Picking Up Perl

Picking Up Perl

A Tutorial Book for New Perl Programmers
0.12th Edition, Covering Perl Version `5.6.0`
Feedback and comments are always welcome. Send them to `pup-comments@ebb.org`
January 2001

Bradley M. Kuhn

Table of Contents

Preface

Purpose of this Book

This book has been created for a number of reasons. The primary reason is to provide a freely redistributable tutorial for the Perl language. In writing this freely redistributable tutorial, it is our hope that the largest number of people can have access to it and share it.

In the Perl community, we have discovered ways to save time by writing Perl programs that make our jobs and lives easier. Surely, Perl is not a panacea, but it has certainly made our lives a little bit better. It is hoped that you can use Perl to make your jobs and lives easier, too.

Acknowledgments

Thanks to all those who sent grammar fixes and feedback over the years. There are far too many of you to name here, but I have tried to list you all in the '`THANKS`' file that comes with the transparent copy of this book.

Obtaining the Most Recent Version

This book is still under development. The most recent version can be obtained at `http://www.ebb.org/PickingUpPerl`.

Audience

This book does not that assume any prior knowledge of Perl. However, a reader familiar with standard computer science concepts such as abstraction, stacks, queues, and hash tables will definitely find her[1] way through this book with ease. In other words, anyone with a knowledge equivalent to a first-year of college computer science courses should find this book very basic. Those of less experience may find this book challenging.

Material Covered

The material covered in this book is designed to prepare the reader to enter the world of Perl programming. This book covers the basic data and control structures of Perl, as well as the philosophies behind Perl programming. The native search patterns used in Perl, called regular expressions, are introduced and discussed.

These concepts are introduced through the use of examples. It is hoped that readers find these examples fun.

[1] Female pronouns are used throughout this book. It can be assumed that they refer to both genders.

Conventions Used in this Book

In this text, a variety of conventions are used to explain the material. Certain typographical and display elements are used for didactic purposes.

Any Perl code that is included directly in flowing text appears like this: `$x = 5`. Any operating system commands discussed in flowing text appear like this: `program`. Operating system files that are discussed directly in the flowing text appear like this: '`file`'. When a technical term of particular importance is first introduced and explained, it appears in emphasized text, like this: *an important term*.

When Perl code examples or operating system commands need to be separated away from the flowing text for emphasis, or because the code is long, it appears like this:

```
my $x = "foo";       # This is a Perl assignment
print $x, "\n";      # Print out "foo" and newline
```

All Perl code shown in this manner will be valid in Perl, version `5.6.0`. In most cases, you can paste code from one of these sections into a Perl program, and the code should work, even under `use strict` and `use warnings`.

Sometimes, it will be necessary to include code that is not valid Perl. In this case, a comment will appear right after the invalid statement indicating that it is not valid, like this:

```
$x = "foo;      # INVALID: a '"' character is missing
```

When code that we set aside forms an entire Perl program that is self-contained, and not simply a long example code section, it will appear like this:

```
#!/usr/bin/perl

use warnings;
use strict;

print "Hello World\n";
```

Finally, when text is given as possible output that might be given as error messages when `perl` is run, they will appear like this:

```
Semicolon seems to be missing
syntax error
```

Keep these standards in mind as you read this book.

1 Getting Started

This chapter begins the introduction to Perl by giving a simple code example. We do not expect the reader to understand this example completely (yet). We present this example to ease the reader into Perl's syntax and semantics.

1.1 A First Perl Program

So, to begin our study of Perl, let us consider a small Perl program. Do not worry that you are not familiar with all the syntax used here. The syntax will be introduced more formally as we continue on through this book. Just try to infer the behavior of the constructs below as best you can.

For our first Perl program, we will ask the user their username, and print out a message greeting the user by name.

```
#!/usr/bin/perl

use strict;                          # important pragma
use warnings;                        # another important pragma
print "What is your username?  ";    # print out the question
my $username;                        # “declare” the variable
$username = <STDIN>;                 # ask for the username
chomp($username);                    # remove “new line”
print "Hello, $username.\n";         # print out the greeting

# Now we have said hello to our user
```

Let us examine this program line by line to ascertain its meaning. Some hand-waving will be necessary, since some of the concepts will not be presented until later. However, this code is simple enough that you need not yet understand completely what each line is doing.

The first line is how the program is identified as a Perl program. All Perl programs should start with a line like `#!/`*path*`/perl`. Usually, it is just `#!/usr/bin/perl`. You should put this line at the top of each of your Perl programs.

In the lines that follow, halfway through each line, there is a ‘`#`’ character. Everything from the ‘`#`’ character until the end of the line is considered a *comment*. You are not required to comment each line. In fact, commenting each line is rare. However, you will find in this text that we frequently put comments on every line, since we are trying to explain to the reader exactly what each Perl statement is doing. When you write Perl programs, you should provide comments, but you need not do so as verbosely as we do in this text.

Note, too, that comments can also occur on lines by themselves. The last line of the program above is an example of that.

Now, consider the code itself, ignoring everything that follows a ‘`#`’ character. Notice that each line (ignoring comments) ends with a ‘`;`’. This is the way that you tell Perl that a *statement* is complete. We'll talk more about statements soon; for now, just consider a statement to be a single, logical command that you give to Perl.

The first line, `use strict`, is called a *pragma* in Perl. It is not something that "explicitly" gets executed, from your point of view as the programmer. Instead, a pragma specifies (or changes) the rules that Perl uses to understand the code that follows. The `use strict;` pragma enforces the strictest possible rules for compiling the code. You should always use this pragma while you are still new to Perl, as it will help you find the errors in your code more easily.

The second line is another pragma, `use warnings`. This pragma tells Perl that you'd like to be warned as much as possible when you write code that might be questionable. Certain features of Perl can confuse new (and sometimes even seasoned) Perl programmers. The `use warnings` pragma, like `use strict`, is a way to tell Perl that you'd like to be warned at run-time when certain operations seem questionable.

So, you might wonder why two separate pragmas are needed. The reason is that they are enforced by Perl at different times. The `use strict` pragma enforces compile-time constraints on the program source code. You can even test them without running the program by using `perl -c` *filename*, where *filename* is the file containing your program. That option does not run your program, it merely checks that they syntax of your program is correct. (To remember this, remember that the letter '`c`' in '`-c`' stands for "check the program".)

By contrast, the `use warnings` pragma controls run-time behavior. With `use warnings`, messages could be printed while your program runs, if Perl notices something wrong. In addition, different inputs to the program can cause different messages to be printed (or suppress such messages entirely).

The third line is the first statement of the program the performs an action directly. It is a call to Perl's built-in `print` function. In this case, it is taking a string (enclosed in double quotes) as its argument, and sending that string to the standard output, which is, by default, the terminal, window, or console from which the program is run.

The next line is a variable *declaration*. When in `strict` mode (set by the `use strict` pragma), all variables must be declared. In this case, Perl's `my` keyword is used to declare the variable `$username`. A variable like `$username` that starts with a `$` is said to be a *scalar* variable. For more information on scalar variables, see Chapter 2 [Working with Scalars], page 7. For now, just be aware that scalar variables can hold strings.

The next line, `$username = <STDIN>` is an assignment statement, which is denoted by the `=`. The left hand side of the assignment is that scalar variable, `$username`, that we declared in the line before it. Since `$username` is on the left hand side of the `=`, that indicates `$username` will be assigned a new value by this assignment statement.

The right hand side of the assignment is a construct that allows us to get input from the keyboard, the default standard input. `STDIN` is called a *file handle* that represents the standard input. We will discuss more about file handles later. For now, just remember that the construct `<STDIN>`, when assigned to a scalar variable, places the next line of standard input into that scalar variable.

Thus, at this point, we have the next line of the input (which is hopefully the username that we asked for), in the `$username` variable. Since we got the contents of `$username` from the standard input, we know that the user hit return after typing her username. The return key inserts a special character, called newline, at the end of the line. The `$username` variable contains the full contents of the line, which is not just the user's name, but also that newline character.

To take care of this, the next thing we do is `chomp($username)`. Perl's built-in function, `chomp`, removes any newline characters that are on the end of a variable. So, after the `chomp` operation, the variable `$username`

The final statement is another `print` statement. It uses the value of the `$username` variable to greet the user with her name. Note that it is acceptable to use `$username` inside of the string to be printed, and the contents of that scalar are included.

This ends our discussion of our small Perl program. Now that you have some idea of what Perl programs look like, we can begin to look at Perl, its data types, and its constructs in detail.

1.2 Expressions, Statements, and Side-Effects

Before we begin introduce more Perl code examples, we want to explain the ideas of an *expression* and a *statement*, and how each looks in Perl.

Any valid "chunk" of Perl code can be considered an *expression*. That expression always evaluates to some value. Sometimes, the value to which expression evaluates is of interest to us, and sometimes it is not. However, we always must be aware that each expression has some "value" that is the evaluation of that expression.

Zero or more expressions to make a *statement* in Perl. Statements in Perl end with a semi-colon. For example, in the Perl code we saw before, we turned the expression, `chomp($userName)`, into a statement, `chomp($userName);` by adding a `;` to the end. If it helps, you can think about the `;`s as separating sets of expressions that you want Perl to evaluate and execute in order.

Given that every expression, even when combined into statements, evaluate to some value, you might be tempted to ask: What does the expression `chomp($userName)` evaluate to? It turns out that expression evaluates to the total number of characters removed from the end of the variable `$userName`. This is actually one of those cases where we are not particularly interested in the evaluation result of the code. In this case, we were instead interested in what is called the *side-effect* of the expression.

The *side-effect* of an expression is some change that occurs as a result of that expression's evaluation. Often, a side-effect causes some change in the state of the running program, such as changing the value of a variable. In the expression `chomp($userName)`, the side-effect is that any newline characters are removed from the end of the variable, `$username`.

Let's now consider a slightly more complex statement, and look for the the expressions and side-effect. Consider the statement, `$username = <STDIN>;` from our first program. In this case, we used the expression, `<STDIN>` as part of a larger expression, namely `$username = <STDIN>`. The expression, `<STDIN>` evaluated to a scalar value, namely a string that represented a line from the standard input. It was of particular interest to us the value to which `<STDIN>` evaluated, because we wanted to save that value in the variable, `$username`.

To cause that assignment to take place, we used the larger expression, `$username = <STDIN>`. The side-effect of that larger expression is that `$username` contains the value that `<STDIN>` evaluated to. That side-effect is what we wanted in this case, and we ignore the value to which `$username = <STDIN>` evaluates. (It turns out that it evaluates to the value contained in `$username` after the assignment took place.)

