CHAPTER

8

The Basics of Perl

Our primary interest in Perl in this book is its widespread use for Common Gateway Interface (CGI) programming. However, it is beneficial to most programmers to become familiar with Perl's capabilities. Perl is a flexible, powerful, widely used programming language—and it would be so even if CGI programming did not exist. This chapter takes you on a quick tour of Perl, introducing most of the important concepts and constructs but leaving out many of the details of the language. In spite of its brevity, however, if you are an experienced programmer, you can learn to write useful Perl programs by studying this chapter. The similarity of Perl to other common programming languages, especially C and JavaScript, makes this relatively easy, at least for those who know one of those languages. If you need more details, there are numerous books dedicated to Perl.

This chapter begins with a description of Perl's scalar values and variables and their use in expressions and assignment statements. Next, it covers control expressions and the collection of control statements available in Perl. Then it introduces Perl's two built-in data structures, arrays and hashes, and references. This is followed by a description of functions and how they are defined and called, including the peculiar way (for a high-level language) that parameters are passed. Finally, the chapter covers Perl's pattern matching and the basic operations of file input and output. Although we attempt to describe Perl in a single chapter, do not be misled into thinking that this is a small or simple language—it is neither.

8.1 Origins and Uses of Perl

Perl began as a relatively small language with the modest purpose of including and expanding on the operations of the text-processing language `awk` and the system administration capabilities of the UNIX shell languages, initially `sh`. Perl was first released in 1987 after being developed and implemented by Larry Wall. Since then it has grown considerably, borrowing features from other languages as well as inventing a few of its own.[1] Along the way, it picked up support for communications using sockets, a module construct, and support for object-oriented programming. Its text pattern-matching capabilities are elaborate and powerful, which is one of the reasons it became the most popular language for CGI programming. Perl's pattern matching has been copied into several other languages, including JavaScript, Ruby, and PHP. Perl is now used for many of the small- to medium-size programming projects formerly done in C.

Perl is a language whose implementation is between compiled (to machine code) and interpreted languages. Perl programs are compiled to an intermediate form, in part to check for errors, but mostly to make possible impressive run-time performance even though it is interpreted. When we refer to `perl`, we mean the Perl language processing system, which compiles and interprets programs. The `perl` system includes a debugger, among other things.

Perl has been ported to every common computing platform, from the various versions of UNIX to Windows and everything in between. Most versions of UNIX come with Perl, and Mac OS X has Perl installed.

8.2 Scalars and Their Operations

Perl has three categories of variables—scalars, arrays, and hashes—each identified by the first character of their names (`$` for scalar variables, `@` for array variables, and `%` for hash variables). This section discusses the important characteristics of the most commonly used kind of variables, namely, scalars. Scalar variables can store three different kinds of values: numbers, character strings, and references, which are addresses. We postpone discussing references until Section 8.7.

1. See `http://history.perl.org/` for more on the evolution of Perl.

The numeric values stored in scalar variables are represented internally in double-precision floating-point form. Although there is a way to suggest (to the compiler) that integer operations be used on scalar numeric values, in most cases the operations are done in double-precision floating point.

8.2.1 Numeric and String Literals

Numeric literals can have the forms of either integers or floating-point values. Integer literals are strings of digits. Floating-point literals can have either decimal points or exponents or both. Exponents are specified with an uppercase or lowercase `e` and a possibly signed integer literal. The following are legal numeric literals:

```
72  7.2  .72  72.  7E2  7e2  .7e2  7.e2  7.2E-2
```

Integer literals can be written in hexadecimal (base 16) by preceding their first digit with either `0x` or `0X`.

String literals can appear in two forms, depending on whether their delimiters are single quotes (`'`) or double quotes (`"`). Single-quoted string literals cannot include characters specified with escape sequences, such as newline characters specified with `\n`.[2] If an actual single-quote character is needed in a string literal that is delimited by single quotes, the embedded single quote is preceded by a backslash, as shown in the following example:

```
'You\'re the most freckly person I\'ve ever met'
```

If an escape sequence is embedded in a single-quoted string literal, each character in the sequence is taken literally as itself. For example, the sequence `\n` in the following string literal will be treated as two characters, a backslash and an `n`:

```
'You have freckles, \n but I don\'t'
```

If a string literal with the same characteristics as single-quoted strings is needed but you want to use a different delimiter, precede the delimiter with `q`, as shown in the following example:

```
q$I don't want to go, I can't go, I won't go!$
```

If the new delimiter is a parenthesis, a brace, a bracket, or an angle bracket, the left element of the pair must be used on the left, and the right element must be used on the right. For example:

```
q<I don't want to go, I can't go, I won't go!>
```

Double-quoted string literals differ from single-quoted string literals in two ways: First, they can include special characters specified with escape sequences; second, embedded variable names are interpolated into the string, which means that their values are substituted for their names. We discuss the first of these differences here; the other will be discussed in Section 8.2.2.

---

2. An *escape sequence* is one or two special characters followed by another character. The special character or characters change the meaning of the following character for that single appearance. For example, in some contexts, `\n` means a newline character, not a backslash followed by the letter *n*.

In many situations, we want to include special characters that are specified with escape sequences in string literals. For example, if we want the words on a line to be spaced by tabs, we use a double-quoted literal with embedded escape sequences for the tab character:

```
"Completion % \t Yards \t Touchdowns \t Interceptions"
```

A double quote can be embedded in a double-quoted string literal by preceding it with a backslash.

A different delimiter can be specified for string literals with the characteristics of double-quoted strings by preceding the new delimiter with `qq`:

```
qq@"Why, I never!", said she.@
```

The null string (one with no characters) can be expressed as `''` or `""`.

8.2.2 Scalar Variables

The names of all scalar variables, whether predefined or programmer defined, begin with dollar signs (`$`). The part of a scalar variable's name that follows the dollar sign is similar to the names of variables in other programming languages. It begins with a letter, which can be followed by any number of letters, digits, or underscore characters. There is no limit to the length of a variable name, and all of its characters are significant.[3] The letters in a variable name are case sensitive, meaning that `$FRIZZY`, `$Frizzy`, `$FrIzZy`, and `$friZZy` are all distinct names, although using all four of these in the same program would be frowned upon by most sensible people. However, by convention, programmer-defined variable names do not include uppercase letters.

As mentioned earlier, double-quoted string literals that contain the names of variables have the values of those variables included in, or interpolated into, the string. Consider the following string literal:

```
"Jack is $age years old"
```

If the value of `$age` is `47`, this string has the following value:

```
"Jack is 47 years old"
```

To place a variable name in a double-quoted string but not have it interpolated, the variable's name is preceded by a backslash. For example:

```
"The variable with the result is \$answer"
```

In Perl, variables often are not explicitly declared; the compiler declares a variable implicitly when it first encounters the variable's name in a program.[4]

A scalar variable that has not been assigned a value by the program has the value `undef`. The numeric value of `undef` is `0`; the string value of `undef` is the null string (`""`).

---

3. Well, at least no practical limit. Actually, a name can have no more than 255 characters.

4. Because a variable's name clarifies its type, explicit variable declarations would serve little purpose, except in functions (see Section 8.8).

Perl includes a large number of predefined, or *implicit*, variables. The names of implicit scalar variables begin with dollar signs. The rest of the name of an implicit variable is often just one more special character, such as an underscore (_), a circumflex (^), or a backslash (\). You will see many uses of these implicit variables in this chapter and the next.

The Perl software system includes extensive documentation. Specific parts of this documentation can be retrieved with the command `perldoc` followed by a topic. For example, a list of all Perl predefined variables can be found by typing the following:

```
perldoc perlvar
```

8.2.3 Numeric Operators

Most of Perl's numeric operators are similar to those in other common programming languages, so they should be familiar to most readers. They are the binary operators + for addition, - for subtraction, * for multiplication, / for division, ** for exponentiation, and % for modulus.[5] In addition, there are the unary operators for addition, subtraction, decrement (--), and increment (++). The decrement and increment operators can be either prefix or postfix.

