```

In this chapter we introduce **else** as a way to do the same thing without deciding twice.

```
if ( $x == 1 ) { print "x is one" }
else { print "x is not one" }
```

16.1 Syntax

The syntax of the **else** option is as follows:

```
if ( condition ) { truepart } else { falsepart }
```

else cannot be used on its own. It must immediately follow an **if** statement. It then becomes part of that **if** statement and provides the alternative that will happen when the condition is not satisfied.

Here is the placement rule: Each "else" statement is a continuation of the previous "if" statement, which it must immediately follow. An "else" statement that does not immediately follow an "if" statement is a syntax error.

16.2 Completeness

Else is important because it helps us avoid overlooking some alternative. For example, if you drive less than 55 you are fine. If you drive more than 55, you are "fined."

```
if ( $speed > 55 ) { print "you are FINED." }
if ( $speed < 55 ) { print "you are fine." }
```

What if you drive exactly 55? We accidentally left out that alternative so nothing will print. It would be safer to have a catch-all in the program.

```
if ( $speed > 55 ) { print "you are FINED." }
else { print "you are fine." }
```

This way we know we have covered every case.

16.3 Exercises

Important Vocabulary: Words worth memorizing.

default: a value or activity that is used when no other value or activity is specified. In an if statement, else identifies the default action.

Answers to starred exercises can be found on page 287.

Exercise 53:* What will the following program print?

```
$x = 3; $y = 9;
if ( $x < $y ) { print "$x is less than $y." }
if ( $x > $y ) { print "$x is greater than $y." }
else { print "$x is equal to $y." }
```

Programming exercises can be found in chapter 25 (page 128).

Chapter 17

Block Structure

Some years ago when I learned programming there was a new technology that was all the rage. It was called "structured programming." (Today the rage is "object-oriented programming.")

It was a response to a now-famous letter published in 1968 by computing pioneer Edsger Dijkstra entitled "Go To Statement Considered Harmful."

The **Go To** (or **goto**, or **branch**) statement gives great power to computer programming. It is fundamental. It allows programs to repeat instructions over and over. It allows programs to select one group of instructions instead of another group. But it is a sharp knife and most of us should not run with it lest we hurt ourselves. Specifically, it increases the opportunity for bugs in our programs, and some of those bugs are very difficult to find and eliminate.

Code written with **goto** statements can become very confused, with the sequence of control jumping around from place to place. Such code is often called **spaghetti code** because it is hard to tell where it is coming from or going to, much like trying to follow the noodles in spaghetti.

The response was to increasingly design computer languages that avoided unrestricted goto statements by using safer alternatives. These safer alternatives came to be called structured programming and relied on considering programs to be assembled from simpler blocks of code.

See http://en.wikipedia.org/wiki/Structured_programming

At this point in the history of programming, structured programming has advanced from being the next new thing to being so pervasive that it is not

even named. It is just assumed.

What's a Block?

Briefly, a block is a collection of code that has exactly one way in and one way out. This keeps things organized so spaghetti code does not result.

How then do we create alternatives, where two ways (or more) exist? It is done through **nesting**.

We want the power of the **goto** statement without the associated bug-creation dangers. Structured programming gives us that.

A linear sequence of statements has one way in and one way out. It makes a natural block. Therefore it can be treated as though it were a single statement. Curly braces (`{}`) are used in Perl and some other languages to collect a sequence of statements into a block.

In chapter 14 (page 87) we introduced the `if/else` control structure. In chapter 26 (page 134) we consider the `while` and `for` control structures.

The `if/else` control structure has one way in and one way out. While the if/else is in progress there are several directions things could go, but each of them is also a block nested inside the `if` statement, each with one way in and one way out.

The `while` loop and the `for` loop are also control structures with one way in and one way out. There are some restricted **goto** statements available for use within loops. Those are **next**, **last**, and **redo**.

Because a block acts as though it were a single statement, it can be used within another block. We say the blocks can be **nested**.

In Perl, because a block ends with `}`, you can generally skip putting a semi-colon before or after a curly brace. Leaving out the semi-colons when not needed can improve readability by reducing clutter in your program.

Chapter 18

Programming Style

We introduced style in chapter 7 (page 55). We continue and expand our discussion here.

Complexity can paralyze us. Yet most computer programs are complex. Programmers deal with that complexity several ways. In this chapter we examine the role of Programming Style in managing complexity. In chapter 36 (page 172) we look at Subroutines as another tool against complexity.

Computers do not care much how a program is written as long as it parses correctly. Humans, on the other hand, benefit greatly from having everyone else follow rules that we call "Programming Style." Most programming shops (places of employment for programmers) have adopted rules of style that all programmers are expected or required to obey. These rules make everyone's code easier to read and understand. The rules in this chapter are my rules.

18.1 Indentation

Structured programming involves organizing your code into blocks. An if/else statement includes two blocks inside itself: first, the true block, and second, the false block.

By convention, inner blocks are indented several spaces in comparison to the outer block of which they are a part.

In the following example, the `print` statement is nested inside the `while`

loop. For that reason, we traditionally indent it a few spaces. All lines at that same level of nesting should have the same amount of indentation. It makes reading the code easier.

```
$x = 1;
while ( $x <= 100 ) {
  print "$x\n";
  $x++ }
print "done!\n";
```

In the following example, the `if/else` statement contains nested `print` statements.

```
if ( $speed <= 55 ) {
  print "Have a nice day!\n";
} else {
  print "Slow down please!\n";
}
```

In the following example, the closing curly braces have been moved a bit. See whether you think the code is as easy to understand.

```
if ( $speed <= 55 ) {
  print "Have a nice day!\n" }
else {
  print "Slow down please!\n" }
```

For programs submitted to me, it is generally recommended to move the closing curly brace up the the preceeding line unless you believe it makes the program more difficult to read and understand.

In the following example, the closing curly braces have been moved again. See whether you think the code is as easy to understand.

```
if ( $speed <= 55 )
{
  print "Have a nice day!\n";
}
else
```

```
{
  print "Slow down please!\n";
}
```

Personally, I find this style a lot more tiresome and difficult to work with. Mostly it is difficult for me because I cannot fit as much of the program on my screen. It's like watching television with a magnifying glass. I want to see the big picture. This obscures it for me. Compact is usually better.

18.2 Special Numbers

Often a program will rely on certain numbers to make decisions, control loops and influence other aspects. The use of these numbers is typically buried in the code. Sometimes the numbers themselves are buried in the code. Invisibility is generally a bad thing. It is better to assign important numbers to variables (or constants) at the top of the program. Consider this example:

```
for ( $i = 1; $i <= 100; $i++ ) { print "$i\n" }
```

Compare it to this:

```
$Limit = 100;
...
for ( $i = 1; $i <= $Limit; $i++ ) { print "$i\n" }
```

The goal here is to easily modify the program in the future. Let's say that due to a decision by senior management, the 100 limit no longer applies, but has been replaced by a 250 limit. To fix the program, you would need to look for every occurrence of 100 and change it (if appropriate) to 250. Or you could just change the one occurrence.

By convention, such variable names for special variables like this one are written with an initial upper-case letter (e.g., `$Limit`). Ordinary variable names start with a lower-case letter (e.g., `$limit`).

The example above may not look so bad, but think about this:

```
for ( $i = 1; $i < 101; $i++ ) { print "$i\n" }
```

The code produces the same output, but there is no 100 present. How do you find it to change it to 251? You would have to read through every line of code. It is much easier if you can do a search to limit the things you must look at.

The goal is to make it easy to fix the program when future changes are needed. If you decide to make something a certain color, ask yourself whether that color might ever need to be changed. If so, make it a variable and assign it a value at the top of your program.

18.3 Internationalization (i18n)*

Special numbers is the tip of the iceberg. What about changing the words as well? There is increasingly a recognition that we can and should separate the logic of the program from the presentation of the user interface. What that means is the underlying programming is probably true whether you are writing your program to be used in China, England, France, or Brasil. But the presentation, or what the user actually sees, may vary a lot.

Internationalization is the porting of your program into the international world. Because that word is sooo long, a common abbreviation has arisen. **i18n** (eye eighteen en) stands for the letter `i` followed by 18 other letters followed by the letter `n`. Eye 18 N means internationalization.

What changes might be required? First of all, the prompt strings may need to change. Instead of saying "Please enter a number" you might need to say it in Spanish or Chinese.

A common way to approach this is to number the strings and put them in a database. Translated strings could be added. Then the specification of a language (or culture) and string number lets us look up and use the proper wording.

What else? Well, the data used might vary. If we are doing personal names in English we may well assume that people have a first, middle, and last name. Maybe also a suffix like Junior. Maybe also a title like Reverend or Doctor. But some cultures use several middle names. In Hawaii it is common to have three middle names. In Spain the family name is a combination from the father's side and the mother's side. Names reflect culture. It may be necessary to have totally different code for names. Or maybe we really only need three things: (a) the full legal name, (b) the sorting name, and (c) the

familiar name.

The character set may need to change. Early programming was done using ASCII and EBCDIC. ASCII is the American Standard Code for Information Interchange. EBCDIC is Extended Binary Coded Decimal Interchange Code. EBCDIC is an IBM code that grew out of the Hollerith codes that were used for the 1890 Census in the United States. ASCII used seven bits per character. Extended ASCII uses eight bits per character. Unicode uses 16 bits. Consult the web for lots of interesting information.

ASCII: http://en.wikipedia.org/wiki/ASCII

EBCDIC: http://en.wikipedia.org/wiki/Ebcdic

Hollerith: http://en.wikipedia.org/wiki/Hollerith

UNICODE: http://en.wikipedia.org/wiki/Unicode

Chapter 19

Many Alternatives: The Elsif Statement

It is common to have more than two choices. There is an **elsif** option that can be helpful. (Notice that **elsif** is not quite spelled the way you might guess.) Here is an example.

```
if ( $speed <= 55 ) { case1 }
elsif ( $speed <= 70 ) { case2 }
elsif ( $speed <= 90 ) { case3 }
else { case4 }
```

For this example, if speed is 80 the first condition, being less than or equal to 55, is false. Therefore case1 is skipped. Next the program checks for less than or equal to 70. That is also false so case2 is skipped. Next the program checks for less than or equal to 90. This is true so case3 is taken. Because case3 was taken, case4 is skipped.

Sometimes this is called an **else/if** or **elsif ladder**.

Here is the revised placement rule: Each line that starts with "els" (elsif or else) is a continuation of the previous "if" statement. It must immediately follow an "if" or "elsif" statement. An "elsif" or "else" statement that does not immediately follow an "if" or "elsif" statement is a syntax error.

Some languages use **switch** and **case** to do what Perl achieves with **elsif**.

19.1 else vs elsif

```
if ( cond ) { body1 } else { body2 }
```

This is the **if/else** statement. If the condition is true then body1 is executed. If not then body2 is executed.

```
if (cond) { body1 } elsif (cond) { body2 } ... else { bodyN }
```

This is the **if/elsif/else** statement. The if and elsif parts are checked, one by one, until a true is found. The matching body is executed. If nothing is true, the else body is executed.

19.2 Exercises

Answers to starred exercises can be found on page 287.

Exercise 54:* What will the following program print?

```
$x = 3; $y = 9;
if ( $x < $y ) { print "$x is less than $y." }
elsif ( $x > $y ) { print "$x is greater than $y." }
else { print "$x is equal to $y." }
```

Exercise 55:* What will the following program print?

```
$x = 5;
if ( $x < 3 ) { print "x is really small." }
elsif ( $x < 4 ) { print "x is small." }
elsif ( $x < 5 ) { print "x is medium." }
elsif ( $x < 6 ) { print "x is large." }
elsif ( $x < 7 ) { print "x is really large." }
else { print "x is huge." }
```

Exercise 56:* What will the following program print?

```
$x = 5;
if ( $x < 3 ) { print "x is really small." }
if ( $x < 4 ) { print "x is small." }
if ( $x < 5 ) { print "x is medium." }
if ( $x < 6 ) { print "x is large." }
```

```
if ( $x < 7 ) { print "x is really large." }
else { print "x is huge" }
```

Exercise 57:* What will the following program print?

```
$x = 5;
if    ( $x < 4 ) { print "a." }
elsif ( $x < 6 ) { print "b." }
elsif ( $x > 4 ) { print "c." }
else             { print "d." }
if    ( $x > 6 ) { print "e." }
elsif ( $x < 4 ) { print "f." }
elsif ( $x < 5 ) { print "g." }
if    ( $x < 6 ) { print "h." }
elsif ( $x > 4 ) { print "i." }
if    ( $x > 6 ) { print "j." }
elsif ( $x > 5 ) { print "k." }
elsif ( $x < 4 ) { print "l." }
if    ( $x > 6 ) { print "m." }
else             { print "n." }
```

Programming exercises can be found in chapter 25 (page 128).

Chapter 20

And, Or, Xor, Not

To this point we have focused on the comparison operators and not really talked much about how they could be combined into larger and more precise statements.

We can begin with a classic statement of mathematics. When some number x is between two other numbers, such as 3 and 5, we can write

$3 < x < 5$

Unfortunately, we cannot write it this way for the computer. As we will see in the precedence chapter (23, page 117), first we would see if 3 is less than x. The result would be True or False. Then we would check to see if True (or False) is less than 5. This is sure to lead to trouble.

We could nest some `if` statements.

```
if ( 3 < $x ) { if ( $x < 5 ) { ... } }
```

That works.

It is good to have an alternative to nested `if` statements. Another approach is to say it like this.

```
if ( 3 < $x and $x < 5 ) { ... }
```

This will compare 3 with x, and then x with 5, as we actually intended. The two results will then be combined using the **and** operator.

The standard **Boolean** connectives are **and**, **or**, and **not**.

Because we have nesting, we do not actually need Boolean connectives, but they often can help by making our programs more concise and clear.

20.1 Logical And

We have **and** (`&&`), also called "**logical and**" (as distinct from "**bitwise and**").

What does **and** mean? We are combining two quantities. How do we do it?

When we say "the sky is blue and I am happy," we are combining two statements. The first is "the sky is blue." The second is "I am happy." We can determine the truth of each statement separately. Look outside. Is the sky blue? If so, then the truth value of the first statement is True.

When both statements are true, the resulting compound statement is true. If either statement is false, the resulting compound statement is false. That is the way that **and** works.

The **truth table** for **and** looks like this. (We can also use the word **and** instead of the ampersands `&&`.)

expression	expression	value
`1 && 1`	`1 and 1`	`1`
`1 && 0`	`1 and 0`	`0`
`0 && 1`	`0 and 1`	`0`
`0 && 0`	`0 and 0`	`0`

We have a convention for what to do if the input numbers are not one or zero. All non-zero numbers are treated as "true." Only zero is treated as "false." In particular, both positive **and** negative numbers are "true."

Thus, `5&&0` is false, `-7&&12` is true, and `0&&4` is false.

20.2 Logical Or

We have **or** (`||`), also called "logical or" (as distinct from "bitwise or"). When the answer is yes, we use `1` (one), representing true. When the answer is no, we use `0` (zero), representing false.

The `|` symbol on my keyboard is found right above the Enter key and right below the Backspace key. It is also called **pipe** or **broken pipe** or **bar**.

What does **or** mean? We are combining two quantities. How do we do it?

When we say "the sky is blue or I am happy," we are combining two statements. When either statement is true, the resulting compound statement is true. If both statements are false, the resulting compound statement is false. That is the way that **or** works.

The **truth table** for **or** looks like this. (We can also use the word **or** instead of the vertical bars `||`.)

expression	expression	value
`1 || 1`	`1 or 1`	`1`
`1 || 0`	`1 or 0`	`1`
`0 || 1`	`0 or 1`	`1`
`0 || 0`	`0 or 0`	`0`

Once again, non-zero inputs are treated as "true." Only zero is treated as "false."

Thus, `5||0` is true, `-7||12` is true, `0||4` is true, and `0||0` is false.

This version of **or** is also called **inclusive or** because it includes the case where both statements are true. In legal writing it is often phrased "and/or." There is another version of **or** called **exclusive or**.

20.3 Exclusive Or

We have **exclusive or** whose **truth table** looks like this:

expression	value
`1 xor 1`	`0`
`1 xor 0`	`1`
`0 xor 1`	`1`
`0 xor 0`	`0`

With **exclusive or** (**xor**) the result is true when one statement or the other, but not both, is true. Exclusive or is also called **parity**. It is true if an odd number of things were true. It is false if an even number of things were true.

expression	value
`1 xor 1 xor 1`	`1`
`1 xor 0 xor 1`	`0`
`0 xor 1 xor 1`	`0`
`0 xor 0 xor 1`	`1`

If you have a large room with light switches at each door, frequently they are wired in an **xor** arrangement, so that flipping any one switch will change the light (from on to off or vice versa).

and, `&&`, **or**, `||`, and **xor** are available as logical operators. They are also available in slightly different form as bitwise operators. These are discussed in the appendix on binary numbers.

20.4 Not

The unary operator `not` can be included to turn a True into a False or vice versa.

Areas for Caution*

C and Perl behave slightly differently. In C, `5||3` is true, represented by 1. In Perl the answer is also true, but it is represented by 5, the number at which short circuit took over. Language designers tend to agree on basic principles but vary on some details. When programming in a new language it is smart to verify that things work as expected before writing much code.

Chapter 21

String Comparison

Strings have different operators than numbers do. If we use the numeric operator, Perl will convert the string into a number before doing the comparison. In most cases, the number will be zero. This can lead to confusing results.

21.1 Several Kinds of Equal

Most people would agree that "5.0" is equal to "5." To know this we would have to know the meaning of those characters. A child might not know they mean the same thing.

There are things that are equal in meaning but not equal in expression. We may say that a "stone" is equal to a "rock," but the word "stone" does not have the same letters in the same order as does the word "rock." Nor does "5.0" have the same characters as "5."

In this chapter we expand on the distinction between mathematical equality and another kind that is called **lexicographic**. A **lexicon** is a dictionary. Lexicographic comparison is based on the order of words in the dictionary. Another word we use in this context is **literal**, which literally means letter by letter.

21.2 Comparison Operators

Here is a table that lists the operators we use to compare strings. Notice that they are composed of letters but the numeric comparison operators were composed of mathematical symbols like `<`.

numeric	string
`<`	lt
`<=`	le
`>`	gt
`>=`	ge
`==`	eq
`!=`	ne

What does it mean for one string to be less than another? We mean that it appears first in the dictionary. Consider the counting numbers: 1, 2, 3. By saying `1<3` we are saying that 1 comes before 3. Similarly, by saying `"eight" lt "five"` we are saying that the word `"eight"` appears earlier in the dictionary than does the word `"five"`.

We can compare any two strings as follows. We begin with the first letter (or character) of each string. If they are different, we are done, and the string whose character is earlier in the alphabet is regarded as less than the string whose character comes later in the alphabet.

If they are the same, we move along to the next character and repeat the process. This continues until a difference is found, or until both strings end at the same time, or until one string ends but the other does not.

The statement `"eight" lt "eighty"` is true because the word `"eighty"` would appear after the word `"eight"` in the dictionary.

When does “eight” come before “seven”? In the dictionary.

Even more interestingly, “9” is less than “10” numerically, but “10” is less than “9” lexicographically. Numerically we are comparing the values. Lexicographically we are comparing the characters that express those values.

21.3 Upper Case, Lower Case

Capital letters can be an area of difficulty. Should upper-case `E` be treated the same as lower-case `e`? Fortunately we can take control of the situation.

uc converts a string to **upper case**.

```
$name = "MacDonald";
$name2 = uc ( $name ) ;# parens are optional
print $name2; # prints MACDONALD
```

lc converts a string to **lower case**.

```
$name = "MacDonald";
$name2 = lc ( $name ) ;# parens are optional
print $name2; # prints macdonald
```

If you are bringing in input and want to immediately convert it to upper or lower case, you can use this short cut and do it in one step instead of two.

```
$name1 = lc <STDIN> ;# convert input to lower case
$name2 = uc <STDIN> ;# convert input to upper case
```

However, if you do it this way, the original input is lost. If you need the original input, use a two-step process.

```
$name1a = <STDIN> ;# capture the original input
$name1b = lc $name1a ;# convert input to lower case
$name2a = <STDIN> ;# capture the original input
$name2b = uc $name1b ;# convert input to upper case
```

21.4 A-I, J-R, S-Z, etc.

How might we compare strings?

Imagine a school where students are assigned to advisors depending on their last name. Students from A-I go to Smith. Students from J-R go to Jones. Students from S-Z go to Knight.

Here is the first attempt at such a program. (It is wrong, by the way.)

```
if ( $name ge "A" && $name le "I" ) { ... Smith ... }
if ( $name ge "J" && $name le "R" ) { ... Jones ... }
if ( $name ge "S" && $name le "Z" ) { ... Knight ... }
```

This looks like a fairly direct translation of the problem. The error is that `$name` is not the first letter of the name. It is the entire name. A name like "Ivan" is not equal to "I". In fact, the program above will correctly assign all students except those whose names start with "I", "R", or "Z".

When we said "A-I" we meant to include all students whose names start with any of those letters, including names like "Ivan." But "Ivan" is not less than or equal to "I." This is an example of an implicit requirement. Although the problem literally says "A" through "I" it means something slightly more inclusive. Therefore, we cannot use `le "I"` as our decision.

There are two solutions. (1) Find a way to isolate the first letter of the name. We will see how to do this when we consider regular expressions in chapter 52 (page 237). (2) Expand the boundaries to cover more names. In this chapter we will take this second approach.

```
if ( $name ge "A" && $name le "Izz" ) { ... Smith ... }
if ( $name ge "J" && $name le "Rzz" ) { ... Jones ... }
if ( $name ge "S" && $name le "Zzz" ) { ... Knight ... }
```

Notice we have added "zz" to the boundary. Now a name like "Ivan" will be correctly decided because "Ivan" is less than or equal to "Izz".

Technically we have not covered all possible names, but we have certainly covered all likely names. Missing still are names that start with "Izz" for instance. Maybe there is someone named "Izzy." That could mess us up. But in general, "Izz" should suffice.

Still better would be an approach that really covers the whole range including highly improbably names.

```
if ( $name ge "A" && $name lt "J" ) { ... Smith ... }
if ( $name ge "J" && $name lt "S" ) { ... Jones ... }
if ( $name ge "S" && $name le "Zzz" ) { ... Knight ... }
```

We don't have anything after "Zzz" such that we could say strictly less than. But that suggests a simplification. There *is* nothing after "Zzz" so

why bother checking? Likewise there is nothing before "A" so why bother checking that either?

```
if ( $name lt "J" ) { ... Smith ... }
if ( $name ge "J" && $name lt "S" ) { ... Jones ... }
if ( $name ge "S" ) { ... Knight ... }
```

One more short cut is begging to be taken. Twice we compare against "J" and twice we compare against "S". We could to that with an `else`-type statements.

```
if ( $name lt "J" ) { ... Smith ... }
elsif ( $name lt "S" ) { ... Jones ... }
else { ... Knight ... }
```

The thing to remember is this: When we cannot explicitly state the end of the range, perhaps we should pick the first thing beyond the range and use that as our standard instead. `le "I"` does not work. `le "Iz"` is better. `le "Izz"` is better still. But `lt "J"` is best of all.

21.5 Barewords

A word about **barewords**. Any word without quotes or `$` or some other thing to characterize it is called a **bareword**, meaning a word that is bare of any indicator. When Perl finds a **bareword** it first tries to see whether it might be a special procedure such as `while` or `else` or `chomp` or a user-defined **subroutine**. After failing to find it on any of those lists it pretends that we put quotes on the word and it treats it as a string.

```
if ( $name lt "N" ) { ... # this is a good example
if ( $name lt N ) { ... # this may work but is wrong
```

Areas for Caution*

Internationalization. Comparing words for dictionary order can vary by language. Be alert to possible problems. However, that is beyond the scope of this book.

21.6 Exercises

Important Vocabulary: Words worth memorizing.

uc: a command to convert a string to all upper case letters.

lc: a command to convert a string to all lower case letters.

bareword: a word without quote marks, $, or some other thing to characterize it. Generally it is treated as though quotes were present. Barewords are dangerous and fragile.

Answers to starred exercises can be found on page 288.

Exercise 58:* What will the following program print?

```
$x = "hello";
$y = "world";
print "a" . ( $x < $y );
print "b" . ( $x > $y );
print "c" . ( $x == $y );
```

Chapter 22

Remainder

We assume you already know about add, subtract, multiply, divide, and assign. Here we add a handy operator for calculating the remainder of division.

22.1 Remainder

Normally when we think of division, we think of calculator division, where `5/2` is `2.5`.

Sometimes we want the remainder.

`5 / 2 = 2 r 1`, five divided by two is two with a remainder of one.

The percent sign (`%`) is used to indicate integer division, but usually we only want the remainder. `11%4` means eleven divided by four, but just the remainder is kept, which is 3.

Remainder is sometimes called **modulus** or **mod**. Division with remainders is sometimes called "clock arithmetic" because 11 o'clock plus three hours equals 2 o'clock on a 12-hour clock: `(11+3)%12` is 2.

When we are trying to be specific, regular division is called floating-point division. With integer division, the decimal point is fixed at the end of the number, never in the middle.

Floating point means that the decimal can be in the middle of the number. Usually when we divide two numbers, we mean floating point divide.

22.2 Cookies and Children

One example that I frequently use to describe integer division is my "cookies and children" example. In this example, we imagine that we have, say, eleven cookies and four children. We wish to divide the cookies.

Now, because we are dealing with children, there are several rules we must follow. (1) We must give each child the maximum number of cookies possible. (2) We must give each child the same number of cookies as every other child. (3) We must not break any of the cookies. (4) Any extra cookies are kept by the "mommy."

`11%4` means eleven cookies, four children. Each child gets two cookies and there are three cookies left over for the mommy. The answer is 3 (the number the mommy gets).

22.3 Calculating Leap Years

Given a year number, tell whether it is a Leap Year or not. Leap Year is the year in which February has 29 days. The rest of the time February has 28 days. (We are using the Gregorian calendar, which is the common calendar used for business throughout the world.)

Rules: Normal years are not leap years. But if the year is divisible by 4, it is a leap year. Unless it is divisible by 100, in which case it is not a leap year. Unless it is divisible by 400, in which case it really is a leap year. Confused yet?

2000 is a leap year. 2100 is not a leap year. 2008 is a leap year. 2009 is not a leap year.

Write the program. You will learn a lot.

Hint: the rules shown above are meant to be understood by a human. If you try to translate them directly into Perl you are likely to make some mistakes. Think carefully.

It generally helps to look for the most simple cases first and deal with them. For leap years there are the non-4s, the 4s, the 100s, and the 400s. 4s are complicated because they can be 100s or 400s. Deal with them last.

22.4 Integer Division*

In Perl, division is always done using floating-point numbers unless you make a special effort to force it to be integer. Don't bother.

One option is to use the directive "use integer;" in your Perl program. It will cause all numbers to be whole numbers.

```
use integer;
```

That is probably overkill for your particular needs. To do an integer divide on just one instance, there are two other options. For one, you can subtract the remainder before doing the division.

```
$leftover = $cookies % $children;
$each = ( $cookies - $leftover ) / $children;
```

Another option is to use the `int` mathematical operator that removes the fractional part of a number and trims it back to just the integer part.

```
$each = int ( $cookies / $children );
```

Negative numbers can be problematic. Is -11/4 equal to -3 with a remainder of 1, or -2 with a remainder of -3? It is clear that the remainder will be between plus-or-minus the denominator (4 in this case). Different programming languages may handle this differently. Fortunately, we are usually dealing with positive numbers.

Chapter 23

Precedence

When we say something like "three plus two times five" there are two ways to interpret it. We could do the plus first, then the times, for an answer of 25. Or we could do the times first, then the plus, for an answer of 13.

When there is more than one reasonable answer, it is called **ambiguity**.

Humans deal with ambiguity fairly well in most cases. Often we do not even notice it. Common sense is called into play and the reasonable alternatives are rated for likelihood. The unlikely alternatives are dropped. If there is only one left, we are happy. If there are two or more reasonable alternatives, we recognize that ambiguity exists.

In a case like $3 + 2 \times 5$, there is not much common sense to call upon. So we have commonly accepted rules. These rules are called the rules of **precedence**.

Precedence comes from the word **precede**, meaning to go before, or to happen first.

For convenience and simplicity, mathematicians have standardized on the precedence of multiplication before addition. It did not have to be that way, but by convention we have agreed as a group to follow that rule.

In the $3 + 2 \times 5$ example, this means we multiply 2 and 5 for a result of 10. We then add 3 for a result of 13.

If we want the other meaning, we use parentheses to alter the precedence. We say $(3 + 2) \times 5$, meaning add 3 and 2 for a result of 5. Then multiply that by 5 for a result of 25.

Writing without parentheses is shorter (more compact). Shorter is more desirable because it is less typing and less cluttered. It is less work to write, but it does require us to memorize the rules of precedence so we will get the same answer as everyone else.

23.1 Precedence Tables

By **convention** (that is, by common agreement), multiply, divide, and remainder all have the same precedence, and we do the operations within that class from left to right.

`8/4/2` means 8 divided by 4, then that divided by 2. `8/4=2` and `2/2=1` so the answer is 1.

If we did not follow the left-to-right precedence, we might have figured it like this: `4/2=2` and `8/2=4` for an answer of 4.

We can see that the order is significant. It makes a difference.

Here is a simple precedence table.

rank	operators	dir
1	`*` `/` `%`	l-r
2	`+` `-`	l-r

This table means that the top-ranking precedence, the number one precedence, is shared by multiply (`*`), divide (`/`), and remainder (`%`). Within that class, precedence is left-to-right (l-r).

Similarly, the second-ranking (number two) precedence is shared by add (`+`) and subtract (`-`). Within that class, precedence is also left to right (l-r). (Most precedence is left to right.)

Here is a more complete precedence table.

rank	operators	dir
1	`++ -- -`	unary
2	`* / %`	l-r
3	`+ - .`	l-r
4	`< <= > >=`	l-r
5	`== !=`	l-r
6	`&&`	l-r
7	`||`	l-r
8	`and`	l-r
9	`or`	l-r

23.2 Unary Operators

`++` and `--` are introduced when we talk about loops. Here we only wish to mention their precedence.

Rarely the `++` or `--` is used in part of a longer expression. This can be confusing and should be avoided. But here is how it works. If the operator comes after the variable name, it is called post-increment (or post-decrement), and the variable is incremented after the rest of the calculation is done. If the operator comes before the variable name, it is called pre-increment (or pre-decrement), and the variable is incremented before the rest of the calculation is done. For example:

```
$a = 5; $b = 4;
$x = $a++ * --$b + 3;
print "$a $b $x\n"; # 6 3 18
```

We can split out the increments and decrements as follows.

```
$a = 5; $b = 4;
--$b; # pre-decrement: 4 -> 3
$x = $a * $b + 3; # 5 * 3 + 3
$a++; # post-increment: 5 -> 6
print "$a $b $x\n"; # 6 3 18
```

Like I said, this can be confusing. The number of times it has been truly useful to me is very small. Avoid combining `++` and `--` in expressions with other things. When possible use them all by themselves.

23.3 Short Circuits

In the case of "logical and" and "logical or" there is a shortcut we can frequently take to reach an answer. This shortcut is called a "short circuit."

To better explain this, consider the following mathematical calculation:

$$13534582934581254 \times 0$$

Can you determine the answer? Can you do so without the aid of a calculator?

We know that when any number is multiplied by zero, the answer is zero. This is a shortcut we can use to determine the result. We may even be able to determine the answer to a calculation before all the parts are known. For example, if I say: $0 \times 1353\ldots$ you can tell me the result before I provide the rest of that big and ugly number. You would have "short circuited" the calculation.

When we use "and" to combine two numbers, if the first number is zero, we know the answer will be false (zero).

When we use "or" to combine two numbers, if the first number is non-zero, we know the answer will be true (one).

In both of these cases, we do not even bother to calculate the rest of the formula. By the rules of computing, we are actually forbidden to calculate the rest of the formula. We must stop when we know the answer.

When does this matter? Here is an example.

"`5/0`" is `error` because dividing by zero is illegal.

"`5%0`" is `error` because dividing by zero is illegal.

"`5%0 || 7`" is `error` because dividing by zero is illegal.

"`7 || 5%0`" is `true`. The divide by zero never happens because short-circuiting tells us the answer: `true or` anything is `true`.

"`5/0 && 0`" is `error` because dividing by zero is illegal.

"`0 && 5/0`" is `false`. The divide by zero never happens because short-circuiting tells us the answer: `false and` anything is `false`.

There is no short-circuit for "exclusive or."

23.4 Assignment

Assignment happens last. It also works right to left instead of left to right. For example, we can initialize several variables at the same time like this.