The concepts of statements, expressions and side-effects will become more clear as we continue. When appropriate, we'll point out various expression and discuss what they evaluate to, and indicate what side-effects are of interest to us.

2 Working with Scalars

Scalar data are the most basic in Perl. Each scalar datum is logically a single entity. Scalars can be strings of characters or numbers. In Perl, you write literal scalar strings like this:

For example, the strings `"foobar"` and `'baz'` are scalar data. The numbers `3`, `3.5` and `-1` are also scalar data.

Strings are always enclosed in some sort of quoting, the most common of which are single quotes, `''`, and and double quotes, `""`. We'll talk later about how these differ, but for now, you can keep in mind that any string of characters inside either type of quotes are scalar string data.

Numbers are always written without quotes. Any numeric sequence without quotes are scalar number data.

In this chapter, we will take a look at the variety of scalar data available in Perl, the way to store them in variables, how to operate on them, and how to output them.

2.1 Strings

Any sequence of ASCII characters put together as one unit, is a string. So, the word `the` is a string. This sentence is a string. Even this entire paragraph is a string. In fact, you could consider the text of this entire book as one string.

Strings can be of any length and can contain any characters, numbers, punctuation, special characters (like '`!`', '`#`', and '`%`'), and even characters in natural languages besides English In addition, a string can contain special ASCII formatting characters like newline, tab, and the "bell" character. We will discuss special characters more later on. For now, we will begin our consideration of strings by considering how to insert literal strings into a Perl program.

To begin our discussion of strings in Perl, we will consider how to work with "string literals" in Perl. The word *literal* here refers to the fact that these are used when you want to type a string directly to Perl. This can be contrasted with storing a string in a *variable*.

Any string literal can be used as an expression. We will find this useful when we want to store string literals in variables. However, for now, we will simply consider the different types of string literals that one can make in Perl. Later, we will learn how to assign these string literals to variables (see Section 2.3 [Scalar Variables], page 15).

2.1.1 Single-quoted Strings

String literals can be represented in primarily three ways in Perl. The first way is in single quotes. Single quotes can be used to make sure that nearly all special characters that might be interpreted differently are taken at "face value". If that concept is confusing to you, just think about single quoted strings as being, for the most part, "what you see is what you get". Consider the following single-quoted string:

```
'i\o';  # The string 'i\o'
```

This represents a string consisting of the character '`i`', followed by '`\`', followed by '`o`'. However, it is probably easier just to think of the string as `i\o`. Some other languages require you think of strings not as single chunks of data, but as some aggregation of a set of characters. Perl does not work this way. A string is a simple, single unit that can be as long as you would like.[1]

Note in our example above that `'i\o'` is an expression. Like all expressions, it evaluates to something. In this case, it evaluates to the string value, `i\o`. Note that we made the expression `'i\o'` into a statement, by putting a semi-colon at the end (`'i\o'`;). This particular statement does not actually perform any action in Perl, but it is still a valid Perl statement nonetheless.

2.1.1.1 Special Characters in Single-quoted Strings

There are two characters in single quoted strings that do not always represent themselves. This is due to necessity, since single-quoted strings start and end with the '`'`' character. We need a way to express inside a single-quoted string that we want the string to contain a '`'`' character.

The solution to this problem is to preceded any '`'`' characters we actually want to appear in the string itself with the backslash ('\' character). Thus we have strings like this:

```
'xxx\'xxx';  # xxx, a single-quote character, and then xxx
```

We have in this example a string with 7 characters exactly. Namely, this is the string: `xxx'xxx`. It can be difficult at first to become accustomed to the idea that two characters in the input to Perl actually produce only one character in the string itself.[2] However, just keep in mind the rules and you will probably get used to them quickly.

Since we have used the '\' character to do something special with the '`'`' character, we must now worry about the special cases for the backslash character itself. When we see a '\' character in a single-quoted string, we must carefully consider what will happen.

Under most circumstances, when a '\' is in a single-quoted string, it is simply a backslash, representing itself, as most other characters do. However, the following exceptions apply:

- The sequence '`\'`' yields the character '`'`' in the actual string. (This is the exception we already discussed above).
- The sequence '`\\`' yields the character '`\`' in the actual string. In other words, two backslashes right next to each other actually yield only one backslash.
- A backslash, by itself, cannot be placed at the end of a the single-quoted string. This cannot happen because Perl will think that you are using the '\' to escape the closing '`'`'.

The following examples exemplify the various exceptions, and use them properly:

```
'I don\'t think so.';          # Note the ' inside is escaped with \
'Need a \\ (backslash) or \?'; # The \\ gives us \, as does \
'You can do this: \\';         # A single backslash at the end
'Three \\\'s: "\\\\\"';        # There are three \ chars between ""
```

In the last example, note that the resulting string is `Three \'s: "\\\"`. If you can follow that example, you have definitely mastered how single-quoted strings work!

[1] Actually, it cannot be longer than your computer has virtual memory, but that is rarely a problem.

[2] C programmers are already probably used to this idea.

2.1.1.2 Newlines in Single-quoted Strings

Note that there is no rule against having a single-quoted string span several lines. When you do this, the string has *newline* characters embedded in it.

A newline character is a special ASCII character that indicates that a new line should be started. In a text editor, or when printing output to the screen, this usually indicates that the cursor should move from the end of the current line to the first position on the line following it.

Since Perl permits the placement of these newline characters directly into single quoted strings, we are permitted to do the following:

```
'Time to
start anew.';   # Represents the single string composed of:
                # 'Time to' followed by a newline, followed by
                # 'start anew.'
```

This string has a total of twenty characters. The first seven are `Time to`. The next character following that is a newline. Then, the eleven characters, `start anew.` follow. Note again that this is **one string**, with a newline as its eighth character.

Further, note that we are not permitted to put a comment in the middle of the string, even though we are usually allowed to place a '`#`' anywhere on the line and have the rest of the line be a comment. We cannot do this here, since we have yet to terminate our single-quoted string with a '`'`', and thus, any '`#`' character and comment following it would actually become part of the single-quoted string! Remember that single-quotes strings are delimited by '`'`' at the beginning, and '`'`' at the end, and everything in between is considered part of the string, included newlines, '`#`' characters and anything else.

2.1.1.3 Examples of Invalid Single-quoted Strings

In finishing our discussion of singled-quoted strings, consider these examples of strings that are **not** legal because they violate the exceptions we talked about above:

```
'You cannot do this: \'; # INVALID: the ending \ cannot be alone
'It is 5 o'clock!'       # INVALID: the ' in o'clock should be escaped
'Three \'s: \\\\\';      # INVALID: the final \ escapes the ', thus
                         #          the literal is  not terminated
'This is my string;      # INVALID: missing close quote
```

Sometimes, when you have invalid string literals such as in the example above, the error message that Perl gives is not particularly intuitive. However, when you see error messages such as:

```
(Might be a runaway multi-line '' string starting on line X)
Bareword found where operator expected
Bareword "foo" not allowed while "strict subs" in use
```

It is often an indication that you have runaway or invalid strings. Keep an eye out for these problems. Chances are, you will forget and violate one of the rules for single-quoted strings eventually, and then need to determine why you are unable to run your Perl program.

2.1.2 A Digression—The print Function

Before we move on to our consideration of double-quoted strings, it is necessary to first consider a small digression. We know how to represent strings in Perl, but, as you may have noticed, the examples we have given thus far do not do anything interesting. If you try placing the statements that we listed as examples in Section 2.1.1 [Single-quoted Strings], page 7, into a full Perl program, like this:

```
#!/usr/bin/perl

use strict;
use warnings;

'Three \\\'s: "\\\\\"'; # There are three \ chars between ""
'xxx\'xxx';             # xxx, a single-quote character, and then xxx
'Time to
start anew.';
```

you probably noticed that nothing of interest happens. Perl gladly runs this program, but it produces no output.

Thus, to begin to work with strings in Perl beyond simple hypothetical considerations, we need a way to have Perl display our strings for us. The canonical way of accomplishing this in Perl is to use the `print` function.

The `print` function in Perl can be used in a variety of ways. The simplest form is to use the statement `print STRING;`, where `STRING` is any valid Perl string.

So, to reconsider our examples, instead of simply listing the strings, we could instead print each one out:

```
#!/usr/bin/perl

use strict;
use warnings;

print 'Three \\\'s: "\\\\\"'; # Print first string
print 'xxx\'xxx';             # Print the second
print 'Time to
start anew.
';    # Print last string, with a newline at the end
```

This program will produce output. When run, the output goes to what is called the *standard output*. This is usually the terminal, console or window in which you run the Perl program. In the case of the program above, the output to the standard output is as follows:

```
Three \'s: "\\\"xxx'xxxTime to
start anew.
```

Note that a newline is required to break up the lines. Thus, you need to put a newline at the end of every valid string if you want your string to be the last thing on that line in the output.

Note that it is particularly important to put a newline on the end of the last string of your output. If you do not, often times, the command prompt for the command interpreter that you are using may run together with your last line of output, and this can be very disorienting. So, **always** remember to place a newline at the end of each line, particularly on your last line of output.

Finally, you may have noticed that formatting your code with newlines in the middle of single-quoted strings hurts readability. Since you are inside a single-quoted string, you cannot change the format of the continued lines within the print statement, nor put comments at the ends of those lines because that would insert data into your single-quoted strings. To handle newlines more elegantly, you should use double-quoted strings, which are the topic of the next section.

2.1.3 Double-quoted Strings

Double-quoted strings are another way of representing scalar string literals in Perl. Like single-quoted strings, you place a group of ASCII characters between two delimiters (in this case, our delimiter is ‘`"`’). However, something called *interpolation* happens when you use a double-quoted string.

2.1.3.1 Interpolation in Double-quoted Strings

Interpolation is a special process whereby certain special strings written in ASCII are replaced by something different. In Section 2.1.1 [Single-quoted Strings], page 7, we noted that certain sequences in single-quoted strings (namely, `\\` and `\'`) were treated differently. This is very similar to what happens with interpolation. For example, in interpolated double-quoted strings, various sequences preceded by a ‘\’ character act different.

Here is a chart of the most common of these:

String	Interpolated As
'\\'	an actual, single backslash character
'\$'	a single $ character
'\@'	a single @ character
'\t'	tab
'\n'	newline
'\r'	hard return
'\f'	form feed
'\b'	backspace
'\a'	alarm (bell)
'\e'	escape
'\033'	character represented by octal value, 033
'\x1b'	character represented by hexadecimal value, 1b

2.1.3.2 Examples of Interpolation

Let us consider an example that uses a few of these characters:

```
#!/usr/bin/perl

use strict;
use warnings;

print "A backslash: \\\n";
print "Tab follows:\tover here\n";
print "Ring! \a\n";
print "Please pay bkuhn\@ebb.org \$20.\n";
```

This program, when run, produces the following output on the screen:

```
A backslash: \
Tab follows:  over here
Ring!
Please pay bkuhn@ebb.org $20.
```

In addition, when running, you should hear the computer beep. That is the output of the '\a' character, which you cannot see on the screen. However, you should be able to hear it.