Except under unusual circumstances, numeric operations are done in double-precision floating point. So, whereas the expression `5 / 2` may evaluate to `2` in many other languages, in Perl it evaluates to `2.5`.

The precedence rules of a language specify which operator is evaluated first when two operators that have different levels of precedence appear in an expression, separated only by an operand. The associativity rules of a language specify which operator is evaluated first when two operators with the same precedence level appear in an expression, separated only by an operand. The precedence and associativity of the numeric operators are given in Table 8.1.

Table 8.1 Precedence and associativity of the numeric operators

Operator	Associativity
++, --	Nonassociative*
unary +, -	Right
**	Right
*, /, %	Left
binary +, -	Left

The operators listed first have the highest precedence.
*An operator is nonassociative if two of them cannot appear in an expression separated only by an operand.

5. `$x % $y` produces the remainder of the value of `$x` after division by `$y`.

8.2.4 String Operators

Perl strings are not stored or treated as arrays of characters; rather, a string is a single unit. The two string operators are described in this section. The most commonly used string functions are introduced in Section 8.2.5.

String catenation is specified with the operator denoted by a period. For example, if the value of `$first` is `"Freddie"`, the value of the expression

```
$first . " Freeloader"
```

is

```
"Freddie Freeloader"
```

The repetition operator is specified with an `x`. It takes a string as its left operand and an expression that evaluates to a number as its right operand. The left operand is replicated the number of times equal to the value of the right operand. For example, the value of

```
"More! " x 3
```

is

```
"More! More! More! "
```

8.2.5 String Functions

Functions and operators in Perl are closely related. In fact, in many cases they can be used interchangeably. For example, if there is a predefined unary operator named `doit` that takes one parameter, it can be treated as an operator, as shown in the following example:

```
doit x
```

Or it could be treated as a function, as shown in the following:

```
doit(x)
```

A function with no parameters can be called with empty parentheses or no parentheses at all.

Table 8.2 lists the most commonly used string functions.

When an operator or function that expects a numeric operand is given a string, the string is implicitly converted (coerced) to a number. If the string does not represent a number, zero is used. When an operator or function that expects a string operand is given a number, the number is coerced to a string.

8.3 Assignment Statements and Simple Input and Output

Among the most fundamental constructs in most programming languages are assignment statements and the statements or functions that provide keyboard input and screen output. The next subsections introduce these as they appear in Perl.

Table 8.2 String functions

Name	Parameter(s)	Actions
`chomp`	A string	Removes any terminating newline characters* from its parameter; returns the number of removed characters
`length`	A string	Returns the number of characters in its parameter string
`lc`	A string	Returns its parameter string with all uppercase letters converted to lowercase
`uc`	A string	Returns its parameter string with all lowercase letters converted to uppercase
`hex`	A string	Returns the decimal value of the hexadecimal number in its parameter string
`join`	A character and the strings catenated together with a list of strings	Returns a string constructed by catenating the strings of the second and subsequent strings together, with the parameter character inserted between them

*The newline character may actually be two characters, as is the case with Windows.

8.3.1 Assignment Statements

Perl's assignment statements are the same as those of C and its descendants. The simple assignment operator is =, used in the following example:

```
$salary = 47500;
```

Compound assignment operators are binary operators with the simple assignment operator catenated to their right side. For example, the statement

```
$sum += $value;
```

is the same as this statement:

```
$sum = $sum + $value;
```

Notice that these sample assignment statements are terminated by semicolons. All Perl statements except those at the end of blocks (see Section 8.4) must be terminated by semicolons.

Comments in Perl are specified using the pound sign (#). Any text following a # on a line is ignored by the compiler (unless the # is in a literal string).

8.3.2 Keyboard Input

All input and output in Perl is uniformly thought of as file input and output. Files have external names but are referenced in programs through internal names, called *filehandles*. There are three predefined filehandles: `STDIN`, which is a program's normal input stream (by default, the keyboard); `STDOUT`, which is a program's normal output stream (by default, the screen); and `STDERR`, which is

a program's normal output stream for error messages (usually also associated with the screen).

The line input operator[6] is different from other operators. It is specified with a pair of angle brackets <>, with its operand, if one is provided, embedded between the brackets. For example, you get a line from `STDIN` as follows:

```
$in_data = <STDIN>;
```

The line input operator gets all characters typed on the keyboard up to and including the newline character. In many cases, the newline character is not wanted, so the following idiom is common:

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

8.3.3 Screen Output

Output is directed to the screen with the `print` function (or operator). We prefer to treat `print` as an operator. The operand for `print` is one or more string literals, separated by commas. No implicit newline character is appended to the last string operand, so if one is needed, it must be included, as shown in the following:

```
print "This is pretty easy \n";
```

Many screen output lines include the value of one or more program variables. Because the variable names in a double-quoted string are interpolated, this is easy. The `printf` function from C is also available in Perl, including format codes such as `%7d` and `%5s`.

The following trivial program illustrates some of what we have discussed so far:

```
# quadeval.pl - A simple Perl program
#  Input: Four numbers, representing the values of
#       a, b, c, and x
# Output: The value of the expression
#       axx + bx + c
# Get input
print "Please input the value of a ";
$a = <STDIN>;
print "Please input the value of b ";
$b = <STDIN>;
print "Please input the value of c ";
$c = <STDIN>;
```

6. Note that the line input operator is also used to get input from places other than the keyboard, as we explain later. The line input operator is sometimes called the *angle operator*; it is also sometimes called the *diamond operator*.

```
print "Please input the value of x ";
$x = <STDIN>;
# Compute and display the result
$result = $a * $x * $x + $b * $x + $c;
print "The value of the expression is: $result \n";
```

Under Windows and UNIX, a Perl program can be run from the operating system command-line prompt, by typing `perl` followed by the filename where the Perl program is stored, as shown in the following:

```
perl quadeval.pl
```

This executes the program named `perl`, which is the Perl compiler/interpreter. It causes the program in the file `quadeval.pl` to be compiled and interpreted. If you want a compilation without the interpretation, just to check the syntactic correctness of your program, include the `-c` flag after the `perl` command. It is always a good idea to include the `-w` flag, which causes `perl` to produce warning messages for a variety of suspicious things it may find in your program. This is useful because Perl is a very forgiving language.

The line input operator can be used to get input from a file specified as a command-line argument, which can appear at the end of the `perl` command. The operator in this case is empty angle brackets. For example, if `perl` is run with the following command:

```
perl –w quadeval.pl quad.dat
```

then the following statement puts the first line from the file `quad.dat` into `$input`:

```
$input = <>;
```

An alternative way to run Perl programs is discussed in Chapter 9, "Using Perl for CGI Programming."

8.4 Control Statements

Perl has a powerful collection of statements for controlling the execution flow through its programs. This section introduces the control expressions and control statements of Perl.

8.4.1 Control Expressions

The expressions upon which statement control flow is based are either scalar-valued expressions, relational expressions, or compound expressions. If the value of a scalar-valued expression is a string, it is true unless it is either the empty

string ("") or a zero string ("0").[7] If the value is a number, it is true unless it is zero (0).

Relational operators can have any scalar-valued expression for their operands. Perl has two sets of relational operators, one for numeric operands and one for string operands. Table 8.3 lists the relational operators.

Table 8.3 Relational operators

Operation	Numeric Operands	String Operands
Is equal to	==	eq
Is not equal to	!=	ne
Is less than	<	lt
Is greater than	>	gt
Is less than or equal to	<=	le
Is greater than or equal to	>=	ge
Compare, returning -1, 0, or +1	<=>	cmp

If a string relational operator is given a numeric operand, the value of that operand is coerced to a string. Likewise, the numeric operators coerce string operands to numbers. In some cases, the value produced by a coercion may not be intuitive. For example, when the string 'George' is used as an operand of >, it is coerced to zero, because zero is produced whenever the string cannot be coerced to a number. Coercions do not produce error or warning messages when an operand must be coerced, even if the coercion is required only because the programmer typed the wrong operand. These coercions are obviously dangerous because they prevent the system from detecting some programmer errors, so be careful to use the relational operator that is appropriate for the operands.