```
$a = $b = $c = 0;
```

The actual order of execution is that zero gets copied into `$c`, which is then copied into `$b`, which is then copied into `$a`.

Chapter 24

Online

In chapter 11 (page 70) we showed how to put simple programs online. By "simple" we meant something that does not work with user input. In this chapter we build on that foundation by showing how to receive simple (closed set) user input, specifically the pressing of buttons, into an online program. In chapter 51 (page 228) we show how to receive multiple and arbitrary (open set) user inputs.

Our example for this chapter is Rock, Paper, Scissors, online.

This is a classic game that uses random choices. Each player picks either rock, paper, or scissors. Then simultaneously they reveal their choices and a winner is determined. Rock beats scissors. Scissors beats paper. Paper beats rock. Although it is very simple to play, it is actually a very challenging strategy game where each opponent tries to guess what the other will select based on analysis of what they have done in the past.

See http://en.wikipedia.org/wiki/Rock-paper-scissors.

See http://en.wikipedia.org/wiki/Jan-ken-pon.

24.1 Rock Paper Scissors, Online, No Inputs

We will start out simple, with no inputs. Instead we simulate two players and we randomly select a throw for each of them.

We start out with some introductory lines. The `#!` line tells the server that this is a script and it is written in a language that can be understood by

the `/usr/local/bin/perl` program. The `content-type` line is the server talking to the browser and telling it that it is receiving a web page written in the `text/html` language. The `<h1>` line simply puts an informative title at the top of the screen.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";
print "<h1>Don Colton's Rock Paper Scissors</h1>\n";
```

We will set up three variables that we can use throughout the program. They represent Rock, Paper, and Scissors. This makes it easier for us to change wordings later, or to fix spelling errors.

```
$r = "ROCK";
$p = "PAPER";
$s = "SCISSORS";
```

We will randomly throw a choice for the first player. We do it as a number between zero and two (inclusive). We then translate it into rock, paper, or scissors. Advanced: Section E.3 (page 320) provides an interesting improvement to this approach utilizing subroutines and arrays.

```
$p1 = int ( rand(3) ); # 0, 1, 2
if ( $p1 == 0 ) { $rps1 = $r }
if ( $p1 == 1 ) { $rps1 = $p }
if ( $p1 == 2 ) { $rps1 = $s }
print "Player 1 throws $rps1<br>\n";
```

We do similar actions to create a throw for player 2.

```
$p2 = int ( rand(3) ); # 0, 1, 2
if ( $p2 == 0 ) { $rps2 = $r }
if ( $p2 == 1 ) { $rps2 = $p }
if ( $p2 == 2 ) { $rps2 = $s }
print "Player 2 throws $rps2<br>\n";
```

By this point we have two variables, `$rps1` and `$rps2`, that hold the results of the two throws. We need to compare the throws and declare a winner. The easy case is when they are both the same.

```
if ( $rps1 eq $rps2 ) { print "It's a DRAW.\n" }
```

The other case is when they are different. We have six alternatives represented by the lines below.

```
if ( $rps1 eq $r && $rps2 eq $s ) {
  print "$r breaks $s! Player 1 wins!\n" }

if ( $rps1 eq $s && $rps2 eq $r ) {
  print "$r breaks $s! Player 2 wins!\n" }

if ( $rps1 eq $r && $rps2 eq $p ) {
  print "$p smothers $r! Player 2 wins!\n" }

if ( $rps1 eq $p && $rps2 eq $r ) {
  print "$p smothers $r! Player 1 wins!\n" }

if ( $rps1 eq $p && $rps2 eq $s ) {
  print "$s cut $p! Player 2 wins!\n" }

if ( $rps1 eq $s && $rps2 eq $p ) {
  print "$s cut $p! Player 1 wins!\n" }
```

24.2 Adding Graphics

Let us upgrade the rock, paper, and scissors variables to display appropriate images. These **img** statements cause actual images to be displayed on the screen. They require three jpg files to be placed in the same directory as our RPS program. The **height** parameters cause the images to be all the same size.

```
$r = "<img src='rock.jpg' alt='rock' height='100'>";
$p = "<img src='paper.jpg' alt='paper' height='100'>";
$s = "<img src='scissors.jpg' alt='scissors' height='100'>";
```

24.3 The SUBMIT Button

In its current form the program can be run over and over again by hitting the browser's reload button.

We can upgrade the program by adding a **submit** button at the top of our program, right after the `<h1>` line.

```
print "<form method='post' action=''>";
print "<input type='submit' name='x' value='Throw Again'>";
```

This does two important things. First it creates a **form**. The main two parameters are the **method**, which we have specified to be **post**, and the **action**, which we have specified to be nothing, or rather, nothing besides the default which is to run the same program again. Without a **form**, no inputs can be sent.

Second creates a **submit** button. The **value** part of the submit button is the actual text that will appear on the button. The **name** part is something we will talk about later when we do CGI programming. For now it does not matter what it is.

The placement of these two lines is as follows. The **form** must be introduced before any inputs can happen. Normally we put it toward the top of the program, possibly right after the content-type line.

The **submit** button can appear any place after the form is introduced.

Technically by using a **form** we have created input for our program. We have not explained how our program receives that input. Indeed, the current program actually ignores any input it receives. The only effect is the **action**, which causes the program to be run again.

Chapter 51 (page 228) covers this material in greater detail.

24.4 Rock Paper Scissors, Online, With Inputs

Next we will do some actual inputs. We will have three submit buttons labeled ROCK, PAPER, and SCISSORS, to allow the human to play against the computer.

Add these lines right after your content-type line:

```
chomp ( $input = <STDIN> );
print "DEBUG: input was (($input))<br>\n";
```

Remove this line (or whatever you made it):

```
print "<input type='submit' name='x' value='Throw Again'>";
```

Add these lines at the bottom of your program:

```
print "<h1>Make Your Throw</h1>\n";
print "<input type='submit' name='x' value='ROCK'>\n";
print "<input type='submit' name='x' value='PAPER'>\n";
print "<input type='submit' name='x' value='SCISSORS'>\n";
```

Run your program. Each time it runs, the button you pressed should be indicated in the debug line at the top. Keep working on it until that happens. Then continue.

Next, we will change the following lines. Here is the old version.

```
$p1 = int ( rand(3) ); # 0, 1, 2
if ( $p1 == 0 ) { $rps1 = $r }
if ( $p1 == 1 ) { $rps1 = $p }
if ( $p1 == 2 ) { $rps1 = $s }
print "Player 1 throws $rps1<br>\n";
```

Here is the new version.

```
if ( $input eq "x=ROCK" ) { $rps1 = $r }
if ( $input eq "x=PAPER" ) { $rps1 = $p }
if ( $input eq "x=SCISSORS" ) { $rps1 = $s }
print "Player 1 throws $rps1<br>\n";
```

With this change, suddenly your program is receiving inputs and acting upon them. You can actually play against the computer.

When you have it working, comment out the DEBUG line.

24.5 User Interface

The user interface is the totality of how the user interacts with the program. It includes whether there is a submit button, and where it is located on the screen, and what it says. It includes whether there are graphic images or simple text. It includes the artistry and graphical design.

We have intentionally left this version of Rock Paper Scissors in a very simplistic form. You the reader could probably make a number of improvements to make the program more fun to use and more effective at communicating with the user.

What can you do to improve it?

Chapter 25

Unit Test: Choices

To demonstrate mastery of this unit, you should be able to write each of these programs (numeric choices, string choices) in 5 to 10 minutes, including testing. As you develop this speed you will be better prepared to master the content in future units and you will be better able to apply your knowledge to opportunities around you.

At this point you should be able to do programs involving if (chapter 14, page 87), else (chapter 16, page 92), elsif (chapter 19, page 101), Boolean operators (chapter 15, page 89), and integer divide and remainder (chapter 22, page 114).

25.1 Vocabulary

The following words are worth memorizing. They are introduced in earlier chapters of this unit. It is assumed that you know them. They may appear in test questions without further explanation.

True: All non-zero numbers and non-zero strings, including negative ones, are regarded as being true.

False: Zero and the empty string are regarded as being false.

default: a value or activity that is used when no other value or activity is specified. In an if statement, else identifies the default action.

uc: a command to convert a string to all upper case letters.

lc: a command to convert a string to all lower case letters.

bareword: a word without quote marks, `$`, or some other thing to characterize it. Generally it is treated as though quotes were present. Barewords are dangerous and fragile.

Answers to starred exercises can be found on page 288.

25.2 Exercises: Numeric

Things to watch out for: (a) remember that `=` is the assignment operator, not a comparison operator. (b) if numbers are specified in the problem, try hard to use those same exact numbers in the answer. (c) `=>` and `>=` are not the same. (d) `3<$x<5` does not work the way you might think.

Easy Numeric Conditionals

Exercise 59:* Prompt for and read in a number. If it is less than zero print "You are in trouble." If not, print "So far, so good."

Exercise 60: Using approved style, write a program to do the following. Prompt for and read in two numbers. If the first is less, print "Things are getting better." If the first is more, print "Things were better before." You can assume the numbers will not be the same.

Exercise 61: Using approved style, write a program to do the following. Prompt for and read in a number. If it divides evenly by 2, print "This number is even." Otherwise print "This number is odd."

Exercise 62: Using approved style, write a program to do the following. Prompt for and read in a number. If it is greater than 10, print "That's a big number." Otherwise print "That's not so big."

Exercise 63: Using approved style, write a program to do the following. Prompt for and read in a number. If it is 100 or more, print "WOW". If not, print "Oh well."

Exercise 64: Using approved style, write a program to do the following. Prompt for and read in a number. If it is an even number, print "(this number) is EVEN." If it is an odd number, print "(this number) is ODD." Replace (this number) by the actual number that was read in.

Exercise 65: Using approved style, write a program to do the following. Prompt for and read in two numbers. Print "The smaller number is X". Print "The bigger number is Y". Replace X by the smaller number. Replace Y by the bigger number. You can assume the numbers will not be the same. Smaller means less than.

Medium: Elsif Numeric Conditionals

Exercise 66: Using approved style, write a program to do the following. Prompt for and read in a distance. If the distance is 500 miles or more, print "you should fly." If it is 100 to 500 miles, print "take the train." If it is less than 100 miles, print "drive your car."

Exercise 67: Using approved style, write a program to do the following. Prompt for and read in a number. If it is 4, print "You get an A." If it is 3, print "You get a B." If it is 2, print "You get a C." If it is anything else, print "You need to try again."

Exercise 68: Using approved style, write a program to do the following. Prompt for and read in a number. If it is between five and ten (inclusive), double it and print out the answer. If it is less, subtract three and print out the answer. If it is greater, add seven and print out the answer.

Exercise 69: Using approved style, write a program to do the following. Ask for and read in two numbers. If they are more than 10 apart, print that they are far apart. If they are between 1 and 10 apart, print that they are close to each other. If they are closer than 1 print that they are almost identical.

Exercise 70: Using approved style, write a program to do the following. Prompt for and read in a number. If the number is a multiple of 2, print "This number is even." Otherwise print "This number is odd." If the number is a multiple of five, print "This number is round." If the number is a multiple of seven, print "This number is lucky." If the number is a multiple of 13, print "This number is unlucky." Put each printed statement on a line by itself. 35 is odd and round and lucky.

Compound Conditionals

Exercise 71:* Using proper programming style, prompt for and read in a number. If it is even and not a multiple of 13, print "Yeehah". Otherwise print "Ouch". Use %. For 26 the answer is Ouch. For 28 the answer is Yeehah. For 31 the answer is Ouch.

Exercise 72: Using approved style, write a program to do the following. Prompt for and read in two numbers. If one is a factor of the other, say "(x) is a factor of (y)." Otherwise say "no factor found." Replace (x) and (y) appropriately. (A factor is a number that divides into another number with no remainder. Examples: With 6 and 3, say 3 is a factor of 6. With 3 and 6, say 3 is a factor of 6. With 4 and 7, say no factor found.)

25.3 Exercises: Strings

Things to watch out for: (a) use the proper comparison operator. `==` and `eq` are not the same. (b) if strings are specified in the problem, try hard to use those same exact strings in the answer. (c) String literals should be in quote marks.

Exercise 73: Using approved style, write a program to do the following. Prompt for and read in three numbers. If any two of the numbers add up to the other one, print "Kangaroo." Otherwise print "Platypus." Kangaroo examples: 1 2 3. 3 2 1. 4 5 9. 4 4 8. -1 1 2. Platypus examples: 1 2 4. 1 1 1. 6 3 2.

Exercise 74: Using approved style, write a program to do the following. Teachers: Read in the name of a class. If it is "CIS101" print "Your teacher is Prof Colton." If it is "CIS350" print "Your teacher is Prof Lee." If it is "CIS470" print "Your teacher is Prof Stanley." Otherwise print "I don't know who your teacher is."

Exercise 75:* Read in a department prefix. If it is "ACCT" print "Accounting". If it is "POSC" print "Political Science". If it is "CIS" print "Computer and Information Sciences". Otherwise print "Unknown".

Exercise 76:* Using approved style, write a program to do the following. A certain dictionary is so big it is divided into two volumes (separate books). The first has words up to and including "leaf". The second has all words after that. Prompt for and read in a word. Tell whether the word would be

in the first volume or the second volume. Assume small letters will be used (not CAPITAL letters).

Exercise 77:* The will-call line for concert tickets is divided A-G, H-N, O-U, and V-Z. Prompt for and read in a name. Convert to upper case. Check which line the person should stand in. Print "You should stand in the H-N line" (but say the correct line for that person).

Exercise 78:* Using approved style, write a program to do the following. Prompt for and read in a name. If it starts with "N" or later, print "Please go to table 2." If it starts with "M" or earlier, print "Please go to table 1."

Exercise 79:* Using approved style, write a program to do the following. Your friends are having a party. Food has been assigned according to last name, with A-L bringing breads, M-Q bringing salads, and R-Z bringing desserts. Prompt for and read in the last name. Print out the food type to be brought.

Exercise 80: Using approved style, write a program to do the following. A certain encyclopedia is divided into twenty volumes (books). In volume 12 the first entry is "Matrimony" and the last entry is "Neoplasm". Prompt for and read in a word. If it would be in volume 12, print "Try Volume 12." If not, print "Don't try Volume 12." Examples: "Elephant" is not. "music" is. "Music" is.

Exercise 81: Using approved style, write a program to do the following. Special half-price tickets for the Superferry will be sold on Friday. Sales times depend on your last name. A-K can buy tickets from 8 AM to 10 AM. L-Q can buy tickets from 11 AM to 1 PM. R-Z can buy tickets from 2 PM to 4 PM. Besides those times tickets are not available. Prompt for and read in a last name. Respond with the time that tickets are available.

Exercise 82: Using approved style, write a program to do the following. The will-call line for concert tickets is divided A-J, K-O, and P-Z. Prompt for and read in a name. Convert to upper case. Check which line the person should stand in. Print "You should stand in the A-J line" (but say the correct line for that person).

Unit III

Repeating Actions

Chapter 26

While Loops

What does it mean when I say "1, 2, 3, ..., 100"?

If you guessed the numbers from 1 to 100, you are right. The three dots has a special meaning. It means (a) there is a pattern, (b) it should be obvious, (c) the pattern continues, (d) the pattern starts and ends as indicated, at 1 and 100.

Obvious. Implicit but obvious. Obvious to humans. Not obvious to computers.

So, how do we explain it to the computer? How do we make the things that are obvious to us obvious to it?

```
print "1, ";
print "2, ";
print "3, ";
(etc.)
print "100";
```

We *could* do that. It would be painful and heartbreaking to write. But we *could* do it. If we make it a thousand or a million it is really not fun to even think about.

How do we explain it to the computer?

26.1 Finding the Pattern

In the 1, 2, 3, example, the pattern is that we add one to the number to get the next number. We could write a program like this.

```
$x = 1;      print "$x, ";
$x = $x + 1; print "$x, ";
$x = $x + 1; print "$x, ";
$x = $x + 1; print "$x, ";
(etc.)
```

Writing our program this way makes the pattern explicit but it also makes the program take about twice as many statements. This is a case where it is worth our time to take a step backward so we can take a big step forward.

26.2 Explicit versus Implicit

Explicit versus **implicit**. It's vocabulary time. Explicit and implicit share the same ending: plicit. Ex means out, like export. Im means in, like import. "ply" is fold. The root is plectere, meaning to plait or braid. It is found in "implicit" and "explicit." When something is implicit, the meaning is folded in, or braided, or plaited. Hidden, but still present. When something is explicit, the meaning is folded out for all to see. The same root is plex in complexity, pli in simplicity and pliers and pliable, and ply in plywood.

Humans tend to like things implicit, up to a point. Example: I say "do you know what time it is?" What do I mean? I mean, "I want to know the time. Please tell it to me." It would be considered direct but rude to say it that way. So we fold the meaning inside. We do not come right out and say what we mean. Instead, we hint at the meaning and expect the other person to figure it out. Humans are funny creatures.

Computers do not do well with implicit. Computers need explicit.

26.3 Repetition

Some actions should be repeated until a condition is satisfied (or while a condition is true). Virtually every programming language provides a way to specify such **looping**, also called **iteration**.

Armed with the explicit pattern, we can now arrange it into a loop.

```
$x = 1;
repeat {
  print "$x, ";
  $x = $x + 1; }
```

The program is suddenly *lots* shorter. Like four statements instead of 200. That's our gigantic step forward. But there are several difficulties yet to overcome.

We are introducing a new word here, "**repeat**." Repeat is not actually part of Perl. It is part of English. So our use of it here is what we call **pseudo** code. Pseudo means fake or imitation. It is really more of a comment.

We need the real word that Perl understands. And we need some way to stop when we get to 100. But at least we have the concept here.

26.4 while Loop

The most simple loop in Perl is the **while** loop. The syntax is very much like the **if** statement, except that after doing the true part (called the body of the loop), the condition is tested again. If it is still true, the body is executed again. And again. And again. Until the condition becomes false.

```
while ( condition ) { body }
```

Here is an example program that prints out the numbers from 1 to 100.

```
$x = 1;
while ( 1 ) {
  print "$x, ";
  if ( $x >= 100 ) { last }
  $x = $x + 1; }
```

We replaced the "repeat" with "while (1)" and we added a line to help us quit when we reach our goal.

Because `1` means true, `while(1)` means while true, or, in other words, forever.

last causes the loop to exit immediately. Execution continues with the next statement (if any) after the loop.

Here is how the program works.

The variable `$x` receives the initial value of 1. Then the **while** loop begins. It executes the body of the loop one time. This will print out the number 1. Then it will check to see if we reached our goal. Then it calculates a new value for `$x` setting it one higher than before.

Next, because this is a loop, we do it all again. We print 2. We check against our goal of 100. We increment.

Next, we print a 3, check the limit, and change `$x` to 4.

dot dot dot

Next, we print a 99, check the limit, and change `$x` to 100.

Next, we print a 100, check the limit, and terminate the loop.

while Condition

Instead of saying `while(1)`, we can replace the `1` with a more complicated condition. Then we can do away with the **if** statement inside the loop.

```
$x = 1;
while ( $x <= 100 ) {
  print "$x, ";
  $x = $x + 1; }
```

The condition, `$x<=100`, keeps us in the loop so long as the condition is true. On the earlier example, we exited the loop when we passed 100. Here we stay in the loop until we reach 100.

Here is how the program works.

The variable `$x` receives the initial value of 1. Then the **while** loop begins. It checks the condition and finds that 1 is less than or equal to 100 (true), so it executes the body of the loop one time. This will print out the number 1. Then it calculates a new value for `$x` setting it one higher than before.

Next, because this is a loop, the condition is checked again. We find that 2 is still less than or equal to 100, so we execute the body again. Then `$x` is incremented to become 3.

Next, we print a 3 and change `$x` to 4.

dot dot dot

Next, we print a 99 and change `$x` to 100.

Next, we print a 100 and change `$x` to 101.

Next, we find that 101 is no longer less than or equal to 100, so we break out of the loop.

26.5 last, next, and redo

Perl provides some options for changing the way the loop runs. The three options are **next**, **last**, and **redo**. These are actually **goto** statements, but very constrained ones.

last causes the loop to be ended immediately. Execution continues with the next statement after the loop. Some languages use **break** to do the same thing.

next causes the remainder of the iteration to be skipped. Execution continues with the condition at the top of the loop. Some languages use **continue** to do the same thing.

redo restarts the loop immediately. The condition is not evaluated again. Instead, execution continues right after the condition at the top of the loop.

26.6 Infinite Loops

Usually an infinite loop is a bad thing. It is something that happens accidentally when your program runs wild and does not stop as it should. You might describe it as "my keyboard locked up" or "the computer stopped responding" when the truth may be that the computer is running just fine, but running a program that is stuck in a loop that will never end. Worst case you can always unplug the computer if you need to.

The **while** loop has a natural way to stop looping. The condition must be true for the loop to continue running.

Sometimes you want a loop to run forever, or to run until some complicated thing is recognized. We can write it like this.

```
while ( 1 ) { block }
```

How do you get out of an infinite loop? Generally, as we saw above, there is some sort of if statement in the loop. When it is satisfied, it will break out.

```
while ( 1 ) {
  ...
  if ( condition ) { last }
  ...
}
```

If the loop involves getting input from the keyboard, you can also rely on the user to simply kill your program when they are tired of it. They may hit control-C or press the red X in the corner of the window. If they have an easy way to kill the program, you don't have to build it in.

```
print "Let's see who can think of the biggest number.\n";
while ( 1 ) {
  print "What is your number: ";
  chomp ( $num = <STDIN> );
  $next = $num + 1; # cheat
  print "My number is $next.  I win!\n";
  print "Let's play again!\n" }
```

Trivia fact: The address for Apple Computer's corporate headquarters is "1 Infinite Loop, Cupertino, CA 95014". Check it out on Google Earth. It really is a loop. Do you think they are making a statement?

The address for Microsoft's corporate headquarters is "1 Microsoft Way, Redmond, WA 98052". Do you think they are making a statement?

26.7 if versus while

Notice that **if** and **while** look a lot alike. **while** and **if** are each control structures. They alter the normal flow of programs from one line to the next. They introduce new things to do.

```
if ( condition ) { body }
```

This is the simple **if** statement. If the condition is true the body is executed exactly once. If not, not.

```
while ( condition ) { body }
```

This is the **while** loop. If the condition is true the body is executed. And again. And again. Until it is no longer true.

else and **elsif** apply to **if** statements. They are not part of the **while** loop. They provide a way to consider several or many alternatives from which only one will be selected.

redo, **next**, and **last** apply to loops. (They apply to the innermost loop that they are inside.) They are not part of the **if** statement. They provide a way to end the iteration or loop early, either restarting with the same item, continuing directly with the next item, or quitting entirely.

26.8 Exercises

Answers to starred exercises can be found on page 289.

Exercise 83:* What is the meaning of **implicit**?

Exercise 84:* What is the meaning of **explicit**?

Exercise 85:* What is the meaning of **efficient**?

Exercise 86:* What is the meaning of **effective**?

Programming exercises can be found in chapter 30 (page 151).

Chapter 27

For Loops

Up until 1956 most programming was done in Assembly language. In 1956 **FORTRAN** became the first high level language to be introduced. (**LISP** was introduced soon after, in 1958.) With the advent of high level languages most engineering programming began to be done in FORTRAN. Wikipedia has an article here:

http://en.wikipedia.org/wiki/Fortran

Why do we care?

As an efficiency, FORTRAN introduced the **DO LOOP** as a way to create a loop by specifying an index variable, a starting value, and an ending value. We care because this model has survived in numerous languages as the **for** loop. Wikipedia has an article here:

http://en.wikipedia.org/wiki/For_loop

When you are first learning to program, **for** loops tend to be a bit confusing. They do not seem to be quite as logical as **while** loops. However, they are commonly used and it is important to be familiar with them and able to use them.

27.1 Comparison of "while" and "for"

Loops are commonly used to run through a list of numbers or examine a set of alternatives. For example, one might wish to print a table of trigonometry values or step through the values stored in an array (see chapter 33, page

163).

With a **while** loop, there is usually a variable called the index variable. It will take on the value of each number in the list, one by one. For example, to work through the list 1 to 100, the following **while** loop could be used.

```
$i = 1; while ( $i <= 100 ) { body; $i++ }
```

This formulation correctly reflects the order in which things happen but has the disadvantage that "the light switch is not by the door; it is on the other side of the room." Specifically, the `$i++` statement is used to move to the next value of `$i` but if the body is large, it can be quite remote from the rest of the control structure (the initializing and testing `$i`).

In the interest of collecting all the control elements into one place, the **for** loop rearranges things as follows.

```
init; while ( condition ) { body; step }
for ( init;   condition; step ) { body }
```

With the **while** loop, the initializations all happen before the loop begins. They are represented here by the word `init`.

Next the loop begins. It checks to see whether there is any work to be done. This test is called the `condition`. If the test is True, the `body` of the loop is executed. Before restarting the loop, the control variable is updated to its new value in the `step` phase.

The **for** loop moves the `init` and `step` pieces up next to the `condition` so they are all visually together. Here is a more specific example.

```
$i = 1; while ( $i <= 100 ) { body; $i++ }
for ( $i = 1; $i <= 100; $i++ ) { body }
```

The setup is done before the loop begins. The condition is handled as before. The step is done at the end (or bottom) of each loop. Typically the step is used to modify the control variable being tested in the condition.

The **for** loop has a major advantage that the setup, condition, and step, are all collected into one place. This makes it easier to see what the program is doing and harder to write buggy programs.

next: The **for** loop has another advantage. In any loop, the **next** command causes the current iteration to end and a new iteration to begin. This is a problem if the increment has not happened yet, but happens at the bottom of the loop. If the increment does not happen, we may well find ourselves in an infinite loop.

With the **for** loop, a **next** command causes the iteration to end. However, the increment will still happen because it is not hidden at the end of the loop. It is part of the loop's control structure.

27.2 Infinite Loops

You can leave out parts of the **for** loop. If there is no initialization, you could write this.

```
for ( ; condition; step ) { body }
```

If you want to create an **infinite loop** with no initialization, no condition, and no step, you can write it like this. In this case the condition is taken to be always True.

```
for (;;) { body }
```

27.3 Summary: if vs while vs for

for is the most complex of the control structures we have studied. The advantage of **for** is that, like the dashboard of an automobile, all the controlling elements are right at the top, easy to find and clearly stated.

The disadvantage is the confusing complexity. Specifically, the word `for` which declares the loop is written before the `init` but the `init` is executed before the loop begins. And the `step` is written before the `body` but the `body` is executed before the `step`. It is the price of having a dashboard.

redo, **next**, and **last** can be used inside **for** loops just as they can in **while** loops.

Chapter 28

Self-Modification

This chapter coveres a number of additional operators commonly used in calculation.

28.1 Plus Plus, Plus Equals

Binary means that two numbers are involved. Most of the operators we use are binary infix operators, meaning that two numbers are involved and the operator goes between them, as in `5+3`.

Unary means that only one number is involved in the expression. We are all familiar with the fact that putting a dash in front of a number converts it to being negative. The dash, also called a minus sign, is a unary operator.

There are two other commonly used unary operators: `++` and `--`. These are used to increment or decrement a variable, that is, increase or decrease its value by exactly one.

`$i++` means the same thing as `$i = $i + 1`.

`++$i` means the same thing as `$i = $i + 1`.

`$i--` means the same thing as `$i = $i - 1`.

`--$i` means the same thing as `$i = $i - 1`.

Warning:

`$i = $i++` does not mean the same thing as `$i = $i + 1`.

We will address this in section 23.2 (page 119), but for now we will say that it (a) finds the value of `$i`, (b) increments `$i`, and (c) copies the original value of `$i` back into `$i`. Thus, the net effect is that it ends up doing nothing. At the same time it looks like it will increment `$i`.

Trivia fact: The language C++ gets its name from its intention to be the successor to C. The language C# gets its name from its intention to be the successor to C++. Notice that the # character looks like four + signs arranged in a grid.

There is a more general version of ++. It is actually pretty commonly used, so you should be familiar with it and comfortable using it.

```
$x += 1; # means $x = $x + 1;
$x -= 1; # means $x = $x - 1;
```

Using this syntax, we can separate out the amount of increment. If we want to count by twos, we could do it like this.

```
$x += 2; # means $x = $x + 2;
```

We can do `op=` for many operators.

```
$x *= 2; # means $x = $x * 2;
$x /= 2; # means $x = $x / 2;
$x %= 2; # means $x = $x % 2;
```

We cannot do `op=` for all operators.

```
$x <= 2; # does NOT mean $x = $x < 2;
$x >= 2; # does NOT mean $x = $x > 2;
```

The reason these operators exist is because their functions can actually be done directly in machine language (also called Assembler or Assembly Language). Some high-level languages (like C and Perl) have made the same capabilities available.

28.2 Dot (Concatenation)

Strings use the dot character to indicate **concatenation**. That is, `"x"."y"` is equal to `"xy"`. It adds two strings by combining them end to end.

```
$x .= "x"; # means $x = $x . "x";
```

`.=` is kind of the opposite of **chop**.

28.3 Precedence of Self-Modification

It would be really confusing, so don't do it, but these operators can be combined with each other. The precedence is right to left. And if you try to get too tricky, it will break. Avoid combining them in the same expression.

```
$a = 5; $b = 4;
print "$a $b\n"; # 5 4
$a += $b += 3;
# add three to b
# then add b to a
print "$a $b\n"; # 12 7
```

Chapter 29

Games and Projects

In this chapter we look at some simple games and projects that can be programmed using nothing more advanced than random numbers, if/else, and loops.

Random numbers are introduced in section 11.6 (page 75) and covered in more detail in chapter E (page 319).

29.1 Hi Lo Game

In this game one person picks a number, maybe between 1 and 100, and the other person has to guess the number in as few guesses as possible. After each wrong guess, the person is told whether the truth is higher or lower.

We will write a program to pick the number and let us guess.

```
$num = int(rand(100)) + 1; # pick a number
print "I am thinking of a number between 1 and 100.\n";
while ( 1 ) { # infinite loop
  print "What is your guess: ";
  chomp ( $guess = <STDIN> );
  if ( $num == $guess ) { print "YOU WIN!!!\n"; last }
  if ( $num < $guess ) { print "$guess is too high.\n" }
  if ( $num > $guess ) { print "$guess is too low.\n" }
}
```

Write it. Run it. Play it.

We can add even more jazz to the program. Let's count the number of guesses the player makes and congratulate her if she gets the answer sooner than expected. Expected would be about 7 guesses. Here are some lines to add to the program. See if you can tell where they should go.

```
$guessCt = 0; # guess count
$guessCt++; # one more guess has happened
if ( $guessCt <= 7 ) { # compare to the expected performance
  print "Wow, only $guessCt guesses.  ";
  print "You're really good at this!\n" }
else {
  print "You found my number in only $guessCt guesses.\n" }
```

How to win: Theoretically you can rule out half the possibilities each time you guess. If you guess 50 and your guess is too high, then you know the number is between 1 and 49. Each time you can guess the middle of the range of numbers that are left. After 0 guesses you have 100 possibilities. After 1 guess you have 49 or 50 possibilities left. After 2 guesses you have 25 left. After 3 guesses, 12 left. After 4 guesses, 6 left. After 5 guesses, 3 left. After 6 guesses, 1 left. By your seventh guess you should be able to always win.

29.2 99 Bottles of Carbonated Non-alcoholic Beverage

There is a well-known US/Canada song that goes like this.

```
99 bottles of (root) beer on the wall,
99 bottles of (root) beer.
Take one down and pass it around,
98 bottles of (root) beer on the wall.
```

Often it is sung on long road trips to keep the kids occupied so they don't keep asking, "Are we there yet?" It then repeats its way down to zero or until people get tired of singing it. Write a program to print out the versus of this song. Prompt for and read in the starting number of bottles.

See also:

http://en.wikipedia.org/wiki/99_Bottles_of_Beer_on_the_Wall

29.3 2 Nim

In games of **Nim** there is a pile of stones. Players take turns removing some number of stones. Typically there are two players. The person who takes the last stone wins.

(If you want to say the person who takes the last stone loses, that is the same as ignoring the last stone and playing for who wins the normal way. Then the other player will be forced to take the last stone.)

In **2 Nim** the rule is that each player takes one or two stones on each turn.

```
print "Let's play Nim!\n";
$stones = int(rand(20)) + 10 ;# 10 to 39 stones
while ( $stones > 0 ) {
  # human takes a turn
  print "There are $stones stones.\n";
  print "How many stones would you like (1 or 2): ";
  chomp ( $takes = <STDIN> );
  $stones -= $takes;
  # computer takes a turn
  print "There are $stones stones.\n";
  $takes = int(rand(2)) + 1;
  print "Computer takes $takes.\n";
  $stones -= $takes;
}
print "Let's play again sometime!\n";
```

As written, this game needs a bit of help. If you key it in and play it you will find that (a) the human could easily cheat, and (b) the computer does not notice when the stones run out. In fact, the computer does not even figure out who won.