Notice that the '\n' character ends a line. '\n' should always be used to end a line. Those students familiar with the C language will be used to using this sequence to mean *newline*. When writing Perl, the word *newline* and the '\n' character are roughly synonymous.

2.1.3.3 Examples of Interpolation (ASCII Octal Values)

With the exception of '\n', you should note that the interpolated sequences are simply shortcuts for actually ASCII characters that can be expressed in other ways. Specifically, you are permitted to use the actual ASCII codes (in octal or hexadecimal) to represent characters. To exemplify this, consider the following program:

```
#!/usr/bin/perl

use strict;
use warnings;

print "A backslash: \134\n";
print "Tab follows:\11over here\n";
print "Ring! \7\n";
print "Please pay bkuhn\100ebb.org \04420.\n";
```

This program generates exactly the same output as the program we first discussed in this section. However, instead of using the so-called "shortcuts" for the ASCII values, we wrote each character in question using the octal value of that character. Comparing the two programs should provide some insight into the use of octal values in double-quoted strings.

Basically, you simply write '`\XYZ`', where *XYZ* is the octal number of the ASCII character desired. Note that you don't always need to write all three digits. Namely, notice that the double-quoted string, `"Ring! \7\n"`, did not require all the digits. This is because in the string, the octal value is immediately followed by another '`\`', and thus Perl could figure out what we meant. This is one of the many cases where you see Perl trying to "do the right thing" when you do something that is technically not completely legal.

However, note that, in the last string, the three digits are required for the sequence (`"\04420"`), because the `20` immediately following the octal code could be easily confused with the octal value preceding it. The point, however, is that as long as you obey the rules for doing so, you can often add characters to your double-quoted strings by simply using the ASCII value.

2.1.3.4 Examples of Interpolation (ASCII Hex Values)

You need not use only the octal values when interpolating ASCII characters into double-quoted strings. You can also use the hexadecimal values. Here is our same program using the hexadecimal values this time instead of the octal values:

```
#!/usr/bin/perl

use strict;
use warnings;

print "A backslash: \x5C\n";
print "Tab follows:\x09over here\n";
print "Ring! \x07\n";
print "Please pay bkuhn\x40ebb.org \x2420.\n";
```

As you can see, the theme of "there's more than one way to do it" is really playing out here. However, we only used the ASCII codes as a didactic exercise. Usually, you should use the single character sequences (like '`\a`' and '`\t`'), unless, of course, you are including an ASCII character that does not have a shortcut, single character sequence.

2.1.3.5 Characters Requiring Special Consideration

The final issue we have yet to address with double-quoted strings is the use of '`$`' and '`@`'. These two characters must always be quoted. The reason for this is not apparent now, but be sure to keep this rule in mind until we learn why this is needed. For now, it is enough to remember that in double-quoted strings, Perl does something special with '`$`' and '`@`', and thus we must be careful to quote them. (If you cannot wait to find out why, you should read Section 2.3.1 [Scalar Interpolation], page 16 and Section 3.4.4 [Array Interpolation], page 29.

2.2 Numbers

Perl has the ability to handle both floating point and integer numbers in reasonable ranges[1].

2.2.1 Numeric Literals

Numeric literals are simply constant numbers. Numeric literals are much easier to comprehend and use than string literals. There are only a few basic ways to express numeric literals.

The numeric literal representations that Perl users are similar to those used in other languages such as C, Ada, and Pascal. The following are a few common examples:

```
42;         # The number 42
12.5;       # A floating point number, twelve and a half
101873.000; # 101,873
.005        # five thousandths
5E-3;       # same number as previous line
23e-100;    # 23 times 10 to the power of -100 (very small)
2.3E-99;    # The same number as the line above!
23e6;       # 23,000,000
23_000_000; # The same number as line above
            # The underscores are for readability only
```

As you can see, there are three basic ways to express numeric literals. The most simple way is to write an integer value, without a decimal point, such as `42`. This represents the number forty-two.

You can also write numeric literals with a decimal point. So, you can write numbers like `12.5`, to represent numbers that are not integral values. If you like, you can write something like `101873.000`, which really simply represents the integral value 101,873. Perl does not mind that you put the extra 0's on the end.

Probably the most complex method of expressing a numeric literal is using what is called *exponential notation*. These are numbers of the form $b \cdot 10^x$, where b is some decimal number, positive or negative, and x is some integer, positive or negative. Thus, you can express very large numbers, or very small numbers that are mostly 0s (either to the right or left of the decimal point) using this notation. However, when you write such a number

[1] There are also standard packages available to handle very large and very small numbers, and packages to force use of integers only for all calculations.

as a literal in Perl, you must write it in the from `bEx`, where `b` and `x` are the desired base and exponent, but `E` is the actual character, `E` (or `e`, if you prefer). The examples of `5E-3`, `23e-100`, `2.3E-99`, and `23e6` in the code above show how the exponential notation can be used.

Finally, if you write out a very large number, such as `23000000`, you can place underscores inside the number to make it more readable.[2] Thus, `23000000` is exactly the same as `23_000_000`.

2.2.1.1 Printing Numeric Literals

As with string literals, you can also use the `print` function in Perl to print numerical literals. Consider this program:

```
#!/usr/bin/perl

use strict;
use warnings;

print 2E-4, ' ', 9.77E-5, " ", 100.00, " ", 10_181_973, ' ', 9.87E9,
      " ", 86.7E14, "\n";
```

which produces the output:

```
0.0002 9.77e-05 100 10181973 9870000000 8.67e+15
```

First of all, we have done something new here with `print`. Instead of giving `print` one *argument*, we have given it a number of arguments, separated by commas. Arguments are simply the parameters on which you wish the function to operate. The `print` function, of course, is used to display whatever arguments you give it.

In this case, we gave a list of arguments that included both string and numeric literals. That is completely acceptable, since Perl can usually tell the difference! The string literals are simply spaces, which we are using to separate our numeric literals on the output. Finally, we put the newline at the end of the output.

Take a close look at the numeric literals that were output. Notice that Perl has made some formatting changes. For example, as we know, the `_`'s are removed from `10_181_973`. Also, those decimals and large integers in exponential notation that were relatively reasonable to expand were expanded by Perl. In addition, Perl only printed `100` for `100.00`, since the decimal portion was zero. Of course, if you do not like the way that Perl formats numbers by default, we will later learn a way to have Perl format them differently (see Section 2.5 [Output of Scalar Data], page 23).

2.3 Scalar Variables

Since we have now learned some useful concepts about strings and numbers in Perl, we can consider how to store them in *variables*. In Perl, both numeric and string values are stored in *scalar variables*.

[2] Language historians may notice that this is a feature from the Ada language.

Scalar variables are storage areas that you can use to store any scalar value. As we have already discussed, scalar values are strings or numbers, such as the literals that we discussed in previous sections.

You can always identify scalar variables because they are in the form `$NAME`, where `NAME` is any string of alphanumeric characters and underscores starting with a letter, up to 255 characters total. Note that `NAME` will be *case sensitive*, thus `$xyz` is a different variable than `$xYz`.

Note that the first character in the name of any scalar variable must be `$`. All variables that begin with `$` are always scalar. Keep this in mind as you see various expressions in Perl. You can remember that anything that begins with `$` is always scalar.

As we discussed (see Section 1.1 [A First Perl Program], page 3), it is best to always declare variables with the `my` function. You do not need to do this if you are not using `strict`, but you should always use `strict` until you are an experienced Perl programmer.

The first operation we will consider with scalar variables is *assignment*. Assignment is the way that we give a value from some scalar expression to a scalar variable.

The assignment operator in Perl is `=`. On the left hand side of the `=`, we place the scalar variable whose value we wish to change. On the right side of the `=`, we place the scalar expression. (Note that so far, we have learned about three types of scalar expressions: string literals, numeric literals, and scalar variables).

Consider the following code segment:

```
use strict;

my $stuff = "My data";  # Assigns "My data" to variable $stuff
$stuff = 3.5e-4;        # $stuff is no longer set to "My data";
                        # it is now 0.00035
my $things = $stuff;    # $things is now 0.00035, also.
```

Let us consider this code more closely. The first line does two operations. First, using the `my` function, it declares the variable `$stuff`. Then, in the same statement, it assigns the variable `$stuff` with the scalar expression, `"My data"`.

The next line uses that same variable `$stuff`. This time, it is replacing the value of `"My data"` with the numeric value of `0.00035`. Note that it does not matter that `$stuff` once contained string data. We are permitted to change and assign it with a different type of scalar data.

Finally, we declare a new variable `$things` (again, using the `my` function), and use assignment to give it the value of the scalar expression `$stuff`. What does the scalar expression, `$stuff` evaluate to? Simply, it evaluates to whatever scalar value is held by `$stuff`. In this case, that value is `0.00035`.

2.3.1 Scalar Interpolation

Recall that when we discussed double-quotes strings (see Section 2.1.3 [Double-quoted Strings], page 11), we noted that we had to backslash the `$` character (e.g., `"\$"`). Now, we discuss the reason that this was necessary. Any scalar variable, when included in a double-quoted string *interpolates*.

Interpolation of scalar variables allows us to insert the value of a scalar variable right into a double-quoted string. In addition, since Perl largely does all data conversion necessary, we can often use variables that have integer and float values and interpolate them right into strings without worry. In most cases, Perl will do the right thing.

Consider the following sample code:

```
use strict;
my $friend = 'Joe';
my $greeting = "Howdy, $friend!";
            # $greeting contains "Howdy, Joe!"
my $cost = 20.52;
my $statement = "Please pay \$$cost.\n";
         # $statement contains "Please pay $20.52.\n"
my $debt = "$greeting  $statement";
         # $debt contains "Howdy, Joe!  Please pay $20.52.\n"
```

As you can see from this sample code, you can build up strings by placing scalars inside double-quotes strings. When the double-quoted strings are evaluated, any scalar variables embedded within them are replaced with the value that each variable holds.

Note in our example that there was no problem interpolating `$cost`, which held a numeric scalar value. As we have discussed, Perl tries to do the right thing when converting strings to numbers and numbers to strings. In this case, it simply converted the numeric value of `20.52` into the string value `'20.52'` to interpolate `$cost` into the double-quoted string.