The first six relational operators in Table 8.3 produce +1 if true, "" if false.

The <=> and cmp operators compare their operands and produce -1 if the left operand is less than the right operand, 0 if they are equal, and +1 if the left operand is greater than the right operand. These operators are useful with the sort operator, which is discussed in Section 8.8.4.

Perl has two sets of operators for the AND, OR, and NOT Boolean operations. These two sets have the same semantics but different precedence levels. The operators with the higher precedence level are && (AND), || (OR), and ! (NOT). Those with the lower precedence are and, or, and not. The precedence of these latter operators is lower than any other operators in Perl, so

7. This can be misinterpreted. The string "0.0" may look like a false value, but because it is not precisely "0", it is true.

regardless of what operators appear in their operands, these operators will be evaluated last.

The precedence and associativity of all operators discussed so far in this chapter are shown in Table 8.4.

Table 8.4 Operator precedence and associativity

Operator	Associativity
`++, --`	Nonassociative
`**`	Right
unary `+`, unary `-`	Right
`*, /, %, x`	Left
`+, -, .`	Left
`>, <, >=, <=, lt, gt, le, ge`	Nonassociative
`==, !=, <=>, eq, ne, cmp`	Nonassociative
`&&`	Left
`||`	Left
`=, +=, -=, *=, **=, /=, %=, &=, &&=, ||=, x=`	Right
`not`	Right
`and`	Left
`or`	Left

Highest-precedence operators are listed first.

Because assignment statements have values (the value of an assignment is the value assigned to the left-side variable), they can be used as control expressions. One common use of this is for an assignment statement that uses `<STDIN>` as its right side. The line input operator returns the empty string when it gets the end-of-file (EOF) character, so this can be conveniently used to terminate loops. For example:

```
while ($next = <STDIN>) { … }
```

The keyboard EOF character is Ctrl+D for UNIX, Ctrl+Z for Windows, and Cmd+. (period) for Macintosh systems.

8.4.2 Selection and Loop Statements

Control statements require some syntactic container for sequences of statements whose execution they are meant to control. Perl uses the block for this container. A block is formed by putting braces (`{}`) around a sequence of state-

ments. Blocks can have local variables, as you will see in Section 8.8. A *control construct* is a control statement and the block whose execution it controls.

Perl's `if` statement is similar to that of other languages. The only thing a bit different is that both the then clause and the else clause must always be blocks. For example, the following construct is illegal:

```
if ($a > 10)
    $b = $a * 2;  # Illegal – not a block
```

To be legal, the then clause must be a block, as shown in this example:

```
if ($a > 10) {
    $b = $a * 2;
}
```

An `if` construct can include `elsif` (note that it is *not* spelled "elseif") clauses, which provide a way of having a more readable sequence of nested `if` constructs. For example:

```
if ($snowrate < 1) {
    print "Light snow \n";
} elsif ($snowrate < 2) {
    print "Moderate snow \n";
} else {
    print "Heavy snow \n";
}
```

Perl has an `unless` statement, which is the same as its `if` statement except that the inverse of the value of the control expression is used. This is convenient if you want a selection construct with an else clause but no then clause. The following construct illustrates an `unless` statement:

```
unless ($sum > 1000) {
    print "We are not finished yet! \n";
}
```

The Perl `while` and `for` statements are similar to those of C and its descendants. The bodies of both must be blocks. The general form of the `while` statement is as follows:

`while` (*control expression*) {
  loop body statement(s)
}

The `until` statement is similar to the `while` statement except that the inverse of the value of the control expression is used.

Perl's `for` statement is most often used for loops controlled by counters. The general form of the `for` statement is as follows:

`for` (*initial expression*; *control expression*; *increment expression*) {
  loop body statement(s)
}

Both the initial expression and the increment expression can be multiple expressions, separated by commas, in which case the value of the whole expression is that of the expression following the last comma. The following `for` statement illustrates its use in forming a simple counter-controlled loop:

```
$sum = 0;
for ($counter = 1; $counter <= 100; $counter++) {
    $sum += $counter;
}
```

The operators `last` and `next` provide a way to exit a loop (or any block). These are exactly like the `break` and `continue` statements of C except they can include labels, which allow more than one loop or block to be exited. Consider the following skeletal example:

```
BIGLOOP:
while (...) {
    while (...) {
        ...
        if (...) {last BIGLOOP;}
        ...
    }
    ...
}
```

In this code, the `last` operator transfers control out of both loops.

Perl has no `switch` statement.

The implicit variable `$_` is frequently used in Perl programs, most often as the default parameter in a function call and as the default operand of an operator. It is also the default target for the input operator. For example, consider the following statement:

```
<STDIN>;
```

This statement gets a line from the keyboard and assigns it to `$_`.

The following example illustrates some of the uses of `$_`:

```
while (<STDIN>) {
    print;
    chomp;
    if ($_ eq "redhead") {
        print "I've finally found one! \n";
    }
}
```

There are three uses of `$_` in this `while` construct: as the target of the input line operator, as the default parameter to `print`, and as the default operand of `chomp`. As you might suspect, the heavy use of `$_`, especially when it is only implied, is not highly regarded by some software developers who normally use other programming languages.

8.5 Fundamentals of Arrays

Arrays in Perl are more flexible than those of most of the other common languages. This is a result of three fundamental differences between Perl arrays and those of other common languages such as C, C++, and Java. First, the length of a Perl array is dynamic—it can grow or shrink any time during program execution. Second, Perl arrays can have absent elements. For example, a Perl array may have only three elements, but those elements may have the subscripts `10`, `100`, and `352`. Third, a Perl array can store different types of scalar data. For example, an array may have some numeric elements and some string elements.

8.5.1 List Literals

A *list* is an ordered sequence of scalar values. A *list literal*, which is a parenthesized list of scalar values, is the way a list value is specified in a program. Arrays store lists, so list literals serve as array literals. Because each list element can be any kind of scalar value, a list literal can be any combination of them. An expression can also be used to describe an element of a list literal. For example:

```
(3.14149 * $radius, "circles", 17)
```

8.5.2 Arrays

We use the term *array* to denote a variable that stores lists. All array names begin with an at sign (`@`), which puts them in a namespace that is different from that of the scalar variable names. Arrays can be assigned list literals or other arrays. Consider the following examples:

```
@list =  ('boy', 'girl', 'dog', 'cat');
@creatures = @list;
```

When an array is assigned to another array, as in the second assignment statement above, a new array is created for the target variable (`@creatures` in the example). This is different from C, where array names without subscripts are treated as pointers, and an array assignment such as the preceding example would result in the target variable being set to the address of the assigned array. In Perl, array names without subscripts are never treated as pointers.

The context of an assignment statement depends on the type of the target variable—if it is a scalar, the context is scalar; if it is an array or a list, it is called list context. If an array is used in scalar context, the array's length (the number of elements that are in the array) is used. For example, the following statement assigns `4` to `$len` because there are four elements in (the previously defined) `@list`:

```
$len = @list;
```

This is a rather odd way to get the length of an array. It is important to remember that it is easy to put an array in scalar context by mistake, and Perl will not detect it as an error.

A list literal that contains only scalar variable names can be the target of a list assignment:

```
($first, $middle, $last) = ("George", "Bernard", "Shaw");
```

Such an assignment is called a *list assignment*. In the example, list assignment is used as shorthand for the following:

```
$first = "George";
$second = "Bernard";
$third = "Shaw";
```

If the target of a list assignment includes an array, all remaining values on the right side go to the array. Therefore, if an array is in the target of a list assignment, it should appear last.