Your task is to improve the program, adding such personal touches as you may find desirable.

How to Win: Try to leave a multiple of three stones. Once you do that, you can always force a win.

29.4 3 Nim

This version is like 2 Nim except you can take up to three stones.

How to Win: Try to leave a multiple of four stones. Once you do that, you can always force a win.

29.5 Variations

With any of these games, you can make some typical variations.

(a) Put the game in a loop so it automatically starts over when it normally would have ended.

```
while ( 1 ) {
  ... insert the whole program here ...
}
```

This is called a **nested loop** because the original loop has been placed inside another loop. The original loop is called the **inner loop** and the new loop is called the **outer loop**.

(b) Count the number of times the computer wins and the number of times the human wins. Report the totals after every game.

(c) Add random comments such as "Wow, great move." and "I did not see that coming."

(d) Add simple graphics. The number of stones in Nim could be shown as a line of "O"s in addition to (or instead of) stating a number.

Chapter 30

Unit Test: Repeated Actions

This chapter is a collection of programming questions that are typical of those you should be able to do by this point in the course. You should be able to do while loops (chapter 26, page 134) and for loops (chapter 27, page 141).

30.1 Vocabulary

The following words are worth memorizing. They are introduced in earlier chapters of this unit. It is assumed that you know them. They may appear in test questions without further explanation.

last causes the remainder of the loop to be skipped. The condition is not checked again. Instead, the loop is ended and execution continues with the next statement after the loop. Some languages use **break** to do the same thing.

next causes the remainder of the loop to be skipped. Execution continues with the condition at the top of the loop. Some languages use **continue** to do the same thing.

redo restarts the loop immediately. Execution continues right after the condition at the top of the loop.

printf: a command to format numbers, usually to a constant width.

nested loop: a set of loops with one inside the other, much like carved Russian Dolls. Time keeping uses nested loops. Seconds are nested within

minutes. Minutes are nested within hours.

inner loop: a loop that is inside another loop.

outer loop: a loop that has another loop inside itself.

30.2 Exercises

Things to watch out for: (a) `$x++` and `$x+1` are not the same. `$x=$x++` does not do what you might think. (b) `while(;)` is wrong. (c) `for(;)` is wrong. (d) `if` and `while` and `for` are not the same. (e) `while();` is usually wrong.

Answers to starred exercises can be found on page 289.

Easy: Simple, Predictable Loops

Exercise 87:* Print the numbers from 1 to 500. Leave space between numbers.

Exercise 88:* Print the numbers from 2 to 100 counting by 2s. 2, 4, 6, 8, 10, 12, ..., 98, 100. Don't worry about spacing or punctuation.

Exercise 89: Using approved style, write a program to do the following. Print the numbers from 100 down to 1. Put each number on a line by itself.

Exercise 90:* Use a while loop to print the numbers from n down to zero. Then print "blast off!" Prompt for and read in the starting number.

Exercise 91: Using approved style, write a program to do the following. Counting by fives (5, 10, 15, ...), print the numbers from 5 to 1000. Put each number on a line by itself.

Exercise 92: Using approved style, write a program to do the following. Prompt for and read in a number. Assume it is more than 100. Print all the numbers from 1 up to and including the number that was read in.

Exercise 93: Using approved style, write a program to do the following. Prompt for and read in a number. Starting at that number, print the next 700 numbers. Then stop. Do not worry about punctuation or spacing in the output, except the numbers should not bump into each other. Example: if the number is 50, you should print from 50 through 749.

Exercise 94: Using approved style, write a program to do the following.

Prompt for and read in a number. Counting by 2s, print the next 100 numbers. (If the number was 77, you should print 79, 81, 83, ... 277.)

Medium: Predictable Loops with Complications

Exercise 95:* Print the numbers from 1 to 500, skipping the multiples of 3. Leave space between numbers.

Exercise 96:* Coin Toss: For each coin toss, randomly select "H" or "T" with equal likelihood. Do 100 coin tosses, showing the results of each.

Exercise 97: Using approved style, write a program to do the following. Print 100 random numbers. Each number should be a digit between 1 and 6 (inclusive) as though you were rolling dice. Example: 2 4 3 4 1 5 3 3 4 2 5 6 4 3 ...

Exercise 98: Using approved style, write a program to do the following. Prompt for and read in a number. Print the numbers from one up the the number entered, except when a number is a multiple of seven, print the word "skip" instead. Leave space between each thing you print.

Exercise 99: Using approved style, write a program to do the following. Prompt for and read in a starting number. Prompt for and read in an ending number. Print all the numbers from start to end, in that order. Be alert to the fact that you could be counting up or you could be counting down.

Exercise 100:* Print the triangular numbers up to 1000. The first few triangular numbers are 1, 3, 6, 10, ... (1, 1+2, 1+2+3, 1+2+3+4, ...).

Exercise 101: Using approved style, write a program to do the following. Prompt for and read in a word. Use a loop and use chop to print all the prefixes of the word, from the full word down to the first letter, one per line. For example, if the word is Washington, print the following:

```
Washington
Washingto
Washingt
Washing
Washin
Washi
Wash
Was
```

```
Wa
W
```

Medium: Unpredictable Loops

Exercise 102: Using approved style, write a program to do the following. Prompt for and read in a name. See whether it is "Joe". If so, print "At last I have found you." If not, try again. Keep trying until you find Joe.

Exercise 103: Using approved style, write a program to do the following. Factors: Prompt for and read in a target number. Use a loop to look at all the numbers from 2 up to but not including the target, in order. For each number that divides exactly into the target, print this statement:

"The number (factor) is a proper factor of (target).\n"

Replace (factor) with the number that divides exactly. Replace (target) with the target number. It should all print on one line.

If you know how to skip numbers that will obviously not work, you are free to do that without actually trying them. If there are no factors, there is nothing to print.

Exercise 104: Using approved style, write a program to do the following. Determine whether a number is prime. Ask for and read in a number. Check all possible factors. Print out each factor greater than 1 and less than the number. If there are no such factors, report that the number is prime. (A factor is something that divides exactly into the number.)

Medium: Counting Songs

Exercise 105: There is a children's play song that goes like this: "5 little monkeys jumping on the bed, one fell off and hurt his head, Momma called the doctor and the doctor said, No More Monkeys jumping on the bed." It repeats with 4 monkeys, then 3, then 2, and finally 1. Using approved style, write a program to print out the versus of this song. Prompt for and read in the starting number of monkeys. Invent a clever ending.

Exercise 106: There is a children's song that goes like this: "5 kids in a bed and the little one said roll over, roll over. So they all rolled over and one fell out, 4 kids in a bed and the little one said ..." It continues until

the little one is all alone: "1 kid in the bed and he-e said, Ahhhh." Using approved style, write a program to print out the versus of this song. Prompt for and read in the starting number of kids in the bed.

Exercise 107:* Using approved style, write a program to do the following. Here is one verse of a hiking song.

```
15 miles we yet must walk.
But all along the way we talk.
Talking brings such happy smiles.
Suddenly there's 14 miles.
```

On the last verse end with:

```
Suddenly there's no more miles.
```

Write a program that, starting at 100, prints all the verses of this song. Leave a blank line between each verse.

Unit IV

Repeated Data

Chapter 31

Lists

Earlier we talked about simple variables. Simple variables hold one thing at a time. Simple variables each have a name.

Sometimes we want a variable that will hold more than one thing. Maybe we want the names of the students in this class. That would be a **list**. Or maybe we want the food items we would need to buy to make a special dinner for someone we love. That would be a list.

Our second big category of variables is the **list**, also known as the **array**. We will use **list** and **array** to mean the same thing although they have a subtle difference in emphasis.

Lists act a lot like simple variables, but they have a sequence of slots (or elements), sort of like cars on a train. Each slot is a simple variable, except it does not have its own name. We will use **slot** and **element** to mean essentially the same thing although they have a subtle difference in emphasis.

```
$x = 5; # simple
@x = ( 100, 90, 92, 86 ); # array
```

In the first example, a simple variable named `x` receives the value `5`. In the second example, an array variable named `x` receives the list of values `100`, `90`, `92`, and `86`. Simple variables are prefixed with a dollar sign: `$`. Array variables are prefixed with an at sign: `@`. Array elements are listed between parentheses and separated by commas.

31.1 Adding Items (push)

We can add items to the list. There are several ways to do it, but one popular way is "**push**."

`push` adds a new last item to the list, making the list one longer.

```
push @list, $item;
```

31.2 Deleting Items (pop, shift)

We can retrieve items from the list, deleting them as we go. There are more ways to do it, but here we will show two: `pop` and `shift`.

pop removes the last item from the list, leaving the list one shorter. Sometimes this is called **LIFO**, meaning "last in, first out."

shift removes the first item from the list, leaving the list one shorter. Sometimes this is called **FIFO**, meaning "first in, first out."

```
$item = pop @list;
$item = shift @list;
```

31.3 Adding Items (unshift)

Push, pop, and shift are the most popular ways of adding to a list and removing from a list.

What's missing? It might be nice occasionally to be able to put something new on the front of the list.

unshift is the opposite of **shift**. It adds a new first item to the list, making the list longer by one.

```
unshift @list, $item;
```

Important Vocabulary: Words worth memorizing.

push: a command to add a new item to the end of an array.

pop: a command to remove the last item from the end of an array.

shift: a command to remove the first item from the front of an array.

unshift: a command to insert a new item at the front of an array.

Chapter 32

Arrays (Indexed)

Earlier we said that each slot is a simple variable, except it does not have its own name. However, the slots or elements of the **array** can be accessed by number. The slot number is called an **index**.

(In the United States, the presidential residence is called the "White House." It is also called 1600 Pennsylvania Avenue. "White House" is a name. 1600 is a number, an index referring to its location as part of a list of properties that are sited along Pennsylvania Avenue. Special things get names. Less special things get numbers, indicating that they are just one among many. A few numbers are special, like zero and one.)

For the array `@x` shown here, the index values are as follows:

```
@x = ( 100, 90, 92, 86 ); # as a list
$x[0] = 100; # as a numbered element
$x[1] = 90;
$x[2] = 92;
$x[3] = 86;
```

It gets confusing here for two reasons. First, it is clear that the slots are numbered in order. But the numbers start at zero instead of one. (A few computer languages allow the index numbers to start at one, but zero is more normal.) Sorry about that zero thing, but it is common so get used to it.

Second, shouldn't it be "`@x[0] = 100;`"? Well, yes, that would make a lot of sense. But slot zero is a simple variable. Slot zero is not an array. When

we say `$something[1]`, we are saying the simple item in slot number one of the array `@something`.

Negative Slots

Perl allows us to count the slots from the front or the back. From the front they are counted starting at zero. From the back we use negative numbers. Negative zero is the same as zero, making it potentially ambiguous. Instead we start counting at the end with negative one.

```
@x = ( 100, 90, 92, 86 ); # as a list
$x[-1] = 86; # as a numbered element
$x[-2] = 92;
$x[-3] = 90;
$x[-4] = 100;
```

This can be handy. Sometimes you want to know what the first item on the list is. That would be `$list[0]`. Sometimes you want to know what the last item on the list is. That would be `$list[-1]`.

Adding by Slot Number

We can also add by putting something directly into a slot by number.

```
$list[10] = $item;
```

This sets or replaces item 10 in the list. If the list had more than ten items, nothing else changes. If the list was shorter, it grows to reach the new last element.

Check out the `splice` function (Google it) for more complex needs.

Effects of Push and Pop

Important Warning: Slot numbers are affected by changes to the size of the list. In particular, if you add to the list using **push** or **unshift**, or if you

delete from the list using **pop** or **shift**, all the slot numbers get updated. Putting something into slot `[4]` and assuming it will stay there could be unreliable. In chapter 45 (page 207) we will look at another alternative called **hash** tables.

Important Vocabulary: Words worth memorizing.

index: a number that tells which slot of the array is being considered. Numbering starts with zero at the front and counts up. Numbering from the back starts at -1 and counts down using negative numbers.

Chapter 33

Walk the List

Let's say we have a list `@x` and we want to walk down the list and print out everything on the list.

We can find out the size of the list by copying it into a simple variable. I am not sure why they decided to use this particular syntax, but it is handy.

```
$size = @list;
```

Armed with this knowledge, we could use a program like this:

```
$size = @x;
for ( $i = 0; $i < $size; $i++ ) {
  print "$x[$i]\n" }
```

We are using `$i` as an index variable. The first time through the loop we print `$x[0]`. The last time through the loop we print `$x[$size-1]`.

We don't actually need the variable `$size`. This works too.

```
for ( $i = 0; $i < @x; $i++ ) {
  print "$x[$i]\n" }
```

33.1 foreach

Déjà vu: Let's say we have a list `@x` and we want to walk down the list and print out everything on the list. We could use a program like this:

```
foreach $item ( @x ) {
  print "$item\n" }
```

This program is quite a bit shorter than the one above. It also runs faster, but not enough faster to worry about. The main advantage is that it is easier to write and easier to understand.

We are using `$item` as an alias variable in a `foreach` loop. The first time through the loop it equals `$x[0]`. The last time through the loop it equals `$x[$size-1]`.

In most cases where I want to walk down a list, I find that `foreach` is by far the most convenient way to get the job done. Your mileage may vary.

33.2 $_ (dollar underscore)

The **foreach** command permits the `$item` part to be implicit. In that case, the variable used is `$_`. We can rewrite our program from above as follows:

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

Important Vocabulary: Words worth memorizing.

foreach: a type of loop that avoids index numbers while still giving access to each array element in turn.

Chapter 34

Split and Join

We can convert the contents of an array into a string and store it in a simple variable. And we can go the other way.

The easiest way is just to interpolate the array into a string. Items will normally appear with spaces between them.

```
$x = "@y";
```

Interpolation can be a handy way to compare two arrays. Compare these alternatives, with and without quotes.

```
if ( @x == @y ) { ... } ;# compares lengths numerically
if ( @x eq @y ) { ... } ;# compares lengths
if ( "@x" eq "@y" ) { ... } ;# compares contents
```

With quotes the arrays are converted into strings. The strings are then compared in the usual way. Without quotes the arrays are converted into numbers that represent their length.

```
@x = ( 1, 2, 3, 4 );
@y = ( 4, 5, 6, 7 );
if ( @x eq @y ) { this will happen }
if ( "@x" eq "@y" ) { this will not }
```

We can also use `join` to combine things. The first thing after join is the string to insert between list items. The remaining things make up the list to be combined.

```
@y = ( 1, 2, 4, 8 );
$x = join " and ", @y;
print $x; # "1 and 2 and 4 and 8"
```

We might want to pick the join string carefully so we can reverse the effect later. We can use `split` to separate a string into items of an array. In this next example we are splitting on space, which means that any time we run into a space, we start a new list item.

```
@y = split " ", "this is a test";
print $y[1]; # will print "is"
```

As a special case, if we split on nothing, the string gets separated into individual letters.

```
@y = split "", "this is a test";
print $y[1]; # will print "h"
```

The split string (the first item after the word `split`) is actually a regular expression. We talk about those in chapter 52 (page 237).

Important Vocabulary: Words worth memorizing.

split: a command to separate a string on certain boundaries thus creating an array.

join: a command to combine the elements of an array thus creating a string.

Chapter 35

Unit Test: Repeated Data

This chapter is a collection of programming questions that are typical of those you should be able to do by this point in the course. You should be able to do arrays and foreach loops (chapter 32, page 160).

35.1 Vocabulary

The following words are worth memorizing. They are introduced in earlier chapters of this unit. It is assumed that you know them. They may appear in test questions without further explanation.

push: a command to add a new item to the end of an array.

pop: a command to remove the last item from the end of an array.

shift: a command to remove the first item from the front of an array.

unshift: a command to insert a new item at the front of an array.

index: a number that tells which slot of the array is being considered. Numbering starts with zero at the front and counts up. Numbering from the back starts at -1 and counts down using negative numbers.

foreach: a type of loop that avoids index numbers while still giving access to each array element in turn.

split: a command to separate a string on certain boundaries thus creating an array.

join: a command to combine the elements of an array thus creating a string.

35.2 Exercises

Things to watch out for: (a) `@x[1]` and `$x[1]` are not the same.

Answers to starred exercises can be found on page 291.

Easy: One Array One Pass

Exercise 108:* Using approved style, write a program to do the following. Use a loop. Prompt for and read in a name. Add the name to the end of an array. Repeat until you get a blank name.

Exercise 109:* Create a loop. On each pass through the loop, prompt for and read in a name. If the name is "William" break out of the loop. Otherwise, add the name to an array of names and continue with the loop.

Exercise 110:* Using approved style, write a program to do the following. Given a string in `$a`, use a while loop and the chop function to convert it into an array named `@a`, one letter per slot. (Chop returns and removes the last letter of a string.)

Exercise 111:* Using approved style, write a program to do the following. Assume that you are given an array @a that has things in it. Go through the array and print out each thing, one per line. Do not print anything else.

Exercise 112:* Using approved style, write a program to do the following. Assume that you are given an array @a that has numbers in it. Go through the array and print out any numbers that are an exact multiple of 5, one per line. Do not print anything else.

Exercise 113:* Using approved style, write a program to do the following. Assume that you are given an array @x. Go through the array and print out every fifth item, one per line. Do not print anything else.

Exercise 114:* Using approved style, write a program to do the following. Assume you have an array named @x. Look through the array to see if one of the entries is the number 5. Report "5 is found" or "5 is not found" depending on which thing is true.

Exercise 115:* Using approved style, write a program to do the following. Assume you are given an array named @visitors. Look through the array and tell whether "Joe" is one of the visitors.

Exercise 116:* Using approved style, write a program to do the following. Assume you are given an array named @things. Write a program that prints out all the things one per line, counting from zero, with a typical line looking like this:

```
The number 5 thing is "hello".
```

Exercise 117:* Assume you already have an array named @ARGV. It is filled with numbers. Write a program that adds up the numbers and reports the total.

Exercise 118:* Fill an array with the numbers from 10 to 900 counting by 10s.

Exercise 119:* Assume that you are given an array @deck that has shuffled cards in it. Deal out five cards each to three players, @Adam, @Bob, and @Charlie, by removing cards from the front of the deck, giving the first card to Adam (by adding it to the @Adam array), the second to Bob, the third to Charlie, the fourth to Adam, and so on in turn until each has five cards.

Medium: Two Arrays or Passes

Exercise 120:* Read in 100 strings one by one and store them in an array. Then read in one more string and check the array to see if it is already present. If present, print "PRESENT". If not present, print "ABSENT". Do not prompt or print anything else.

Exercise 121:* Using approved style, write a program to do the following. Perfect Shuffle: Assume you have two arrays, @left and @right. Assume they have the names of playing cards, such as "2D" for the Two of Diamonds. Combine them into one new array called @cards that consists of the first card from @left followed by the first card from @right, followed by the next card from @left and the next from @right. You cannot assume how many cards there are in each pile, but you can assume they will both run out of cards at the same time.

Exercise 122:* Using approved style, write a program to do the following. Assume that you are given an array @a that has data in it. Construct an array @b with the same data but in reverse order. Do not use indexing (things like [2]). Use things like push, pop, shift, unshift, and foreach.

Exercise 123:* Using approved style, write a program to do the following.

Use a loop to read in names one by one. Examine each name. If it is not "Joe" keep reading in names. When you find "Joe" stop reading in names. Go back through the list and print out the names you have read. Do not use indexing (`[$i]`). Instead use things like push, pop, and foreach.

Hard: Nested Loops or Complications

Exercise 124:* Using approved style, write a program to do the following. Read in a list of numbers, one per line, until you get a blank line. Add up the numbers and print out the total. Also print out the average. Then tell how many numbers are above average and how many are below.

Exercise 125:* Given two lists of words, @old and @new, compare them to find out whether the same word is present in both arrays. If such a word exists, find it and print it out. If no such word exists, print "no duplicates found". If more than one set of duplicates exists, report which ever one you want.

Exercise 126: Using approved style, write a program to do the following. Merge: Assume you have two arrays, @left and @right, each filled with sorted numbers, low to high. Combine (merge) them into one new array called @both that combines all the numbers still in sorted order. Start by taking the lower number from left and right. Repeat by taking the next lower number from left and right. Continue until both sides are empty. Assume that one side will run out before the other does. Assume that some of the numbers may be the same. You are free to destroy left and right in the process of creating both.

Unit V

Organizing with Subroutines

Chapter 36

Subroutines

We have been using built-in subroutines from the beginning of the course. The first one we used was `print`. The printing process is rather elaborate and involved. We don't have to understand it personally. We have a subroutine that we can call upon to do that work. Somebody else wrote it a long time ago. It is part of the standard library of subroutines.

Subroutines give us a way to manage and reduce complexity. This involves is a small loss in efficiency for the computer. More statements must be carried out to achieve the same goal. But this can be a really big win for the programmers. Simple statements mean less bugs. Less bugs is a worthy trade-off for the small loss of efficiency.

Would you rather have a program that runs fast but is wrong, or a program that runs slow and is correct?

The actual trade-off is for a program that runs just slightly slower and has less bugs.

Subroutines are a good deal.

In this unit we learn how to write our own subroutines.

36.1 Clean Your Room

When my oldest child was about eight years old, we told her to clean her room. For some reason she went inside her room, sat on the floor, and cried. (At least that's how I remember it.) So I sat down next to her. "What's

wrong?" "I don't know how to clean my room." "Okay, I'll help you." At this point she brightened up, possibly expecting that I would do the work for her. But instead I said, "Close your eyes." "Okay." "Now stick out your hand and put it on the floor." "Okay." "Open your eyes and tell me what you are touching." "It's a story book." "Great. Where do story books go?" "On the book shelf." "Great. Go put it there and come back." "Okay."

At this point the process repeated. "Close your eyes." "Sigh." "I see you have the idea. When we say 'Clean your room' we mean that you should pick up things, one by one, and put them where they go. Dirty clothes go in the hamper. Clean clothes go in your dresser or closet. Books go in the book case. Everything has a place. Putting things away is cleaning your room."

(On the other hand, Mark Twain said "Have a place for everything and keep the thing somewhere else. This is not advice, it is merely custom.")

In computer terms I was defining a subroutine. I wanted to call that subroutine by saying "Clean your room." I wanted it to be easy to call. I did not want to explain how to clean each time the task needed to be performed.

36.2 The Subroutine Call

A **subroutine** is a set of instructions that has been given a special name. It is also known as a **function** or a **method** or a **procedure**. It is defined in one place and **called** from one or more other places.

The most important part of a subroutine is **how it is called.** The **call** should be designed to look meaningful and logical.

When you call it you are allowed to tell it things. Those things are called **parameters** or **arguments**. When it finishes its work it can report to you the results of its work.

In the room cleaning example, the subroutine name is "clean" and the parameter is "your room." The same cleaning procedure could be applied to the living room or play room or garage or back yard. A somewhat different procedure might apply to the kitchen or the bathroom where simply putting things away is not enough.

In computer terms, we would say:

```
$status = clean ( "your room" );
```

or maybe:

```
$status = clean ( "your room", "now" );
```

We would be looking for a status of "all finished." In my story the first time I called the subroutine, the status came back "I don't know how."

36.3 Syntax of the Call

When a subroutine is called, a list of zero or more **parameters**, also known as **arguments**, is **passed** (provided). Upon its return, an answer may (or may not) be sent back, either nothing or a **scalar** or an **array**. Here are some examples of calls.

```
name ( arg, arg, arg, ... );
$result = name ( arg, arg, arg, ... );
@result = name ( arg, arg, arg, ... );
```

Zero or more parameters includes zero as a possibility. **rand()** is an example of a subroutine that will run without any parameters being specified.

36.4 Syntax of the Definition

The **subroutine** is defined with the introductory word **sub** followed by the name of the subroutine. The name follows the same rules as variables: start with a letter, continue with letters and/or digits, underscore counts as a letter. Next comes the body of the subroutine which is surrounded by curly braces. Here is an example of a subroutine definition.

```
sub abc123 { body }
```

The body is given as a set of statements to be executed whenever the subroutine is called.

The input arguments can be examined by the subroutine. They arrive in an array named `@_` ("at underscore"). We will say more about that in chapter 37 (page 178).

Other information can be retrieved or written using **global** variables. We will say more about that in chapter 38 (page 182).

36.5 Return

The subroutine can return a result, either a **scalar** (simple variable) or an **array** (list), using the **return** command.

Here is a brief example that adds two numbers and returns the result.

```
sub add2 {
  my $res ;# local variable to hold result
  my ( $n1, $n2 ) = @_;
  $res = $n1 + $n2;
  return $res }
```

Here is how we might call it.

```
$sum = add2 ( 5, 8 );
print "$sum\n"; # will print 13
$sum = add2 ( 21, 34 );
print "$sum\n"; # will print 55
```

Here is a brief (not terribly useful) example that returns an array filled with numbers from 1 to some limit. Mainly it shows how to return an array.

```
sub one2n {
  my ( $i, @result );
  my ( $limit ) = @_;
  for ( $i = 1; $i <= $limit; $i++ ) {
    push @result, $i }
  return @result }
```

As usual, there is far, far more that we are not saying. But this should get you started. Beyond that, remember that Internet search engines are your friend.

36.6 Code Factoring

By **code factoring** we mean the dividing out of common elements. We are simplifying our program by breaking it into meaningful pieces. Although it

is theoretically possible to divide up code almost anywhere, there are natural boundaries that should be found and used.

In mathematics we might see an expression like this:

$5x + 10$

Another way to write it is to recognize that there is a factor of `5` present. We can rewrite the expression like this:

$5(x + 2)$

The new expression has two factors, 5 and $x + 2$.

Similarly programs can be divided into independent pieces that interact with each other.

Here is an example with code.

```
print "Please enter your name: "; chomp ( $name = <STDIN> );
print "Please enter your age: "; chomp ( $age = <STDIN> );
```

Notice the repetition. We can factor that out and create two calls. (You should **always** design the subroutine call first. This is called the **interface**. Then design the subroutine to support it. Don't let the tail wag the dog.)

```
$name = ask ( "your name" );
$age = ask ( "your age" );
```

We support these with a subroutine definition.

```
sub ask {
  my ( $q, $ans );
  ( $q ) = @_;
  print "Please enter $q: ";
  chomp ( $ans = <STDIN> );
  return $ans }
```

The beauty of it is that the subroutine can be carefully constructed and tested one time. Then it can be used many times. Once it is built, it becomes a black box that you do not ever have to look at again. (We explain the phrase "black box" a bit later.)

With the subroutine in hand, the main program becomes more simple. Simplicity is an important virtue for programming. Consider these two lines, side by side.

```
print "Please enter your name: "; chomp ( $name = <STDIN> );
$name = ask ( "your name" );
```

Both lines do the same thing. The second line is probably closer to the way a human would like to see it. The first is probably closer to the way the computer would like to see it. Who should win? That's just the tip of the iceberg. Subroutines make life much easier. They let us organize. This is very valuable.

As we write code, we should be asking ourselves whether the code we are writing is so specialized that it will probably not be useful anyplace else, or could it be generalized so our efforts can find a home in other programs.

Although we can talk about it at length, probably the best way to learn it is by discovery. As you write a few subroutines, your understanding will increase and you will make better choices.

Chapter 37

Arguments

The input arguments to a subroutine can be examined by the subroutine. They arrive in a predefined array named `@_` ("at underscore").

37.1 Direct Access

The input arguments can be accessed directly where they are, by using a `$_[$x]` type of notation that indexes directly into the arguments array. If the first three parameters are name, address, and city, you could copy them like this:

```
$name = $_[0];
$address = $_[1];
$city = $_[2];
```

There is a nice trick you can use to copy all the parameters into named variables at the same time. If the first three parameters are name, address, and city, you could copy them like this:

```
( $name, $address, $city ) = @_;
```

WARNING: Consider the following example.

```
$name = @_; # wrong
```

Rather than copying the first parameter into `$name` it actually copies the count of the number of parameters into `$name`. If we are only expecting one parameter, then `$name` will probably be `1` when the statement finishes.

We really need the parentheses so that the left side of the equals sign is an array, like this:

```
( $name ) = @_; # correct
```

Here is a bigger example.

Let's say that we prompt for input, read in a number, and verify that it is between 1 and 10 (inclusive). Or an age between 14 and 99. If it is, we go about our business. But if not, we print an error message and prompt for the number again.

We decide we want to call it like this:

```
$num = askAZ ( "a number", 1, 10 );
$age = askAZ ( "your age", 14, 99 );
```

We can modify our subroutine (or make a new one) that handles the additional functionality. We elect to create a new one named `askAZ`, which reminds us we are asking something and we are providing the limits (the A to Z) in which it must fall.

```
sub askAZ {
  my ( $q, $min, $max, $ans );
  ( $q, $min, $max ) = @_;
  while ( 1 ) {
    print "Please enter $q: ";
    chomp ( $ans = <STDIN> );
    if ( $min <= $ans && $ans <= $max ) { return $ans }
    print "Please enter a number between $min and $max.\n"
  } }
```

Notice we have four variables in our subroutine. Three are used to hold information given by the caller. One is used to hold the answer we will return to the caller.

Also we created an infinite loop. It will continue asking the question until the user provides an acceptable answer.

We are removing a complexity burden from the main program. We want the main program to appear and be simple as simple as possible. We want the subroutines to be generic so they can be used in other programs we might write later.

37.2 Unlimited Parameters

The commands **pop**, **shift**, and **foreach** can be used directly on the `@_` array.

Say we wanted to add up the numbers that were given to us. We want to be able to say

`$x = total ( 1` 2, 3, 4, 5);

and have `$x` end up with the number 15. Following is a subroutine that will do that job.

```
sub total {
  my ( $total, $num );
  $total = 0;
  foreach $num ( @_ ) { $total += $num }
  return $total }
```

Notice in this subroutine that we use the input array directly. We walk down the array and as we encounter each entry we add it to the total.

We could also compute the average. The subroutine is almost the same.

```
sub average {
  my ( $total, $num );
  $total = 0;
  foreach $num ( @_ ) { $total += $num }
  return $total / @_ }
```

All we have done here is rename the subroutine and make one additional calculation right at the end. We are dividing by `@_`, which is treated as the number of items in the parameter list. Of course, there is a risk of dividing by zero. We should take care of that by including a line like the following:

```
  if ( @_ == 0 ) { return 0 }
```

We could also combine the two subroutines, letting “average” call “total” as follows.

```
sub average {
  if ( @_ == 0 ) { return 0 }
  return total ( @_ ) / @_ }
```

Chapter 38

Global versus Local

In Perl all variables are **global** unless you do something to prevent it. This is actually a bad thing, but to change it now would break too many programs. It would turn lots of programs into stroke victims. So we muddle on.

The opposite of **global** is **local**. Newer programming language tend to avoid global variables just as modern languages tend to avoid the **goto** statement. Global variables make an end run around the formal interface and allow access to things that would normally be private.

When a variable is **global** each time we say its name we refer to the same storage bin.

```
$x = 5;
...
print $x;
...
sub abc {
  print $x;
  ... }
```

In this example, the variable `$x` is set equal to 5. Some time later in the program it is printed. Because it has the same name, it means the same storage location, which still has (presumably) the 5 that we put there earlier.

Later, inside the subroutine, when `$x` is mentioned, it still means the same storage location. It still prints a 5.

More recent languages tend to dislike this behavior because it leads to buggy programs.

Consider the family next door. Pretend they have a boy named "Joe." When you say "I am going over to Joe's house" everybody knows what you mean. Joe's name is global, at least around you.

Imagine that another family moves in on the other side. Pretend they also have a boy named "Joe." To solve the neighborhood problem maybe you would call them "big Joe" and "little Joe" or something. But inside their home, each one is still just "Joe" to his family.

The name "Joe" is defined locally in the context of the family where it is used.

We can force a local interpretation of the variable by explicitly declaring it by saying **my**.

```
sub abc { $x = 10; print "$x\n" }
$x = 5;
print "$x\n";
abc;
print "$x\n";
```

This will print "5, 10, 10" because when `abc` runs, it will change the global value of `$x`.

```
sub abc { my $x; $x = 10; print "$x\n" }
$x = 5;
print "$x\n";
abc;
print "$x\n";
```

This will print "5, 10, 5" because when `abc` runs, it will change the local value of `$x`. The difference is the `my $x;` in the second program.

The word **my** causes `$x` to belong to the subroutine instead of the whole program. This is called **declaring a variable**.