Interpolation is not only used when assigning to other scalar variables. You can use a double-quoted string and interpolate it in any context where a scalar expression is appropriate. For example, we could use it as part of the `print` statement.

```
#!/usr/bin/perl

use strict;
use warnings;

my $owner  = 'Elizabeth';
my $dog    = 'Rex';
my $amount = 12.5;
my $what   = 'dog food';

print "${owner}'s dog, $dog, ate $amount pounds of $what.\n";
```

This example produces the output:

```
Elizabeth's dog, Rex, ate 12.5 pounds of dog food.
```

Notice how we are able to build up a large string using four variables, some text, and a newline character, all contained within one interpolated double-quoted string. We needed only to pass **one argument** to `print`! Recall that previously (see Section 2.2.1.1 [Printing Numeric Literals], page 15) we had to separate a number of scalar arguments by commas to pass them to print. Thus, using interpolation, it is very easy to build up smaller scalars into larger, combined strings. This is a very convenient and frequently used feature of Perl.

You may have noticed by now that we did something very odd with `$owner` in the example above. Instead of using `$owner`, we used `${owner}`. We were forced to do this because following a scalar variable with the character `'` would confuse Perl.[3] To make it clear to Perl that we wanted to use the scalar with name `owner`, we needed to enclose `owner` in curly braces (`{owner}`).

In many cases when using interpolation, Perl requires us to do this. Certain characters that follow scalar variables mean something special to Perl. When in doubt, however, you can wrap the name of the scalar in curly braces (as in `${owner}`) to make it clear to Perl what you want.

Note that this can also be a problem when an interpolated scalar variable is followed by alpha-numeric text or an underscore. This is because Perl cannot tell where the name of the scalar variable ends and where the literal text you want in the string begins. In this case, you also need to use the curly braces to make things clear. Consider:

```
use strict;

my $this_data = "Something";
my $that_data = "Something Else ";

print "_$this_data_, or $that_datawill do\n"; # INVALID: actually refers
                                              # to the scalars $this_data_
                                              # and $that_datawill

print "_${this_data}_, or ${that_data}will do\n";
           # CORRECT: refers to $this_data and $that_data,
           #          using curly braces to make it clear
```

2.3.2 Undefined Variables

You may have begun to wonder: what value does a scalar variable have if you have not given it a value? In other words, after:

```
use strict;
my $sweetNothing;
```

what value does `$sweetNothing` have?

The value that `$sweetNothing` has is a special value in Perl called `undef`. This is frequently expressed in English by saying that `$sweetNothing` is undefined.

The `undef` value is a special one in Perl. Internally, Perl keeps track of which variables your program has assigned values to and which remain undefined. Thus, when you use a variable in any expression, Perl can inform you if you are using an undefined value.

For example, consider this program:

[3] The `'` character is a synonym for `::` which is used for packages, a topic not covered in this text.

```
#!/usr/bin/perl

use strict;
use warnings;

my $hasValue = "Hello";
my $hasNoValue;

print "$hasValue $hasNoValue\n";
```

When this program is run, it produces the following output:

```
Use of uninitialized value at line 8.
Hello
```

What does this mean? Perl noticed that we used the uninitialized (i.e., undefined) variable, `$hasNoValue` at line 8 in our program. Because we were using `warnings`, Perl warned us about that use of the undefined variable.

However, Perl did not crash the program! Perl is nice enough not to make undefined variables a hassle. If you use an undefined variable and Perl expected a string, Perl uses the empty string, `""`, in its place. If Perl expected a number and gets `undef`, Perl substitutes `0` in its place.

However, when using `warnings`, Perl will always warn you when you have used an undefined variable at run-time. The message will print to the standard error (which, by default, is the screen) each time Perl encounters a use of a variable that evaluates to `undef`. If you do not use `warnings`, the warnings will not print, but you should probably wait to turn off `warnings` until you are an experienced Perl programmer.

Besides producing warning messages, the fact that unassigned variables are undefined can be useful to us. The first way is that we can explicitly test to see if a variable is undefined. There is a function that Perl provides called `defined`. It can be used to test if a variable is defined or not.

In addition, Perl permits the programmer to assign a variable the value `undef`. The expression `undef` is a function provided by Perl that we can use in place of any expression. The function `undef` is always guaranteed to return an undefined value. Thus, we can take a variable that already has a value and make it undefined.

Consider the following program:

```
#!/usr/bin/perl

use strict;
use warnings;

my $startUndefined;
my $startDefined = "This one is defined";

print "defined \$startUndefined == ",
      defined $startUndefined,
      ", defined \$startDefined == ",
      defined $startDefined, "\n";

$startUndefined = $startDefined;
$startDefined = undef;

print "defined \$startUndefined == ",
      defined $startUndefined,
      ", defined \$startDefined == ",
      defined $startDefined, "\n";
```

Which produces the output:

```
defined $startUndefined == , defined $startDefined == 1
defined $startUndefined == 1, defined $startDefined ==
```

Notice a few things. First, since we first declared `$startUndefined` without giving it a value, it was set to `undef`. However, we gave `$startDefined` a value when it was declared, thus it started out defined. These facts are exemplified by the output.

To produce that output, we did something that you have not seen yet. First, we created some strings that “looked” like the function calls so our output would reflect what the values of those function calls were. Then, we simply used those functions as arguments to the `print` function. This is completely legal in Perl. You can use function calls as arguments to other functions.

When you do this, the innermost functions are called first, in their argument order. Thus, in our `print` statements, first `defined $startUndefined` is called, followed by `defined $startDefined`. These two functions each evaluate to some value. That value then becomes the argument to the `print` function.

So, what values did `defined` return? We can determine the answer to this question from the printed output. We can see that when we called `defined` on the variable that we started as undefined, `$startUndefined`, we got no output for that call (in fact, `defined` returned an empty string, `""`). When we called `defined` on the value that we had assigned to, `$startDefined`, we got the output of `1`.

Thus, from the experiment, we know that when its argument is not defined, `defined` returns the value `""`, otherwise known as the empty string (which, of course, prints nothing to the standard output when given as an argument to `print`).

In addition, we know that when a variable is defined, `defined` returns the value `1`.

Hopefully, you now have some idea of what an `undef` value is, and what `defined` does. It might not yet be clear why `defined` returns an empty string or `1`. If you are particularly curious now, see Section 4.2 [A Digression—Truth Values], page 31.

2.4 Operators

There are a variety of operators that work on scalar values and variables. These operators allow us to manipulate scalars in different ways. This section discusses the most common of these operators.

2.4.1 Numerical Operators

The basic numerical operators in Perl are like others that you might see in other high level languages. In fact, Perl's numeric operators were designed to mimic those in the C programming language.

First, consider this example:

```
use strict;
my $x = 5 * 2 + 3;      # $x is 13
my $y = 2 * $x / 4;     # $y is 6.5
my $z = (2 ** 6) ** 2;  # $z is 4096
my $a = ($z - 96) * 2;  # $a is 8000
my $b = $x % 5;         # 3, 13 modulo 5
```

As you can see from this code, the operators work similar to rules of algebra. When using the operators there are two rules that you have to keep in mind—the rules of *precedence* and the rules of *associativity*.

Precedence involves which operators will get evaluated first when the expression is ambiguous. For example, consider the first line in our example, which includes the expression, `5 * 2 + 3`. Since the multiplication operator (`*`) has precedence over the addition operator (`+`), the multiplication operation occurs first. Thus, the expression evaluates to `10 + 3` temporarily, and finally evaluates to `13`. In other words, precedence dictates which operation occurs first.

What happens when two operations have the same precedence? That is when associativity comes into play. Associativity is either left or right[4]. For example, in the expression `2 * $x / 4` we have two operators with equal precedence, `*` and `/`. Perl needs to make a choice about the order in which they get carried out. To do this, it uses the associativity. Since multiplication and division are left associative, it works the expression from left to right, first evaluating to `26 / 4` (since `$x` was `13`), and then finally evaluating to `6.5`.

Briefly, for the sake of example, we will take a look at an operator that is left associative, so we can contrast the difference with right associativity. Notice when we used the exponentiation (`**`) operator in the example above, we had to write `(2 ** 6) ** 2`, and not `2 ** 6 ** 2`.

What does `2 ** 6 ** 2` evaluate to? Since `**` (exponentiation) is **right associative**, first the `6 ** 2` gets evaluated, yielding the expression `2 ** 36`, which yields `68719476736`, which is definitely not `4096`!

[4] Some operators are not associative at all (see Section 2.7 [Summary of Scalar Operators], page 24).

Here is a table of the operators we have talked about so far. They are listed in order of precedence. Each line in the table is one order of precedence. Naturally, operators on the same line have the same precedence. The higher an operator is in the table, the higher its precedence.

Operator	**Associativity**	**Description**
**	right	exponentiation
*, /, %	left	multiplication, division, modulus
`+`, `-`	left	addition, subtraction

2.4.2 Comparison Operators

Comparing two scalars is quite easy in Perl. The *numeric* comparison operators that you would find in C, C++, or Java are available. However, since Perl does automatic conversion between strings and numbers for you, you must differentiate for Perl between numeric and string comparison. For example, the scalars `"532"` and `"5"` could be compared two different ways—based on numeric value or ASCII string value.

The following table shows the various comparison operators and what they do. Note that in Perl `""`, `0` and `undef` are false and anything else as true. (This is an over-simplified definition of true and false in Perl. See Section 4.2 [A Digression—Truth Values], page 31, for a complete definition.)

The table below assumes you are executing `$left <OP> $right`, where `<OP>` is the operator in question.

Operation	**Numeric Version**	**String Version**	**Returns**
less than	`<`	`lt`	1 iff. `$left` is less than `$right`
less than or equal to	`<=`	`le`	1 iff. `$left` is less than or equal to `$right`
greater than	`>`	`gt`	1 iff. `$left` is greater than `$right`
greater than or equal to	`>=`	`ge`	1 iff. `$left` is greater than or equal to `$right`
equal to	`==`	`eq`	1 iff. `$left` is the same as `$right`
not equal to	`!=`	`ne`	1 iff. `$left` is not the same as `$right`
compare	`<=>`	`cmp`	-1 iff. `$left` is less than `$right`, 0 iff. `$left` is equal to `$right` 1 iff. `$left` is greater than `$right`

Here are a few examples using these operators.

```
use strict;
my $a = 5; my $b = 500;
$a < $b;                 # evaluates to 1
$a >= $b;                # evaluates to ""
```

```
$a <=> $b;                # evaluates to -1
my $c = "hello"; my $d = "there";
$d cmp $c;                # evaluates to 1
$d ge  $c;                # evaluates to 1
$c cmp "hello";           # evaluates to ""
```

2.4.3 Auto-Increment and Decrement

The auto-increment and auto-decrement operators in Perl work almost identically to the corresponding operators in C, C++, or Java. Here are few examples:

```
use strict;
my $abc = 5;
my $efg = $abc-- + 5;       # $abc is now 4, but $efg is 10
my $hij = ++$efg - --$abc;  # $efg is 11, $abc is 3, $hij is 8
```

2.4.4 String Operators

The final set of operators that we will consider are those that operate specifically on strings. Remember, though, that we can use numbers with them, as Perl will do the conversions to strings when needed.