All Perl array elements use integers as subscripts, and the lower-bound subscript of every array is zero. Array elements are referenced through subscripts delimited by brackets (`[ ]`). A subscript can be any numeric-valued expression. Because an array element is always a scalar, the scalar version of the array's name is used when a subscript is attached:

```
@list = (2, 4, 6, 8);
$second = $list[1];  # Sets $second to 4
```

Though it makes some sense, this use of a scalar name to reference an element of an array is confusing to most beginning Perl programmers.

It is essential to remember that scalar and array variables are in distinct namespaces. When a scalar name is followed by a subscript, it is no longer in the scalar namespace—the subscript moves it to the array namespace. Therefore, there is no connection between `$a` and `$a[5]`.

The length of an array is dynamic; the highest subscript to which a value has been assigned determines the current length of the array:

```
@list = ("Monday", "Tuesday", "Wednesday", "Thursday");
$list[4] = "Friday";
```

Here, the length of `@list` is now 5. Note that the length of an array is determined by the highest used subscript, which may be unrelated to the number of elements in the array.

Consider the following code:

```
@list = (2, 4, 6);
$list[27] = 8;
```

Now `@list` has four elements, but its length is 28. It has 24 vacant positions, which do not have elements.

The last subscript of `@list` can be referenced as `$#list`. So, the length of `@list` is `$#list + 1`. The last subscript of an array can be assigned to set its length to whatever you want, as shown in the following example:

```
$#list = 999;
```

As discussed previously, two different contexts of a variable name or expression exist: scalar and list. An expression assigned to a scalar variable is in scalar context; an expression assigned to an array or list is in list context. Some of Perl's operators force either scalar or list context on their operands. Likewise, some functions force either scalar or list context on their parameters. Scalar context can be forced with the pseudo function `scalar`, as shown here:

```
scalar(@list)
```

There is no way to force list context on an expression.

8.5.3 The foreach Statement

The `foreach` statement is used to process the elements of an array. For example, the following code will divide all of the values in `@list` by 2:

```
foreach $value (@list) {
    $value /= 2;
}
```

The scalar variable in a `foreach` becomes an alias (a new name) for each of the array's elements, one at a time. So, although the assignment statement in this `foreach` body appears to be changing the scalar variable `$value`, it is in fact changing the values of the elements of the array `@list`.

The scalar variable that appears in the `foreach` statement is local to the `foreach` construct. So, if the program has another variable with the same name, the one in the `foreach` will not interfere with the one used outside the construct.

If the array specified in a `foreach` statement has vacant spaces, `foreach` behaves as if the vacant elements existed and had the value `""`. For example, suppose an array is defined as follows:

```
$list[1] = 17;
$list[3] = 34;
```

Now consider the following loop:

```
foreach $value (@list) {
    print "Next: $value \n";
}
```

This loop produces the following:

```
Next:
Next: 17
Next:
Next: 34
```

8.5.4 Built-In Array Functions

As with other Perl functions, these can be treated as either operators or functions—operators if the parameters are not parenthesized, functions otherwise. It is common to need to place new elements on one end or the other of an array. Perl has four functions for these purposes: `unshift` and `shift`, which deal with the left end of arrays (lowest subscript); and `pop` and `push`, which deal with the right end of arrays (highest subscript).

The `shift` function removes and returns the first element of its given array parameter. For example, the following statement removes the first element of `@list` and places it in `$first`:

```
$first = shift @list;
```

The subscripts of all of the other elements in the array are reduced by 1 as a result of the `shift` operation. The `pop` function removes and returns the last element of its given array operand. In this case, there is no change in the subscripts of the array's other elements.

The `unshift` function takes two parameters: an array and a scalar or list. The scalar or list is appended to the beginning of the array. This results in an increase in the subscripts of all other array elements. The `push` function also takes an array and a scalar or a list. The scalar or list is added to the high end of the array:

```
@list = (2, 4, 6);
push @list, (8, 10);
```

The value of `@list` is now `(2, 4, 6, 8, 10)`.

Either `pop` and `unshift` or `push` and `shift` can be used to implement a queue in an array, depending on the direction the queue is intended to grow.

8.5.5 Built-In List Functions

List functions take lists as parameters. Because lists are immutable, these functions cannot change their list parameters. Instead, they yield new lists.

The `split` function is used to break strings into parts using a specified character as the basis for the split. The resulting substrings are placed in a specified array, as shown in the following example:

```
$stoogestring = "Curly Larry Moe";
@stooges = split(" ", $stoogestring);
```

The three elements of `@stooges` are now `"Curly"`, `"Larry"`, and `"Moe"`.

The `sort` function takes an array parameter and uses a string comparison to sort the elements of the array alphabetically in the returned list. For example:

```
@new_names_list = sort @names_list;
```

Note that `sort` does not alter its parameter, so in this statement, `@names_list` is unchanged.

Recall that if either operand of a string comparison (relational) operator is not a string, it is coerced to a string before the comparison operation takes place. It is possible to supply a comparison function to allow `sort` to sort all kinds of data. Do not use the `sort` operator to sort numbers if you do not supply a comparison function. If you do, `sort` will convert the numbers to strings, and you probably will not get what you expected. Section 8.8.4 discusses the use of the sort function for other orders and for nonstring array elements.

The `qw` function can be used on a sequence of unquoted strings to quote all of them, as shown in the following example:

```
qw(peaches apples pears kumquats)
```

This call to `qw` produces the following:

```
("peaches", "apples", "pears", "kumquats")
```

Notice that spaces, not commas, separate the list elements in the call to `qw`.

The `die` list operator is similar in form to the `print` function, which also can be considered a list operator. It takes a variable number of string parameters, catenates them, sends the result to `STDERR`, and terminates the program. The implicit variable `$!` stores the number of the most recent error that has occurred. It is useful for debugging to include `$!` in the parameter to `die`. For example:

```
die "Error -- division by zero in function fun2 - $!";
```

8.5.6 An Example

The following example illustrates a simple use of an array. A file of names, whose name is specified on the command line, is read. The names are converted to all uppercase letters, and the array is sorted and displayed.

```
# process_names.pl - A simple program to illustrate
#                    the use of arrays
#  Input: A file, specified on the command line, of
#         lines of text, where each line is a person's
#         name
# Output: The input names, after all letters are converted
#         to uppercase, in alphabetical order

$index = 0;

#>>> Loop to read the names and process them
while($name = <>) {

#>>> Convert the name's letters to uppercase and put it in
#>>>  the names array
    $names[$index++] = uc($name);
```

```
}

#>>> Display the sorted list of names
print "\nThe sorted list of names is:\n\n\n";
foreach $name (sort @names) {
    print ("$name \n");
}
```

8.6 Hashes

Associative arrays are arrays in which each data element is paired with a key, which is used to find the data element. Because hash functions are used to find specific elements in an associative array, Perl associative arrays are called *hashes*. The two fundamental differences between arrays and hashes are as follows: First, arrays use numeric subscripts to address specific elements, whereas hashes use string values (the keys) for element addressing. Second, the elements in arrays are ordered by subscript, but the elements in hashes are not. In a sense, elements of an array are like those in a list, whereas elements of a hash are like those in a set, where order is irrelevant. The actual arrangement of the elements of a hash in memory is determined by the internal hash function used to insert and access them.

Names of hash variables begin with percent signs (%), which places them in their own namespace. List literals may be used to initialize hash variables. The symbols => can be used between a key and its associated data element, or value, in a list literal used to initialize a hash variable. Commas can also be used, but they do nothing to connote the fact that it is a literal hash value:

```
%kids_ages = ("John" => 38, "Genny" => 36, "Jake" => 22,
              "Darcie" => 21);
```

Arrays also can be assigned to hashes, with the sensible semantics that the odd-subscripted elements of the array become the values of the hash, and the even-subscripted elements of the array become the keys of the hash. Hashes also can be assigned to arrays. When an array is assigned to a hash or a hash is assigned to an array, a copy process is used. Therefore, after an array is assigned to a hash, for example, subsequent changes to the hash do not affect the array.