Why bother?

As programs get big, the parts become more and more isolated from each other, just as children grow up, move out, and establish their own lives.

Subroutines grow up and become moderately independent parts of the program.

We want it to be that way.

We want them to be independent because changes in one subroutine should not have surprising effects in other parts of the program. When you use global variables your program is fragile to the extent that variable names could accidentally be re-used.

Say one programmer writes subroutine xyz and uses variable abc. Another programmer writes subroutine pqr and also uses variable abc. They are dancing on each other's feet. The program will break and it will be hard to find out why it happened.

There are two typical solutions. One solution is to assign each programmer a range of variable names. Joe's variables could all start with `$j_` followed by the rest of the variable name, hence, `$j_abc`. Bob's variable names could all start with `$b_`, resulting in `$b_abc`. Problem solved.

Another solution is to locally declare each variable before or when it is first used in the subroutine, like `my $abc`. Then both programmers can use it and nobody needs to know or coordinate and assign a naming scheme.

When you use **my** your variables become **local**. Otherwise your variables will probably be **global**.

Chapter 39

Black Boxes

A subroutine should be a black box. A **black box** has an interface that is public and potentially a lot of hidden components that are supposed to be left alone by most of us.

Say I give you a magical device that will cook food. I call it a Microwave Oven. I show you how to work it. You use it happily for several years. Do you know how it works?

Well, a microwave has a user **interface**. Most people who say "sure, I know how a microwave works" really mean that they know how to press the buttons. They know the interface. Show them a new microwave and they can figure out how to work it.

But at a deeper level, most people have no idea how a microwave really works, or why it makes sparks when you put metal inside. No metal is allowed! Those are just the rules. If you were an electrical engineer, maybe you would understand why those are the rules, but for the rest of us, there are "no user serviceable parts inside." So don't open the box. You'll probably just break it.

An object (or situation) like this is called a **black box**.

A black box is any device that does a task for us but we don't really know how it works. We just know it works. Black refers to the fact that we cannot see inside it to tell how it works. In contrast, a **white box** or **clear box** is one that we *can* look inside.

Computers are a black box to most people. Microwaves. Clocks. Electricity. Rockets. Government. Car engines. Any number of things.

And yet we live our lives and we use these things. We rely on them. But we don't understand them.

It's probably better that way. Who has time to go to medical school before having a baby? And yet you would want your doctor to have been to medical school.

We love the fact that there are things we can rely on without knowing exactly how they work.

SOMEBODY has to know how those things work. And it is wonderful when we can be that person. A race car driver probably understands his car engine quite well. He's an expert. That knowledge might be useful once in a while. A computer expert may really know what is happening when he clicks the mouse on a web page.

SOMEBODY has to know, but it does not have to be me.

Because I don't have to know, it reduces the complexity in my life. I flip the switch. The light comes on. It works whether I understand or not. This is a good thing.

Computer programs can be the same way. Programs become complex in order to do the wonderful things we have come to expect. Complexity means they are hard to fix when they break. We would like to divide that complexity into black boxes. Experts can each understand their own black box. We can use those black boxes to assemble a solution.

`print` is a black box that we use all the time in programming. `<STDIN>` is another black box.

Making subroutines is the way we can create our own black boxes for use in our own programs. That way we don't have to reinvent the wheel each time we need a wheel. We can just cut and paste a subroutine into our new program and start using it right away.

39.1 Formal Interface

One beauty of the black box is that we can swap it out for a new and improved black box. We can buy a faster, better computer and still use it the way we did before. As long as the user interface is about the same, we will be okay.

What if we change the user interface? What if we upgrade our operating

system or move forward to the next version of Office software? Suddenly the changes in the interface cause serious frustration. It's like being a stroke victim. We have to relearn common tasks.

So it is very important to think about the formal user interface each subroutine will have. That's why I said earlier that you must always design the subroutine call first. Then design the subroutine to support it.

We want to conceal all the complexity of the subroutine. We do not want the complexity to be part of the formal interface. That way, any changes to the innards of the subroutine should not cause the rest of the program to break. We don't want our program to be a stroke victim.

The formal interface consists first of the subroutine name. You want something that is clear and concise. Not too long and verbose. Easy to understand when you see it.

Second, the interface involves the **parameters** (**arguments**) that will be used. The subroutine itself represents a set of actions. Those actions might be applied in different circumstances, or with different specific details. The easily changed parts should be represented by parameters.

Third, the interface involves the **return** value or values. When the subroutine finishes, what should it say? Maybe nothing. Maybe "I finished successfully" or "I had error 13" (whatever that is). Or maybe it should give us information we were seeking, like "The current time is 10:03 AM."

Fourth, the interface involves **side effects**, which are those things not already mentioned. Common side effects are use of `print` and/or `<STDIN>`. Other side effects might be the utilization or updating of **global variables**.

Chapter 40

Games and Projects

In this chapter we look at some simple games and projects that use our knowledge up through subroutines.

40.1 Games Menu

Make each game you have written into a subroutine. Then make a menu that offers a list of your games to the player and runs the game they select.

40.2 Input Edits

It is common that we want an input. The human can type whatever they want. Once the program receives it, we want the input to be converted into some standard form that is easy to use.

We may want to invent a subroutine that gathers answers to yes/no questions. Call the subroutine `yesno`. Give it one input, being the question that is asked. It returns one answer, being either "y" or "n" depending on what the user entered.

```
sub yesno {
  my ( $q ) = @_ ;# grab the question
  while ( 1 ) { # repeat until success
    print "$q: "; # ask the question
```

```
    chomp ( $Ans = <STDIN> ); # get the answer
    $ans = lc $Ans ;# convert to lower case
    if ( $ans eq "y" ) { return "y" }
    if ( $ans eq "yes" ) { return "y" }
    if ( $ans eq "n" ) { return "n" }
    if ( $ans eq "no" ) { return "n" }
    print "\nYou said \"$Ans\" but I don't understand.\n";
    print "I really only understand 'y' and 'n'.\n";
    print "Please try again.\n\n";
  }
}
```

Make this subroutine work. Test it.

40.3 Input Edits with Default

Modify the subroutine to take one more input. The second input will be the default answer. If the user just presses Enter and nothing else, the default value should be used.

The human should be notified what the default value will be. Normally this is done by putting the alternatives after the question, and making the default a capital letter.

Example: If the question is "Would you like to eat?" and the default is yes, the question might be presented to the use as follows.

```
print "Would you like to eat? [Yn] ";
print "Would you like to eat? (Y,n) ";
print "Would you like to eat? (Y/n) ";
```

Usually the first alternative, `[Yn]`, is used. See regular expressions in chapter 52 (page 237) for an explanation. But any method can be used that you believe will be clear to the human.

40.4 Input Edits with Alternatives

Modify the subroutine to take many more inputs. The first parameter is the question. The second parameter is be the default answer. Instead of

just “yes” and “no” as acceptable alternatives, use additional parameters to specify a list of other acceptable alternatives. Loop until one of the acceptable alternatives is entered.

Chapter 41

Unit Test: Organizing

This chapter is a collection of programming questions that are typical of those you should be able to do by this point in the course. You should be able to do subroutines (chapter 36, page 172).

41.1 Vocabulary

The following words are worth memorizing. They are introduced in earlier chapters of this unit. It is assumed that you know them. They may appear in test questions without further explanation.

argument: information explicitly passed to a subroutine as part of the subroutine call.

parameter: information explicitly passed to a subroutine as part of the subroutine call.

global variable: information implicitly available to the subroutine and throughout the program.

local variable: information available only within a specific subroutine.

return: the completion of a subroutine; also the information explicitly passed back from the subroutine upon completion.

41.2 Exercises

Things to watch out for: (a) make sure your variables are local unless otherwise requested by the problem. (b) When subroutines provide an answer, usually it is returned, not printed. `print` and `return` are not the same thing. (c) When subroutines get input, usually it is arguments or parameters, not STDIN. `<STDIN>` and input parameters are not the same thing. Parameters and arguments *are* the same thing. (d) `my $x = @_;` is not the same thing as `my ($x) = @_;`. Without parentheses you get the number of parameters. With the parentheses you get the first parameter.

Answers to starred exercises can be found on page 294.

Easy: Simple In, Process, Out

Exercise 127:* Write a subroutine named add3 that receives as input three parameters. It adds them together and returns the result.

Exercise 128:* Write a subroutine called stars that takes two inputs: the number of stars to write and the character to be the star. Return a string composed of those characters. Example: stars(10,"*") should return "**********".

Exercise 129:* Write a subroutine that adds up the numbers given to it and returns their total. If no inputs are provided, return zero.

Exercise 130:* Write a subroutine named "average" that accepts an argument array and calculates and returns the mathematical average (mean) of the array. If the array is empty, return zero.

Exercise 131:* Write a subroutine that finds the largest and smallest numbers among its parameters. It should return "smallest largest" where smallest and largest are replaced by the correct numbers. Assume at least one number is provided as an input.

Exercise 132: Using approved style, write a program to do the following. Write a subroutine named "max" that takes a list of numbers and returns the greatest (maximum) one. You can assume there is at least one entry in the argument list.

Exercise 133: Using approved style, write a program to do the following. Write a subroutine named "min" that takes a list of numbers and returns

the least (minimum)t one. You can assume there is at least one entry in the argument list.

Exercise 134: Using approved style, write a program to do the following. Write a subroutine named "pick" that takes a list of arguments and randomly picks one and returns it. Each argument should have the same chances of being picked.

Exercise 135:* Write a subroutine named dice that returns two random whole numbers between 1 and 6, as though a pair of fair dice had been rolled. Return them as `"$x $y"` where $x is one roll and $y is the other.

Exercise 136: Using approved style, write a program to do the following. Write a subroutine named "cointoss" that takes one parameter, the number of coins to toss and returns an array with that many random choices of "heads" and "tails" as though a fair coin had been tossed for each one.

Medium:

Exercise 137:* Write a subroutine name numType that accepts a number as its only parameter and returns "lucky", "unlucky", or "normal" as the result. A number is lucky if it divides by 7 but not by 13. A number is unlucky if it divides by 13 but not by 7. A number is normal if it divides by both or neither.

Exercise 138: Using approved style, write a program to do the following. Write a subroutine named dice that takes one parameter, the number of dice to roll, and returns an array with that many random integers from 1 to 6, as though a fair dice had been rolled for each one.

Exercise 139: Write a subroutine named "pick" that accepts an argument array and randomly returns one of the arguments. Each slot should be equally likely. Sample usage: `$coin=pick("heads","tails");`

Exercise 140: You are given the following broken subroutine. It is supposed to accept one input parameter, multiply it by itself, and return the answer. (That answer is called the square of the number.)

```
subroutine square { $x == @_; returns $x * $x }
```

Fix the subroutine. Then write a program that uses it to print the squares of numbers from 1 to 100.

Hard:

Exercise 141:* Write a subroutine named yesno. It accepts one argument: a question. It returns one result: the answer "yes" or "no". Inside the subroutine it asks the user the question. It listens for an answer. If the answer is yes or no it returns the answer. If not it complains and then repeats the process until it finally gets a good answer.

Exercise 142:* Write a subroutine that asks a user to pick either Rock, Paper, or Scissors. Keep asking until you get a valid answer. Return that answer.

Exercise 143:* Reorganize the following program by creating an appropriate subroutine and calling it. Give the complete final program with subroutine. Here is the original program:

```
print "Please enter the name of a boy: ";
chomp ( $person = <STDIN> );
print "Please enter a color: ";
chomp ( $color = <STDIN> );
print "Please enter a number: ";
chomp ( $number = <STDIN> );
print "Please enter a food: ";
chomp ( $food = <STDIN> );
print "Please enter a day of the week: ";
chomp ( $day = <STDIN> );
print "A story for you:

$person has a Surprising Day

On $day morning, $person woke up.
He was hungry. He decided to eat $food.
He was so hungry he probably ate $number of them.
Afterward he was feeling a little $color.
His mom gave him some medicine and he felt better.

";
```

Unit VI

Complex Programs

Chapter 42

Complex Programs

Up until this point we have tried to keep each unit as simple as possible. We introduced **if** statements but we did not do much with nested if statements. We introduced loops but did not do much with nested loops.

In this unit we explore a few of the things that can be done when we allow the programs to become a bit more complex. Hopefully you are ready for that now.

Chapter 43

Nested Loops

We can have loops within loops.

43.1 Counting the Hours

Let's look at the telling of time. We have hours, minutes and seconds. 60 minutes per hour. 60 seconds per minute. Here we print sixty seconds.

```
for ( $sec = 0; $sec < 60; $sec++ ) {
  print "$sec\n"; }
```

The first ten numbers do not look right because traditionally seconds are formatted as a two-digit number. We can solve that with **printf**, which we briefly explain here and at greater length in appendix B (page 298).

printf uses `%..d` to format numbers. The `02` you will see below indicates leading zeroes are requested and a minimum of two digits should be printed (more if necessary).

```
for ( $sec = 0; $sec < 60; $sec++ ) {
  printf "%02d\n", $sec; }
```

Now let's start building that up into a clock. This code will print the times from 1:00 up to 3:59.

```
for ( $sec = 0; $sec < 60; $sec++ ) {
  printf "1:%02d\n", $sec; }
for ( $sec = 0; $sec < 60; $sec++ ) {
  printf "2:%02d\n", $sec; }
for ( $sec = 0; $sec < 60; $sec++ ) {
  printf "3:%02d\n", $sec; }
```

Clearly there is a pattern here. We should exploit it. We should use another loop to provide the minutes.

```
for ( $min = 0; $min < 60; $min++ ) {
  for ( $sec = 0; $sec < 60; $sec++ ) {
    printf "%02d:%02d\n", $min, $sec; } }
```

And what is to stop us from going to hours as well?

```
for ( $hour = 1; $hour <= 12; $hour++ ) {
  for ( $min = 0; $min < 60; $min++ ) {
    for ( $sec = 0; $sec < 60; $sec++ ) {
      printf "%02d:%02d:%02d\n", $hour, $min, $sec; } }
```

Let's add AM and PM.

```
foreach $ap ( "AM", "PM" ) {
  for ( $hour = 1; $hour <= 12; $hour++ ) {
    for ( $min = 0; $min < 60; $min++ ) {
      for ( $sec = 0; $sec < 60; $sec++ ) {
        printf "%02d:%02d:%02d $ap\n", $hour, $min, $sec }}}}
```

We have AM and PM moving the most slowly. Within each AM or PM we have 12 hours. Within each hour we have 60 minutes. Within each minute we have 60 seconds.

Hours are nested within half days. Minutes are nested within hours. Seconds are nested within minutes.

43.2 Starbox

Let's print a geometric figure, a box of stars.

```
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

This box of stars is ten on a side. We can break it down into an outer loop that prints rows of stars making up the whole square, plus an inner loop that prints individual stars making up one row.

Let's start with the inner loop that prints one row.

```
$size = 10;
for ( $col = 0; $col < $size; $col++ ) { print "* " }
print "\n";
```

This prints `$size` stars, each followed by a space.

Now let's put an outer loop around it.

```
$size = 10;
for ( $row = 0; $row < $size; $row++ ) {
  for ( $col = 0; $col < $size; $col++ ) { print "* " }
  print "\n"; }
```

We can easily modify this program so it prints boxes of arbitrary size.

43.3 Right Triangle

Let's print a geometric figure, a right triangle of stars. This one is a bit more complicated. The rows are not of equal length. We will modify the inner loop so it stops at `$row` instead of `$size`. Row 1 will stop after 1 star. Row 5 will stop after 5 stars.

```
$size = 5;
for ( $row = 0; $row < $size; $row++ ) {
  for ( $col = 0; $col <= $row; $col++ ) { print ``* `` }
  print ``\n''; }
```

When the program is run, the result looks like this.

```
*
* *
* * *
* * * *
* * * * *
```

43.4 Centered Triangle

A centered triangle might look nicer.

```
    *
   * *
  * * *
 * * * *
* * * * *
```

What can we do to make that happen?

In this case we need to print some spaces before each row of stars. The bottom row has no spaces in front. One up has one space. Two up has two spaces. The top row has the most spaces.

We can invent a formula for the number of spaces. It turns out to be `$size-$row`. On a size 5 triangle, row 5 has zero spaces and row 1 has 4 spaces.

Let's add another loop inside the outer loop and right before the inner loop.

```
$size = 5;
for ( $row = 0; $row < $size; $row++ ) {
  for ( $col = 1; $col <= $size-$row; $col++ ) { print " " }
  for ( $col = 0; $col <= $row; $col++ ) { print "* " }
  print "\n"; }
```

Notice in this case we have one outer loop and two inner loops. We could express it like this:

```
( () () )
```

The inner loops are side by side. One happens first. The other happens second. They are both inside the outer loop. When the outer loop runs five times, each time both inner loops will run.

In terms of style, we use indenting to show that the second and third **for** loops are sub-parts of the outer loop.

43.5 Times Tables

When I was growing up, I had to memorize some multiplication facts. We called it the "times tables." They looked something like this.

```
 X |  1  2  3  4  5
---+---------------
 1 |  1  2  3  4  5
 2 |  2  4  6  8 10
 3 |  3  6  9 12 15
 4 |  4  8 12 16 20
 5 |  5 10 15 20 25
```

Let's write a program to print such a table. What are the components?

First, we have a top line that consists of numbers 1 through 5. It idenfies the columns. Plus there is a heading in front of the line.

Second, we have a line of dashes.

Third, we have the body of the table, consisting of five lines. Each line consists of five "products." Each line also has a heading identifying the row number.

Here is a program to print the table.

```
$size = 5; # how big to make the table

# first row
print " X |"; # print the upper left corner
```

```
# print the numbers across the top
for ( $col = 1; $col <= $size; $col++ ) {
  printf "%3d", $col }
print "\n";

# second row
print "---+"; # print the upper left corner
# print a line of dashes
for ( $col = 1; $col <= $size; $col++ ) { print "---" }
print "\n";

# rest of the rows
for ( $row = 1; $row <= $size; $row++ ) {
  printf "%2d |", $row;
  for ( $col = 1; $col <= $size; $col++ ) {
    printf "%3d", $row * $col }
  print "\n" }
```

Notice the style. We have included comments to tell us what is happening. We have included some blank lines to visually cluster the other lines according to which line is being printed. And for the rest of the rows we have nested loops identified by the indentation of the program.

Programming exercises can be found in chapter 30 (page 151).

Important Vocabulary: Words worth memorizing.

nested loop: a set of loops with one inside the other, much like carved Russian Dolls. Time keeping uses nested loops. Seconds are nested within minutes. Minutes are nested within hours.

inner loop: a loop that is inside another loop.

outer loop: a loop that has another loop inside itself.

Chapter 44

Games and Projects (Loops)

In this chapter we look at some simple games and projects that can be programmed using nothing more advanced than random numbers, if/else, and loops.

Random numbers are introduced in section 11.6 (page 75) and covered in more detail in chapter E (page 319).

44.1 Calendar

Not actually a game, this program is to print out a one-month calendar. For inputs you can ask the human what day the month starts and how many days the month has. Then print out a calendar that looks like this.

```
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

Variation: Instead of the small version of the calendar, print big squares like this:

```
APRIL 2009
```

```
  SUN     MON     TUE     WED     THU     FRI     SAT
+-------+-------+-------+-------+-------+-------+-------+
|       |       |       |     1 |     2 |     3 |     4 |
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
+-------+-------+-------+-------+-------+-------+-------+
|     5 |     6 |     7 |     8 |     9 |    10 |    11 |
(et cetera)
```

Advanced: Given the month and year, figure out what day the month starts on. You can probably find a formula on the web. Or you can puzzle it out for yourself. Here are the facts: There are 365.2425 days per calendar year. Leap years happen on multiples of 4, but not multiples of 100, but yes multiples of 400. In a leap year, February has 29 days. Otherwise February has 28 days.

44.2 Double Nim

In games of **Nim** there is a pile of stones. Players take turns removing some number of stones. Typically there are two players. The person who takes the last stone wins.

In **Double Nim** the rule is that each player takes up to twice as many stones as the previous player took. However, the first player is not allowed to take all the stones on the first grab.

Your task is to modify 1 2 Nim so it allows each player to take up to twice as many stones as the prior person took.

How to Win: This one is tricky. It calls for a **Fibonacci** decomposition of the number of stones. The Fibonacci numbers are 1, 1, 2, 3, 5, 8, etc, where each number is the sum of the prior two.

The Fibonacci decomposition is the sum of the largest possible Fibonacci numbers that add up to the existing pile of stone.

Example: If there are 30 stones, we need to know all Fibonacci numbers up to 30. They are 1, 1, 2, 3, 5, 8, 13, 21, and 34. Okay, we overshot the goal. The largest Fibonacci number in 30 is 21. That leaves 9. The largest Fibonacci number in 9 is 8. That leaves 1, which is itself a Fibonacci number.

Therefore, the Fibonacci decomposition of 30 is 21 + 8 + 1.

The optimum strategy is to take the last number in the decomposition. For 30, take 1. For 29, take 8.

If you are unable to take the number you want, just take any random number that you are allowed to take and hope your opponent will make a mistake.

44.3 Tic Tac Toe

This classic game takes a few more lines to program but can be a very satisfying achievement. Here are the issues:

(a) How will you show the human what the grid looks like?

(b) How will the human tell you where they want to move?

(c) How do you recognize the game to be over?

(d) How do you make the computer be less stupid?

Let's start with the grid. One option is that we can show it like this.

```
 1 | 2 | 3
---+---+---
 4 | 5 | 6
---+---+---
 7 | 8 | 9
```

Moves: The human can tell us their move by giving us the number for the cell they want to occupy. After the human takes cell number 1, the grid might look like this.

```
 H |   |
---+---+---
   |   |
---+---+---
   |   |
```

We could trust that the human remembers what the cell numbers are, since they have probably not scrolled off the screen yet.

Each location could be represented by a variable, `$c1` through `$c9`, for instance. This makes the grid look something like this.

```
print "
 $c1 | $c2 | $c3
---+---+---
 $c4 | $c5 | $c6
---+---+---
 $c7 | $c8 | $c9";
```

We can check for a win as follows.

```
if ( "$c1$c2$c3" eq "HHH" ) { print "Human Wins" }
if ( "$c4$c5$c6" eq "HHH" ) { print "Human Wins" }
if ( "$c7$c8$c9" eq "HHH" ) { print "Human Wins" }
if ( "$c1$c4$c7" eq "HHH" ) { print "Human Wins" }
if ( "$c2$c5$c8" eq "HHH" ) { print "Human Wins" }
if ( "$c3$c6$c9" eq "HHH" ) { print "Human Wins" }
if ( "$c1$c5$c9" eq "HHH" ) { print "Human Wins" }
if ( "$c3$c5$c7" eq "HHH" ) { print "Human Wins" }
```

Repeat with `"CCC"` and "Computer Wins."

Quickly you can see that the program is easy to write, but is going to have a lot of lines.

That's where it gets fun. There is a pattern here. The pattern can be exploited to make the program lots shorter. But it is not easy to see how.

Hint: This will be easier to write once we have studied arrays (chapter 32, page 160).

Chapter 45

Hashes

Arrays and hashes each store values. Arrays do it by numeric index. Hashes do it by key string. Arrays preserve order. Hashes do not. Arrays use push, pop, shift, and unshift. Hashes use keys, values, each, and delete.

If you need to preserve order, use an array.

If you need to store and retrieve by key, use a hash.

A **hash** is also called an **associative array** or a **dictionary**, or even a poor mans's **database**. It is like an indexed array, but the index is called a **key**, and instead of being a number, can be a string. This is a huge "plus." On the other hand, **hash** does not have any concept of order, so push, pop, shift, and unshift do not make any sense. **hash** only supports keys and values.

The introductory symbol for a **scalar** is the `$` dollar sign. The introductory symbol for an **array** is the `@` commercial at sign. The introductory symbol for a **hash** is the `%` percent sign. Using a `#` might have been more mnemonic (memorable), but `#` is already in use for comments.

With an array the index is surrounded by square brackets, like this: `[1]`. With a hash the index (or key) is surrounded by curly brackets, like this: `{1}`.

With a **hash** we can store, retrieve, delete and list by key.

Lots more information can be had by Googling "perl hash".

45.1 Store and Retrieve

In this example, "`1,2`" is the key and "`Florida`" is the value.

```
  $x{1,2} = "Florida";
  $x{xyz} = "New York";
```

Let's store payroll information by name.

```
while ( 1 ) {
  print "name: ";
  chomp ( $name = <STDIN> );
  print "income: ";
  chomp ( $income = <STDIN> );
  $payroll{$name} = $income; }
```

Once the `%payroll` hash is filled, we can use it to look up information about any employee.

```
while ( 1 ) {
  print "name: ";
  chomp ( $name = <STDIN> );
  print "The income of $name is $payroll{$name}.\n";
```

45.2 List by Key, Value, or Both

We can extract an array from a hash. We can get a list of keys. We can get a list of values. We can get a list of keys and values.

```
  @x = keys %whatever; # keys in random order
  @x = values %whatever; # values in random order
  @x = each %whatever; # key/value pairs in random order
```

We can print the contents of a hash using the keys.

```
  foreach $k ( keys %whatever ) {
    print "key: $k, value: $whatever{$k}\n" }
```

We can print the contents of a hash using both keys and values.

```
foreach ( $k, $v ) ( each %whatever ) {
  print "key: $k, value: $v\n" }
```

45.3 Lookup

We can find whether a specific key is in the hash. **exists** checks the key. **defined** checks the value. A key can be exist but have an undefined value.

```
if ( exists $whatever{key} ) { key is there }
if ( defined $whatever{key} ) { value is defined }
if ( $whatever{key} ) { value is true }

$whatever{key} = 1; # true, defined, exists
$whatever{key} = "xyz"; # true, defined, exists
$whatever{key} = 0; # false, defined, exists
$whatever{key} = ""; # false, defined, exists
$whatever{key} = $x[7]; # false, not defined, exists
delete ( $whatever{key} ); # false, not defined, not exist
```

Chapter 46

Games and Projects (Hashes)

In this chapter we look at some simple games and projects that use our knowledge up through arrays and hashes.

46.1 Animal

In this game you think of an animal and the computer tries to guess it by asking you yes/no questions. This is an example of a **decision tree**.

The basic dialog goes something like this.

```
Let's play Animal!
Are you thinking of an animal? yes
Does it live on land? yes
Horse? yes
I win.  Let's play again!
```

`%kb` is a hash that contains our knowledge base of everything we know. Each line is the "what next" based on answers already received. That series is called a decision tree. Also we update the decision tree when we learn new information.

```
$kb{''} = "Does it live on land"; # ask when you know nothing
```

```
$kb{y} = "Horse";
$kb{n} = "Whale";
```

`$alist` is the answer list, or list of answers to the questions we have already asked. We update it as the questioning goes along.

```
print "Let's play ANIMAL!\n";
while ( 1 ) { # start a round
  print "\nYou think of an animal. I will guess it.\n";
  $alist = ""; # start over with no answers
  while ( 1 ) { # keep asking questions
    $save = $alist; # basis for updates later
    print "$kb{$save}? "; # ask question (or guess)
    chomp ( $ans = <STDIN> ); # assume y/n answer
    $alist .= $ans; # update our answer list
```

After each question, we are closer to reaching a conclusion. Eventually we will run out of questions. When that happens, if the last answer was "y" then we guessed correctly. If not, we try to learn a new animal.

```
    if ( exists $kb{$alist} ) { next }
    if ( $ans eq "y" ) { print "Hurray!  I win!\n"; last }
    # must be a new animal
    $old = $kb{$save};
    print "I give up.  What animal was it? ";
    chomp ( $new = <STDIN> );
    print "Oh, $new. Wow. I never would have guessed.\n";
    print "What question could I ask to tell the
difference between an $old and a $new? ";
    chomp ( $question = <STDIN> );
    print "For $new what would the answer be? ";
    chomp ( $yesno = <STDIN> );
```

At this point we have a new animal and a new question. We can update the knowledge base.

```
    $kb{$save} = $question;
    $kb{$save . "y"} = $old;
```

```
    $kb{$save . "n"} = $old;
    $kb{$save . $yesno} = $new;
    last }
  print "Let's play again!\n" }
```

Write it. Run it. Play it. Figure out how to improve it. (It's pretty much bare bones and could use a lot of loving care.)

Idea for debugging: Add a "list" command that tells all the information in your database. Use `keys` to get a list of the keys. Use `sort` to sort the keys into ascending order. Use `foreach` to walk the list and print each matching key and value.

Idea for improvement: Read the chapter C (page 310) on File I/O. Add a "save" command to your game. Write out the knowledge base. Next time the game starts read in the knowledge base so you do not lose all the knowledge learned from the users of your game.

46.2 Exploration

This is a variation of a classic game called **Zork** or **Adventure**. In this game, you are at one of several locations. A description of your surroundings is printed. You can then move in various directions. Your location is updated and the new surroundings are described.

In the classic versions of the game, you can also pick up and use objects, but we will not do that here. Instead, we will just explore.

Like Animal, our choices lead us forward. Unlike Animal, we can back up and even wander in circles. So instead of indicating our location by `$alist`, we will use `$at` that holds some id marker for the room we are in.

Another big difference is that the world does not normally change as we explore it (although it could). Everything is set up in advance. The fun and excitement is in creating a fantasy world.

Each location or room is described by data.

```
$kb{0,short} = "Home";
$kb{0,long} = "You are in a beautiful meadow.
North you hear a waterfall.  South you see a river.
East is a mountain.  West is more forest.";
```

```
$kb{0,N} = 1;
$kb{0,S} = "river";
$kb{0,W} = "forest2";
$kb{1,short} = "Waterfall";
$kb{1,long} = "The waters thunder down as you gaze
on in amazement. This is truly a place of great beauties.
But somehow you feel as though there are hidden secrets.
South is a path.";
$kb{1,S} = 0;
$kb{1,N} = 2;
$kb{2,short} = "Hidden Cave";
$kb{2,long} = "You have discovered a mysterious cave
behind the rushing torrents of water. It seems that no
one has been here in ever so long. It is cool and peaceful.
Above you is a ledge.  Perhaps you can climb up to it.";
$kb{2,S} = 1;
$kb{2,U} = 3;
```

This is clearly a fantasy location. You could start by describing the actual house or village where you grew up. Before you key in your knowledge base you may want to start by drawing a map so you can keep everything straight more easily.

My knowledge base starts with the visitor at location `$at=0`. Yours could be different. Mine uses numbers to identify locations. Yours could be different.

When the game starts, and as you enter each new location, you would normally print out the short name of the place, `$kb{$at,short}`, followed by the long name. Then you wait for inputs.

If the user types in some form of the word "North" you set `$dir` to "N" and set `$at` to be `$kb{$at,$dir}`. Similarly for "South" use "S", for "East" use "E", for "West" use "W", for "Up" use "U", and for "Down" use "D". In fact, you can invent your own directions, like "NE", "In", "Under", or whatever.

If the user selects a direction that is not (currently) possible, you could respond with something like: "I'm sorry. I don't seem to be able to go that direction at this time."

If you want to remember where you have been, as you enter each room you can update the knowledge base by saying something like `$kb{$at,seen}=1`. Then, the next time you visit that place you could get by with only saying

the short description.

Chapter 47

Unit Test: Complex

To demonstrate mastery of this unit, you should be able to write programs that combine the skills learned in previous units.

47.1 Exercises

New Operators

Exercise 144: Using approved style, write a program to do the following. Cash Register: Prompt for and read in the total cost of the purchase. Prompt for and read in the amount of money tendered by the customer. Print out the change to be returned, both as a total dollar amount and also as a list of coins. Assume the change will never be more than 99 cents and your coins include 25 cent, 10 cent, 5 cent, and 1 cent coins. Tell how many of each coin to give to the customer. Use the largest coins possible.

Exercise 145: Write a program to do the following. Prompt for and read in a distance in inches. Convert it to yards, feet, and inches. (There are 36 inches in a yard. There are 12 inches in a foot.) Sample input: 100. Correct output: 2 yards, 2 feet, 4 inches.) Do not worry about spacing or exact wording as long as the correct numbers come out in an understandable way.

Exercise 146: Using approved style, write a program to do the following. Leap Year: Given a year number, tell whether it is a Leap Year or not. Leap Year is the year in which February has 29 days. The rest of the time February has 28 days. (We are using the Gregorian calendar, which is the

common calendar used for business throughout the world.)

Rules: Normal years are not leap years. But if the year is divisible by 4, it is a leap year. Unless it is divisible by 100, in which case it is not a leap year. Unless it is divisible by 400, in which case it really is a leap year. Examples: 2000 is a leap year. 2100 is not a leap year. 2008 is a leap year. 2009 is not a leap year.