The string operators that you will see and use the most are `.` and `x`. The `.` operator is string concatenation, and the `x` operator is string duplication.

```
use strict;
my $greet = "Hi! ";
my $longGreet  = $greet x 3;   # $longGreet is "Hi! Hi! Hi! "
my $hi = $longGreet . "Paul.";  # $hi is "Hi! Hi! Hi! Paul."
```

2.4.5 Assignment with Operators

It should be duly noted that it is possible to concatenate, like in C, an operator onto the assignment statement to abbreviate using the left hand side as the first operand. For example,

```
use strict;
my $greet = "Hi! ";
$greet  .= "Everyone\n";
$greet  = $greet . "Everyone\n"; # Does the same operation
                                 # as the line above
```

This works for any simple, binary operator.

2.5 Output of Scalar Data

To output a scalar, you can use the `print` and `printf` built-in functions. We have already seen examples of the `print` command, and the `printf` command is very close to that in C or C++. Here are a few examples:

```
use strict;
my $str  = "Howdy, ";
my $name = "Joe.\n";
```

```
print $str, $name;    # Prints out: Howdy, Joe.<NEWLINE>
my $f = 3e-1;
printf "%2.3f\n", $f; # Prints out: 0.300<NEWLINE>
```

2.6 Special Variables

It is worth noting here that there are some variables that are considered “special” by Perl. These variables are usually either read-only variables that Perl sets for you automatically based on what you are doing in the program, or variables you can set to control the behavior of how Perl performs certain operations.

Use of special variables can be problematic, and can often cause unwanted side effects. It is a good idea to limit your use of these special variables until you are completely comfortable with them and what they do. Of course, like anything in Perl, you can get used to some special variables and not others, and use only those with which you are comfortable.

2.7 Summary of Scalar Operators

In this chapter, we have looked at a number of different scalar operators available in the Perl language. Earlier, we gave a small chart of the operators, ordered by their precedence. Now that we have seen all these operators, we should consider a list of them again, ordered by precedence. Note that some operators are listed as “nonassoc”. This means that the given operator is not associative. In other words, it simply does not make sense to consider associative evaluation of the given operator.

Operator	**Associativity**	**Description**
`++`, `--`	nonassoc	auto-increment and auto-decrement
`**`	right	exponentiation
`*`, `/`, `%`	left	multiplication, division, modulus
`+`, `-`, `.`	left	addition, subtraction, concatenation
`<`, `>`, `<=`, `>=`, `lt`, `gt`, `le`, `ge`	nonassoc	comparison operators
`==`, `!=`, `<=>`, `eq`, `ne`, `cmp`	nonassoc	comparison operators

This list is actually still quite incomplete, as we will learn more operators later on. However, you can always find a full list of all operators in Perl in the *perlop* documentation page, which you can get to on most systems with Perl installed by typing ‘`perldoc perlop`’.

3 Arrays

Now that we have a good understanding of the way scalar data and variables work and what can be done with them in Perl, we will look into the most basic of Perl's natural data structures—arrays.

3.1 The Semantics of Arrays

The arrays in Perl are semantically closest to lists in Lisp or Scheme (sans cons cells), however the syntax that is used to access arrays is closer to arrays in C. In fact, one can often treat Perl's arrays as if they were simply C arrays, but they are actually much more powerful than that.

Perl arrays grow and shrink dynamically as needed. The more data you put into a Perl list, the bigger it gets. As you remove elements from the list, the list will shrink to the right size. Note that this is inherently different from arrays in the C language, where the programmer must keep track and control the size of the array.

However, Perl arrays are accessible just like C arrays. So, you can subscript to anywhere within a given list at will. There is no need to process through the first four elements of the list to get the fifth element (as in Scheme). In this manner, you get the advantages of both a dynamic list, and a static-size array.

The only penalty that you pay for this flexibility is that when an array is growing very large very quickly, it can be a bit inefficient. However, when this must occur, Perl allows you to pre-build an array of certain size. We will show how to do this a bit later.

A Perl array is always a list of scalars. Of course, since Perl makes no direct distinction between numeric and string values, you can easily mix different types of scalars within the same array. However, everything in the array must be a scalar[1].

Note the difference in terminology that is used here. Arrays refer to *variables* that store a list of scalar values. Lists can be written as literals (see Section 3.2 [List Literals], page 25) and used in a variety of ways. One of the ways that list literals can be used is to assign to array variables (see Section 3.3 [Array Variables], page 26). We will discuss both list literals and array variables in this chapter.

3.2 List Literals

Like scalars, it is possible to write lists as literals right in your code. Of course, as with inserting string literals in your code, you must use proper quoting.

There are two primary ways to quote list literals that we will discuss here. One is using `()`, and the other is using what is called a quoting operator. The quoting operator for lists is `qw`. A quoting operator is always followed by a single character, which is the "stop character". It will eat up all the following input until the next "stop character". In the case of `qw`, it will use each token that it finds as an element in a list until the second "stop character" is reached. The advantage of the `qw` operator is that you do not need to quote strings in any additional way, since `qw` is already doing the quoting for you.

[1] It is possible to make an array of arrays using a concept called "references", but that topic is beyond the scope of this book.

Here are a few examples of some list literals, using both `()` and the `qw` operator.

```
();                    # this list has no elements; the empty list
qw//;                  # another empty list
("a", "b", "c",
  1,  2,  3);          # a list with six elements
qw/hello world
  how are you today/; # another list with six elements
```

Note that when we use the `()`, we have to quote all strings, and we need to separate everything by commas. The `qw` operator does not require this.

Finally, if you have any two scalar values where all the values between them can be enumerated, you can use an operator called the `..` operator to build a list. This is most easily seen in an example:

```
(1 .. 100);     # a list of 100 elements: the numbers from 1 to 100
('A' .. 'Z');   # a list of 26 elements: the uppercase letters From A to Z
('01' .. '31'); # a list of 31 elements: all possible days of a month
                #    with leading zeros on the single digit days
```

You will find the `..` operator particularly useful with slices, which we will talk about later in this chapter.

3.3 Array Variables

As with scalars, what good are literals if you cannot have variables? So, Perl provides a way to make array variables.

3.3.1 Array Variables

Each variable in Perl starts with a special character that identifies what type of variable it is. We saw that scalar variables always start with a '`$`'. Similarly, all array variables start with the character, '`@`', under the same naming rules that are used for scalar variables.

Of course, we cannot do much with a variable if we cannot assign things to it, so the assignment operator works as perfectly with arrays as it did with scalars. We must be sure, though, to always make the right hand side of the assignment a list, not a scalar! Here are a few examples:

```
use strict;
my @stuff  = qw/a  b  c/;           # @stuff a three element list
my @things = (1, 2, 3, 4);          # @things is a four element list
my $oneThing = "all alone";
my @allOfIt = (@stuff, $oneThing,
               @things);            # @allOfIt has 8 elements!
```

Note the cute thing we can do with the `()` when assigning `@allOfIt`. When using `()`, Perl allows us to insert other variables in the list. These variables can be either scalar or array variables! So, you can quickly build up a new list by "concatenating" other lists and scalar variables together. Then, that new list can be assigned to a new array, or used in any other way that list literals can be used.

3.3.2 Associated Scalars

Every time an array variable is declared, a special set of scalar variables automatically springs into existence, and those scalars change along with changes in the array with which they are associated.

First of all, for an array, `@array`, of n elements. There are scalar variables `$array[0]`, `$array[1]`, ..., `$array[n-1]` that contain first, second, third, ..., nth elements in the array, respectively. The variables in this format are full-fledged scalar variables. This means that anything you can do with a scalar variable, you can do with these elements. This provides a way to access array elements by subscript. In addition, it provides a way to change, modify and update individual elements without actually using the `@array` variable.

Another scalar variable that is associated to any array variable, `@array`, is `$#array`. This variable always contains the *subscript* of the last element in the array. In other words, `$array[$#array]` is always the last element of the array. The length of the array is always `$#array + 1`. Again, you are permitted to do anything with this variable that you can normally do with any other scalar variable; however, you must always make sure to leave the value as an integer greater than or equal to -1. In fact, if you know an array is going to grow very large quickly, you probably want to set this variable to a very high value. When you change the value of `$#array`, you not only resize the array for your use, you also direct Perl to allocate a specific amount of space for `@array`.

Here are a few examples that use the associated scalar variables for an array:

```
use strict;
my @someStuff = qw/Hello and
                  welcome/;     # @someStuff: an array of 3 elements
$#someStuff = 0;                # @someStuff now is simply ("Hello")
$someStuff[1] = "Joe";          # Now @someStuff is ("Hello", "Joe")
$#someStuff  = -1;              # @someStuff is now empty
@someStuff   = ();              # does same thing as previous line
```

3.4 Manipulating Arrays and Lists

Clearly, arrays and lists are very useful. However, there are a few more things in Perl you can use to make arrays and lists even more useful.

3.4.1 It Slices!

Sometimes, you may want to create a new array based on some subset of elements from another array. To do this, you use a slice. Slices use a subscript that is itself a list of integers to grab a list of elements from an array. This looks easier in Perl than it does in English:

```
use strict;
my @stuff = qw/everybody wants a rock/;
my @rock  = @stuff[1 .. $#stuff];      # @rock is qw/wants a rock/
my @want  = @stuff[ 0 .. 1];           # @want is qw/everybody wants/
@rock     = @stuff[0, $#stuff];        # @rock is qw/everybody rock/
```

As you can see, you can use both the `..` operator and commas to build a list for use as a slice subscript. This can be a very useful feature for array manipulation.

3.4.2 Functions

Perl also provides quite a few functions that operate on arrays. As you learn more and more Perl, you will see lots of interesting functions that work with arrays.

Now, we'll discuss a few of these functions that work on arrays: `push`, `pop`, `shift`, and `unshift`.

The names `shift` and `unshift` are an artifact of the Unix shells that used them to "shift around" incoming arguments.