An individual value element of a hash can be referenced by "subscripting" the hash name with a key. Braces are used to specify the subscripting operation. The name of a reference to a hash element begins, of course, with a dollar sign instead of the percent sign (because it is a scalar value):

```
$genny_age = $kids_ages{"Genny"};
```

New values are added to a hash by assigning the value of the new element to a reference to the key of the new element, as shown in the following example:

```
$kids_ages{"Aidan"} = 7;
```

An element is removed from a hash with the `delete` operator, as shown here:

```
delete $kids_ages{"Genny"};
```

A hash can be set to empty in two ways: First, an empty list can be assigned to the hash. Second, the `undef` operator can be used on the hash. These are illustrated with the following statements:

```
%kids_ages = ();
undef %kids_ages;
```

The `exists` operator is used to determine whether an element with a specific key is in a hash. For example:

```
if (exists $kids_ages{"Freddie"}) ..
```

The keys and values of a hash can be extracted into arrays with the operators `keys` and `values`, respectively:

```
foreach $child (keys %kids_ages) {
  print "The age of $child is $kids_ages{$child} \n";
}
@ages = values %kids_ages;
print "All of the ages are: @ages \n";
```

If a hash variable is embedded in a double-quoted string literal, its keys and values are not interpolated into the string. To display all of the keys and values of a hash, first assign it to an array and then print the array.

Perl has a predefined hash named `%ENV` that stores operating system environment variables. Environment variables are used to store information about the system on which `perl` is running. The environment variables and their respective values in `%ENV` can be accessed by any Perl program. The keys of `%ENV` are the names of the environment variables; the values are, of course, their values. All of the environment variables and their values of a specific system can be displayed with the following:

```
foreach $key (sort keys %ENV) {
    print "$key = $ENV{$key} \n";
}
```

In Chapter 9, "Using Perl for CGI Programming," we will make use of environment variables.

8.7 References

A *reference* is a scalar variable that references another variable or a literal. So, the value of a reference variable is an address. Although Perl's references are related to the pointers in C and C++, they are less flexible and much safer. The address

of an existing variable is obtained with the backslash operator on the name of that variable:

```
$age = 42;
$ref_age = \$age;
@stooges = ("Curly", "Larry", "Moe");
$ref_stooges = \@stooges;
```

A reference to a list literal can be created by putting the literal value in brackets, as follows:

```
$ref_salaries = [42500, 29800, 50000, 35250];
```

A reference to a hash literal is created by putting the literal value in braces:

```
$ref_ages = {
    'Curly' => 41,
    'Larry' => 38,
    'Moe' => 43,
};
```

A reference can specify two different values: its own, which is an address, or the value at that address. The process of making an appearance of a reference variable specify the latter is called *dereferencing*. All dereferencing in Perl is explicit. So, if you want the value to which a reference points rather than the address value of the reference, you must use different syntax on the reference's name. There are two ways to do this in Perl. First, an extra dollar sign can be appended to the beginning of the reference's name. For example, the value of `$$ref_age` is `42` rather than the address of `$age`.

If the reference is to an array or hash, there is a second way to specify dereferencing, which is to use the `->` operator between the variable's name and its subscript. For example, the following two assignment statements are equivalent:

```
$$ref_stooges[3] = "Maxine";
$ref_stooges -> [3] = "Maxine";
```

8.8 Functions

Subprograms are central to the usefulness of any programming language. Perl's subprograms are all functions, as in its ancestor language, C. This section describes the basics of Perl functions.

8.8.1 Fundamentals

A *function definition* includes the function's header and a block of code that describes its actions. Neither parameter specifications nor a return value type are part of a Perl function definition. A *function header* is the reserved word `sub` and the function's name. A *function declaration* is a message to the compiler that

the given name is a function that will be defined somewhere in the program. Syntactically, a function declaration is the function header without the block. Because forward calls to functions are legal, it does not matter where their definitions appear in a program. We prefer to put them at the beginning.

A function that returns a value that is to be used immediately is called in the position of an operand in an expression (or as the whole expression). A function that either does not return a value or returns a value that is not to be used can be called by a standalone statement.

A function that has been previously declared can be treated as a list operator, meaning that calls to it do not need to include the parentheses.

A function definition can specify the value it returns in two ways, implicitly and explicitly. The `return` function takes an expression as its parameter. The value of the expression is returned when the `return` is executed. A function can have any number of calls to `return`, including none. If there are no calls to `return` in a function or if execution arrives at the end of the function without encountering a `return`, its returned value is the value of the last expression evaluated in the function. For clarity's sake, we recommend that the last statement of every function is a `return`.

8.8.2 Local Variables

Variables that are implicitly declared have global scope—that is, they are visible in the entire program. It is usually best for variables used in a function that are not used outside the function to have local scope, meaning that they are visible and can be used only within the block of the function. Such variables are declared to have local scope in a function with a `my` declaration.[8] Initial values of local variables can be part of their declaration as follows:

```
my $count = 0;
```

If more than one variable is declared in a `my` declaration, they must be placed in parentheses, as shown in the following example:

```
my($count, $sum) = (0,0);
```

Notice the use of the list assignment to initialize these local variables.

If the name of a local variable conflicts with that of a global variable, the local variable is used. This is the advantage of local variables: When you make up their names, you do not need to be concerned that a global variable with the same name may exist in the program.

Perl includes a second kind of local variable, which is declared with the `local` reserved word. The scope of such variables is dynamic, which makes them very different from `my` variable, whose scope is static. We do not discuss this second kind of local variable here. When we say a variable is local, we mean it has local scope and is defined with `my`.

8. Actually, `my` is a function, but it seems clearer to consider it a declaration.

8.8.3 Parameters

The parameter values that appear in a call to a function are called *actual parameters*. The parameter names used in the function, which correspond to the actual parameters, are called *formal parameters*. Two common models of parameter transfers are used in the linkage between a function and its caller: pass by value and pass by reference. Pass-by-value parameters, which are usually implemented by sending values to the function, provide one-way communication to the called function. Pass-by-reference parameters, which are often implemented by passing the addresses of variables to the function, provide two-way communication between the function and its caller. Unless two-way communication is necessary, pass-by-value parameters should be used.

All Perl parameters are communicated through a special implicit array, `@_`. When a function is called, the values of the actual parameters specified in the call are placed in `@_`. If an actual parameter is an array, all of the array's elements are placed in `@_`. When a hash is used as an actual parameter, its value is flattened into an array, and its values are moved to `@_`. Every function has its own version of `@_`, so if a function calls another function, the values in the caller's `@_` are not affected by the call.

If the values in `@_` are manipulated directly in the called function, the parameters have pass-by-reference semantics, as shown in the following example:

```
sub plus10 {
    $_[0] += 10;
}
plus10($a);
```

The call to `plus10` results in `10` being added to `$a`. If you call this function with a literal—by mistake, of course—it has no effect, not even that of producing an error message.

Pass-by-value parameters are implemented by assigning the passed values in `@_` to local variables. For example:

```
sub fun_eval {
    my($a, $b, $c, $x) = @_;
    return $a * $x * $x + $b * $x + $c;
}
```

This function evaluates a given quadratic equation.

References to variables can be used as actual parameters, which provides pass-by-reference semantics. For example, the following function builds a new array from the nonnegative values from a given array.

```
sub squeeze {
    my $ref_list = $_[0];
    my $value, @new;
    foreach $value (@$ref_list) {
        if($value > 0) {
```

```
            push(@new, $value);
        }
    }
    return @new;
}
```

The following is an example of a call to `squeeze`:

```
squeeze(\@mylist);
```

8.8.4 The sort Function, Revisited

Recall that the `sort` function, as introduced in Section 8.5.5, takes a single array parameter and sorts that array, treating the elements as strings (coercing elements that are not strings to strings). This does not work for numbers because, when coerced to strings, `124` belongs before `2`.