Nested Loops

Exercise 147:* Factor Counter: Use nested loops. For each number from 1 to 100, print out the number and the count of factors it has. Here are the first few answers: 1 1, 2 2, 3 2, 4 3, 5 2, 6 4, 7 2. "6 4" means that 6 has four factors. They are 1, 2, 3, and 6. A factor is any number that divides into the other number and leaves no remainder.

Exercise 148:* Use nested loops to print out this figure:

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

Exercise 149:* Use a pair of nested loops to print this triangle:

```
9876543210
876543210
76543210
6543210
543210
43210
3210
210
10
0
```

Exercise 150:*

Multiplication Table. Use a pair of nested loops. Prompt for and read in a number. Print the times tables up to that number, each number taking four spaces. Example, if the number is 5, print the following:

```
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25
```

Hint: To make $x always take four spaces when printed, try this:

```
$x = sprintf ( "%4d", $x );
```

Exercise 151: Using approved style, write a program to do the following. Print out the prime numbers from 2 through 1000. Use an outer loop to consider all reasonable candidates. Use an inner loop to consider all reasonable factors. A factor is a number that divided exactly into the other number (with a remainder of zero). Example: 2 3 5 7 11 13 17 19 23 ...

Unit VII

Publishing

Chapter 48

Web Hosting

Your instructor may provide you with web hosting in conjunction with your enrollment in this class. If not, you can get something on the open market. Prices are very reasonable lately.

To put content up on the web, the cheapest alternative is to buy a hosting package from a large web hosting service provider.

At this writing (Feb 2009) such packages are available for around $6 per month on a two-year contract. They provide unlimited storage for your website (but not to back up your local files, just for web content). Some provide a command-line login. You can get a Linux or Windows server. Typically they throw in a doman name for free, but that can be a trick to keep you with them since the name is not always transferrable if you change hosts.

Use your favorite search engine. Search on "web hosting reviews". See what comes up. Read a few reviews.

Your first goal is to figure out what criteria are important to you.

Does the host allow you to have CGI programs? You want that to be "yes."

Does the host support programming in Perl? You want "yes."

Does the host provide a command-line (ssh) login? You want "yes."

What control panel do they provide, if any? **cPanel(R)** is popular but there are many others. Do they let you try it out? Many do.

Can you set up an email address? You want "yes." How many?

Can you host more than one domain? You want "yes."

How much hosting space is provided? Lots is better than little.

How much bandwidth can you use per month? Lots is better than little.

How many people are you sharing with? (You won't find that in the advertising but you may find it in the reviews or blogs.) If lots are sharing the machine your performance will be miserable.

Are you limited in the types of content you can post? All restrict copyright violation such as posting "warez" (games hacked to remove their copy protection) and pirated music or video. Most places restrict Adult Content (pornography). Many also restrict spamming. Some restrict large files such as streaming videos and sound recordings.

Decide what things are important to you and look for a good provider.

Chapter 49

Passwords

49.1 Methods of Authentication

Computers facilitate access to resources. Before computers we dealt with humans, typically on a familiar basis. They knew us. We did not have to prove who we were when carrying out business. When you borrow $5 from your friend, he probably does not ask to see your ID.

But when you ask for $20 from the cash machine, it wants your ID card and your PIN number. Why is that? Because the computer does not know who you are.

Is it possible for you to give someone your bank card and have them get the $20 from the cash machine? Sure, if you trust them. Give them your card. Tell them your PIN. Let them handle the transaction.

The purpose of the bank card and the PIN number is authentication. Anyone who has both those items will be presumed to have the right to use them.

The key to a door is typically a single token of authority.

The bank card and pin are two tokens that must be used together.

In general, a token is something you are, something you have, or something you know.

Things you are would include finger prints and retina scans. It could include the length and shape of your fingers and toes. It could include the way you walk or the shape of your body. It could include your voice print. Sometimes these are called biometrics.

Things you have would include keys, driver's licenses, and ID cards.

Things you know would also be called "shared secrets." When I am trying to prove to you that I am me, I could tell you something that only you and I know. That would be evidence of my identity. The combination of a lock is a shared secret. The PIN number with your bank card is a shared secret.

49.2 Creating a Password

Cracked: 11151982, 4glory, ALOHA, carlos, cheerio, daniel8, is351, surfer, marathon, monster, tevita

How do you pick a good password? And who cares?

There are many sources of password advice. This is a short summary that should help you pick a good password.

Hackers are constantly attacking web servers and other network resources in the attempt to "own" them so they can be used for sending spam, carrying out distributed denial of service attacks, or doing other bad things. Most days we are probed hundreds of times by hackers looking to break in. We do not want to become a victim.

To stay one step ahead of the hackers, on our departmental web hosting machine we run our own password hacking program to test all passwords for ease of break-in. If we break your password, we assume a hacker could too. So we warn you to secure your account better, and if you don't fix it we suspend your account to protect ourselves.

The cracked list above a partial list of actual student-chosen passwords that were cracked by our password hacking program. Those would be examples of bad passwords.

Here are some rules for picking a good password:

* The key thing is that your password should be (a) easy for YOU to remember, (b) hard for someone who sees it to remember, and (c) hard for anyone to guess.

* Especially avoid dictionary words. These are the first things that hackers try. Dictionary is not limited to English.

* I recommend that you use the initial lettes of a phrase you can remember. For example, "It was the best of times. It was the worst of times" might

become “iwtBotiwtWot”.

* Modify your password by replacing some letters with digits or other special characters. For example, “iwtBotiwtWot” might become “1wtB0t1wtW0t” where we replace the “i” with a digit “1” and the letter “o” with a digit “0”.

* Here are some character swaps to get you thinking: `A=4=@ B=%=8=6 E=3 G=6 I=1(one)=l(el)=! K=1<=x O(oh)=0(zero) q=9 er=0r S=$=5 T=7=+`. Anything that looks graphically similar would work.

* Change your password if you think it has been discovered. Some places require passwords to be changed every six weeks or at some other set interval. I have seen no evidence that this improves security. Many users simply alternate between two passwords, as in Aloha1 and Aloha2. Others append the date to their password, as in AlohaJan, AlohaFeb, AlohaMar, etc. Is that really any more secure? Well, maybe a little. But password cracking programs will probably not be fooled for long.

Chapter 50

HTML

Warning: **CSS** (Cascading Style Sheets) is the right way to do things. But it is beyond the scope of this book. Instead we will **simplify a great deal** and teach you enough to get started. Take a course or read a book about making web pages to learn more.

HTML is used to **mark up** a document (content). The markup specifies formatting for the content. HTML stands for "hyper text markup language." We will briefly identify the main markup elements you need to know. Remember that we are (here) greatly simplifying the world of HTML, and you can find more powerful and complex markup commands discussed elsewhere. Our purpose here is simply to introduce you and get you started.

`<html>` starts your document. `</html>` ends your document. Everything else goes between those tags.

`<head>` starts the heading portion of your document. `</head>` ends the heading portion of your document. The heading portion includes the title of your web page, which should be between the `<head>` and `</head>` tags.

`<title>` starts the title portion of your document. `</title>` ends the title portion of your document. Naturally the title goes between them. The title may be displayed on the frame of the browser or in the tab marker for the web page.

`<body>` starts the content (body) portion of your document. `</body>` ends the content (body) portion of your document. All the content of your document goes between those two tags.

`<h1>` and `</h1>` delimit a main heading in your document. The words of

the heading go between them. Headings range from h1 to h6.

`<p>` is used to end a paragraph.

`<br>` is used to break to a new line in the same paragraph.

That should get us started. Here is a sample web page using html.

```
<html><head><title>Aloha</title>
</head><body>
<h1>Aloha from BYUH!</h1>
This is a short web page!<p>
Well, that's all folks!<p>
</body></html>
```

50.1 Web Forms

Web forms provide the input to CGI programs. They consist of blanks into which the user can type information, and buttons or boxes of several types that can be checked or pressed. A web page can have any number of forms.

50.1.1 `<form>`

Each form starts with a `<form>` command and ends with a `</form>` command. Forms cannot be nested within one another. Here is a sample `<form>` command:

```
<form method="post" action="bar.cgi">
```

50.1.2 `<input>`

Within the form, the most important item is the `<input ...>` item. These create the data entry areas, the check boxes, and the buttons that can be pressed to communicate with your CGI program.

```
<input type="button"   name="x" value="y">
<input type="checkbox" name="x" value="y">
<input type="file"     name="x" value="y">
<input type="hidden"   name="x" value="y">
```

```
<input type="image"    name="x" value="y">
<input type="password" name="x" value="y">
<input type="radio"    name="x" value="y">
<input type="reset"    name="x" value="y">
<input type="submit"   name="x" value="y">
<input type="text" name="x" value="y" size="20">
```

There are many resources on the web to show you examples of the `<input>` command. Each input will have a name and a value. The name and value are sent to your CGI program.

50.1.3 Example

In this example, there is a form with three visible inputs: nuts, bolts, and enter. When the user keys in values for nuts and bolts, and presses the enter button, a string of information is sent to the CGI program.

```
<form method="post" action="bar.cgi">
<input type="hidden" name="f" value="1">
<input type="text" name="nuts" value="" size="20">
<input type="text" name="bolts" value="" size="20">
<input type="submit" name="done" value="enter">
</form>
```

If the user keys in the values `19` for nuts and `27` for bolts, and clicks on the `enter` button, the CGI program will receive a single line of standard input with exactly the following content:

```
f=1&nuts=19&bolts=27&done=enter
```

A program can be used to read this line and extract the data from it.

50.2 Tables*

When you want to align the elements of your web page into neat columns, it is handy to use a table. You can start a table using a `<table>` command, and end it using a `</table>` command. Within the table, there are rows,

each of which starts with a `<tr>` (table row) command and ends with `</tr>` at the end of the row. Within the row are data items. Each is introduced by a `<td>` (table data) command and ended by a `</td>` at the end of the data item. Here is a sample table.

```
<table>
  <tr>
    <td>upper left</td>
    <td>upper right</td>
  </tr>
  <tr>
    <td>lower left</td>
    <td>lower right</td>
  </tr>
</table>
```

You can nest tables inside one another.

50.3 Validator*

The `w3.org` consortium provides a free web page validation service. Go to their web site at

http://validator.w3.org/

and type in the URI of your web page. It will tell you if your page has the correct syntax or not, and how to fix it.

This chapter does not teach you enough to be able to write HTML that will validate properly. We skip a lot.

Chapter 51

Forms: Web-based Input

Chapter 11 (page 70) showed how to put simple programs online. By "simple" we meant something that does not work with user input. Chapter 24 (page 122) built on that foundation by showing how to receive simple (closed set) user input, specifically the pressing of buttons. In this chapter we will show how to receive multiple and arbitrary (open set) user inputs.

51.1 Counting to N: Offline versus Online

Here is a simple program.

```
print "Professor Colton's Counting Program\n";
print "count how high? ";
chomp ( $high = <STDIN> );
for ( $i = 1; $i <= $high; $i++ ) { print " $i" }
print "\nDone!\n";
```

We wish to count to N online.

This gets a bit tricky because web programs do not stop in the middle to ask for input. Instead, we will run the program twice. The first time we will ask for input. The second time we will process it.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";
```

```
print "Professor Colton's Counting Program\n";
chomp ( $input = <STDIN> );

if ( $input eq "" ) { # nothing was entered
  print "count how high? ";
  print "<form method='post'>";
  print "<input name='high'>";
  exit }

$input =~ /high=(.*)/;
$high = $1;

for ( $i = 1; $i <= $high; $i++ ) { print " $i" }
print "\nDone!\n";
```

I will explain the new parts.

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

This line accepts one line of input sent from the browser to your program on the server.

`if ( $input eq "" ) { # nothing was entered`

This line checks to see if the input was blank. If it was blank, we will ask the question. Otherwise we will generate the answer. If you use `chomp` on your input, check for `""`. If you do not use `chomp` on your input, check for `"\n"`.

`print "<form method='post'>";`

This line tells the browser that it is expected to send our program the inputs provided by the user. Notice there are no spaces around the equals sign.

`print "<input name='high'>";`

This line tells the browser to make an input box (**field**) on the web page and to name it `high`. Notice there are no spaces around the equals sign. Notice there are quotes (single or double) around the field name.

Every input needs a **field name**. The field name should be composed of letters and digits like a normal variable name. It is possible to use other characters as well, but that will make things tricky. As you start out, avoid anything but letters and digits in field names. Also give each field a different

name. Later, when you know how to properly extract and escape things, you can break these rules.

```
exit
```

This line tells the server to stop running your program. Everything you have printed will be sent to the browser. Nothing else will be done at this time.

```
$input =~ /high=(.*)/;
```

This line tells the server to look through the line of input it got. It is looking for the information sent by the browser. If the user keyed in the number `55`, the browser will send `high=55` to the server and the server will feed it to your program. Your program will look for `high=` followed by a number.

At this point, consider the `(.*)` part to just be magic.

```
$high = $1;
```

This line takes the number found and copies it to a variable named `$high` for later use.

Input variations: We can check for various configurations of input by include an `if` statement. We could do something like this:

```
if ( $input =~ /high=(.*)/ ) { $high = $1; ...
```

This would try to match the input to a pattern consisting of the letters "high=" followed by any characters. If the match is successful, the body of the `if` statement would be executed. If not, we would fall through to the next thing.

51.2 Regular Expression Fundamentals

The extraction phase uses a technique called **regular expressions**. Whole books have been written about regular expressions. They are very powerful. We explain regular expressions in chapter 52 (page 237). Until then here is a quick preview.

== is used to compare two numbers.

eq is used to compare two strings.

`=~` (equals tilde) is used to compare a string (on the left side) against a pattern (on the right side). The pattern is called a regular expression. It is possible to also extract the exact characters that matched the pattern.

. (dot) is a Regular Expression that means "match a single character".

.* is a Regular Expression that means "match zero or more single characters". The star (asterisk) means zero or more of the previous thing.

(.*) is a Regular Expression that means "match zero or more single characters and extract the results". The parenthesis mean group and extract.

\d is a Regular Expression that means "match a digit".

\d* is a Regular Expression that means "match zero or more digits". The star (asterisk) means zero or more of the previous thing.

(\d*) is a Regular Expression that means "match zero or more digits and extract the results". The parenthesis mean group and extract.

[+] is a Regular Expression that means "match a plus character". Because plus has special meaning (as one or more), to match a literal plus we place square brackets around it establish a character class context where the plus does not have that special meaning.

`[^&]` is a Regular Expression that means "match any character except ampersand (`&`)". The caret (`^`) means "everything but" in the character class.

`([^&]*)` is a Regular Expression that means "match zero or more characters except ampersand (`&`) and extract them".

51.3 Testing

Sometimes we need to test our online program without having it be online. We can run the program from the command line. When the program starts, if it is expecting any input, it should pause to wait for one line of STDIN.

In the above example, the first time the program runs there is no input. We simulate that by just pressing ENTER when the program starts. That will provide one blank line of input, the same as when your program is first starting up. Your program should then run and produce the HTML output that would normally be sent to the browser.

Make sure `content-type: text/html` is the first line of output. Make sure it is followed by a blank line. (There are other alternatives but they are beyond the scope of this book.)

Make sure the remaining output is the HTML that you were expecting.

Make sure there are no error messages.

The second time the program runs, there should be input. We can simulate such input by typing in one line through STDIN. In the example above, the web form has one input field with a name of `high`. We can simulate entry of the number `12` in that blank by typing the following:

```
high=12
```

The general rule is that each input is expressed as a name, an equals sign, and a value, NO SPACES. The name must be exactly the same as the **field name** that was specified in the input form.

If there are several input fields, we can string them together using the **&** (**ampersand**) symbol. Let's say the form had the following input fields:

```
<input name='x'>
<input name='y'>
<input name='z'>
```

If we want a value of 5 for x, 7 for y, and 12 for z, we could type in the following line of simulated input.

```
x=5&y=7&z=12
```

Notice there are no spaces. There are NEVER ANY SPACES in CGI input. There are never any spaces added around the equals signs or the ampersand signs.

Also, order may make a difference. The browser will send you the fields in the order that they occurred on the form. When you are simulating the running of your program, you should do the same thing.

51.4 Prototype

The following example shows how to write an online program.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";
chomp ( $input = <STDIN> );

(things to do every time)
```

```
if ( $input eq "" ) { # nothing received
  (things to do when no input was found)
  exit }

(extract and clean up the inputs)

(things to do when input was provided)
```

51.5 Adding Two Numbers

We will now show how to get more than one input.

Here is the offline version of the program.

```
print "Professor Colton's Adding Program\n";
print "first number? ";
chomp ( $first = <STDIN> );
print "second number? ";
chomp ( $second = <STDIN> );
$answer = $first + $second;
print "The total is $answer\n";
```

Here is the online version of the same program.

```
#! /usr/local/bin/perl --
print "content-type: text/html\n\n";
chomp ( $input = <STDIN> );

# things to do every time
print "Professor Colton's Adding Program\n";

if ( $input eq "" ) { # nothing was entered
  # things to do when input was not provided
  print "<form method='post'>";
  print "first number? ";
  print "<input name='first'>";
  print "second number? ";
  print "<input name='second'>";
```

```
  exit }

# extract the inputs
$input =~ /first=(.*)&second=(.*)/;
$first = $1;
$second = $2;

# things to do when input was provided
$answer = $first + $second;
print "The total is $answer\n";
```

The interesting part here is extracting the inputs. Everything else should be fairly straightforward.

Notice that each field is listed along with its own `=` sign and with its own number.

Notice that the first field is extracted into `$1` and the second field is extracted into `$2`.

If the fields just contained letters and digits, we are done. However, if the fields might include spaces or special characters we have another step. We have to replace plusses and percent codes.

51.6 Special Characters (plus and percent)

CGI input defines certain characters to be field separators. Specifically the ampersand (`&`) is used to separate name-value pairs from one another, and the equals sign (`=`) is used to separate names from values. But what happens if a name or a value includes one of these special characters? They must be quoted somehow.

Browsers quote special charcters. Letters, digits, and a few other characters usually come through exactly as typed, but most special characters are replaced by **percent codes**. Before use, those codes must be converted back into their original characters.

The most noticable replacement is that each **space** is replaced at the browser by the **plus** sign. Thus, "this is a test" gets transmitted as `this+is+a+test`.

At the server you can reverse this effect by using the following command.

```
$item =~ s/[+]/ /g;
```

The `=~` indicates a pattern match. The `s/` indicates that a substitution will take place. Between the first pair of `//` we have a regular expression for the plus character. Between the last pair of `//` we have a string literal for a space. The `g` at the end indicates that the change is to be made globally throughout the `$item` and not just on the first occurrence.

The other replacement is that many special characters are replaced by **percent codes** related to hexidecimal ASCII coding. Thus, "a+b" gets transmitted as `a%2Bb`. (The percent code for plus is `%2B`.)

At the server you can reverse this effect by using the following command.

```
$item =~ s/%(..)/pack('c',hex($1))/eg;
```

The `/%(..)/` is a regular expression to find a percent sign followed by two characters. Those two characters will be hex digits. They are extracted and can be accessed through the special variable `$1`. The `eg` at the end indicates both that this operation is (g) globally applied and that the replacement is to be (e) executed rather than literally inserted. The **pack** command converts a number into something else. The `'c'` causes the conversion to be into a character. The **hex** command takes a single argument, `$1` as a hexidecimal number, and converts it into a normal base-10 number.

The entire effect of the command is to replace all the percent codes in `$item` with their corresponding original characters.

Order makes a difference. The plus substitution must happen before the percent code substitution. Otherwise, a true plus would be accidentally converted into a space.

Also, any data extraction must happen before the percent code conversion. Otherwise, ampersands and equals signs that appear in the input could be mistaken for field separators.

51.7 Debugging

Plus signs, percent codes, ampersands, only one line of input, the order of fields, and other things can make it difficult to debug a CGI program. It can be very helpful to insert a DEBUG line at the top of your program. That

way you can see exactly what your program is seeing. (The first two lines are already in your program. They are shown here just to provide context.)

```
print "content-type: text/html\n\n"; # already there
chomp ( $in = <STDIN> ); # already there
print "<p>DEBUG: (($in))</p>\n"; # NEW
```

The DEBUG line can appear after the STDIN and content-type lines have occurred. Later, when you want to stop seeing the debug line, just comment it out:

```
# print "<p>DEBUG: (($in))</p>\n"; # NEW
```

Later, if you are having difficulty again and need to see what is going on, activate it again by removing the # from the start of the line.

51.8 Exercises

Exercise 152: Using the techniques shown in this chapter, try converting a few programs from local (offline) to online.

Chapter 52

Regular Expressions

In chapter 51 (page 228) we introduced regular expressions as a way to extract data sent from the browser to a program running on a web server. In this chapter we look at regular expressions in depth.

Regular expressions give us the ability to do open set recognition. **Closed set** means that there are only a few specified options, like rock, paper, and scissors. **Open set** means that there are an unlimited set of options (or nearly unlimited). Rock, paper, scissors, hand grenades, cuddly toys, tacos, perfume, what ever you can think of. It is easy for a program to foresee and deal with each alternative in a closed set. It is more challenging to deal with the unlimited number of alternatives in an open set.

Regular Expressions are used for processing blocks of text. They have other uses, but we will focus on the text uses here. We will use the syntax of PERL. Similar syntax applies to TCL and other languages that support regular expressions.

A block of text is also called a **string**. It is composed of a sequence of characters. Typically we think of the string as being words designed for human view.

We have two main goals. The first goal is to recognize a string as belonging to a certain category, such as phone numbers or female names. As a human, if I show you a phone number, you are likely to be able to identify it as a phone number, even if you have never seen that phone number before.

The second goal is to extract useful information from the string. This information may be located and identified by its surrounding context.

There can be other goals, such as the modification of the string to cause it to meet certain standards. For instance, we could find all instances of two or more spaces in a row and replace each instance with a single space. Or we could look for common spelling errors and replace them with their corrections.

Because these types of problems occurs again and again, computer scientists have been motivated to find a good solution. The solution is called **regular expressions**.

52.1 Recognition

The first goal is to recognize a string as being a member of a certain category. Is it a social security number? Is it a vehicle license plate number? Is it a possible word in English? As a human, you are likely to be able to answer yes or no in each case with a high degree of accuracy. Let's begin by focusing on license plates.

52.1.1 License Plates

In Hawaii currently a vehicle license plate generally consists of three letters followed by three digits. There are other configurations, including so-called vanity plates that can be almost anything. Let's focus on typical plates. Say your computer program input is `ABC123` and you must determine if that meets the pattern. As a human you can easily say yes. But as a computer program, how easy is it? One could build a program with a lot of **if** statements, like this:

```
# Plan A: check every plate
# about 17.5 million lines long
$okay = 0; # default
if ( $plate eq "AAA000" ) { $okay = 1 }
if ( $plate eq "AAA001" ) { $okay = 1 }
if ( $plate eq "AAA002" ) { $okay = 1 }
... you can guess what goes in between
if ( $plate eq "ZZZ999" ) { $okay = 1 }
```

This would be a really long program. About 17.5 million lines. You would probably like to avoid writing it. So here is an alternative that looks at each

character separately.

```
# Plan B: check every letter
# about 127 lines long
$char6 = chop ( $plate );
$okay6 = 0;
if ( $char6 eq "0" ) { $okay6 = 1 }
if ( $char6 eq "1" ) { $okay6 = 1 }
...
if ( $char6 eq "9" ) { $okay6 = 1 }
$char5 = chop ( $plate );
$okay5 = 0;
if ( $char5 eq "0" ) { $okay5 = 1 }
...
$char1 = chop ( $plate );
$okay1 = 0;
if ( $char1 eq "A" ) { $okay1 = 1 }
if ( $char6 eq "B" ) { $okay1 = 1 }
...
if ( $char6 eq "Z" ) { $okay1 = 1 }
$okay = 1; # default
if ( $char1 != 0 ) { $okay = 0 }
...
```

Validating one digit takes 12 lines. Validating one letter takes 28 lines. That totals 127 lines. You would probably agree it is a big improvement over Plan A above.

52.1.2 Character Classes

To recognize a digit, we can use the following regular expression.

```
if ( $ch6 =~ m/[0123456789]/ ) { $ok6 = 1 }
```

This deserves some explanation. The first interesting thing is the `=~` operator, which in PERL indicates that a regular expression is being matched or substituted. (It can also indicate a translation operation, but that is beyond the scope of this chapter.)

The `m/../` part indicates that a matching operation is happening. The `m` means match. (An `s` is used for substitution.) The slashes are delimiters that surround the regular expression.

The `[0123456789]` part indicates a character class. It matches when any of the characters in that list matches.

This gives rise to yet another plan.

```
# Plan C
# about 14 lines long
$ok = 0;
$ch6 = chop ( $plate );
if ( $ch6 =~ m/[0123456789]/ ) { $ok++ }
$ch5 = chop ( $plate );
if ( $ch5 =~ m/[0123456789]/ ) { $ok++ }
$ch4 = chop ( $plate );
if ( $ch4 =~ m/[0123456789]/ ) { $ok++ }
... do something for the letters
if ( $ok == 6 ) { success! }
```

You can see that this is a big improvement over Plan B. It shortens the testing of each digit by nine lines. It shortens the program considerably. The result is about 14 lines, which almost 90% smaller than 127 lines.

52.1.3 Character Ranges

It is common to want to match a character to an uninterrupted range of characters, such as zero to nine, or A to Z. Regular expressions typically include a short-cut way to express the same thought given above by using a dash to indicate a character range.

```
if ( $ch6 =~ m/[0-9]/ ) { $ok6 = 1 }
```

Characters (digits and letter and punctuation) are stored in the computer as a sequence of binary digits (bits: zeros and ones). One typical code is called ASCII. Another is UNICODE. Character ranges work because the digits zero to nine take up adjacent positions in the character code.

```
0 in ASCII is 0110000
```

```
1 in ASCII is 0110001
2 in ASCII is 0110010
3 in ASCII is 0110011
4 in ASCII is 0110100
5 in ASCII is 0110101
6 in ASCII is 0110110
7 in ASCII is 0110111
8 in ASCII is 0111000
9 in ASCII is 0111001
```

If you are familiar with binary numbers, you will see that the ASCII codes for "0" to "9" are the binary numbers for 48 through 57. The main thing is that the range includes everything we do want and nothing we don't want.

Thinking about letters, we might have considered doing this:

```
$ch1 =~ m/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/
```

Fortunately the letters A to Z form a range. We can say `[A-Z]`. And little a to little z form a range (`[a-z]`). Unfortunately those two ranges are separated so we cannot say `[A-z]` and have it work correctly.

Here is an improved program

```
# Plan D
# about 14 lines long
$ok = 0;
$ch6 = chop ( $plate );
if ( $ch6 =~ m/[0-9]/ ) { $ok++ }
$ch5 = chop ( $plate );
if ( $ch5 =~ m/[0-9]/ ) { $ok++ }
$ch4 = chop ( $plate );
if ( $ch4 =~ m/[0-9]/ ) { $ok++ }
$ch3 = chop ( $plate );
if ( $ch3 =~ m/[A-Z]/ ) { $ok++ }
$ch2 = chop ( $plate );
if ( $ch2 =~ m/[A-Z]/ ) { $ok++ }
$ch1 = chop ( $plate );
if ( $ch1 =~ m/[A-Z]/ ) { $ok++ }
if ( $ok == 6 ) { success! }
```

52.1.4 Matching Several Characters

Regular expressions are not limited to matching a single character at a time. We can match several characters at the same time. In this regular expression, we are looking for letter, letter, letter, digit, digit, digit.

```
$plate =~ m/[A-Z][A-Z][A-Z][0-9][0-9][0-9]/
```

Notice that we eliminate the chops and we can make the whole process happen in a single statement:

```
if ( $plate =~ m/.../ ) { success! }
```

This is an incredible improvement, but language tinkerers seem to never be satisfied until they have squeezed the last drop of redundancy out of something.

52.1.5 Multipliers

Here is another improvement.

```
if ( $plate =~ m/[A-Z]{3}[0-9]{3}/ ) ...
```

This improvement uses `{3}` as a multiplier to say that `[A-Z]` must happen exactly three times, and `[0-9]` must also happen exactly three times.

We can use the multiplier to give us some flexibility on the length. Here is a regular expression to match a license plate that consists of between 1 and 7 letters or digits in any order.

```
if ( $plate =~ m/[A-Z0-9]{1,7}/ ) ...
```

Notice that we have included two character ranges in the character class, and we have given a multiplier of `{1,7}` which means a minimum of 1 and a maximum of 7 repetitions.

To match three or more of something, we can say `{3,}`. To match less than five of something, we can say `{0,4}`.

There are three special cases that are so popular they have an even shorter short cut.

```
{0,}   is   *
{1,}   is   +
{0,1}  is   ?
```

When something can be repeated zero or more times, we put a star (*) in the regular expression. For one or more times, we can put a plus (+). If something is optional, meaning it can occur zero or one times, we can put a question mark (?).

Here is a regular expression for a number with a decimal point:

```
[+-]?[0-9]+[.][0-9]+
```

Notice it has an optional sign (`[+-]?`), followed by one or more digits (`[0-9]+`), followed by a dot (`[.]`), followed by one or more digits. If we are using dash (`-`) itself as one of the characters, we simply list it last (or first) in the character class.

52.1.6 A Small Lie

It is time to correct a small oversight in our discussion. To keep things simple, we have not talked about position anchors. There are two of them commonly used with regular expressions. The start-of-string anchor is caret (`^`). The end-of-string anchor is dollar (`$`).

Without these position anchors, all our regular expressions mentioned above will match any string that **contains** the target. For instance,

```
$line =~ m/[+-]?[0-9]+[.][0-9]+/
```

will match any `$line` that has a decimal point number anywhere inside. To restrict our attention to lines that have nothing else before the number, we can say this:

```
$line =~ m/^[+-]?[0-9]+[.][0-9]+/
```

Notice the caret at the front of the regular expression. To restrict our attention to lines that have nothing else **after** the number, we can say this:

```
$line =~ m/[+-]?[0-9]+[.][0-9]+$/
```

To restrict our attention to lines that have nothing else before or after the number, we can say this:

```
$line =~ m/^[+-]?[0-9]+[.][0-9]+$/
```

52.1.7 More Shortening

By the way, the `m` in the `m/.../` construct is optional. We can leave it out if we are using slashes as the delimiters. It is another short cut.

Also, there is a short cut for digit. Instead of saying `[0-9]`, we can simply say `\d`. There are a number of similar short cuts.

52.2 Data Extraction

Say we want the middle two digits of a US Social Security number. We can recognize the number like this:

```
/^\d{3}-\d{2}-\d{4}$/
```

This means: Must begin at the start of the string. Match three digits. Match one dash. Match two digits. Match another dash. Match four digits. Must end at the end of the string.

By placing parenthesis around the part we wish to extract, we can isolate parts of the string that interest us.

```
$ssn =~ /^\d{3}-(\d{2})-\d{4}$/
$middle = $1;
```

In this case, there are parenthesis around the `\d{2}`. Because these are the first set of parenthesis, the matching part is stored in special variable `$1`.

Here is a longer example:

```
$ssn = "321-54-9876";
```

```
$ssn =~ /^(\d{3})-(\d\d)-(\d{4})$/;
$first = $1;
$middle = $2;
$last = $3;
```

In this case, `$first` will end up containing `321`. `$last` will end up containing `9876`.

In chapter 51 (page 228) we look at ways to use regular expressions to unpack the data that comes to us from forms on web pages. We can then use that data to respond to the user.

Chapter 53

State: Persistent Data

Computers run programs to interact with people. Program operation is characterized by flurries of activity to satisfy a request, and long periods of silence in between, waiting until the user is ready to talk again. The user may view this as an ongoing interactive **session**. However, during the periods of silence, the computer usually turns to other tasks.

This chapter introduces the concept of a **persistent** data, also known as **state**, also known as a **session**. It identifies several ways to recover the state of a transaction so that the session can be established or continued, including visible fields, hidden fields, and http cookies.

While computer activity is under way, information is held in variables and arrays. When a program resumes operation after a long period of silence, it restarts with no memory of what it did before. Sometimes it needs to know what happened before so it can do the proper thing next. When that is the case, data storage needs to be involved.

Current information (what the user is giving us now) helps us know what the user wants to do next.

The long-term information that we need to remember from one run to the next is called **state** or **persistent** data. It includes the history of our past dealings with this particular user. It helps us decide how to respond to current requests.