3.4.2.1 Arrays as Stacks

What more is a stack than an unbounded array of things? This attitude is seen in Perl through the `push` and `pop` functions. These functions treat the "right hand side" (i.e., the end) of the array as the top of the stack. Here is an example:

```
use strict;
my @stack;
push(@stack, 7, 6, "go");   # @stack is now qw/7 6 go/
my $action = pop @stack;    # $action is "go", @stack is (7, 6)
my $value = pop(@stack) +
            pop(@stack);    # value is 6 + 7 = 13, @stack is empty
```

3.4.2.2 Arrays as Queues

If we can do stacks, then why not queues? You can build a queue in Perl by using the `unshift` and `pop` functions together.[2] Think of the `unshift` function as "enqueue" and the `pop` function as "dequeue". Here is an example:

```
use strict;
my @queue;
unshift (@queue, "Customer 1"); # @queue is now ("Customer 1")
unshift (@queue, "Customer 2"); # @queue is now ("Customer 2" "Customer 1")
unshift (@queue, "Customer 3");
         # @queue is now ("Customer 3" "Customer 2" "Customer 1")
my $item = pop(@queue);         # @queue is now ("Customer 3" "Customer 2")
print "Servicing $item\n";       # prints:  Servicing Customer 1\n
$item = pop(@queue);            # @queue is now ("Customer 3")
print "Servicing $item\n";       # prints:  Servicing Customer 2\n
```

This queue example works because `unshift` places items onto the front of the array, and `pop` takes items from the end of the array. However, be careful using more than two arguments on the `unshift` when you want to process an array as a queue. Recall that `unshift` places its arguments onto the array *in order* as they are listed in the function call. Consider this example:

```
use strict;
my @notAqueue;
unshift(@notAqueue, "Customer 0", "Customer 1");
```

[2] For another way to do this, see the exercises in this section.

```
                                # @queue is now ("Customer 0", "Customer 1")
unshift (@notAqueue, "Customer 2");
                         # @queue is now ("Customer 2", "Customer 0", "Customer 1")
```

Notice that this variable, `@notAqueue`, is not really a queue, if we use `pop` to remove items. The moral here is to be careful when using `unshift` in this manner, since it places it arguments on the array *in order*.

3.4.3 The Context—List vs. Scalar

It may have occurred to you by now that in certain places we can use a list, and in other places we can use a scalar. Perl knows this as well, and decides which is permitted by something called a *context*.

The context can be either list context or scalar context. Many operations do different things depending on what the current context is.

For example, it is actually valid to use an array variable, such as `@array`, in a scalar context. When you do this, the array variable evaluates to the number of elements in the array. Consider this example:

```
use strict;
my @things = qw/a few of my favorite/;
my $count  = @things;                   # $count is 5
my @moreThings = @things;               # @moreThings is same as @things
```

Note that Perl knows not to try and stuff `@things` into a scalar, which does not make any sense. It evaluates `@things` in a scalar context and given the number of elements in the array.

You must always be aware of the context of your operations. Assuming the wrong context can cause a plethora of problems for the new Perl programmer.

3.4.4 Array Interpolation

Array variables can also be evaluated through interpolation into a double-quoted string. This works very much like the interpolation of scalars into double-quoted strings (see Section 2.3.1 [Scalar Interpolation], page 16). When an array variable is encountered in a double-quoted string, Perl will join the array together, separating each element by spaces. Here is an example:

```
use strict;
my @saying = qw/these are a few of my favorite/;
my $statement = "@saying things.\n";
          # $statement is "these are a few of my favorite things.\n"
my $stuff = "@saying[0 .. 1] @saying[$#saying - 1, $#saying]  things.\n"
          # $stuff is "these are my favorite things.\n"
```

Note the use of slices when assigning `$stuff`. As you can see, Perl can be very expressive when we begin to use the interaction of different, interesting features.

4 Control Structures

The center of any imperative programming language is control structures. Although Perl is not purely an imperative programming language, it has ancestors that are very much imperative in nature, and thus Perl has inherited those same control structures. It also has added a few of its own.

As you begin to learn about Perl's control structures, realize that a good number of them are syntactic sugar. You can survive using only a subset of all the control structures that are available in Perl. You should use those with which you are comfortable. Obey the "hubris" of Perl, and write code that is readable. But, beyond that, do not use any control structures that you do not think you need.

4.1 Blocks

The first tool that you need to begin to use control structures is the ability to write code "blocks". A block of code could be any of the code examples that we have seen thus far. The only difference is, to make them a block, we would surround them with `{}`.

```
use strict;
{
my $var;
Statement;
Statement;
Statement;
}
```

Anything that looks like that is a block. Blocks are very simple, and are much like code blocks in languages like C, C++, and Java. However, in Perl, code blocks are decoupled from any particular control structure. The above code example is a valid piece of Perl code that can appear just about anywhere in a Perl program. Of course, it is only particularly useful for those functions and structures that use blocks.

Note that any variable declared in the block (in the example, `$var`) lives only until the end of that block. With variables declared `my`, normal lexical scoping that you are familiar with in C, C++, or Java applies.

4.2 A Digression—Truth Values

We have mentioned truth and "true and false" a few times now; however, we have yet to give a clear definition of what truth values are in Perl.

Every expression in Perl has a truth value. Usually, we ignore the truth value of the expressions we use. In fact, we have been ignoring them so far! However, now that we are going to begin studying various control structures that rely on the truth value of a given expression, we should look at true and false values in Perl a bit more closely.

The basic rule that most Perl programmers remember is that `0`, the empty string and `undef` are false, and everything else is true. However, it turns out that this rule is not actually completely accurate.

The actual rule is as follows:

Everything in Perl is true, except:

- the strings `""` (the empty string) and `"0"` (the string containing only the character, 0), or any string expression that evaluates to either `""` (the empty string) or `"0"`.
- any numeric expression that evaluates to a numeric `0`.
- any value that is not defined (i.e., equivalent to `undef`).

If that rule is not completely clear, the following table gives some example Perl expressions and states whether they are true or not:

Expression	**String/Number?**	**Boolean value**
0	number	false
0.0	number	false
0.0000	number	false
""	string	false
"0"	string	false
"0.0"	string	**true**
undef	N/A	false
42 - (6 * 7)	number	false
"0.0" + 0.0	number	**false**
"foo"	string	true

There are two expressions above that easily confuse new Perl programmers. First of all, the expression `"0.0"` is true. This is true because it is a string that is not `"0"`. The only string that is not empty that can be false is `"0"`. Thus, `"0.0"` must be true.

Next, consider `"0.0" + 0.0`. After what was just stated, one might assume that this expression is true. However, this expression is **false**. It is false because `+` is a numeric operator, and as such, `"0.0"` must be turned into its numeric equivalent. Since the numeric equivalent to `"0.0"` is `0.0`, we get the expression `0.0 + 0.0`, which evaluates to `0.0`, which is the same as `0`, which is false.

Finally, it should be noted that all references are true. The topic of Perl references is beyond the scope of this book. However, if we did not mention it, we would not be giving you the whole truth story.

4.3 The if/unless Structures

The `if` and `unless` structures are the simplest control structures. You are no doubt comfortable with `if` statements from C, C++, or Java. Perl's `if` statements work very much the same.

```
use strict;
if (expression) {
    Expression_True_Statement;
    Expression_True_Statement;
    Expression_True_Statement;
} elsif (another_expression) {
    Expression_Elseif_Statement;
    Expression_Elseif_Statement;
    Expression_Elseif_Statement;
} else {
    Else_Statement;
```

```
        Else_Statement;
        Else_Statement;
    }
```

There are a few things to note here. The `elsif` and the `else` statements are both optional when using an `if`. It should also be noted that after each `if (expression)` or `elsif (expression)`, a code *block* is required. These means that the `{}`'s are mandatory in all cases, even if you have only one statement inside.

The `unless` statement works just like an `if` statement. However, you replace `if` with `unless`, and the code block is executed only if the expression is *false* rather than true.

Thus `unless (expression) { }` is functionally equivalent to `if (! expression) { }`.

4.4 The while/until Structures

The `while` structure is equivalent to the `while` structures in Java, C, or C++. The code executes while the expression remains true.

```
    use strict;
    while (expression) {
        While_Statement;
        While_Statement;
        While_Statement;
    }
```

The `until (expression)` structure is functionally equivalent `while (! expression)`.

4.5 The do while/until Structures

The `do/while` structure works similar to the `while` structure, except that the code is executed at least once before the condition is checked.

```
    use strict;
    do {
        DoWhile_Statement;
        DoWhile_Statement;
        DoWhile_Statement;
    } while (expression);
```

Again, using `until (expression)` is the same as using `while (! expression)`.

4.6 The for Structure

The `for` structure works similarly to the `for` structure found in C, C++ or Java. It is really syntactic sugar for the `while` statement.

Thus:

```
    use strict;
    for(Initial_Statement; expression; Increment_Statement) {
        For_Statement;
        For_Statement;
        For_Statement;
    }
```

is equivalent to:

```
use strict;
Initial_Statement;
while (expression) {
    For_Statement;
    For_Statement;
    For_Statement;
    Increment_Statement;
}
```

4.7 The foreach Structure

The `foreach` control structure is the most interesting in this chapter. It is specifically designed for processing of Perl's native data types.

The `foreach` structure takes a scalar, a list and a block, and executes the block of code, setting the scalar to each value in the list, one at a time. Consider an example:

```
use strict;
my @collection = qw/hat shoes shirts shorts/;
foreach my $item (@collection) {
    print "$item\n";
}
```

This will print out each item in collection on a line by itself. Note that you are permitted to declare the scalar variable right with the `foreach`. When you do this, the variable lives only as long as the `foreach` does.

You will find `foreach` to be one of the most useful looping structures in Perl. Any time you need to do something to each element in the list, chances are, using a `foreach` is the best choice.

5 Associative Arrays (Hashes)

This chapter will introduce the third major Perl abstract data type, associative arrays. Also known as hashes, associative arrays provide native language support for one of the most useful data structures that programmers implement—the hash table.

5.1 What Is It?

Associative arrays, also frequently called *hashes*, are the third major data type in Perl after scalars and arrays. Hashes are named as such because they work very similarly to a common data structure that programmers use in other languages—hash tables. However, hashes in Perl are actually a direct *language supported* data type.

5.2 Variables

We have seen that each of the different native data types in Perl has a special character that identify that the variable is of that type. Hashes always start with a `%`.

Accessing a hash works very similar to accessing arrays. However, hashes are not subscripted by numbers. They can be subscripted by an arbitrary scalar value. You simply use the `{}` to subscript the value instead of `[]` as you did with arrays. Here is an example:

```
use strict;
my %table;
$table{'schmoe'} = 'joe';
$table{7.5}  = 2.6;
```

In this example, our hash, called, `%table`, has two entries. The key `'schmoe'` is associated with the value `'joe'`, and the key `7.5` is associated with the value `2.6`.