The `sort` function can be used more flexibly by giving it the code to use to compare array elements. This code appears as a block between `sort` and the array parameter. Because the block is not a parameter, there is no comma between it and the array to be sorted. The comparison code block must evaluate to a negative number if the first value being compared belongs before the second, zero if the two values are equal, and a number greater than zero if the two values must be interchanged. The two values to be compared are referenced in the comparison with the names `$a` and `$b`. These variables, which act as formal parameters, are used in pass-by-reference mode. Therefore, they should not be changed in the comparison code. The two relational operators, `<=>` and `cmp`, are normally used for the comparison. Recall that they return exactly what is needed by the sort process.

For example, to sort numbers in the array `@list` into ascending order, the following could be used:

```
@new_list = sort {$a <=> $b} @list;
```

To sort the same array into descending order, the two variables `$a` and `$b` are interchanged in the comparison, as shown in the following:

```
@new_list = sort {$b <=> $a} @list;
```

Likewise, to sort strings in the array `@names` into reverse alphabetic order, the following could be used:

```
@new_names = sort {$b cmp $a} @names;
```

The comparison process for a `sort` can be defined in a function, in which case the function's name takes the place of the block in the call to `sort`.

8.8.5 An Example

What follows is an example of a program that uses a function to compute the median of a given array of numbers. The address of the array is passed to the function and is moved to a local variable there. There is no need to pass the

length of the array because the function can easily determine the array's length. The precise specification for the function is given in its initial comments.

```
# tst_median.pl - a program to test a function that
#                 computes the median of a given array

# median - a function
# Parameter:
#     A reference to an array of numbers
# Return value:
#     The median of the array, where median is the
#      middle element of the sorted array, if the
#      length is odd; if the length is even, the median
#      is the average of the two middle elements of the
#      sorted array

sub median {
    my $ref_list = $_[0];

#>>> Compute the length of the passed array
    my $len = $#$ref_list + 1;

#>>> Sort the parameter array
    @list = sort {$a <=> $b} @$ref_list;

#>>> Compute the median
    if ($len % 2 == 1) {  # length is odd
        return $list[$len / 2];
    } else {  # length is even
        return ($list[$len / 2] + $list[$len / 2 - 1]) / 2;
    }
}  #>>> End of function median

#>>> Begin main program
#>>> Create two test arrays, one with odd length and one with
#>>>  even length
@list1 = (11, 36, 5, 20, 41, 6, 8, 0, 9);
@list2 = (43, 77, 11, 29, 8, 51, 9, 18);

#>>> Call median on both arrays and display the results
$med = median(\@list1);
print "The median of the first array is: $med \n";
$med = median(\@list2);
print "The median of the second array is: $med \n";
```

The output of this program is as follows:

```
The median of the first array is: 9
The median of the second array is: 23.5
```

8.9 Pattern Matching

Regular expressions in JavaScript were discussed in Chapter 4, "The Basics of JavaScript." Because JavaScript regular expressions are based directly on those of Perl, readers who are not familiar with regular expressions are referred to Sections 4.12.1 to 4.12.3. The pattern-matching operations of Perl are different from those of JavaScript, so they are discussed here.

8.9.1 The Basics of Pattern Matching

In Perl, the pattern-matching operation is specified with the operator `m`. When slashes are used as delimiters, the `m` operator is not required. Therefore, just a slash-delimited regular expression is a complete pattern expression in Perl. The string against which the matching is attempted is, by default, the implicit variable `$_`. The result of evaluating a pattern-matching expression is either `true` or `false`. The following is a pattern match against the value in `$_`:

```
if (/rabbit/) {
    print
        "The word 'rabbit' appears somewhere in \$_ \n";
}
```

The pattern-matching operation does not need to be always against the string in `$_`. The binding operator, `=~`, can be used to specify any string as the one against which the pattern will be matched. For example, consider the following:

```
if ($str =~ /^rabbit/) {
    print "The value of \$str begins with 'rabbit' \n";
}
```

A restricted form of the `split` function was introduced in Section 8.5.5. In that section, the first parameter to `split` was a single character. However, the first parameter also can be any pattern. For example, we could have the following:

```
@words = split /[ .,]\s*/, $str;
```

This statement puts the words from `$str` into the `@words` array. The words in `$str` are defined to be terminated with either a space, a period, or a comma, any of which could be followed by more whitespace characters.

The following sample program illustrates a simple use of pattern matching and hashes. The program reads a file of text in which the words are separated by

whitespace and some common kinds of punctuation such as commas, periods, semicolons, and so forth. The objective of the program is to produce a frequency table of the words found in the input file. A hash is an ideal way to build the word-frequency table. The keys can be the words, and the values can be the number of times they have appeared. The `split` operator provides a convenient way to split each line of the input file into its words. For each word, the program uses `exists` on the hash to determine whether the word has occurred before. If so, its count is incremented; if not, the word is entered into the hash with a count of `1`.

```
# word_table.pl
#  Input: A file of text in which all words are separated by white-
#         space or punctuation, possibly followed by whitespace,
#         where the punctuation can be a comma, a semicolon, a
#         question mark an exclamation point, a period, or a colon.
#         The input file is specified on the command line
# Output: A list of all unique words in the input file,
#         in alphabetical order

#>>> Main loop to get and process lines of input text
while (<>) {

#>>> Split the line into words
    @line_words = split /[\.,;:!\?]\s*/;

#>>> Loop to count the words (either increment or initialize to 1)
    foreach $word (@line_words) {
        if (exists $freq{$word}) {
            $freq{$word}++;
        } else {
            $freq{$word} = 1;
        }
    }
}
#>>> Display the words and their frequencies
print "\n Word \t\t Frequency \n\n";
foreach $word (sort keys %freq) {
    print " $word \t\t $freq{$word} \n";
}
```

Notice that the two normally special characters, `.` (period) and `?` (question mark), are not backslashed in the pattern for `split` in this program. The reason is that, as mentioned previously, the normally special characters for patterns (metacharacters) are not special in character classes.

8.9.2 Remembering Matches

The part of the string that matched a part of the pattern can be saved in an implicit variable for later use. The part of the pattern whose match you want to save is placed in parentheses. The substring that matched the first parenthesized part of the pattern is saved in `$1`, the second in `$2`, and so forth. As an example, consider the following:

```
"4 July 1776" =~ /(\d+) (\w+) (\d+)/;
print "$2 $1, $3 \n";
```

This displays the following:

```
July 4, 1776
```

In some situations, it is convenient to be able to reference the parts of the string that preceded the match, the part that matched, or the part that followed the match. These three strings are available after a match through the implicit variables `$`` `, `$&`, and `$´`, respectively.

8.9.3 Substitutions

Sometimes the substring of a string that matched a pattern must be replaced by another string. Perl's substitute operator is designed to do exactly that. The general form of the substitute operator is as follows:

`s/`*Pattern*`/`*New_String*`/`

Pattern is the same as the patterns used by the match operator. *New_String* is what is to replace the part of the string that matched the pattern. Consider the following example:

```
$str = "It ain't going to rain no more, no more";
$str =~ s/ain't/is not/;
```

This changes `"ain't"` to `"is not"` in `$str`

The substitute operator can have two modifiers, `g` and `i`. The `g` modifier tells the substitute operator to find all matches in the given string and replace all of them:

```
$str = "Rob, Robbie, and Robette were siblings";
$str =~ s/Rob/Bob/g;
```

This changes `$str` to `"Bob, Bobbie, and Bobette were siblings"`.

The `i` modifier, which tells the pattern matcher to ignore the case of letters, can also be used with the substitute operator, as shown in the following code:

```
$str = "Is it Rose, rose, or ROSE?";
$str =~ s/Rose/rose/ig;
```

This changes `$str` to `"Is it rose, rose, or rose?"`.