53.1 Medical Records

Consider medical records. Each time you visit a healthcare facility or provider (for example, a hospital emergency room or a medical doctor), you are seeking to cure an illness or prevent future problems. Medical tests may be performed and analysis done. The results are entered into your medical chart. If there is no chart, a new one must be prepared. If there is a chart it can be consulted to find out your medical history. The chart contains details the might be important in the future, including data to help us see trends. The chart might be kept by you or by your healthcare provider.

Imagine that you keep your medical records. What are the benefits and difficulties?

Benefits of Keeping Your Own Records

One benefit is that each time you visit a new doctor, you can bring along your records. That saves time because the doctor does not need to request the records from the previous provider. In case of emergency, especially if you are traveling away from home, the time saved could make an important difference in your survival. Another benefit is that you can see exactly what is written in those records. This might allow you to uncover any errors that have crept in. It might also allow you to learn more about your medical conditions, enabling you to research them on the Internet. This could make you a more informed consumer of health care services. Another benefit is that your doctor would not need to maintain records on each patient. Whole rooms full of files might be turned into more useful space.

Difficulties with Keeping Your Own Records

One difficulty is that you might lose the records. In this case, a backup should be kept somewhere. Another difficulty is that you might find out you are going to die. You might not want to know what is really wrong with you. People differ on how much of the truth they want to hear. You may differ even from day to day. Another difficulty is you might change the records because you don't like what they say. You don't want people to know that you have some particular problem. Or you may want to receive some special treatment. The opportunity for mischief could make medical personnel less trusting of what they find in your records.

53.2 Persistent Data

Similar to the medical records example, persistent data is involved in many of our dealings with computers.

Many examples come to mind: financial records including tax returns and bank balances, school records including transcripts and grades, and travel documents including passports and visas.

Each time your CGI program starts up, it receives one line of input sent by the browser. That line of input contains the contents of each field that was on the form (screen) being sent. That line might be really, really long. One simple solution is to include all the needed data amongst the information being sent from the screen.

This solves the problem. Nearly. There are two real issues with it. First, there is clutter. And second, there is trust.

53.3 Clutter

Should we clutter up the screen? No.

For input to be transmitted from the browser to the server, it must be part of the **form** that appears on the screen.

(Actually, **cookies** can also be used but we will not discuss them in this book except to say (a) they are an option, (b) some people turn them off because they do not trust them, and (c) if you understand the rest of this section, you should be able to do the research on cookies to find out everything you need to know to utilize them.)

It can be distracting to a user when their screen has too much data that is visible. The user interface is very important and too much complexity will be confusing and possibly frightening to the user.

Fortunately we have a solution. We can utilize **hidden** fields. These are fields with a name and a value just like all other fields. They are part of the form but they do not appear on the screen.

Following is an example of a hidden field that contains a user name and password previously entered by the user.

```
<input type='hidden' name='user' value='fred'>
```

```
<input type='hidden' name='pass' value='aloha'>
```

With these two fields included in the form, each time the form is submitted the CGI program will receive the following as part of the input provided by the browser:

```
user=fred&pass=aloha
```

This should allow the server to access `fred`'s data as needed to do the tasks for which it was designed.

There is (theoretically) no limit to the number of hidden fields you can include, or how much data can be in each one. Practically speaking there may be memory or buffer size limits for all fields. I once ran into a limit around 4192 characters. If this might be a problem for you, do some experimenting to find out. It may depend on your server. It may depend on the user's browser.

53.4 Trust

Should we trust the user? It depends.

How would a bank feel about a customer who walked in and said, "Hey, I have ten thousand dollars on deposit in your bank, and I would like half of it now please." How could they know if he is telling the truth?

The bank may not trust you to remember exactly how much money is on deposit. You could be lying about it. Depending on the size of the request, the bank will normally consult it's own trusted records for that piece of information.

If you are a customer of that bank, they will have a file on you. That file will probably be available based on an account number, and possibly based on the account owner's name.

The bank might say, "Hello. May I please see some identification?" You provide a driver's license or passport. The bank compares the picture and checks the ID documents for signs of tampering. If nothing seems to be strange, they will conclude you are who you say and they will grant you the privileges that the account owner should have.

In this case, the bank trusts the user to provide some identity information. It then **authenticates** or verifies that information by comparing to:

(a) **something you know**, a shared secret, like what is your PIN number or your mother's maiden name, or what was the date and amount of your last deposit,

(b) **something you possess**, a physical artifact, like a special card or a passbook or a key to the deposit box,

(c) **something you are**, also known as **biometrics**, like a finger print or a photograph of you that they may have on file,

(d) **something you can do**, like create a signature similar to the one they have on file, or respond to a **challenge** with a suitable response. In this case I may give you a random challenge and you may need to come up with the proper response. Someone overhearing the process would still be unable to steal your identity unless the exact same challenge question was asked.

Each of these authentication methods suffers from limitations. A password or key can be copied. An ID card can be forged.

After proving your identity, you are granted access to your account.

With a computer system, frequently **authentication** is done by a user name and a **password**. The user name identifies the account. The password proves that you know the shared secret.

Whole books have been written about authentication and **identity theft**, but the information above should more than suffice. If you want to know more, search the Internet using the key words given above.

53.5 Data Kept by the User

We trust the user to assist in establishing their identity. And we trust the user to make their request.

Normally we do not trust the user to accurately remember the history of our dealings with them. We use our own internal records for that. And we use our own internal policies to decide what to do next.

An online shopping cart is an interesting example. The shopping cart is really only an extended method for the user to make a request. The request consists of a list of items to be purchased and details of who will pay and

where to send the merchandise. Authentication can come last. If the shopping cart is abandoned and no sale takes place, we do not need to remember anything. If a sale does occur, we need to make a proper record of it in case questions later arise.

For an online shopping cart, until the purchase is complete we can trust the user to maintain all of the data about the upcoming transaction. However, we would probably verify some things just before completing the transaction. Are the claimed prices accurate? Do we still have sufficient stock on hand? Will the credit card be honored?

Hidden fields provide one way for a program to let the user be responsible for their data without too much clutter.

53.6 Data Kept by the Provider

Computer files, including especially Databases, provide a way for the program to keep its own information. Chapter 55 (page 255) will introduce databases and show how to use them to store and retrieve trusted persistent data by keeping it where the user cannot mess it up.

Chapter 54

HTTP Cookies

In chapter 53 (page 246) we introduced the concept of a **session**, and that there are several ways to recover the state of a transaction so that the session can be established or continued. Displayed fields (such as user name and password) can be sent from the web page. Hidden fields can also be sent from the web page. **Cookies** can also be sent, not actually *from* a web page, but in conjunction with it. This chapter discusses Cookies.

See also http://en.wikipedia.org/wiki/HTTP_cookie.

Cookies are kept by the user, or specifically the user's browser. Because of this there are several advantages and several disadvantages.

54.1 Advantages of Cookies

Because cookies are maintained by the user, they need not be held by the service provider. The service provider (that would be you writing the CGI program) may have thousands or millions of customers or clients, and yet not need to keep track of any of them. This can greatly reduce the data storage burden for the service provider.

Cookies reach farther than Hidden Fields: Hidden fields are restricted to operation on the form on the page for which they were transmitted and no other page. Cookies are sent on each web page request to a related website. This means that cookies can share data between websites and between web pages. The user can have two or more pages open in different tabs or browser windows and information can transfer between pages, even days apart. This

can be very handy.

54.2 Disadvantages of Cookies

Because cookies are held by the user, they are at risk of tampering. Essentially, cookies cannot be fully trusted. They can be used for tracking or light-weight relationship management, but anything serious should probably be kept by the provider and stored in a database instead. Data encryption can provide a level of protection against some risks. If you encrypt the data you have sent, when it comes back and decrypts successfully you can probably trust that the data has not been tampered with. But can you trust that it came from the same person as you expect?

Another disadvantage of cookies is that some users do not trust them. I don't have statistics, but my sense is that this was once a larger percentage of users but seems to be smaller now. Still, some users will either refuse cookies (in which case you can reciprocate by refusing service), or they will delete the cookies later, perhaps each time they close their web browser, making them look like a new customer each time they visit.

54.3 How To Set A Cookie

Cookies can be sent from the web server to the brower. Specifically cookies can be sent as part of the header information in conjunction with the **content-type** line.

Cookies can be tagged to indicate which web sites or web pages belong with it. They can also be tagged to indicate expiration date. The expiration can be at the end of the session or at a specific date in the future. Users can, of course, override those expiration dates and cause the cookie to be deleted earlier than expected.

Without an expiration date, the cookie will expire when the browser is closed.

```
print "content-type: text/html\n";
print "set-cookie: name=value;\n";
print "\n"; # blank line to end the header
```

With an expiration date, the cookie requests to persist until the specified date.

```
print "content-type: text/html\n";
print "set-cookie: name=value; expires=(date);\n";
print "\n"; # blank line to end the header
```

The `(date)` part would of course be replaced by a date written in some reasonable format.

Cookies can also be set by a JavaScript (or similar) program that is embedded in the web page sent by the web server.

54.4 How To Read A Cookie

When a web page request is issued, the browser looks through its list of cookies to see which ones are related to the request. It then automatically sends those to the server similar to the way that **form** data is sent.

The server collects the cookies and passes them along to your program. You can find the cookies in the `ENV(HTTP_COOKIE)` **environment variable**.

```
$cookies = $ENV{HTTP_COOKIE};
```

See also http://en.wikipedia.org/wiki/Environment_variables.

Cookies appear to be subject to the same **percent code** encryption as form fields. They appear to be separated by `;␣` (semi-colon, space). They can be separated by code like the following:

```
foreach $nv ( split /; /, $ENV{HTTP_COOKIE} ) {
  $nv =~ s/[+]/ /g;
  ( $name, $value ) = split /=/, $nv;
  $name =~ s/%(..)/pack('c',hex($1))/eg;
  $value =~ s/%(..)/pack('c',hex($1))/eg;
  # do something with the name and the value
  print "cookie: <b>$name</b> = $value<br/>\n";
}
```

Chapter 55

Database

In chapter 53 (page 246) we introduced the need for persistent data. Examples included financial records, school records, and travel documents.

In each case, the main documents were not trusted to the user. Instead, the main documents were kept by the service provider or other authority. The user proved his identity and was then granted rights based on the trusted data. This could include the ability to pass through an immigration checkpoint or take money out of a bank account.

55.1 Databases for Trusted Persistent Storage

Databases provide persistent storage. This means your programs can remember things from one run to the next. The dominant database approach of today (2009) is called SQL (pronounced "ess-cue-ell" or "sequel").

The SQL approach is to organize data into **tables**. This approach works fairly well. Tables are collections of **rows** and **columns**, much like a very simple spreadsheet in Excel. The **columns**, also called **fields** or **attributes**, describe the data to be stored.

In this chapter, we learn to use an SQL database system called MySQL. The programming interface also applies to other database engines, such as Oracle. We are using MySQL because it runs fast and is essentially free, so you can run it on your own machine at home if you desire.

55.2 MySQL by Hand

There are three main ways to work with a database. First, we use the web interfaces provided by cPanel and phpMyAdmin. Second, we use the command line interface to directly enter queries. Third, we use the Perl database interface (DBI) to talk to the database server. In the long run, you will do most of your database work using another program that you will write, such as a CGI program. When you are debugging, or when you are doing one-time kinds of things like creating tables, working by hand can be a good solution.

55.2.1 Connect to the MySQL Server

Use this command (e.g., from cs.byuh.edu) to connect to MySQL.

```
mysql -p -h HHH -u UUU
```

Replace `HHH` with the host name. Replace `UUU` with your own username. When you are prompted for a password, type it in. Your instructor should tell you your hostname, username, and password. You will be able to change your password later.

The `-p` tells mysql to prompt you for a password.

The `-h` tells mysql to what hostname to connect to.

The `-u` tells mysql what username to use.

55.2.2 How To Quit

Once you have successfully connected to the mysql server, you should see the following prompt:

```
mysql>
```

This means that the mysql server is waiting for you to enter a command. The first command we will enter is `exit`, so that we will know how to quit when we are done. `quit` also works.

```
mysql> exit
```

55.2.3 A Few Alerts

Pay attention to capital and lower-case letters. On some platforms big and little letters are interchangable, so "DonC" and "donc" are treated the same. This is called "case insensitive." On other platforms the big and little letters are distinguished, so "DonC" and "donc" are different. This is called "case sensitive." Be aware of the difference.

The up-arrow is your friend. Press it several times. Press the down-arrow several times. Try the other arrow keys (left and right). When you have a line looking the way you want it, you can press ENTER from anyplace in the line. You do not need to be at the end of the line. Using these keys will speed up your work.

55.2.4 Change Your Password

The following command allows you to change your password to something you like better. The quotes, equal sign, parentheses, and semi-colon are all required as shown.

```
mysql> set password = password("whatever");
```

Change your password. Type `exit;` to quit. Then log back in using the new password.

Avoid using any of your existing passwords. This is because your mysql password is going to end up in your programs, right in plain text. People may see it, so you may want to make it hard to remember. I recommend you invent something totally random, like "vlksH36E." Nobody will guess that. Even if they see it, nobody will remember it. And you will hardly ever need to type it in.

Certain characters can cause difficulty in a password (or a table name, for that matter). Characters to avoid include space, backslash (\), at (`@`), dollar (`$`), and quote (`"`). There may be others. These characters are "meta-characters" or "escape characters" which means that they have a special meaning. They escape from the normal meaning. For instance, when we print "`\n`" we do not expect to get a backslash and an n. We expect to get a newline. The backslash is special. Now that you have been warned, I will admit that you can use these special characters, but it requires extra care. For most people it is easier to avoid them.

55.2.5 What Databases Exist?

See what databases exist on this "host." Use the following command. It should give you a list of the databases that currently exist.

```
mysql> show databases;
```

55.2.6 Creating a Database

You will probably not do this, but later you may want to set up your own server and databases. Here are the commands to create a database and to grant access rights to a user.

```
mysql> create database DDD;
mysql> grant all on DDD.* to UUU
    -> identified by "PPP";
```

In this example, `DDD` is the name of the database. `UUU` is the username, which is limited to 16 characters. `PPP` is the password. Apparently a user can only have one password, so if you issue the command again with a different password, the first password is replaced.

Notice the `grant` command was split over two lines. When `mysql` did not find a semi-colon (`;`) at the end of the `grant` line, it prompted for more input using `->` instead of `mysql>`. When you get a `->` prompt unexpectedly, maybe type a semi-colon and press enter.

55.2.7 Focus on Your Database

When you arrive in mysql, the instructor will have already granted you all rights within your own database. Tell mysql to focus on your database by typing the following command:

```
mysql> use DDD
```

where "`DDD`" is replaced by the name of the database assigned to you. `mysql` should then respond with "`Database changed.`" Notice that in most commands, a semi-colon ("`;`") is required after the command. On this command the semi-colon seems to be optional.

55.2.8 Databases Contain Tables

You will be working with tables within your database. You can see a list of the existing tables by using this command:

```
mysql> show tables;
```

Initially there should be nothing there. You should see a message saying something like "`Empty set.`"

55.2.9 Create a Table

Create a table by doing something like this:

```
mysql> create table scores (
    ->   student varchar(50),
    ->   score int(6)
    ->   );
```

In this example, `scores` is the name of the table. It has two columns. One is called `student` and can hold a string of up to 50 characters. The second is called `score` and can hold a number up to nine digits, with a default printing width of six digits.

You can put that command all on one line if you like, or spread it out over multiple lines like shown above. mysql will continue prompting you for the remainder of your command until you put in the semi-colon "`;`" to tell it that you are done.

Now show tables again. You should see your new table.

55.2.10 Enter Data into Your Table

Insert something into your table by using commands like these.

```
insert into scores values ( "Fred", 100 );
insert scores values ( "Bob", 70 );
insert into scores ( score, student )
  values ( 95, "Anne" );
insert scores set student="Don", score=75;
```

55.2.11 Display the Data in Your Table

See what you have in your table by using a command like this. Star (*) lists all columns in the default order, but you can assert more control if you want.

```
mysql> select * from scores;
mysql> select student from scores;
mysql> select score from scores;
mysql> select student, score from scores;
mysql> select score, student from scores;
```

You can also control the order in which your information is returned.

```
mysql> select * from scores order by score;
```

55.2.12 Think Beyond the Example

Invent your own table with three or more columns. Insert into it three or more rows. Be creative. When you have your table built and populated, do a "select *" on the table and show your instructor. (Table names and column names do not allow spaces.)

55.3 MySQL by Program

The previous section told how to work with a database by hand. In this section, we show how to work with it from another program. We will illustrate using the Perl DBI (data base interface). We will not show you everything that is possible. Instead, we focus on a few simple commands that will allow you to do some interesting things.

55.3.1 Connect to MySQL

In your Perl program, you can use the following lines to connect to and disconnect from a MySQL database. You are telling the computer the same things you had to specify by hand (in the section above) when you connected to MySQL. The difference is that this protocol is easier for programming, and the by-hand protocol is easier for humans.

```
use DBI; # to include needed definitions
$db = "DDD";
$host = "HHH";
$username = "UUU";
$password = "PPP";
$x = DBI->connect( "DBI:mysql:$db:$host",
  $username, $password, {RaiseError=>1} );
```

Of course, you should replace `DDD`, `HHH`, `UUU`, and `PPP` with the correct information. If your host is the "localhost" you can leave that part off from the connect statement. RaiseError is an example of information you can send as part of the connect process. If the connect fails, `$x` will be false. You can say `if($x)` to test whether the connection worked.

After connecting, you can access any of the tables and data in the database. When you are done, you should disconnect as follows.

```
$x->disconnect(); # when you are done
```

55.3.2 Issue a Query

When you are connected to the database, you can issue queries (inquiries, requests, commands). Data retrieval is probably the most common activity. There is a four-step process for retrieving data from a table. The first two steps are prepare and execute.

```
$query = "select * from inv"; # or ...
$query = "select * from inv order by cost";
$y = $x->prepare($query); # introduce task
if ( !$y ) { die "query failed\n" }
$y->execute(); # carry out the task
```

Replace `inv` with the actual name of your table. It is `scores` in the examples above.

55.3.3 Viewing Results

If your query creates results (like `select` does), then after you have executed the query, the results are ready for you to process. You can fetch the results

one row at a time. If there are many rows of output, you need to do this in a loop so you can fetch each of them. Here are two ways to get the data from each row. In the first way, we retrieve all the columns for one row into an array.

```
while ( @z = $y->fetchrow_array() ) {
  print $z[1]; # print the second column
  ( $xxx, $yyy, $zzz, $frog ) = @z;
  print $frog; # print the first column
}
```

Of course, the names `$xxx`, etc. can be whatever variable names you wish to use. In the second way, we retrieve all the columns into a hash. Columns are identified by their formal name within the table.

```
while ( $z = $y->fetchrow_hashref() ) {
  print $z->{price}; # print price column
}
```

After processing the rows we want, we can end at any time.

```
$y->finish(); # when done with this query
```

55.3.4 Display Your Data

Write a program that displays the rows from the table you created earlier. Connect. Query. Display results. Disconnect. Quit. At first, do not worry if your data do not line up neatly. But do separate them by at least a few spaces so they don't run together.

To make your data line up in neat columns, you can use the `\t` tab character (a quick and dirty solution), or you can format the data to a specific width, using `printf`. Here is an example.

```
$format = "student: %-20s score: %5.0f\n";
printf ( $format, $student, $score );
```

55.4 $x and $y ??

In the previous discussion, we introduced some "objects" that are used to access the data and carry out queries. They were called `$x` and `$y`. This section briefly explains what they are and how they are used.

Imagine you are shopping at Amazon.com. You sit down at a computer, start your browser, and type in the Amazon url. Then you start shopping. You have a "shopping cart." When you find something you want, you add it to your cart. Eventually you pay and check out, or you abandon the cart.

Now imagine that while you are shopping, you move to another computer, surf over to Amazon, and check your shopping cart. Will you see the things that you picked on the first computer? Probably not. Thousands of people are shopping Amazon at any moment. They each have their own shopping cart.

`$x` is like that shopping cart. When you connect to a database, you get back a ticket (`$x`). Every time you want to add to your shopping cart, or remove from it, or pay for it, you need to identify your shopping cart. You need to say "`$x->prepare`" rather than just "`prepare`". `$x` represents the shopping cart.

When shopping at Amazon, you can actually have two or three shopping carts by using different computers, or even by starting another browser window on the same computer. When accessing a database, you can have several sessions going at the same time:

```
$x1 = DBI->connect( ... );
$x2 = DBI->connect( ... );
$x3 = DBI->connect( ... );
```

Each of these connections has its own `$x`, its own shopping cart, so to speak. You can use one to do one thing while you use another to do something else. One could be connected to an automotive parts warehouse while another is connected to a credit card processing facility and a third is connected to a customer database.

55.5 Sample dbselect Program

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

```
# read a line of cgi
chomp ( $in = <STDIN> );

use DBI;
$x = DBI->connect (
  "DBI:mysql:DDD:mysql.cs.byuh.edu",
  "UUU", "PPP" );

# show tables
print "Content-type: text/html

<html><head><title>DB Select</title>
</head><body>
<h1>DB Select</h1>

Please select a table for viewing.

<form method=post action=\"\">
";

$y = $x->prepare ( "show tables" );
$y->execute();
while ( 1 ) {
  ( $table ) = $y->fetchrow_array();
  if ( !defined($table) ) { last }
  print " <input type=submit";
  print " name=table value=\"$table\">\n";
}
$y->finish();

# if table is specified list its contents
if ( "&$in&" =~ /&table=([^&]+)&/ ) {
  $table = $1;
  print "<h1>Contents of Table $table</h1>
<table border=1>
";

  $y = $x->prepare (
    "select * from $table" );
```

```
  $y->execute();
  while ( 1 ) {
    @z = $y->fetchrow_array();
    if ( @z == 0 ) { last }
    print "<tr>";
    foreach $cell ( @z ) {
      print "<td>$cell" }
    print "\n";
  }
  $y->finish();
}

# end of processing
$x->disconnect();
print "</table></body></html>\n";
exit;
```

55.6 Advanced Queries

Here is some additional information you may find useful.

55.6.1 Column Data Types Allowed

In our example above, we created a table with two columns: student and score. Student was followed by the note "varchar(50)" and score was followed by the note "int(6)". Varchar and int are called data types or column types. Here is a more complete sample of the column types allowed in mySQL. For a complete list, consult the mySQL book.

tinyint	-128 .. 127 (one byte)
smallint	-32768 .. 32767 (two bytes)
mediumint	-8388608 .. 8388607 (three byte)
int	9 digits (four bytes)
bigint	20 digits (eight bytes)
float	like C, four bytes
double	like C, eight bytes
decimal(m,d)	string, m+2 bytes
char(m)	string, m bytes
varchar(m)	string, 1 to m+1 bytes
tinytext	up to 256 bytes
text	up to 65536 bytes
date	YYYY-MM-DD, three bytes
time	hh:mm:ss, three bytes
datetime	eight bytes
timestamp	four bytes (auto updating)
year	one byte

55.6.2 Updating a Row

You will need an update query. In this example, `mystuff` is a table name, `desc` is the column to be changed, `yadda` is a new value, and `ID` is the column to be matched.

```
update inven set desc="yadda" where ID=37;
update inven set desc="yadda"
  where ID=37 and price=33.91;
update inven set desc="yadda", price=99.99
  where ID=19;
```

55.6.3 Deleting a Row

You will need a delete query. In this example, `inventory` is a table name, `ID` is the column to be matched, and `99` is a value. If you leave off the "where" part, all rows in your table will be deleted.

```
delete from inventory where ID='99';
delete from inventory where ID like '201%';
delete from inventory where cost=price;
```

55.6.4 How to Add a Column

To modify an existing table, use the `alter table` query. Here are some samples of things you can do:

```
alter table foo add price int after cost;
alter table bar change price float;
alter table bletch drop cost;
```

55.6.5 How to Delete a Table

To delete a table, use the `drop table` query.

```
drop table bletch;
```

55.6.6 Not Case Sensitive

MySQL is **not** case sensitive, so name your tables and rows with that in mind. However, perl **is** case sensitive, so within any programs you write be consistent in how you capitalize things.

55.7 Doing More

If you wish to go beyond this short introduction to database you may want to buy these books.

- **Recommended:** *MySQL*, by: Paul DuBois. New Riders press. ISBN 0-7357-0921-1. SRP $49.99.
- **Alternate:** *Programming the Perl DBI*, by: Alligator Descartes and Tim Bunce. O'Reilly press. ISBN 1-56592-699-4. SRP $34.95.

"MySQL" is more expensive, but really covers both SQL and the Perl DBI well. I recommend it highly.

Write a web-based program that prints Hippopotamus five times.

Write a web-based program that asks for a number, adds one to it, and prints the result.

Unit VIII

Projects

Chapter 56

Projects

Doing a project is a great way to become empowered. Our nominal goal is that each student be able to build something fun and useful. The real goal is to enrich the student by giving them the ability to create the programs they need without always relying on others.

What is the right size for a project at this point in your skills development? This unit contains a few pre-defined projects that could be appropriate for demonstrating and improving your programming skills. They are given as examples. They have served in the past as actual assignments.

Because out-of-classroom projects by their nature are done without supervision, there is some risk that students will get inappropriate help. To guard against this, project points can only be earned by students who have already performed sufficiently well on the in-class exams and activities.

56.1 Invent a Project

Doing pre-defined projects can be boring and can lead to some inappropriate sharing of code. This does not enhance learning. So instead here is a list of requirements that your project should satisfy.

Fun: Your program should be fun. A game would be ideal. Fun is a subjective judgment, so we will trust you on this. If you think it is fun, we will agree that it is fun.

Creative: Do something creative and unique. If it looks like the project your neighbor already turned in, that would not qualify.

Online: It must run online as a web application. Anyone in the world should be able to run your program.

Authorship: The code comments and the program output should clearly identify you as the author and owner of the program.

Images: The program should appropriately use pictures, typically by way of an HTML `<img>` statement. Ideally the pictures would change depending, for example, on the progress of the game.

Input: Your program must accept multiple inputs, for example buttons or text fields, to allow the user to interact with it. Hidden fields count as inputs.

State: Your program must have some sort of meaningful state that it carries forward. Some or all of the state must be carried in hidden fields that are actually important to your program's operation.

Chapter 57

Risk Roller

Chapter 58

Hangman Project

We will create a working online hangman game. In case you are not familiar with this game, you can read a description on Wikipedia.

http://en.wikipedia.org/wiki/Hangman_(game)

An example game, written by the author of this book and very similar to the game you are asked to write, can be played here.

http://hangman.doncolton.com/play/

A Hangman (and Crossword) Solver, also written by the author of this book, can be found here.

http://hangman.doncolton.com/

58.1 Requirements

The first requirement is that the resulting game should be fun to play. This, of course, is true of any game. The fun-ness aspect is not explicitly graded, but it should be kept in mind as you design, develop, and implement your game.

Your program must be written in Perl.

Your program must run as a CGI program on the web.

The URL for your program may be assigned to you. If so, you must make your program work from that URL.

Every time your program runs, it should create a web page that clearly identifies you as the author and Hangman as the game.

When your program is first started, and after each game ends, it should attempt to start a new game. The player should have at least these two options: (1) let the program pick a word to be guessed, and (2) let the user enter his own word to be guessed.

When the user lets the program pick a word, the word should be randomly selected from a list within your program. The list must have at least ten different words to choose from.

When the game is under way, your program must display a graphic illustration (for example, a hangman's scaffold) showing how many times the player has made a mistake. The game starts with zero mistakes. When six mistakes have happened, the game is lost.

When the game is under way, your program must also display letters and dashes to indicate the letters that have been correctly guessed and the letters that have not yet been guessed.

When the game is under way, your program must also display the letters that were incorrectly guessed, in the order those guesses were made.

When six mistakes have been made, the game is lost, and your program must announce that the game is lost and reveal the hidden word. It must also attempt to start a new game as indicated above.

When all letters have been correctly guessed, your program must announce that the game is won. It must also attempt to start a new game as indicated above.

58.2 Suggestions

It is recommended but not required that there be a way for the player to end the game early. If the player takes this option, the hidden word whould be revealed and your program must attempt to start a new game.

It is recommended but not required that your program allow spaces to appear in the hidden word, thus making it a hidden phrase.

It is recommended but not required that your program have a list of words that are challenging to guess.

It is recommended but not required that your program allow the player to enter more than one letter at once. For example, the player could be allowed to enter the entire word or phrase.

It is recommended but not required that your program ignore guesses that have already been considered. Thus, if the player already guessed "e", whether it was right or not, if they guess "e" again it should be ignored.

You may want to provide hints in the form of a list of the most common letters, maybe in order of popularity.

You may want to provide an alphabetical list of letters that have not already been guesses.

You may want to provide random taunts, telling the player how close they are to losing, or making fun of wrong choices. Similarly you may want to provide random statements of encouragement, telling the player how wise and intelligent they are for the good choices they have made.

You may want to use another graphic instead of a hangman's gallows. Perhaps apples on a tree, as mentioned in Wikipedia. Perhaps time through an hour glass. Perhaps a fuse on a bomb. Think of something unique and fun.

58.3 Design

Your program must comply with indentation and other style requirements specified by the instructor.

The standard for indentation is that each level of nesting will have two or more spaces at the front of each line, and that the indentation will be uniform for all lines at that level of nesting.

Subroutines are to be placed after the main program. They should not be nested inside the main program or inside each other.

"`use strict;`" must be invoked to eliminate accidentally global variables.

There should be a subroutine that will return a word to be guessed. This subroutine should be called when the user requests the program to provide such a word. You may want to have an array of words and to select one at random.

There should be a subroutine that will accept as input the target word and the list of letters that have been tried. It should return letters and dashes

(or underscores) to indicate what the user should be shown.

There should be a subroutine that will accept as input the target word and the list of letters that have been tried. It should return a list of wrong letters, in the original order.

There should be a subroutine that will accept input (whatever you deem appropriate) and will return a count of the number of errors that have been made.

All "state" information needed to continue the game should be included in the web page that your program puts up. Thus, many people should be able to simultaneously play your game without interfering with each other. It is not required that the state information be encrypted, but it must at least be hidden to the extent that a casual player cannot see it. If "view page source" reveals it, that is not a problem.

58.4 Sample Code

No, I am not going to give you the source code for the program itself. What I will give you is the HTML code that my program produces.

Reminder: The HTML code here is simplified. It is not a good example of high-quality HTML code. It is an example of the minimum code you might use to achieve the goals of the assignment.

First we have the initial screen.

```
<body bgcolor='lime'><form method='post'>
<h1>Don Colton's Hangman Game</h1>

<img src='hang0.png' width='280' height='291'><br>
<h1>Type in a word or phrase
 for someone else to guess,<br>
or let me pick something (if you dare).</h1>
<input type='submit' name='go1' value='Use this :'>
<input name='phrase' value='type something here'
 size='20'><br>
<input type='submit'name='go2a'
 value='Pick something EASY for me.'>
<input type='submit'name='go2b'
```

```
 value='Pick something CHALLENGING for me.'>
```

It shows several **submit** buttons that allow the user to select their path forward.

Next we have a game underway.

```
<body bgcolor='lime'><form method='post'>
<h1>Don Colton's Hangman Game</h1>
<img src='hang1.png' width='280' height='291'>
<h1>phrase: _ O _ _ O O _</h1>
<h1>miss: U</h1>
<input type='hidden' name='bgcolor' value='lime'>
<input type='hidden' name='phrase' value='DOGWOOD'>
<input type='hidden' name='guesses' value='UO'>
your next guess: <input name='guess' value=''>
<input type='submit' name='go3'
 value='submit'><br>
<input type='submit' name='go4'
 value='I give up. Just tell me.'>
```

It shows several **hidden** fields: bgcolor (background color), phrase (the secret word to be guessed), guesses (the prior guesses). It also has a blank where the next letter can be guessed and two submit buttons.

The **hidden** fields carry along all the information, also known as **state**, that is necessary for the program to make its next calculation. In this case we need to know the secret word and the letters guessed thus far. From that we can decide how many correct letters have been guessed. We can also decide what mistakes have been made.

And we can tell by the number of mistakes which hangman picture to display. Notice the **img** field that indicated `hang0.png` initially and now indicates `hang1.png` after one incorrect guess was registered.

Here we have the successful guessing of the secret word.

```
<body bgcolor='lime'><form method='post'>
<h1>Don Colton's Hangman Game</h1>
<img src='hang3.png' width='280' height='291'>
<h1>phrase: D O G W O O D</h1>
```

```
<h1>miss: U C H</h1>
<h1>You escaped the hangman! Play Again?</h1>
<input type='submit' name='go1' value='Use this :'>
<input name='phrase' value='type something here'
 size='20'><br>
<input type='submit'name='go2a'
 value='Pick something EASY for me.'>
<input type='submit'name='go2b'
 value='Pick something CHALLENGING for me.'>
```

Notice that we have blended the ending of one game with the questions required to start the next game.