Just like with array elements, hash elements can be used anywhere a scalar variable is permitted. Thus, given a *%table* built with the code above, we can do the following:

```
print "$table{'schmoe'}\n";    # outputs "joe\n"
--$table{7.5};                 # $table{7.5} now contains 1.6
```

Another interesting fact is that all hash variables can be evaluated in the list context. When done, this gives a list whose odd elements are the keys of the hash, and whose even elements are the corresponding values. Thus, assuming we have the same `%table` from above, we can execute:

```
my @tableListed = %table;  # @tableListed is qw/schmoe joe 7.5 1.6/
```

If you happen to evaluate a hash in scalar context, it will give you `undef` if no entries have yet been defined, and will evaluate to true otherwise. However, evaluation of hashes in scalar context is not recommended. To test if a hash is defined, use `defined(%hash)`.

5.3 Literals

"Hash literals" per se do not exist. However, remember that when we evaluate a hash in the list context, we get the pairs of the hash unfolded into the list. We can exploit this to do hash literals. We simply write out the list pairs that we want placed into the hash. For example:

```
use strict;
my %table = qw/schmoe joe 7.5 1.6/;
```

would give us the same hash we had in the previous example.

5.4 Functions

You should realize that any function you already know that works on arrays will also work on hashes, since you can always evaluate a hash in the list context and get the pair list. However, there are a variety of functions that are specifically designed and optimized for use with hashes.

5.4.1 Keys and Values

When we evaluate a hash in a list context, Perl gives us the paired list that can be very useful. However, sometimes we may only want to look at the list of keys, or the list of values. Perl provides two optimized functions for doing this: `keys` and `values`.

```
use strict;
my %table = qw/schmoe joe smith john simpson bart/;
my @lastNames  = keys %table;    # @lastNames is: qw/schmoe smith simpson/
my @firstNames = values %table;  # @firstNames is: qw/joe john bart/
```

5.4.2 Each

The `each` function is one that you will find particularly useful when you need to go through each element in the hash. The `each` function returns each key-value pair from the hash one by one as a list of two elements. You can use this function to run a `while` across the hash:

```
use strict;
my %table = qw/schmoe joe smith john simpson bart/;
my($key, $value);  # Declare two variables at once
while ( ($key, $value) = each(%table) ) {
    # Do some processing on $key and $value
}
```

This `while` terminates because `each` returns `undef` when all the pairs have been exhausted. However, be careful. Any change in the hash made will “reset” the `each` function for that hash.

So, if you need to loop and change values in the hash, use the following `foreach` across the keys:

```
use strict;
my %table = qw/schmoe joe smith john simpson bart/;
foreach my $key (keys %table) {
    # Do some processing on $key and $table{$key}
}
```

5.5 Slices

It turns out you can slice hashes just like you were able to slice arrays. This can be useful if you need to extract a certain set of values out of a hash into a list.

```
use strict;
my %table = qw/schmoe joe smith john simpson bart/;
my @friends = @table{'schmoe', 'smith'};   # @friends has qw/joe john/
```

Note the use of the `@` in front of the hash name. This shows that we are indeed producing a normal list, and you can use this construct in any list context you would like.

5.6 Context Considerations

We have now discussed all the different ways you can use variables in list and scalar context. At this point, it might be helpful to review all the ways we have used variables in different contexts. The table that follows identifies many of the ways variables are used in Perl.

Expression	**Context**	**Variable**	**Evaluates to**
`$scalar`	scalar	`$scalar`, a scalar	the value held in `$scalar`
`@array`	list	`@array`, an array	the list of values (in order) held in `@array`
`@array`	scalar	`@array`, an array	the total number of elements in `@array` (same as `$#array + 1`)
`$array[$x]`	scalar	`@array`, an array	the `($x+1)`th element of `@array`
`$#array`	scalar	`@array`, an array	the subscript of the last element in `@array` (same as `@array -1`)
`@array[$x, $y]`	list	`@array`, an array	a slice, listing two elements from `@array` (same as `($array[$x], $array[$y])`)
`"$scalar"`	scalar (interpolated)	`$scalar`, a scalar	a string containing the contents of `$scalar`
`"@array"`	scalar (interpolated)	`@array`, an array	a string containing the elements of `@array`, separated by spaces
`%hash`	list	`%hash`, a hash	a list of alternating keys and values from `%hash`
`$hash{$x}`	scalar	`%hash`, a hash	the element from `%hash` with the key of `$x`
`@hash{$x, $y}`	list	`%hash`, a hash	a slice, listing two elements from `%hash` (same as `($hash{$x}, $hash{$y})`

6 Regular Expressions

One of Perl's original applications was text processing (see Section A.1 [A Brief History of Perl], page 45). So far, we have seen easy manipulation of scalar and list data is in Perl, but we have yet to explore the core of Perl's text processing construct—regular expressions. To remedy that, this chapter is devoted completely to regular expressions.

6.1 The Theory Behind It All

Regular expressions are a concept borrowed from automata theory. Regular expressions provide a a way to describe a "language" of strings.

The term, *language*, when used in the sense borrowed from automata theory, can be a bit confusing. A *language* in automata theory is simply some (possibly infinite) set of strings. Each string (which can be possibly empty) is composed of a set of characters from a fixed, finite set. In our case, this set will be all the possible ASCII characters[1].

When we write a regular expression, we are writing a description of some set of possible strings. For the regular expression to have meaning, this set of possible strings that we are defining should have some meaning to us.

Regular expressions give us extreme power to do pattern matching on text documents. We can use the regular expression syntax to write a succinct description of the entire, infinite class of strings that fit our specification. In addition, anyone else who understands the description language of regular expressions, can easily read out description and determine what set of strings we want to match. Regular expressions are a universal description for matching regular strings.

When we discuss regular expressions, we discuss "matching". If a regular expression "matches" a given string, then that string is in the class we described with the regular expression. If it does not match, then the string is not in the desired class.

6.2 The Simple

We can start our discussion of regular expression by considering the simplest of operators that can actually be used to create all possible regular expressions[2]. All the other regular expression operators can actually be reduced into a set of these simple operators.

6.2.1 Simple Characters

In regular expressions, generally, a character matches itself. The only exceptions are regular expression special characters. To match one of these special characters, you must put a \ before the character.

For example, the regular expression `abc` matches a set of strings that contain `abc` somewhere in them. Since `*` happens to be a regular expression special character, the regular expression `\*` matches any string that contains the `*` character.

[1] Perl will eventually support unicode, or some other extended character format, in which case it will no longer merely be ASCII characters.

[2] Actually, Perl regular expressions have a few additional features that go beyond the traditional, simple set of regular expressions, but these are an advanced topic.

6.2.2 The * Special Character

As we mentioned `*` is a regular expression special character. The `*` is used to indicate that zero or more of the previous characters should be matched. Thus, the regular expression `a*` will match any string that contains zero or more `a`'s.

Note that since `a*` will match any string with zero or more `a`'s, `a*` will match *all* strings, since all strings (including the empty string) contain at least zero `a`'s. So, `a*` is not a very useful regular expression.

A more useful regular expression might be `baa*`. This regular expression will match any string that has a `b`, followed by one or more `a`'s. Thus, the set of strings we are matching are those that contain `ba`, `baa`, `baaa`, etc. In other words, we are looking to see if there is any "sheep speech" hidden in our text.

6.2.3 The . Character

The next special character we will consider is the `.` character. The `.` will match any valid character. As an example, consider the regular expression `a.c`. This regular expression will match any string that contains an `a` and a `c`, with any possible character in between. Thus, strings that contain `abc`, `acc`, `amc`, etc. are all in the class of strings that this regular expression matches.

6.2.4 The | Character

The `|` special character is equivalent to an "or" in regular expressions. This character is used to give a choice. So, the regular expression `abc|def` will match any string that contains either `abc` or `def`.

6.2.5 Grouping with ()s

Sometimes, within regular expressions, we want to group things together. Doing this allows building of larger regular expressions based on smaller components. The `()`'s are used for grouping.

For example, if we want to match any string that contains `abc` or `def`, zero or more times, surrounded by a `xx` on either side, we could write the regular expression `xx(abc|def)*xx`. This applies the `*` character to everything that is in the parentheses. Thus we can match any strings such as `xxabcxx`, `xxabcdefxx`, etc.

6.2.6 The Anchor Characters

Sometimes, we want to apply the regular expression from a defined point. In other words, we want to anchor the regular expression so it is not permitted to match anywhere in the string, just from a certain point.

The anchor operators allow us to do this. When we start a regular expression with a `^`, it anchors the regular expression to the beginning of the string. This means that whatever the regular expression starts with *must be* matched at the beginning of the string. For example, `^aa*` will not match strings that contain one or more `a`'s; rather it matches strings that *start* with one or more `a`'s.

We can also use the `$` at the end of the string to anchor the regular expression at the end of the string. If we applied this to our last regular expression, we have `^aa*$` which now matches *only* those strings that consist of one or more `a`'s. This makes it clear that the regular expression cannot just look anywhere in the string, rather the regular expression must be able to match the entire string exactly, or it will not match at all.

In most cases, you will want to either anchor a regular expression to the start of the string, the end of the string, or both. Using a regular expression without some sort of anchor can also produce confusing and strange results. However, it is occasionally useful.

6.3 Pattern Matching

Now that you are familiar with some of the basics of regular expressions, you probably want to know how to use them in Perl. Doing so is very easy. There is an operator, `=~`, that you can use to match a regular expression against scalar variables. Regular expressions in Perl are placed between two forward slashes (i.e., `//`). The whole `$scalar =~ //` expression will evaluate to `1` if a match occurs, and `undef` if it does not.

Consider the following code sample:

```
use strict;
while ( defined($currentLine = <STDIN>) ) {
    if ($currentLine =~ /^(J|R)MS speaks:/) {
        print $currentLine;
    }
}
```

This code will go through each line of the input, and print only those lines that start with “JMS speaks:” or “RMS speaks:”.

6.4 Regular Expression Shortcuts

Writing out regular expressions can be problematic. For example, if we want to have a regular expression that matches all digits, we have to write:

```
(0|1|2|3|4|5|6|7|8|9)
```

It would be terribly annoying to have to write such things out. So, Perl gives an incredible number of shortcuts for writing regular expressions. These are largely syntactic sugar, since we could write out regular expressions in the same way we did above. However, that is too cumbersome.

For example, for ranges of values, we can use the brackets, `[]`'s. So, for our digit expression above, we can write `[0-9]`. In fact, it is even easier in perl, because `\d` will match that very same thing.