All of the other details of Perl regular expressions can be found by typing the following:

```
perldoc perlre
```

8.9.4 The Transliterate Operator

Perl has a transliterate operator, `tr`, which translates a character or character class to another character or character class, respectively. For example, the following statement replaces all semicolons in `$str` with colons:

```
$str =~ tr/;/:/;
```

This particular operation can also be done with the substitute operator, as follows:

```
$str =~ s/;/:/g;
```

The following statement transforms all uppercase letters in `$str` to lowercase letters:

```
$str =~ tr/A-Z/a-z/;
```

Specific characters can be deleted from a string by using a null substitution character. For example, this next statement deletes all commas and periods from the string in `$str`:

```
$str =~ tr/\,\.//;
```

The transliterate operator also can be used on `$_` by omitting the left side and the binding operator.

8.10 File Input and Output

Files are referenced through program variables called *filehandles*, whose names do not begin with special characters. To make them more readable, filehandles are often spelled using all uppercase letters. Filehandles are initialized with the `open` function, which opens a file for use and assigns the filehandle. The `open` function establishes the relationship between a filehandle and the file's external (operating system) name. Files can be opened for input or either of two kinds of output. The first parameter to `open` is the filehandle; the second specifies both the file's external name and how it will be used. The file's name is either a literal string or a string-valued expression. The file's usage is specified by appending one or two special characters to the beginning of the file's name. The most common of these are shown in Table 8.5.

Table 8.5 File use specifications

Character(s)	Meaning
<	Input (the default)
>	Output, starting at the beginning of the file
>>	Output, starting at the end of the existing data on the file
+>	Input from and output to the file

The > file use specification indicates the file is to be opened for output and writing is to begin at the file's beginning (overwriting any data currently written there). If the file does not exist, it is implicitly created. >> also indicates the file is to be opened for output, but the new data to be appended to the end of the file's current data. The +< file use specification is used when a file's data is to be read, processed, and rewritten to the same file, replacing the data that was read. As with <, if the file does not exist, it is an error.

Every file has an internal file pointer that points to the position in the file where the next read or write will take place. The > and < file use specifications initialize the file pointer to the beginning of the file. The >> file use specification initializes the file pointer to the end of the current data in the file.

Because `open` can fail, it is often used with the `die` function, as shown in the following example:

```
open(INDAT, "<temperatures") or
     die "Error – unable to open temperatures $!";
```

This use of `die` after the `or` operator works for two reasons: First, `or` has lower precedence than the call to `open`. Second, `open` returns false if it fails.

Files are closed with the `close` function.

One line of text can be written to a file with the `print` function, using the file's filehandle as the first parameter, as follows:

```
print OUTDAT "The result is: $result \n";
```

Notice that no comma appears between the filehandle and the string.

Lines of text can be read from a file using the line input operator, including the filehandle between the angle brackets, as shown in the following example:

```
$next_line = <INDAT>;
```

Multiple lines can be read from a file with the `read` function. Because of the overhead of starting a file read or write operation, large files are input most quickly by reading more than one line at a time. The general form of a call to `read` is as follows:

`read(`*filehandle*`,` *buffer*`,` *length* `[,` *offset*`]);`

The *offset* is optional, as indicated by the brackets. The *buffer* is a scalar variable into which the lines that have been read are placed. The *length* parameter is the number of bytes of input to be read. When included, the offset is the distance from the beginning of the buffer where the input is to go. When not included in the call to `read`, the offset is zero. The `read` function returns the number of bytes that it read and placed in the buffer. Newlines count in the `read` operation.

Suppose you have a file whose filehandle is `ANIMALS`, which has lines of 50 characters, not counting the newline. You could read five lines from this file with the following statement:

```
$chars = read(ANIMALS, $buf, 255);
```

If `$chars` is less than `255` after this statement is executed, there were not 255 characters left in the file.

The lines in the buffer can be separated into separate elements of an array with the following statement:

```
@lines = split /\n/, $buf;
```

Some applications read a file, modify the data read from the file, and then rewrite the file with the modified data. The +> file use specification allows a file to be both read and written, but after the file has been read, its file pointer is left at the end of the data that has been read. To rewrite the file, its file pointer must be moved back to the file's beginning. This can be done with the `seek` function, which takes three parameters: the filehandle of the file, an offset, and a base position in the file. The possible base position values are 0, 1, or 2, specifying the beginning of the file, the current position of the file pointer in the file, or the end of the file, respectively. The offset is used to specify the number of bytes from the given base position. A positive value is an offset toward the end of the file (from the base position); a negative value is an offset in the direction of the beginning of the file. Most commonly, `seek` is used to rewind a file,[9] which sets the file pointer to the beginning of the file. This is specified with the following:

```
seek(filehandle, 0, 0);
```

When files are used to store data for CGI programs, they frequently need to be protected against corruption caused by multiple simultaneous writes. This is done with file locks, which are discussed and exemplified in Chapter 9.

8.11 An Example

The next sample program illustrates some of the features of Perl described in this chapter. The program reads employee records from a file and computes some statistics on the contents of the file. The precise specification for the program is included at its beginning as documentation.

```
# wages.pl - An example program to illustrate some of the
#            features of Perl
#  Input: A file of lines of employee data, where each line has
#         name:age:department code:salary
# Output: 1. The names of all employees whose names end with "son"
#         2. Percentage of employees under 40 years old
#         3. Average salary of employees under 40 years old
#         4. An alphabetical list of employees who are under 40
#            years old and who have salaries more than $40,000
```

9. Rewind is a holdover word from the days of magnetic tape, when setting the file pointer to the beginning required the tape to be rewound.

```
#>>> Open the data file and display a header for employees
#>>>  whose names end in 'son'
open(EMPLOYEES, "employees.txt") || die "Can't open employees $!";
print "Names that end in 'son'\n\n";

#>>> Loop to read and process the employee data
while (<EMPLOYEES>) {

#>>> Increment the number of employees and chop off the newline
    $total_employees++;
    chomp;

#>>> Split the input line into its four parts
    ($name, $age, $dept, $salary) = split(/:/);

#>>> If the name ends in 'son', print the name
    if ($name =~ /son$/) {
        print "$name \n";
    }

#>>> If the employee is under 40, count him or her and add his or
#>>>  her salary to the sum of such salaries
    if ($age < 40) {
        $under_40++;
        $salary_sum += $salary;

#>>> If the salary was over 40,000, add the person and his or her
#>>>  salary to the hash of such people
        if ($salary > 40000) {
            $sublist{$name} = $salary;
        }
    }
}
#>>> If there was at least one employee, continue
if ($total_employees > 0) {

#>>> If there was at least one under 40, continue
    if ($under_40 > 0) {

#>>> Compute and display the % of employees under 40 and their
#>>>  average salaries
        $percent = 100 * $under_40 / $total_employees;
        print "\nPercent of employees under 40 is: $percent \n";
        $avg = $salary_sum / $under_40;
        print "Average salary of employees under 40 is: $avg \n";
```

```
#>>> If there was at least one under 40 who earned a
#>>>  salary > 40,000, continue
        if (keys(%sublist)) {

#>>> Sort and display the names of the employees under 40 with
#>>>  with salaries > 40,000
            print "Sorted list of employees under 40",
                  " with salaries > \$40,000 \n";
            @sorted_names = sort (keys(%sublist));
            print "\nName \t\t Salary\n";
            foreach $name (@sorted_names) {
                print "$name \t \$$sublist{$name} \n";
            }
        }
        else {
            print "There were no employees under 40 who earned";
            print "over $40,000 \n";
        }  #>>> of if (keys(%sublist))
    }
    else {
        print "There were no employees under 40 \n";
    }  #>>> of if ($under_40 > 0)
}
else {
    print "There were no employees\n";
}  #>>> of if ($total_employees > 0)
```

Summary

Perl began as a relatively small language that included the capabilities of `awk` and `sh`, but it since has evolved to a full-blown programming language.