Here we have the unsuccessful outcome of not guessing the secret word.

```
<body bgcolor='lime'><form method='post'>
<h1>Don Colton's Hangman Game</h1>
<img src='hang6.png' width='280' height='291'>
<h1>phrase: D O _ _ O O D</h1>
<h1>miss: U C X J M Q</h1><h1>Sorry!
 The secret was ``DOGWOOD''.  Play Again?</h1>
<input type='submit' name='go1' value='Use this :'>
<input name='phrase' value='type something here'
 size='20'><br>
<input type='submit'name='go2a'
 value='Pick something EASY for me.'>
<input type='submit'name='go2b'
 value='Pick something CHALLENGING for me.'>
```

Notice that again we have blended the ending of one game with the questions required to start the next game.

The goal of the program is to create web pages such as these.

Chapter 59

Hotter, Colder

59.1 Computer Picks

This game is called **Hotter**, **Colder**. In this game, the computer thinks of a number between 1 and 100. The human tries to guess it.

On the first guess, the only two answers are "YOU WIN!!!" and "Nope, try again."

On subsequent guesses, the possible answers are things like "YOU WIN!!!", "You are getting warmer", and "You are getting colder".

Here is an off-line version of the game.

```
$num = int(rand(100)) + 1; # pick a number
print "I am thinking of a number between 1 and 100.\n";
print "What is your guess: ";
chomp ( $guess = <STDIN> );
$dist = abs ( $num - $guess );
if ( $dist == 0 ) { print "YOU WIN!!!\n"; exit }
print "Nope, try again.\n";
while ( 1 ) { # infinite loop
  $prev = $dist; # remember what they guessed last time
  print "What is your guess: ";
  chomp ( $guess = <STDIN> );
  $dist = abs ( $num - $guess );
  if ( $dist == 0 ) { print "YOU WIN!!!\n"; last }
```

```
    if ( $dist < $prev ) { print "You are getting warmer.\n" }
    if ( $dist > $prev ) { print "You are getting colder.\n" }
    if ( $dist == $prev ) { print "No change. Try again.\n" }
  }
```

The challenge is to convert this to an online game, complete with pictures and a recounting of the history of guesses.

The user interface must provide a place for the human to type in a guess and a way to submit it.

The web page must maintain a history of guesses, both a displayed version for the human to see and a hidden (or cookie) version that makes up the session / state / persistent data.

Idea: have a button for "I give up, tell me".

Idea: have a way for one human to pick the number that another human will guess. The computer can act as the referee.

Idea: Use graphics to indicate hotter or colder.

Idea: Let the user pick different limits instead of 1 and 100.

Idea: Use a cookie to remember the best score or the top scores from that computer. When someone gets a new high score invite them to join the list.

Idea: Change the statements to things like "You are getting REALLY hot" or "You are FREEZING" depending on just how close the guess was.

59.2 Human Picks

How about a challenge. Computer against human. Human gets to decide who guesses first.

This version is an interesting challenge for the programmer. Allow the human to pick a number and ask the computer to guess. Similar rules as before: display a history of guesses and their results.

Have several buttons for the human to press: "You are getting hotter", "You are getting colder", "No change", and "You win!"

Strategy: After the first guess, theoretically you can rule out half the possibilities each time you guess. Theoretically. It is much more tricky than a game like High/Low, which makes it a bit more challenging to play.

You might guess 33 first, and then 66 on the next guess. If 66 is hotter, we know the number is closer to 66, which would make it 50 or more. Otherwise it is 49 or less. You might want to modify your guesses by a random factor to disguise your strategy.

Unit IX

Appendices

Appendix A

Answers to Selected Exercises

Answers to Selected Exercises

Introduction

1: Yes. I love Wikipedia. You can't prove things but you can get a lot of insight.

2: Yes. I love Google.

3: 1987, but who cares? It was a long time ago. That's all you need to know.

4: Why are you looking here? Use Google or something.

5: Star indicates advanced material. Advanced means I usually do not expect a typical 100-level student to know it, but I might expect an A student to be interested in it.

Chapter 1

6: "sleep 5" pauses your program for the number of seconds that you specify, in this case, five seconds.

7: Reading from STDIN causes your program to wait for input, preventing

its termination.

8: Perl programs are indicated by the .pl ending on the file name.

9: Print does nothing. print (lower case) prints text. With a capital P it is an error.

Chapter 2

10: Human communication is based on common sense, directed by words. Computer communication is based on words alone because computers do not understand common sense. Computers do what you say, not what you mean.

11: Syntax is the wording, the form, the outward appearance.

12: Semantics is the meaning, the substance, the inward content.

Chapter 3

13: "chomp" removes the newline from the contents of a variable. When input is received from the keyboard, it generally ends with a newline. Usually the newline is not really wanted.

14: "chop" removes the last character from the contents of a variable. If the contents is freshly received input, the last character will be a newline. "chomp" is safer for removing such newlines.

15: Nothing. Well, almost nothing. The words are often used interchangably and usually mean the same thing. Technically there are two control codes related to this. One is CR, code 13, hex 0D. The other is LF, code 10, hex 0A. Originally CR repositioned the cursor (print head) to the start of the line and LF moved the cursor down a line. Some people found this to be wasteful and redundant. They went with one or the other. This has resulted in a world of minor confusion.

Chapter 4

16: Legal variable names have a dollar-sign as a prefix, then they start with a letter, then they continue with zero or more letters or digits. Underscores count as letters.

17: Good variable names are, first of all, legal. Also they should be mean-

ingful to the programmer and anyone else that my need to understand the program. Hungarian notation is one suggestion for making good variable names.

18: Case sensitive (for variable names) means that $ABC and $abc are not the same variable. Case makes a difference. BIG LETTERS are called Upper Case or capital letters. little letters are called lower case.

19: Assignment is written as "variable = value", where the name on the left side of the command must be a variable, an equals sign must follow, and then an expression involving constants and/or variables. It must be written in that order. It would be incorrect syntax to write "value = variable".

20: Interpolation is the inserting of some variable into the middle of a string.

21: Concatenation is the combining of adjacent strings to make a longer string. The concatenation operator is a dot (full stop, period).

Chapter 6

22: (line 1) Print should be print. (line 2) chomp needs an open paren next. equals sign needed after inches. closing paren after STDIN. (line 3) star (multiply) needed after 2.54. semicolon needed at end of line. (line 4) Print should be print. line 4: double-quote needed before ;

Chapter 8

23: 10, 16, 7

Chapter 13

25: Nice Pet.

```
print "Please enter the name of an animal:  ";
chomp ( $name = <STDIN> );
# if you do not chomp, the next line will be split.
print "I think $name would make a nice pet.\n";
```

26: Zoo Favorite.

```
print "Please enter a zoo animal name:  ";
chomp ( $a = <STDIN> );
print "I love to see the $a at the zoo.";
# the chomp is necessary to get the right result.
```

30: Tell about an animal at school.

```
print "Enter a kind of animal:  "; chomp ( $a = <STDIN> );
print "Enter a person's name:  "; chomp ( $n = <STDIN> );
print "Yesterday I took my pet $a to school.\n";
print "My friend $n laughed all day.\n";
```

31: Add Five.

```
print "Please enter a number:  ";
$num = <STDIN>;
$result = $num + 5; print $result;
```

32: Subtract Ten:

```
print "Please enter a number:  ";
chomp ( $num = <STDIN> );
$num = $num - 10;
print "Subtracting 10 gives $num.\n";
```

33: Multiply by 2:

```
print "Please enter a number:  ";
chomp ( $num = <STDIN> );
$num = $num * 2;
print "Multiplying by 2 gives $num.\n";
```

34: Add Two Numbers:

```
print "Please enter a number:  ";
chomp ( $num1 = <STDIN> );
print "Please enter a number:  ";
chomp ( $num2 = <STDIN> );
$ans = $num1 + $num2;
print "The sum of $num1 and $num2 is $ans.\n";
```

35: Double It:

```
print "Please enter a number:  ";
chomp ( $num = <STDIN> );
$num = $num * 2;
print "Doubling it gives $num.\n";
```

36: Before and After.

```
print "Please enter a number:  "; $n = <STDIN>;
$before = $n - 1; $after = $n + 1;
print "$before comes before it.  $after comes after it.\n";
```

37: Add Three Numbers:

```
print "Enter a number:  ";
$num1 = <STDIN>;
print "Enter a number:  ";
$num2 = <STDIN>;
print "Enter a number:  ";
$num3 = <STDIN>;
$answer = $num1 + $num2 + $num3;
print "The numbers add up to $answer.\n";
```

39: Airplane Empty Seat Count:

```
print "How many passengers are on board?  ";
$pass = <STDIN>;
$empty = 240 - $pass;
print "There are $empty empty seats.\n";
```

40: Convert miles to kilometers.

```
print "Please enter miles:  "; $miles = <STDIN>;
$km = $miles * 1.6;
print "That makes $km kilometers.\n";
```

41: Save Ten Percent.

```
print "Enter the husband's paycheck amount:  ";
chomp ( $h = <STDIN> );
print "Enter the wife's paycheck amount:  ";
chomp ( $w = <STDIN> );
$tot = $h + $w;
$sav = $tot * 0.10;
print "Savings is $sav\n";
```

42: Phonecard.

```
print "Please enter the connect fee:  "; $cf = <STDIN>;
print "Please enter the per-minute rate:  "; $rate = <STDIN>;
print "Please enter the minutes connected:  "; $min = <STDIN>;
$cost = $cf + $rate * $min;
print "The call will cost $cost\n";
```

Chapter 14

49: It prints "9 is greater than 3."

50: It prints "hello is equal to world." The numeric operators convert

"hello" and "world" into numbers. Since no actual digits are found at the front of the strings, the numeric values are each zero, and are equal to each other. To correct the program we should use "lt", "gt", and "eq" for our comparison operators.

51: Trick question. It prints "9 is greater than 3.3 is equal to 3." The third if statement uses a single equals instead of a double equals. Single is for assignment. Double is for comparison. Therefore, it copies the value of y into x. 3 is true so the print happens also.

Chapter 15

52: "ab1c". Actually Perl uses the empty string for false instead of using zero, so it would really be ("", 1, "").

Chapter 16

53: It prints "3 is less than 9.3 is equal to 9." The reason is that the first if is true, so we get less. The second if is false, so we do not get greater, but the else takes over and we get the equals. The else only applies to the second if.

Chapter 19

54: It prints "3 is less than 9." The reason is that the first if is true, so we get less. We then skip the elsif and else.

55: It prints "x is large." The first three conditions are false so their blocks are skipped. The fourth condition is true so its block is executed. Because a true condition was found, the remaining else line are skipped.

56: It prints "x is large.x is really large." The first three conditions are false so their blocks are skipped. The fourth condition is true so its block is executed. The fifth condition is true so its block is executed. To avoid executing the fifth block we would need to connect the statements using elsif (or nesting). Then the remaining blocks would have been skipped.

57: It prints "b.h.n." Each line that starts with "els" (elsif or else) is a continuation of the previous if statement (and must immediately follow an if or elsif). Therefore, the whole program breaks out into five if ladders. The first ladder has four lines (abcd). The second ladder has three lines (efg). The third ladder has two lines (hi). The fourth ladder has three lines (jkl). The fifth ladder has two lines (mn). In the first block, a is false, b is

true, and c and d are never considered. In the second block, all three are false and there is no else clause so nothing prints. In the third block, h is true and i is skipped. In the fourth block nothing is true. In the fifth block nothing is true so the else is executed.

Chapter 21

58: The answer is "abc1". Actually Perl uses the empty string for false instead of using zero, so it would really be ("", "", 1). But the big question is why not (1, 0, 0). It is because we used mathematical operators instead of string operators. The numeric operators convert "hello" and "world" into numbers before comparing them. Since no actual digits are found at the front of the strings, the numeric values are each zero, and are equal to each other. To correct the program we should use "lt", "gt", and "eq" for our comparison operators.

Chapter 25

59: Trouble.

```
print "Please enter a number:  ";
chomp ( $num = <STDIN> );
if ( $num < 0 ) { print "You are in trouble." }
else { print "So far, so good." }
```

71: Yeehah, Ouch.

```
print "Please enter a number:  "; $num = <STDIN>;
if ( $num % 2 == 0 && $num % 13 != 0 ) {
print "Yeehah" } else { print "Ouch" }
```

75: Department Names:

```
print "Please enter a department prefix:  ";
chomp ( $dept = <STDIN> );
if ( $dept eq "ACCT" ) { print "Accounting" }
elsif ( $dept eq "POSC" ) { print "Political Science" }
elsif ( $dept eq "CIS" ) {
.  print "Computer and Information Sciences" }
else { print "Unknown" }
```

76: Dictionary, leaf:

```
print "What word do you seek?  ";
chomp ( $word = <STDIN> );
if ( $word le "leaf" ) { print "First Volume\n" }
```

```
else { print "Second Volume\n" }
```

77: Concert Lines:

```
print "Please enter a name:  ";
chomp ( $name = <STDIN> );
$name = uc $name;
if ( $name lt "H" ) { $line = "A-G" }
elsif ( $name lt "O" ) { $line = "H-N" }
elsif ( $name lt "V" ) { $line = "O-U" }
else { $line = "V-Z" }
print "You should stand in the $line line";
```

78: Table by Names:

```
print "Please enter a name:  "; $name = <STDIN>;
if ( $name ge "N" ) { print "Please go to table 2.\n" }
if ( $name lt "N" ) { print "Please go to table 1.\n" }
```

79: Party food assignments:

```
print "name:  ";
$name = <STDIN>;
if ( $name lt "M" ) { print "bring bread.\n" }
elsif ( $name lt "R" ) { print "bring salad.\n" }
else { print "bring dessert.\n" }
```

Chapter 26

83: "implicit" means folded or braided in, so the meaning is present but hidden.

84: "explicit" means folded or braided out, so the meaning is obvious.

85: "efficient" means fast or not overly wasteful.

86: "effective" means having the proper effect. In other words, working correctly.

Chapter 30

87: Print 1 to 500.

```
$x = 1; # start at 1
while ( $x <= 500 ) { # quit when we pass 500
.  print "$x "; # print the number and a space
.  $x = $x + 1; } # move along to the next number
```

88: Print 2 to 100 by 2s.

```
for ( $i = 2; $i <= 100; $i += 2 ) { print $i }
```

90: Count from n down to 0, then blast off.

```
print "Starting number:  "; $num = <STDIN>;
while ( $num > 0 ) { # quit when we reach 0
.  print "$num\n"; # print the number
.  $num = $num - 1; } # move along to the next number
print "blast off!\n";
```

95: Print 1 to 500, skipping multiples of 3.

```
$x = 1; # start at 1
while ( $x <= 500 ) { # quit when we pass 500
.  if ( $x % 3 != 0 ) { # only non-multiples of three
.  .  print "$x " } # print the number and a space
.  $x = $x + 1; } # move along to the next number
```

96: Coin Toss 100 times.

```
for ( $i = 1; $i <= 100; $i++ ) {
.  if ( rand(2) < 1 ) { print "H" }
.  else { print "T" }
}
```

100: Triangular Numbers to 1000.

```
$total = 0; $i = 1;
while ( 1 ) {
.  $total = $total + $i;
.  $i = $i + 1;
.  if ( $total > 1000 ) { last }
.  print "$total, " }
```

107: Walking Verse.

```
for ( $m1 = 100; $m1 > 1; $m1-- ) {
.  $m2 = $m1 - 1;
.  print "$m1 miles we yet must walk.
But all along the way we talk.
Talking brings such happy smiles.
Suddenly there's $m2 miles.\n\n" }
print "1 miles we yet must walk.
But all along the way we talk.
Talking brings such happy smiles.
```

```
Suddenly there's no more miles.\n";
```

Chapter 35

108: Add names to an array.

```
while ( 1 ) { # infinite loop
.  print "Please enter a name:  "; # prompt for a name
.  chomp ( $name = <STDIN> ); # read in a name
.  if ( $name eq "" ) { last } # quit upon blank name
.  push @x, $name; # Add the name to the end of an array.
}
```

109: Add names Until William.

```
for (;;) { # infinite loop
.  print "Please enter a name:  ";
.  chomp ( $name = <STDIN> );
.  if ( $name eq "William" ) { last }
.  push @names, $name }
```

110: Use chop to convert string to array.

```
while ( $a ne "" ) {
.  unshift @a, chop ($a) }
```

111: Print Array Items.

```
foreach $x (@a) { print "$x\n" }
```

112: Print multiples of 5.

```
foreach $x (@a) {
.  if ( $x % 5 == 0 ) { print "$x\n" } }
```

113: Print every fifth item.

```
for ( $i = 4; $i < @x; $i += 5 ) {
.  print "$x[$i]\n" }
```

114: Find 5 in an Array.

```
foreach $num (@x) {
.  if ( $num == 5 ) { print "5 is found\n"; exit } }
print "5 is not found\n";
```

– Alternate Solution (without exit) –

```
$found = 0;
foreach $num (@x) {
.  if ( $num == 5 ) { $found = 1; last } }
```

```
if ( $found == 1 ) { print "5 is found\n" }
else { print "5 is not found\n"; }
```

115: Is Joe a Visitor?

```
foreach $name (@visitors) {
.  if ( $name eq "Joe" ) { print "Joe visited.\n"; exit } }
print "Joe did not visit.\n";
```

116: List Things.

```
$num = 0;
foreach $thing (@things) {
.  print "The number $num thing is \"$thing\".\n";
.  $num++ }
```

– Alternate Solution –

```
for ( $i = 0; $i < @things; $i++ ) {
.  print "The number $i thing is \"$things[$i]\".\n"; }
```

117: Adding ARGV.

```
$total = 0;
foreach $x (@ARGV) { $total += $x }
print "total:  $total\n";
```

118: Ten to nine hundred.

```
for ( $i = 10; $i <= 900; $i += 10 ) {
.  push @x, $i }
```

119: Deal Five Cards.

```
for ( $i = 1; $i <= 5; $i++ ) {
.  $card = shift @deck; push @Adam, $card;
.  $card = shift @deck; push @Bob, $card;
.  $card = shift @deck; push @Charlie, $card; }
```

– Alternate Solution –

```
for ( $i = 1; $i <= 5; $i++ ) {
.  push @Adam, shift @deck;
.  push @Bob, shift @deck;
.  push @Charlie, shift @deck; }
```

120: Found in 100.

```
for ( $i = 1; $i <= 100; $i++ ) {
.  $x = <STDIN>;
.  push @a, $x }
```

```
$y = <STDIN>;
$found = 0;
foreach $x (@a) {
.  if ( $y eq $x ) { $found++ } }
if ( $found ) { print "PRESENT\n" }
else { print "ABSENT\n" }
```

121: Perfect Shuffle.

```
@cards = (); @t1 = @left; @t2 = @right;
while ( @t1 ) { push @cards, shift @t1, shift @t2 }
```

122: Reverse Array.

```
@b = (); @temp = @a;
while ( @temp ) { push @b, pop @temp }
```

123: Names until Joe.

```
$name = ""; @names = (); # start empty
while ( $name ne "Joe" ) { # watch for Joe
.  chomp ( $name = <STDIN> );
.  push @names, $name }
foreach $name (@names) { print "$name\n" }
```

124: Total, average, above, below.

```
@nums = (); # start with an empty array
while ( 1 ) { # infinite loop
.  # Read in a list of numbers, one per line
.  chomp ( $num = <STDIN> );
.  # until you get a blank line.
.  if ( $num eq "" ) { last }
.  push @nums, $num }
# Add up the number and print out the total.
$total = 0;
foreach $num (@nums) { $total += $num }
print "The total is $total\n";
# Also print out the average.
$average = $total / @nums;
print "The average is $average\n";
# Then tell how many numbers are above average ...
$above = 0; $below = 0;
foreach $num (@nums) {
.  if ( $num > $average ) { $above++ }
```

```
.  if ( $num < $average ) { $below++ }
}
print "$above numbers are above average.\n";
print "$below numbers are below average.\n";
```

125: Present in Both.

```
foreach $w1 ( @old ) {
.  foreach $w2 ( @new ) {
.  .  if ( $w1 eq $w2 ) { print $w1; exit }
.  }
}
print "no duplicates found";
```

Chapter 41

127: Add Three.

```
sub add3 {
.  my ( $a1, $a2, $a3 ) = @_;
.  my $res = $a1 + $a2 + $a3;
.  return $res }
```

128: Stars.

```
sub stars {
.  my ( $count, $star ) = @_;
.  my $res = '';
.  while ( $count > 0 ) {
.  .  $res .= $star;
.  .  $count-- }
.  return $res }
```

129: Add All.

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

130: Subroutine average.

```
sub average { my ( $sum, $n );
.  if ( @_ == 0 ) { return 0 }
.  $sum = 0; foreach $n (@_) { $sum += $n }
.  return $sum / @_ }
```

131: Extremes.

```
sub extremes {
.  my $max = $_[0];
.  my $min = $_[0];
.  foreach ( @_ ) {
.  .  if ( $_ < $min ) { $min = $_ }
.  .  if ( $_ > $max ) { $max = $_ }
.  }
.  return "$min $max" }
```

135: Subroutine dice.

```
sub dice { my ( $x, $y );
.  $x = int ( rand(6) ) + 1;
.  $y = int ( rand(6) ) + 1;
.  return "$x $y" }
```

137: Lucky, Unlucky, Normal.

```
sub numType { my ( $num ) = @_;
.  if ( $num % 7 == 0 && $num % 13 != 0 ) { return "lucky" }
.  if ( $num % 7 != 0 && $num % 13 == 0 ) { return "unlucky" }
.  return "normal" }
```

141: subroutine yesno.

```
sub yesno { my ( $q, $ans );
.  ( $q ) = @_;
.  for (;;) {
.  .  print "$q:  ";
.  .  chomp ( $ans = <STDIN> );
.  .  if ( $ans eq "yes" ) { return $ans }
.  .  if ( $ans eq "no" ) { return $ans }
.  .  print "Please answer yes or no.\n" } }
```

142: Subroutine Rock.

```
sub Rock { my ( $choice );
.  for (;;) {
.  .  print "Please pick Rock, Paper, or Scissors.\n";
.  .  chomp ( $choice = <STDIN> );
.  .  if ( $choice eq "Rock" ) { return $choice }
.  .  if ( $choice eq "Paper" ) { return $choice }
.  .  if ( $choice eq "Scissors" ) { return $choice }
.  } # end of for loop
```

```
} # end of subroutine
```

143: The Food Story.

```
sub ask { my ( $res );
.  my ( $q ) = @_;
.  print "Please enter $q:  ";
.  chomp ( $res = <STDIN> );
.  return $res }
$person = ask ( "the name of a boy" );
$color = ask ( "a color" );
$number = ask ( "a number" );
$food = ask ( "a food" );
$day = ask ( "a day of the week" );
print "A story for you:

$person has a Surprising Day

On $day morning, $person woke up.
He was hungry.  He decided to eat $food.
He was so hungry he probably ate $number of them.
Afterward he was feeling a little $color.
His mom gave him some medicine and he felt better.

";
```

147: Factor Counter.

```
for ( $n = 1; $n <= 100; $n++ ) {
.  $ct = 0; for ( $f = 1; $f <= $n; $f++ ) {
.  .  if ( $n % $f == 0 ) { $ct++ }
.  print "$n $ct," } }
```

148: Triangle Up.

```
$row = 1;
while ( $row <= 9 ) {
.  $col = 1;
.  while ( $col <= $row ) {
.  .  print $col; $col = $col + 1; }
.  print "\n";
.  $row = $row + 1; }
```

149: Triangle Down.

```
for ( $row = 9; $row >= 0; $row-- ) {
.  for ( $col = $row; $col >= 0; $col-- ) {
.  .  print $col }
.  print "\n" }
```

150: Times Tables.

```
print "table max:  "; $max = <STDIN>;
for ( $row = 1; $row <= $max; $row++ ) {
.  for ( $col = 1; $col <= $max; $col++ ) {
.  .  printf ".  print "\n" }
```

Appendix B

Formatted Printing: printf

`printf` is the C language function to do formatted printing. The same function is also available in PERL. This appendix explains how `printf` works, and how to design the proper formatting specification for any occasion.

B.1 Background

In the early days, computer programmers would write their own subroutines to read in and print out numbers. It is not terribly difficult, actually. Just allocate a character array to hold the result, divide the number by ten, keep the remainder, add x30 to it, and store it at the end of the array. Repeat the process until all the digits are found. Then print it. Too easy, right?

But even though it was easy (for Einstein), it still took some effort. And what about error checking, and negative numbers? So the computer programmers brought forth libraries of prerecorded functions. And it was good. Eventually the most popular of these functions were canonized into membership in the "standard" libraries. Number printing was popular enough to gain this hallowed honor.

This meant that programmers did not have to reinvent the number-printing subroutine again and again. It also meant that everybody's favorite options tried to make it into the standard.

Thus was `printf` born.

B.2 Simple Printing

In the most simple case, `printf` takes one argument: a string of characters to be printed. This string is composed of characters, each of which is printed exactly as it appears. So `printf("xyz");` would simply print an x, then a y, and finally a z. This is not exactly "formatted" printing, but it is still the basis of what `printf` does.

B.2.1 Naturally Special Characters

To identify the start of the string, we put a double-quote (") at the front. To identify the end of the string we put another double-quote at the end. But what if we want to actually print a double-quote? We can't exactly put a double-quote in the middle of the string because it would be mistaken for the end-of-string marker. Double-quote is a special character. The normal print-what-you-see rules do not apply.

Different languages take different approaches to this problem. Some require the special character to be entered twice. C uses backslash (virgule, \) as an escape character to change the meaning of the next character after it. Thus, to print a double-quote you type in backslash double-quote. To print a backslash, you must escape it by typing another backslash in front of it. The first backslash means "give the next character its alternate meaning." The second backslash has an alternate meaning of "print a backslash."

Without a backslash, special characters have a natural special meaning. With a backslash they print as they appear. Here is a partial list.

\	escape the next character
\\	print a backslash
"	start or end of string
\"	print a double quote
'	start or end a character constant
\'	print a single quote
%	start a format specification
\%	print a percent sign

B.2.2 Alternately Special Characters

On the other hand we have characters that normally print as you would expect, but when you add a backslash, then they become special. An example is the newline character. To print an `n`, we simply type in an `n`. To print a newline, we type in a `\n`, thus invoking the alternate meaning of `n`, which is newline. Here is a partial list.

`\a`	audible alert (bell)
`\b`	backspace
`\f`	form feed
`\n`	newline (linefeed)
`\r`	carriage return
`\t`	tab
`\v`	vertical tab

B.3 Format Specifications

The real power of `printf` is when we are printing the contents of variables. Let's take the format specifier `%d` for example. This prints a number. So, a number must be provided for printing. This is done by adding another argument to the `printf` statement, as shown here.

```
int age;
age = 25;
printf ( "I am %d years old\n", age );
```

In this example, `printf` has two arguments. The first is a string: `"I am %d years old\n"`. The second is an integer, `age`.

B.3.1 The Argument List

When `printf` processes its arguments, it starts printing the characters it finds in the first argument, one by one. When it finds a percent it knows it has a format specification. It goes to the next argument and uses its value, printing it according to that format specification. It then returns to printing a character at a time (from the first argument).

It is okay to include more than one format specification in the `printf` string. In that case, the first format specification goes with the first additional argument, second goes with second, and so forth. Here is an example:

```
int x = 5, y = 10;
printf ( "x is %d and y is %d\n", x, y );
```

B.3.2 Percent

Every format specification starts with a percent sign and ends with a letter. The letters are chosen to have some mnemonic meaning. Here is a partial list:

`%c`	print a single **c**haracter
`%d`	print a **d**ecimal (base 10) number
`%e`	print an **e**xponential floating-point number
`%f`	print a **f**loating-point number
`%g`	print a **g**eneral-format floating-point number
`%i`	print an **i**nteger in base 10
`%o`	print a number in **o**ctal (base 8)
`%s`	print a **s**tring of characters
`%u`	print an **u**nsigned decimal (base 10) number
`%x`	print a number in he**x**idecimal (base 16)
`%%`	print a percent sign (`\%` also works)

To print a number in the simple way, the format specifier is simply `%d`. Here are some sample cases and results.

`printf`	produces
`("%d",0)`	`0`
`("%d",-7)`	`-7`
`("%d",1560133635)`	`1560133635`
`("%d",-2035065302)`	`-2035065302`

Notice that in the simple, `%d` way, there is no pre-determined size for the result. `printf` simply takes as much space as it needs.

B.3.3 The Width Option

As I mentioned above, simply printing numbers was not enough. There were special options that were desired. The most important was probably the width option. By saying `%5d` the number was guaranteed to take up five spaces (more if necessary, never less). This was very useful in printing tables because small and large numbers both took the same amount of space. Nearly all printing was monospaced in those days, which means that a `w` and an `i` both took the same amount of space. This is still common in text editors used by programmers.

To print a number with a certain (minimum) width, say 5 spaces wide, the format specifier is `%5d`. Here are some sample cases and results. (We will use the ␣ symbol to explicitly indicate a space.)

printf	produces
("%5d",0)	␣␣␣␣0
("%5d",-7)	␣␣␣-7
("%5d",1560133635)	1560133635
("%5d",-2035065302)	-2035065302

Notice that for shorter numbers, the result is padded out with leading spaces. For excessively long numbers there is no padding, and the full number is printed.

In normal usage, one would make the field wide enough for the biggest number one would ever expect. If your numbers are usually one, two, or three digits long, then `%3d` is probably adequate. In abnormal usage, one could end up printing a number that is too big for the field. `printf` makes the decision to print such numbers fully, even though they take too much space. This is because it is better to print the right answer and look ugly than to print the wrong answer and look pretty.

B.3.4 Filling the Extra Space

When printing a small number like `27` in a `%5d` field, the question then became where to put the `27` and what to put in the other three slots. It could be printed in the first two spaces, the last two spaces, or maybe the middle two spaces (if that can be determined). The empty spaces could be filled with the blank character, or perhaps stars (`***27` or `27***` or `**27*`), or dollar signs (`$$$27`), or equal signs (`===27`), or leading zeros (like `00027`).

These extra characters are often called "check protection" characters because the are intended to prevent bad guys from changing the dollar amount on a printed check. It is relatively easy to change a space into something else. It is harder to change a star, a dollar sign, or an equal sign.

`printf` provides space fill (left or right) and zero fill (left only). If you want check protection or centering you need to make other arrangements. But even without check protection or centering `printf` still has an impressive (and bewildering) collection of options.

B.3.5 The Justify Option

Using `printf` numbers can be left-justified (printed in the left side of the field) or right-justified (printed in the right side of the field). The most natural way to print numbers seems to be right-justified with leading spaces. That is what `%5d` means: print a base-10 number in a field of width 5, with the number right-aligned and front-filled with spaces.

To make the number left-aligned, a minus sign is added to the format specifier. To print a number 5 spaces wide and left-justified (left-aligned) the format specifier is `%-5d`. Here are some sample cases and results.

`printf`	produces
`("%-5d",0)`	`0␣␣␣␣`
`("%-5d",-7)`	`-7␣␣␣`
`("%-5d",1560133635)`	`1560133635`
`("%-5d",-2035065302)`	`-2035065302`

As before, for shorter numbers, the result is padded out with spaces. For longer numbers there is no padding, and the number is not shortened.

B.3.6 The Zero-Fill Option

To make things line up nice and pretty, it is common to write a date using leading zeros. We can write May 5, 2003 in the US as 05/05/2003. We could also write it as 2003.05.05. Notice that in both cases, the leading zeros do not change the meaning. They just make it line up nicely in lists.

When a number is zero-filled, the zeros always go in front, and the resulting number is both left- and right-justified. In this case the minus sign has no

effect. To print a zero-filled number 5 spaces wide the format specifier is `%05d`. Here are some sample cases and results.

printf	produces
("%05d",0)	00000
("%05d",-7)	-0007
("%05d",1560133635)	1560133635
("%05d",-2035065302)	-2035065302

Shorter numbers are padded out with leading zeros. Longer numbers are unchanged.

B.3.7 Fun With Plus Signs

Negative numbers always print with a minus sign. Positive numbers and zero usually do not print with a sign, but you can request one. A plus (+) in the format specifier makes that request.

To print a signed number 5 spaces wide the format specifier is `%+5d`. Here are some sample cases and results.

printf	produces
("%+5d",0)	␣␣␣+0
("%+5d",-7)	␣␣␣-7
("%+5d",1560133635)	+1560133635
("%+5d",-2035065302)	-2035065302

Notice that zero is treated as a positive number. Shorter numbers are padded. Longer numbers are unchanged.