There are lots of these kinds of shortcuts. They are listed in the ‘`perlre`’ online manual. They are listed in many places, so there is no need to list them again here.

However, as you learn about all the regular expression shortcuts, remember that they can all be reduced to the original operators we discussed above. They are simply short ways of saying things that can be built with regular characters, `*`, `()`, and `|`.

7 Subroutines

Until now, all the Perl programs that we have written have simply a set of instructions, line by line. Like any good language, Perl allows one to write modular code. To do this, at the very least, the language must allow the programmer to set aside subroutines of code that can be reused. Perl, of course, provides this feature.

Note that many people call Perl subroutines "functions". We prefer to use the term "functions" for those routines that are built in to Perl, and "subroutines" for code written by the Perl programmer. This is not standard terminology, so you may hear others use subroutines and functions interchangeably, but that will not be the case in this book. We feel that it is easier to make the distinction if we have two different terms for functions and subroutines.

Note that user subroutines can be used anywhere it is valid to use a native Perl function.

7.1 Defining Subroutines

Defining a subroutine is quite easy. You use the keyword `sub`, followed by the name of your subroutine, followed by a code block. This friendly subroutine can be used to greet the user:

```
use strict;
sub HowdyEveryone {
   print "Hello everyone.\nWhere do you want to go with Perl today?\n";
}
```

Now, anywhere in the code where we want to greet the user, we can simply say:

```
&HowdyEveryone;
```

and it will print that message to the user. In fact, in most cases, the `&` for invoking subroutines is optional.

7.2 Returning Values

Perhaps we did not want our new subroutine to actually print the message. Instead, we would like it to return the string of the message, and then we will call `print` on it.

This is very easy to do with the `return` statement.

```
use strict;
sub HowdyEveryone {
   return "Hello everyone.\nWhere do you want to go with Perl today?\n";
}
print &HowdyEveryone;
```

7.3 Using Arguments

A subroutine is not much good if you cannot give it input on which to operate. Of course, Perl allows you to pass arguments to subroutines just like you would to native Perl functions.

At the start of each subroutine, Perl sets a special array variable, `@_`, to be the list of arguments sent into the subroutine. By standard convention, you can access these variables through `$_[0 .. $#_]`. However, it is a good idea to instead immediately declare a list of variables and assign `@_` to them. For example, if we want to greet a particular group of people, we could do the following:

```
use strict;
sub HowdyEveryone {
   my($name1, $name2) = @_;
   return "Hello $name1 and $name2.\n" .
          "Where do you want to go with Perl today?\n";
}
print &HowdyEveryone("bart", "lisa");
```

Note that since we used `my`, and we are in a new block, the variables we declared will live only as long as the subroutine execution.

This subroutine leaves a bit to be desired. It would be nice if we could have a custom greeting, instead of just "Hello". In addition, we would like to greet as many people as we want to, not just two. This version fixes those two problems:

```
use strict;
sub HowdyEveryone {
   my($greeting, @names) = @_;
   my $returnString;

   foreach my $name (@names) {
       $returnString .= "$greeting, $name!\n";
   }

   return $returnString .
          "Where do you want to go with Perl today?\n";
}
print &HowdyEveryone("Howdy", "bart", "lisa", "homer", "marge", "maggie");
```

We use two interesting techniques in this example. First of all, we use a list as the last parameter when we accept the arguments. This means that everything after the first argument will be put into `@names`. Note that had any other variables followed `@names`, they would have remained undefined. However, scalars before the array (like `$greeting`) do receive values out of `@_`. Thus, it is always a good idea to only make the array the last argument.

Appendix A Background of Perl

In this appendix, we introduce a brief historical, sociological and psychological reasons for why Perl works the way it does. We also include a brief history of Perl.

When learning a new language, it is often helpful to learn the history, motivations, and origins of that language. In natural languages such as English, this helps us understand the culture and heritage of the language. Such understanding leads to insight into the minds of those who speak the language. This newly found insight, obtained through learning culture and heritage, assists us in learning the new language.

This philosophy of language instruction can often be applied to programming languages as well. Although programming languages grow from a logical or mathematical basis, they are rarely purely mathematical. Often, the people who design, implement and use the language influence the language, based on their own backgrounds. Because of the influence the community has upon programming languages, it is useful, before learning a programming language, to understand its history, motivations, and culture. To that end, this chapter examines the history, culture, and heritage of the Perl language.

A.1 A Brief History of Perl

Larry Wall, the creator of Perl, first posted Perl to the '`comp.sources`' Usenet newsgroup in late 1987. Larry had created Perl as a text processing language for Unix-like operating systems. Before Perl, almost all text processing on Unix-like systems was done with a conglomeration of tools that included AWK, '`sed`', the various shell programming languages, and C programs. Larry wanted to fill the void between "manipulexity" (the ability of languages like C to "get into the innards of things") and "whipuptitude" (the property of programming languages like AWK or '`sh`' that allows programmers to quickly write useful programs).

Thus, Perl, the Practical Extraction and Report Language[1], was born. Perl filled a niche that no other tool before that date had. For this reason, users flocked to Perl.

Over the next four years or so, Perl began to evolve. By 1992, Perl version 4 had become very stable and was a "standard" Unix programming language. However, Perl was beginning to show its limitations. Various aspects of the language were confusing at best, and problematic at worst. Perl worked well for writing small programs, but writing large software applications in Perl was unwieldy.

The designers of the Perl language, now a group, but still under Larry's guidance, took a look around at the other languages that people were using. They seemed to ask themselves: "Why are people choosing other languages over Perl?" The outcome of this self-inspection was Perl, version 5.

The first release of version 5 came in late 1994. Many believed that version 5 made Perl "complete". Gone were the impediments and much of the confusion that were prevalent in version 4. With version 5, Perl was truly a viable, general purpose programming language and no longer just a convenient tool for system administrators.

[1] Perl has also been given the name, Pathologically Eclectic Rubbish Lister. Choose the one with which you feel more comfortable. Some even insist that Perl is no longer an acronym for anything. This argument has merit, since Perl is never written PERL.

A.2 Perl as a Natural Language

Natural languages, languages (such as English) that people use on a daily basis to communicate with each other, are rich and complete. Most natural languages allow the speaker to express themselves succinctly and clearly. However, most natural languages are also full of arcane constructs that carry over from the language's past. In addition, for a given natural language, it is impossible to fully master the vocabulary and grammar because they are very large, extremely complex, and always changing.

You may wonder what these facts about natural languages have to do with a programming language like Perl. Surprising to most newcomers to Perl, the parallels between Perl and a natural language like English are striking. Larry Wall, the father of Perl, has extensive scholastic training as a linguist. Larry applied his linguistic knowledge to the creation of Perl, and thus, to the new student of Perl, a digression into these language parallels will give the student insight into the fundamentals of Perl.

Natural languages have the magnificent ability to provide a clear communication system for people of all skill levels and backgrounds. The same natural language can allow a linguistic neophyte (like a three-year-old child) to communicate herself nearly completely to others, while having only a minimal vocabulary. The same language also provides enough flexibility and clarity for the greatest of philosophers to write their works.

Perl is very much the same. Small Perl programs are easy to write and can perform many tasks easily. Even the newest student of Perl can write useful Perl programs. However, Perl is a rich language with many features. This allows the creation of simple programs that use a "limited" Perl vocabulary, and the creation of large, complicated programs that seem to work magic.

When studying Perl, it is helpful to keep the "richness" of Perl in mind. Newcomers find Perl frustrating because subtle changes in syntax can produce deep changes in semantics. It can even be helpful to think of Perl as another natural language rather than another programming language. Like in a natural language, you should feel comfortable writing Perl programs that use only the parts of Perl you know. However, you should be prepared to have a reference manual in hand when you are reading code written by someone else.

The fact that one cannot read someone else's code without a manual handy and the general "natural language" nature of Perl have been frequently criticized. These arguments are well taken, and Perl's rich syntax and semantics can be confusing to the newcomer. However, once the initial "information overload" subsides, most programmers find Perl exciting and challenging. Discovering new ways to get things done in Perl can be both fun and challenging! Hopefully, you will find this to be the case as well.

A.3 The Slogans

Clearly, Perl is a unique language. This uniqueness has brought forth a community and an ideology that is unprecedented with other languages. One does not have to be a member of this community or agree with this ideology to use Perl, but it helps to at least understand the ideology to get the most out of Perl.

The common Perl slogans have become somewhat famous. These slogans make up the "Perl ethic"—the concepts that guide the way Perl itself is built, and the way most Perl programs are written.

"There's more than one way to do it". This slogan, often abbreviated TMTOWTDI (pronounced TIM-toady), is common among many programmers, but Perl takes this idea to its logical conclusion. Perl is rich with non-orthogonality and shortcuts. Most major syntactic constructs in Perl have two or three exact equivalents. This can be confusing to newcomers, but if you try to embrace this diversity rather than be frustrated by it, having "more than one way to do it" can be fun.

"The Swiss Army chain-saw". This is the somewhat "less friendly" summary of the previous term. Sometimes, all these diverse, powerful features of Perl make it appear that there are too many tools that are too powerful to be useful all together on one "Swiss Army knife". However, eventually, most Perl users find all these different "chain-saw"-style tools on one "Swiss Army" knife are a help rather than a hindrance.

"Perl makes easy jobs easy, and the hard jobs possible." This is a newer phrase in the Perl community, although it was originated by Alan Kay decades ago, but it is quite valid. Most easy tasks are very straightforward in Perl. As the saying goes, most programmers find that there are very few jobs that Perl cannot handle well. However, despite what the saying might indicate, Perl is not a panacea; the programmer should always choose the right tool for the job, and that right tool may not always be Perl.

"Perl promotes laziness, impatience and hubris." These seem like strange qualities to be promoting, but upon further analysis, it becomes clear why they are important.

Lazy programmers do not like to write the same code more than once. Thus, a lazy programmer is much more likely to write code to be reusable and as applicable in as many situations as possible.

Laziness fits well with impatience. Impatient programmers do not like to do things that they know very well the computer could do for them. Thus, impatient programmers (who are also lazy) will write programs to do things that they do not want to have to do themselves. This makes these programs more usable for more tasks.

Finally, laziness and impatience are insufficient without hubris. If programmers have hubris, they are much less likely to write unreadable code. A good bit of hubris is useful—it makes programmers want to write code that they can show off to friends. Thus, hubris, when practiced in the conjunction with laziness and impatience, causes programmers to write reusable, complete and readable code. In other words, it is possible to exploit these three "bad" traits to obtain a "good" outcome.

Appendix B GNU Free Documentation License

B.1 ADDENDUM: How to use this License for your documents

General Index