Perl's scalar values can be numbers, strings, or references. Numeric literals can be in either integer or floating-point form. Strings can be delimited by either single or double quotes. Variables that appear in double-quoted strings are interpolated into the string, but in single-quoted strings they are not. The names of all scalar variables begin with dollar signs. Numeric expressions can use the usual complement of arithmetic operators.

There are only two string operators: catenation (`.`) and repetition (`x`). However, there is a large collection of functions and operators that perform the most commonly needed operations on strings. Assignment statement operators can be simple or compound, where the compound operators combine a binary operator with the assignment operator. Keyboard input is obtained with the line input operator using `STDIN`. Screen output is created with the `print` function.

Perl includes two sets of relational operators, one for numeric operands and one for string operands. If an operand of the wrong type appears on one of these, it is coerced to the proper type. Perl also includes two sets of Boolean operators, the only difference being the level of precedence of the two (`&&`, `||`, and `!` have higher precedence; `and`, `or`, and `not` have lower precedence). In all of the control constructs, the statements whose execution is to be controlled are specified by a block. The two selection statements are `if` and its complement `unless`. The two logically controlled loops are `while` and `until`. The `for` statement can also be used as a logically controlled loop but is usually used for counting loops. Perl has no `case` or `switch` statement.

An array is a variable that stores lists. A list literal is a parenthesized list of scalar expressions or values. The names of all arrays begin with at signs (`@`). Individual elements of arrays are referenced through subscripts, which are numeric-valued expressions delimited by brackets. Arrays, like scalars, do not need to be declared—that is, they are implicitly declared. The length of an array is determined by the highest subscript with which it has been assigned a value. The `foreach` statement provides a convenient way of processing all of the elements of an array. The `shift`, `unshift`, `pop`, and `push` operators provide simple ways of adding or removing an end element of an array.

A hash is an associative array, which is a data structure in which each element is actually a key/value pair. The values are the stored data; the keys are used to find specific data values. There are no hash literals, so list literals with alternating keys and values are used. A particular value element of a hash is referenced by the hash's name followed by the key's name, delimited by braces. Hash elements can be removed with the `delete` operator.

A reference variable stores the address of a variable or a literal. References to variables are created with the backslash operator appended to the beginning of the variable's name. A reference to a list literal is created by placing the list literal in brackets. A reference to a hash literal is created by placing the list of hash elements in braces. A reference can be dereferenced by appending a dollar sign to the beginning of its name. In the cases of arrays and hashes, the `->` operator can also be used for dereferencing.

A function consists of the function header, which is the reserved word `sub` and the function's name, and a block that defines its actions. Local variables in functions are created with the `my` declaration. Function parameters are passed through the implicit array `@_`. To achieve pass-by-reference semantics, `@_` can be directly manipulated in the function. To achieve pass-by-value semantics, the values in `@_` are assigned to new local variables at the beginning of the function. Pass-by-reference semantics can also be achieved by passing references.

Perl has a powerful pattern matcher. Typically, the pattern is delimited by slashes and is matched against `$_`. The result of a pattern match is either `true` or `false`. String matches of subpatterns can be remembered in predefined variables by parenthesizing those subpatterns. The substitution operator is a powerful tool for modifying text. The transliteration operator is used to do literal translations of either characters or character classes.

Files are referenced through program variables called filehandles, which do not begin with any special characters. Files can be opened for input, output to the current end of the file, output to the beginning of the file, or input and output. These are specified with the `open` function, whose parameters are the filehandle and the file's external name as a string literal. Using a filehandle as the first parameter to `print` causes its output to go to that file. A single line from a file can be read using the line input operator with the filehandle. The `read` function can be used to read multiple lines from a file.

Review Questions

8.1 What are the three categories of Perl variables?

8.2 How many numeric data types does Perl have?

8.3 What is the purpose of the `qq` operator?

8.4 In what two ways do single-quoted string literals differ from double-quoted string literals?

8.5 What is the numeric value of `undef`?

8.6 Describe the operands and the actions of the `x` string operator.

8.7 Describe the parameters and actions of the `chomp` function.

8.8 Describe the parameters and actions of the `join` function.

8.9 What is a filehandle?

8.10 Under what conditions is a string used in a Boolean context considered to be true?

8.11 Why does Perl have two sets of relational operators?

8.12 What is the difference between Perl's two sets of Boolean operators?

8.13 What exactly does the expression `<STDIN>` do when executed?

8.14 In what three fundamental ways do Perl arrays differ from the arrays of other common high-level programming languages?

8.15 If an array's name appears in scalar context, what is its value?

8.16 What two predefined Perl functions can be used to implement a queue in an array?

8.17 What are the two fundamental ways in which hashes differ from arrays?

8.18 What statement adds the element `(joe, 42)` to the hash `%guys`?

8.19 How do you get the address of the scalar variable `$fruit`?

8.20 How do you get the address of a list literal?

8.21 What is a function declaration?

8.22 How do you create local variables in a function?

8.23 What are actual parameters? What are formal parameters?

8.24 What are the two ways a value can be returned from a function?

8.25 How does the `i` modifier change pattern matching?

8.26 Describe the two parameters for the substitute operator.

8.27 Describe the transliterate operator.

8.28 Describe the four file use specifications.

8.29 Under what circumstances is the `read` function used?

8.30 What is a file pointer?

8.31 What is one common use of the `seek` function?

Exercises

Write, test, and debug (if necessary) Perl programs for the following specifications.

8.1 *Input*: Three numbers, `a`, `b`, and `c`, each on its own line, from the keyboard

Output: The value of the expression `10ab-((c-1)/17.44)`

8.2 *Input*: A text file, specified on the command line, in which each line contains one number

Output: The second smallest number in the file, along with its position in the file, with 1 being the position of the first number

8.3 *Input*: Three names, on separate lines, from the keyboard

Output: The input names in alphabetical order, without using arrays

8.4 *Input*: A file of lines of text, specified on the command line

Output: Every input line that has more than 10 characters (not counting the newline) but fewer than 20 characters (not counting the newline) that contains the string `"ed"`

8.5 *Input*: A list of numbers in a file specified on the command line

Output: Two lists of numbers, one with input numbers that are greater than zero, and one with those that are less than zero (ignore the zero-valued numbers)

Method: You must first build two arrays with the required output numbers before you display any of them.

8.6 *Input*: A file that contains English words, where each word is separated from the next word on a line by one space, specified on the command line

Output: A table, in which the first column has the unique words from the input file and the second column has the number of times the word appeared in the file; no word can appear twice in the table

Method: Your program must use two arrays to store the table, one for the words and one for the frequency values.

8.7 *Input*: A file in which each line contains a string of the form `name+sales`, where in some cases the sales will be absent (but not the plus sign), specified on the command line

Output: A list of the names and sales numbers that remain after the following processing:

a. Names with sales numbers are added to a hash when they are first found, along with their sales numbers
b. Names with absent sales numbers are deleted from the hash if they are already there
c. When a name appears that is already in the hash, the new sales number is added to the old sales number (the one already in the hash)

8.8 *Input*: A file specified on the command line that contains text

Output: The input text after the following modifications have been made:

a. All multiple spaces are reduced to single spaces
b. All occurrences of `Darcy` are replaced with `Darcie`
c. All lines that begin with ~ are deleted
d. All occurrences of `1998` are replaced with `1999`

8.9 *Input*: A file specified on the command line that contains a C program

Output: For each line of the input:

a. The number of words (variables and reserved words) on the line
b. The number of numeric literals without decimal points on the line
c. The number of numeric literals with decimal points on the line
d. The number of braces and parentheses on the line

Write functions for the specifications in Exercises 8.10, 8.11, and 8.12

8.10 *Parameter*: An array of strings, passed by value

Return value: A list of the unique strings in the parameter array

8.11 *Parameter*: An array of numbers

Return value: The average and median of the parameter array

8.12 *Parameter*: A reference to an array of strings

Return value: A list of the unique strings in the parameter array