Plus and minus are not related. Both can appear in a format specifier.

B.3.8 The Invisible Plus Sign

This one is a little bizarre. It is an invisible plus sign. Instead of printing a plus on positive numbers (and zero), we print a space where the sign would go. This can be useful in printing left-justified numbers where you want the minus signs to really stand out. Notice these two alternatives.

printf	produces
("%+-5d",0)	+0␣␣␣␣
("%+-5d",-7)	-7␣␣␣␣
("%+-5d",1560133635)	+1560133635
("%+-5d",-2035065302)	-2035065302

printf	produces
("%␣-5d",0)	␣0␣␣␣␣
("%␣-5d",-7)	-7␣␣␣␣
("%␣-5d",1560133635)	␣1560133635
("%␣-5d",-2035065302)	-2035065302

Remember from above that the format specifier %-5d we get the following results (shown again for easier comparison).

printf	produces
("%-5d",0)	0␣␣␣␣␣
("%-5d",-7)	-7␣␣␣␣
("%-5d",1560133635)	1560133635
("%-5d",-2035065302)	-2035065302

Notice that the plus signs disappear, but the sign still takes up space at the front of the number.

Notice also that we can combine several options in the same format specifier. In this case, we have combined the options plus, minus, and five, or space, minus, and five, or just minus and five.

B.3.9 Plus, Space, and Zero

Here is another example of combining several options at the same time.

Using the format specifier %␣05d or %0␣5d we get the following results.

printf	produces
("%␣05d",0)	␣0000
("%␣05d",-7)	-0007
("%␣05d",1560133635)	␣1560133635
("%␣05d",-2035065302)	-2035065302

Using the format specifier `%+05d` or `%0+5d` we get the following results.

printf	produces
`("%+05d",0)`	`+0000`
`("%+05d",-7)`	`-0007`
`("%+05d",1560133635)`	`+1560133635`
`("%+05d",-2035065302)`	`-2035065302`

When we combine plus and space at the same time, the space arranges for room for a sign and the plus uses it. It is the same as if the space was not even specified. The plus takes priority over the space.

B.3.10 Summary

The options are also called "flags" and among themselves they can appear in any order. Here is a partial list.

flag	effect
none	print normally (right justify, space fill)
`-`	left justify
`0`	leading zero fill
`+`	print plus on positive numbers
␣	invisible plus sign

After the options, if any, the minimum field width can be specified.

B.4 Printing Strings

The `%s` option allows us to print a string inside a string. Here is an example.

```
char * grade;
if ( year == 11 ) grade = "junior";
printf ( "%s is a %s\n", "Fred", grade );
```

The left-justify flag applies to strings, but of course the zero fill, plus sign, and invisible plus sign are meaningless.

printf	produces
("%5s","")	␣␣␣␣␣
("%5s","a")	␣␣␣␣a
("%5s","ab")	␣␣␣ab
("%5s","abcdefg")	abcdefg

printf	produces
("%-5s","")	␣␣␣␣␣
("%-5s","a")	a␣␣␣␣
("%-5s","ab")	ab␣␣␣
("%-5s","abcdefg")	abcdefg

B.5 Floating Point

Floating point numbers are those like `3.1415` that have a decimal point someplace inside. This is in contrast to ordinary integers like `27` that have no decimal point.

All the same flags and rules apply for floating point numbers as do for integers, but we have a few new options. The most important is a way to specify how many digits appear after the decimal point. This number is called the **precision** of the number.

For ordinary commerce, prices are often mentioned as whole dollars or as dollars and cents (zero or two digits of precision). For gasoline, prices are mentioned as dollars, cents, and tenths of a cent (three digits of precision). Here are some examples of how to print these kinds of numbers. Let `e=2.718281828`.

printf	produces
("%.0f",e)	3
("%.0f.",e)	3.
("%.1f",e)	2.7
("%.2f",e)	2.72
("%.6f",e)	2.718282
("%f",e)	2.718282
("%.7f",e)	2.7182818

Notice that if a dot and a number are specified, the number (the precision) indicates how many places should be shown after the decimal point.

Notice that if no dot and precision are specified for `%f`, the default is `%.6f` (six digits after the decimal point).

Notice that if a precision of zero is specified, the decimal point also disappears. If you want it back, you must list it separately (after the `%f` format specifier).

We can specify both a width and a precision at the same time. Notice especially that `5.2` means a total width of five, with two digits after the decimal. **It is very common and natural to think it means five digits in front of the decimal and two digits after, but that is not correct.** Be careful.

printf	produces
("%5.0f",e)	␣␣␣␣3
("%5.0f.",e)	␣␣␣␣3.
("%5.1f",e)	␣␣2.7
("%5.2f",e)	␣2.72
("%5.7f",e)	2.7182818

We can also combine precision with the flags we learned before, to specify left justification, leading zeros, plus signs, etc.

printf	produces
("%5.1f",e)	␣␣2.7
("%-5.1f",e)	2.7␣␣
("%+5.1f",e)	␣+2.7
("%+-5.1f",e)	+2.7␣
("%05.1f",e)	002.7
("%+05.1f",e)	+02.7
("%␣05.1f",e)	␣02.7
("%-␣5.1f",e)	␣2.7␣

B.6 Designing The Perfect Spec

If you are designing a `printf` format specifier, the first step is to decide what kind of a thing you are printing. If it is an integer, a float, a string, or a character, you will make different choices about which basic format to use.

The second big question is how wide your field should be. Usually this will be the size of the biggest number you ever expect to print under normal circumstances. Sometimes this is controlled by the amount of space that is provided on a pre-printed form (such as a check or an invoice).

Decide what you would like printed under a variety of circumstances. In this appendix we have often illustrated the results by using a small positive number, a small negative number, an oversized positive number, and an oversized negative number. You should include this options as well as large (but not oversized) numbers. Design your format for the biggest number that you would normally expect to see.

B.7 Conclusion

The `printf` function is a powerful device for printing numbers and other things stored in variables. With this power there is a certain amount of complexity. Taken all at once, the complexity makes `printf` seem almost impossible to understand. But the complexity can be easily unfolded into simple features, including width, precision, signage, justification, and fill. By recognizing and understanding these features, `printf` will become a useful and friendly servant in your printing endeavors.

Appendix C

File I/O

Up to this point we have talked about receiving our inputs from the keyboard and writing our outputs to the screen.

Computer programming takes a more general view of input and output. Specifically, you can prepare your inputs in advance and store them in a file. Similarly, you can create outputs and write them to a file.

More information is available by searching the web. Here we present just a minimal walk-through of the possibilities.

C.1 Redirection

Working from the command line we can generally reassign the inputs and outputs by using the `>` and `<` symbols. Here is an example using our `Hello` program.

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

First we run the program directly from the command line by typing `./Hello`. The output appears immediately.

```
./Hello
Hello, World!
```

Next we run the program directly from the command line while redirecting output.

```
./Hello > asdf
```

Notice that the output does not appear as before. Instead a file `asdf` is created and has as its contents the `Hello, World!` that we saw before.

`cat` is the command for typing out the contents of a file. (The word `cat` is short for concatenate.)

```
cat < asdf
Hello, World!
```

We can use `cat` to show us our program as well.

```
cat < Hello
print "Hello, World!\n";
```

While redirection is going on, our program does not need to do anything special to use files. The special things are all being handled outside of our program. In fact, it is difficult (but not impossible) to even find out that input and/or output has been redirected.

C.2 Explicit Reading

It is also possible to explicitly open a file for reading at the same time that normal input comes from the keyboard.

You will remember that normal input comes from `<STDIN>`, or "standard input." We can open another "channel" with a different name and get input from a file.

```
open IN, "asdf";
chomp ( $line = <IN> );
close IN;
```

We can read repeatedly to bring in more than one line of data.

```
open IN, "asdf";
while ( $line = <IN> ) { print "$line" }
close IN;
```

The word `asdf` is a file name. It could be any name allowed by the operating system.

The word `IN` is called a file handle. By convention, file handles are written in CAPITAL letters, but that is not required. In fact, the word `IN` is not special in any way. We could have called it `FILE1` or something else.

C.3 Explicit Writing

```
open FILE2, ">asdf2";
for ( $i = 0; $i < 100; $i++ ) { print FILE2 "$i\n" }
close FILE2;
```

You can use `cat` to check the contents of `asdf2` to see that it contains the numbers from zero to 99.

The word `asdf2` is the name of the file being written. The `>` sign in front of the file name is Perl's way of saying that we are writing to the file instead of reading it. Perl uses the same symbol that is used on the command line for redirecting output.

The word `FILE2` is not special in any way. We could have called it `MY_OUTPUT` or something else.

Normally when you write to a file any prior contents are erased. It is possible to open a file in "append" mode, so that that new contents are added to the end of the existing file. This could be useful if the file is a log file, for example, used for making note of things that have happened. Instead of using one `>` symbol when opening the file, we use two of them. This also works on the command line.

```
open FILE2, ">>asdf2"; # append
```

C.4 Close is Optional

Each file you open uses a certain amount of your computer's resources while the file is open. Normally when you are done with the file handle, you should close it to release the resources.

When your program ends its execution, resources that were in use are automatically released. Therefore, if you forget to close the file, it will be closed for you by the operating system when your program ends.

If your program crashes, however, the operating system may not have the chance to properly and methodically close your file. In a case like that, some of the output you sent may not actually be recorded.

Even though closing the file is somewhat optional, it is a good habit to make sure you do it at the right time.

C.5 Multiple Files

It is possible to open several files for reading and several other files for writing. Naturally each one should have its own unique name.

Appendix D

Patterns

In chapter 9 (page 64) we talked about inventing formulas to explain things to the computer.

In chapter 26 (page 134) we talked about "1, 2, 3, ..., 100". The three dots mean: (a) there is a pattern, and (b) it should be obvious.

In this chapter we look at a few patterns and try to explicitly say what the pattern is. We look at one specific strategy for finding the pattern.

D.1 The Difference Method

The best way to predict the future is to say it will be the same as the past has been. Usually you will be right. Tomorrow's weather will be like it was today, usually. Or, "it is crazy to do the same thing as before but to expect different results."

By definition, a pattern repeats. Generally it repeats with some small variation from one time to the next.

In analyzing to find a pattern, the first goal is to get an idea of what exactly is repeating. We use this information to account for how things are. Then we move forward to see what remains to be explained.

D.2 Constant

```
1, 1, 1, 1, ...
```

Pretty consistent. This is just a bunch of ones dropping off the assembly line. Nothing is changing. Constant. What comes next? Another 1? Boring.

D.3 Odd

```
1, 3, 5, 7, ...
```

Slightly more interesting. These are the odd numbers. Let's focus on what is the same and what is different, number to number.

The same: each thing is a number. Each thing is a whole number (integer). Each thing is a positive number. The numbers are getting bigger.

Different: the numbers are different.

Let's try to see how different they are. Using the "today's weather" theory, can we predict the next number if we know the current number?

From 1, the next number is 3. The difference is 2.

From 3, the next number is 5. The difference is 2.

From 5, the next number is 7. The difference is 2.

2, 2, 2, ... We found a pattern in the differences. This seems to account for everything. We can assume that the differences will continue to follow this pattern. Therefore,

From 7 the difference is 2 so the next number is 9.

From 9 the difference is 2 so the next number is 11.

More generally, from `$x` the next number is `$x+2`.

D.4 Squares

```
1, 4, 9, 16, 25, ...
```

You may have recognized that these are the first five square numbers (one times one, two times two, ..., five times five). Pretend that it was not so obvious. Let's analyze it the way we did above.

Same: they are numbers. They are whole numbers. They are positive. They are increasing.

Let's look at the differences.

From 1 the next number is 4. The difference is 3.

From 4 the next number is 9. The difference is 5.

From 9 the next number is 16. The difference is 7.

From 16 the next number is 25. The difference is 9.

The differences seem to have a consistent pattern:

```
3, 5, 7, 9, ...
```

In fact, it is the pattern we saw earlier. And we can predict the future. The next difference should be 11.

From 25, the difference is 11, so the next number is 36.

From 36, the difference is 13, so the next number is 49.

D.5 Fibonacci

```
1, 1, 2, 3, 5, 8, 13, ...
```

This is the Fibonacci sequence (pronounced fib oh NOT chee), credited to a famous Italian guy named Leonardo Pisano Bigollo (1180-1250). Apparently his father was named Bonacci, and fibonacci means son of Bonacci. See Wikipedia if you want to know more.

Analysis: they are numbers. They are whole numbers. They are positive. They are increasing.

Differences: 0, 1, 1, 2, 3, 5.

Does it look familiar? The set of differences is basically just the same sequence as before.

Can we predict the future? Next after 13 should be 13+8.

People actually use this sequence in playing the stock market and in gambling. See Wikipedia.

D.6 Triangles

Let's count the stars in a triangle.

```
    *  row 1
   * *  row 2
  * * *  row 3
 * * * *  row 4
* * * * *  row 5
```

The first row forms a really small triangle. It has one star.

The first two rows form a triangle. It has three stars.

The first three rows form a triangle. It has six stars.

The first four rows form a triangle. It has ten stars.

The first five rows form a triangle. It has fifteen stars.

```
1, 3, 6, 10, 15, ...
```

Can we tell how many stars will be in a 100-row triangle? Would you like to count it by hand? Gauss had a famous solution. (Johann Carl Friedrich Gauss, 1777-1855. See Wikipedia.)

Let's do the analysis. It is obvious from the description of the problem that each triangle is larger than the last, and that the difference is just one more than the last difference.

Let's take the differences.

```
3-1=2, 6-3=3, 10-6=4, 15-10=5, ...
2, 3, 4, 5, ...
```

What happened to 1? We have 2, 3, 4, 5, ..., which is an obvious pattern, but it does seem like there should have been a 1 on the front.

If we start with a zero-row triangle we can complete the series.

```
1-0=1, 3-1=2, 6-3=3, 10-6=4, 15-10=5, ...
1, 2, 3, 4, 5, ...
```

A 100-row triangle would be 100 stars bigger than a 99-row triangle. We can write it like this:

```
triangle(100) = triangle(99) + 100
```

We can generalize. Instead of being specific to the number 100, we can try to state the relationship no matter what the number is.

```
triangle(n) = triangle(n-1) + n
```

This is called a `recursive` definition. **Recursion** is defining something that is large in terms of the same sort of thing when it was smaller. We use recursive definitions all the time. Here is an example.

```
Like father, like son.
```

Here is a program to calculate the number of stars in a 100-row triangle.

```
$total = 0;
for ( $row = 1; $row <= 100; $row++ ) {
  $total += $row }
print $total;
```

This is an iterative solution. Iterative means that we are using a loop. Each time we pass through the loop we are updating `$total` which holds our running total. When we have processed 100 rows we should have our answer.

How can we be sure it is working? Let's print out the intervening answers? This should give us the answers from one to 1000.

```
$total = 0;
for ( $row = 1; $row <= 1000; $row++ ) {
  $total += $row;
  print "The $row-row triangle has $total stars.\n" }
```

We can verify that the first few answers are correct. Then, based on that we can guess that the rest are correct. (Well begun is half done.)

Appendix E

Random Numbers

Games and business simulations both benefit greatly from the use of random numbers. What exactly are they?

A random number is a number that you cannot predict exactly. Like flipping a coin could yield heads or tails, picking a random number could yield something to help you make a choice.

In Perl, random numbers are available from the `rand` function. It returns a uniformly distributed number between zero and one (but never equal to one, and probably never actually equal to zero).

By uniformly distributed, we mean that on average, ten percent of the time the number will be between 0 and 0.1. Ten percent of the time the number will be between 0.8 and 0.9.

E.1 Roll the Dice

Often we want a whole number (an integer). For example, we may want two numbers between 1 and 6 to represent the roll of two dice. To get one of 1, 2, 3, 4, 5, and 6, with equal probability, we can use the following formula.

```
$roll = 1 + int ( rand() * 6 );
```

In this example, `rand` returns a number between zero and one. Multiplying it by six gives us a number between zero and six. Taking the `int` part throws away any digits after the decimal. Thus, 5.31376 gets turned into 5.

This gives us six equally likely values: 0, 1, 2, 3, 4, and 5. By adding 1 the numbers are now in the desired range.

E.2 Pick a Card

Let's say we want to randomly pick an item from a list. In this example, `@cards` is an array that contains five cards. We are using `9H` to stand for the nine of hearts, `TS` to stand for the ten of spades, etc. The first line initializes the array to contain five cards. The second line picks one of the cards. `@cards` is treated as the number of cards (5). `rand` picks a number from 0.01 to 4.99 (more or less). `int` converts the number to 0, 1, 2, 3, or 4 with equal probability. `$cards(number)` identifies a specific card. `$cards[0]` is `9H`, `$cards[1]` is `3C`, etc.

```
  @cards = ( "9H", "3C", "KD", "JD", "TS" );
  $pick = $cards[int(rand(@cards))];
```

If we want a random card from a full, standard 52-card deck, we can do it like this:

```
sub PickACard {
  my @suits = split "", "SHDC"; # Spades, Hearts, etc
  my @values = split "", "A23456789TJQK"; # Ace through King
  my $suit = $suits[int(rand(@suits))];
  my $value = $values[int(rand(@values))];
  return "$value$suit" }
```

E.3 Flip a Coin

If we have a standard subroutine to randomly pick an item from a list, we can use that to flip a coin or do many other things.

```
sub pick { return $_[int(rand(@_))] }
```

What will this subroutine do, exactly? First, `@_` contains the alternatives from which we will select. `rand(@_)` gives us a number between zero and

the number of alternatives, not inclusive. `int` turns it into a whole number between zero and N-1. `$_[]` gives us the item in that slot. `return` sends it back.

So, how to flip a coin? Easy.

```
$coin = pick ( "heads", "tails" );
$rps = pick ( "rock", "paper", "scissors" );
```

E.4 Normal Distribution*

The `rand` function returns a number that is uniformly distributed between zero and one (or some other endpoint). Many things in nature follow a distribution that is shaped like a bell curve. We can simulate that by adding six uniform numbers together. We could use something like that to describe a random person.

```
# pretend males average 178 cm tall, std dev 2cm
$height = randND ( 178, 2 );
```

It calls `randND` a subroutine I wrote that you can use to generate normally distributed random numbers with a specified mean and standard deviation. If you really care about your random numbers, you may want to look on the Web for something more accurate, but this is not bad.

```
sub randND { # rand normally distributed
  my ( $mean, $stdev ) = @_;
  # add up six uniformly distributed random numbers (-1 to 1)
  # the result has a mean of 0.0 and stdev of sqrt(2)
  my $r = rand(2)+rand(2)+rand(2)+rand(2)+rand(2)+rand(2)-6;
  $r *= $stdev / sqrt(2); # scale to desired stdev
  $r += $mean; # scale to desired mean
  return $r }
```

To verify the accuracy of the distribution, you may want to generate a bunch of them and then calculate their mean and deviation. Here is a subroutine I wrote that you can use to calculate the mean and standard deviation of an array of numbers.

```
sub stdev {
  my ( $sum, $N, $n, $avg, $SS );
  $sum = 0; $N = @_; $SS = 0;
  foreach $n (@_) { $sum += $n }
  $avg = $sum / $N;
  foreach $n (@_) { $SS += ($n - $avg) * ($n - $avg) }
  $stdev = sqrt ( $SS / ($N - 1 ));
  return ( $avg, $stdev ) }
```

E.5 What is Random Good For?

Two major examples come to mind. In the first example, the computer is playing a game against a human, or is providing a setting in which a game is played. Of course the game would become quite boring if the computer's actions were easily predicted. We would like to have the ability to say: half of the time turn left, and the other half of the time turn right. Or: roll a dice, and one sixth of the time the result should be a 1.

The second example is more complex. Monte Carlo simulation is a business tool that relies on random numbers. In a typical scenario, a business may say: we see a 30% chance the economy will go down, and a 30% chance the economy will go up; we see a 40% chance that our competitors will respond to our price cuts. If the economy goes down and our competitors do not respond, this will be the result. (Et cetera.) With Monte Carlo, the computer randomly tries all possibilities and averages out the results so the business can know whether they should drop their prices or not.

The second example is to show that random numbers are not just for games. But games are easier to work with, so that is what we will use here.

E.6 Truly Random or Not?

Traditionally random numbers have been generated by doing a calculation. Say for instance that we start with some number x. Take the square root. Throw away the first two digits. Keep the next four. That becomes your new random number. Repeat the process for another random number.

```
1234 -> 35.128336
```

```
1283 -> 35.818989
8189 -> 90.493093
4930 -> 70.213958
2139 -> 49.249324
2493 -> ...
```

The numbers 1234, 1283, 8189, 4930, 2139, 2493, ... look pretty random. Because we know how they were derived, we know they are not random. But they still look pretty random, and could be used to turn left or right randomly: If the number is even, turn left, odd turn right. 1234 is even, so we go left. 1283 is odd, so we go right. Left, Right, Right, Left, Right, Right.

There are many algorithms for generating such random numbers. When random numbers are generated in this manner, they are called PSEUDO random. PSEUDO is latin for "fake."

Truly random numbers can be generated by some physical phenomena such as radioactive decay as measured by a Geiger counter. In the INTEL pentium chip, there is a true random number generator based on the random properties of thermal behavior of the CPU chip. Or something like that.

E.7 srand: Seeding the Sequence

Perl and most other languages with random number generators provide a way that you can get the same sequence of random numbers over and over again. This can be very valuable for testing.

The starting parameters for the sequence are called the **seed**.

The procedure for initializing the **seed** is called **srand**.

Appendix F

The True Meaning of Solution

Does this problem have a solution? The words **solve** and **solution** have roots that mean "to loosen" or disolve. Buried in this word is the concept that a problem can be broken or disassembled into smaller parts, each of which will be easier to answer separately.

Another interesting word is **tractable** whose opposite is **intractable**. Tract comes from trahere which is to draw or pull. To draw a piece of artwork is to pull a pencil or brush. A tractor pulls a plow or other piece of equipment on the farm.

A problem that is tractable can be pulled apart into more simple pieces. A problem that is intractable still may have an answer, but not a solution.

An answer but not a solution?

We are playing with definitions here. To solve as commonly used means to find an answer. Commonly any answer is called a solution. Commonly when something is intractable it has no answer. But literally there is a subtle difference that we will examine in this chapter.

The difference is whether we simplify or not.

F.1 Boys, Girls, and Dogs

Time for a story. Consider this sentence: "The boy cried."

Hopefully its meaning is pretty obvious. Some specific boy engaged in the act of crying. We may want to know more about this story, as boys do not normally cry.

More data: "The girl hit the boy."

Now we may understand why the boy cried. I might cry too.

We can combine these two sentences. Combining makes things more complicated, but we do it all the time. Aside from making things more complex, it frequently offers some economy or reduction in the total cost of the item. That is, buying a car one part at a time is probably more expensive than buying a whole car all at once.

Consider this sentence: "The boy the girl hit cried."

One sentence is **nested** in the middle of the other. We have combined the two previous sentences and decreased the total word count from eight to six. That's a twenty-five percent improvement in economy. More miles per gallon. More meaning per word. The improvement came by using "the boy" only once instead of twice.

Is the sentence still comprehensible? Is it still clear? Most people would say that it is.

Enter the dog. "The dog bit the girl." Ouch.

We could say, "The girl the dog bit hit the boy." Complexity is rising but we can still pull it apart at full speed. The story is still clear. Let's complete the story, a tragedy in one line.

"The boy the girl the dog bit hit cried."

At this point, most people have suddenly exceeded their ability to take in this sentence at a run. The brain cannot take it. They grind to a halt and puzzle their way through it. Maybe they give up. Although it is perfectly good English, it is just a bit much to chew on. The words are **nested** three levels deep.

Much better to take this sentence apart, reduce the complexity, and tell the whole story using more words.

"The dog bit the girl.
The girl hit the boy.
The boy cried."

Now that's a version that we can take on the fly, chew it as it comes, and

still be ready for more.

F.2 Complexity

Like **tractable** and **solution** the word **complexity** is fun. Literally it means to be folded or braided together. The word "ply" is fold. The same root is plectere, meaning to plait or braid. It is found in **simplicity**, literally meaning "without braids or tangles." It is found in "plywood" and "pliers" and "pliable."

It is found in **implicit** and **explicit**. When something is implicit, the meaning is folded or braided in. Hidden, but still present. When something is explicit, the meaning is folded out for all to see.

F.3 Statements are Sentences

Each statement in a program is like a sentence in a story. The semi-colon at the end of the line indicates the end of the statement. The period at the end of a line indicates the end of the sentence.

Sentences can be **nested** inside one another, sometimes several levels deep. Just like that, computer program statements can be nested inside each other, any number of levels deep. Usually this nesting happens in the middle of an `if` statement or a `while` loop. It is not always easy to simplify this complexity.

Fortunately for us, the computer will understand nested statements just fine.

Unfortunately for us, the programmers who maintain the program may get lost, just as in "the boy the girl the dog bit hit cried."

Appendix G

Binary Numbers

Several number bases are common in the Information Technology field, including networking. Binary (base two) is the underlying model for most data communications. Octal (base eight) and hexadecimal (base 16, also called "hex") are popular shorthands for presenting binary data. Decimal (base ten) is the number base we most commonly use. It is useful and occasionally important to be able to convert numbers from one base to another.

G.1 Bitwise Operators

`and`, `&&`, `or`, `||`, and `xor` were discussed logical operators. They are also available in slightly different form as bitwise operators.

Logical treats the number as a whole as being either True or False. Bitwise treats each bit in the number individually as being True or False. This is most often used in equipment control but also shows up in computer graphics, turning on and off pixels on the screen.

Bitwise operation views each number as a collection of binary bits. The number `5`, for instance, would be seen as `101`.

The bitwise operators `&`, `|`, and `^` are used to calculate the and, or, and xor of the bits of binary numbers.

expression	value
`5 & 9`	`1`
`5 | 9`	`13`
`5 ^ 9`	`12`

Stating it in binary, it may be more obvious.

expression	value
`0101 & 1001`	`0001`
`0101 | 1001`	`1101`
`0101 ^ 1001`	`1100`

G.2 What is Binary?

In base ten, the ordinary numbers we use are composed of the standard ten digits: 1, 2, 3, 4, 5, 6,7 ,8, 9, and 0. The fact that there are ten different digits is what makes it base ten.

When we count in base 10, we use single-digit numbers up to 9. Then we use all the two-digit numbers: 10 to 19, 20 to 29, 30 to 39, ..., 90 to 99. For each first digit, we run through the entire set of second digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Notice that 0 to 9 is similar but not the same as 1 to 9, plus 0. Generally we count from 1, but from a strictly mathematical point of view, starting from 0 would make more sense.

After the two-digit numbers, we advance to three-digit numbers. 100 to 199, 200 to 299, ..., 900 to 999. The pattern is probably clear.

With binary numbers, we are limited to only two digits, zero and one. We count as follows: 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, and so on.

power	base 10	base 2
2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	1 0000
2^5	32	10 0000
2^6	64	100 0000
2^7	128	1000 0000
2^8	256	1 0000 0000
2^9	512	10 0000 0000
2^{10}	1024	100 0000 0000
2^{11}	2048	1000 0000 0000

G.3 Binary Shortcut

How do we know that 2^{16} is about 64 thousand? To quickly answer such questions it is helpful to know two facts: (a) the powers of two up to 10, and (b) the fact that 2^{10}=1024 is almost one thousand.

Exact Powers of Two				Approximate	
2^1 =	2	2^6 =	64	2^{10} =	thousand
2^2 =	4	2^7 =	128	2^{20} =	million
2^3 =	8	2^8 =	256	2^{30} =	billion
2^4 =	16	2^9 =	512	2^{40} =	trillion
2^5 =	32	2^{10} =	1024	2^{50} =	quadrillion

In the case of 2^{16} we know that 2^6 is 64, so 2^{16}=$2^6 \times 2^{10}$ is about 64 thousand and 2^{26}=$2^6 \times 2^{10} \times 2^{10}$ is about 64 million.

Similarly, 2^4 is 16, so 2^{14} is about 16 thousand and 2^{24} is about 16 million.

G.4 32 Bits

On the Internet, things are organized by bits. Take the following string of bits.

1011 1010 0100 1110 0101 0010 0000 0111

Generally we write IPv4 addresses in base 10. The first eight bits (1011 1010) are called the first octet. They translate to 186. The second eight bits (0100 1110) are called the second octet. They translate to 78. The third eight bits (0101 0010) are called the third octet. They translate to 82. The fourth eight bits (0000 0111) are called the fourth octet. They translate to 7. Generally we would write this address in "dotted-quad" notation, with each octet in base 10, like this.

186.78.82.7

G.5 Binary Numbers

The binary number system uses only two digits: zero and one. Often instead of calling them digits, they are called bits, which is short for bi(nary digi)ts. With this restriction, instead of counting 1, 2, 3, 4, 5, ... we count 1, 10, 11, 100, 101, ... It is a fully-functional numbering system, able to represent even numbers like π (3.1415) as 11.001 (with a bunch more digits). It takes about three times as many bits to express a number in binary as it does in base 10.

Data transmission on the Internet is generally thought of in terms of binary numbers because the underlying physical phenomena can be comfortably described in that way. Because the underlying nature is generally dealt with in binary, it is important for networking practitioners to also work in binary when appropriate.

G.5.1 Divide and Shift

Divide and Shift is one easy method to convert a number from base 10 to binary. Divide it by two repeatedly. Each time you divide, take the remainder and add it to the front of the binary number you are constructing. Here for example we will convert 2005 (the year this tutorial was written).

Taking 2005, we divide by two to get 1002.5. But let's keep it to whole numbers. Dividing we get 1002 with a remainder of 1. We copy the remainder to the front of the binary number we are building.

2005		our starting number
1002	1	divide and copy
501	01	divide and copy
250	101	continue
125	0101	continue
62	1 0101	continue
31	01 0101	continue
15	101 0101	continue
7	1101 0101	continue
3	1 1101 0101	continue
1	11 1101 0101	continue
0	111 1101 0101	continue
0	0111 1101 0101	okay, stop

We could keep putting zeroes at the front, but there is no point.

2005 in base 10 becomes 111,1101,0101 in base 2.

To convert a number from base 2 to base 10, do the same thing in reverse. Double the base 10 number and add the first bit of the base 2 number. We will covert our second octet from above.

0	0100 1110	our starting number
0	100 1110	double and add zero
1	00 1110	double and add one
2	0 1110	double and add zero
4	1110	double and add zero
9	110	double and add one
19	10	double and add one
39	0	double and add one
78		double and add zero

0100,1110 in base 2 becomes 78 in base 10.

G.5.2 Powers of Two

Powers of Two is another easy method to convert a number from base 10 to binary. We start with a table of powers of two. We subtract the largest power repeatedly. Then we add up the results. Again we will convert 2005.

2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	1 0000
2^5	32	10 0000
2^6	64	100 0000
2^7	128	1000 0000
2^8	256	1 0000 0000
2^9	512	10 0000 0000
2^{10}	1024	100 0000 0000
2^{11}	2048	1000 0000 0000

2005		our starting number
1024	100 0000 0000	largest power of two
981		after we subtract
512	10 0000 0000	largest power of two
469		after we subtract
256	1 0000 0000	largest power of two
213		after we subtract
128	1000 0000	largest power of two
85		after we subtract
64	100 0000	largest power of two
21		after we subtract
16	1 0000	largest power of two
5		after we subtract
4	100	largest power of two
1		after we subtract
1	1	largest power of two
0		done

1024	100 0000 0000
512	10 0000 0000
256	1 0000 0000
128	1000 0000
64	100 0000
16	1 0000
4	100
1	1
2005	111 1101 0101

Similarly we can add up the powers of two to get the answer in base 10.

78	0100 1110
2	10
4	100
8	1000
64	100 0000

$64 + 8 + 4 + 2 = 78$, our answer.

Students should be able to convert numbers using either method, and should be confident with at least one method.

G.5.3 Numbers to Memorize

To speed up calculations, it is helpful to memorize the sequence of powers of two, at least through 1024. When I say memorize the sequence, I don't mean that you remember that 64 is the sixth number, but rather that 64 comes after 32 and before 128. When we learn the alphabet, we often do not know that T is the 20th letter, but we know it is part of the sequence "Q R S T U V W."

It is also helpful to know the powers of two subtracted from 256, which are also the negative powers of two; a series of all ones followed by a series of all zeroes.

Negative Powers of Two

255	1111 1111	256-1
254	1111 1110	256-2
252	1111 1100	256-4
248	1111 1000	256-8
240	1111 0000	256-16
224	1110 0000	256-32
192	1100 0000	256-64
128	1000 0000	256-128

Index