Chapter 2. Scalar Data

2.1. What Is Scalar Data?

In English, as in many other spoken languages, we're used to distinguishing between singular and plural. As a computer language designed by a human linguist, Perl is similar. As a general rule, when Perl has just one of something, that's a *scalar*.

A *scalar* is the simplest kind of data that Perl manipulates. Most scalars are either a number (like 255 or 3.25e20) or a string of characters (like `hello` or the Gettysburg Address). Although you may think of numbers and strings as very different things, Perl uses them nearly interchangeably.

A scalar value can be acted upon with operators (like addition or concatenate), generally yielding a scalar result. A scalar value can be stored into a scalar variable. Scalars can be read from files and devices, and can be written out as well.

2.2. Numbers

Although a scalar is most often either a number or a string, it's useful to look at numbers and strings separately for the moment. We'll cover numbers first, and then move on to strings.

2.2.1. All Numbers Are the Same Format Internally

As you'll see in the next few paragraphs, you can specify both integers (whole numbers, like 255 or 2001) and floating-point numbers (real numbers with decimal points, like 3.14159, or 1.35 x 1025). But internally, Perl computes with double-precision floating-point values. This means that there are no integer values internal to Perl -- an integer constant in the program is treated as the equivalent floating-point value. You probably won't notice the conversion (or care much), but you should stop looking for distinct integer operations (as opposed to *floating-point* operations), because there aren't any.

2.2.2. Floating-Point Literals

A literal is the way a value is represented in the source code of the Perl program. A literal is not the result of a calculation or an I/O operation; it's data written directly into the source code.

Perl's floating-point literals should look familiar to you. Numbers with and without decimal points are allowed (including an optional plus or minus prefix), as well as tacking on a power-of-10 indicator (exponential notation) with E notation. For example:

```
1.25
255.000
255.0
7.25e45  # 7.25 times 10 to the 45th power (a big number)
-6.5e24  # negative 6.5 times 10 to the 24th
         # (a big negative number)
-12e-24  # negative 12 times 10 to the -24th
         # (a very small negative number)
-1.2E-23 # another way to say that - the E may be uppercase
```

2.2.3. Integer Literals

Integer literals are also straightforward, as in:

```
0
2001
-40
255
61298040283768
```

That last one is a little hard to read. Perl allows underscores for clarity within integer literals, so we can also write that number like this:

```
61_298_040_283_768
```

It's the same value; it merely looks different to us human beings. You might have thought that commas should be used for this purpose, but commas are already used for a more-important purpose in Perl (as we'll see in the next chapter).

2.2.4. Nondecimal Integer Literals

Like many other programming languages, Perl allows you to specify numbers in other than base 10 (decimal). Octal (base 8) literals start with a leading `0`, hexadecimal (base 16) literals start with a leading `0x`, and binary (base 2) literals start with a leading `0b`. The hex digits `A` through `F` (or `a` through `f`) represent the conventional digit values of ten through fifteen. For example:

```
0377       # 377 octal, same as 255 decimal
0xff       # FF hex, also 255 decimal
0b11111111 # also 255 decimal (available in version 5.6 and later)
```

Although these values look different to us humans, they're all three the same number to Perl. It makes no difference to Perl whether you write `0xFF` or `255.000`, so choose the representation that makes the most sense to you and your maintenance programmer (by which we mean the poor chap who gets stuck trying to figure out what you meant when you wrote your code. Most often, this poor chap is you, and you can't recall whay you did what you did three months ago).

When a non-decimal literal is more than about four characters long, it may be hard to read. For this reason, starting in version 5.6, Perl allows underscores for clarity within these literals:

```
0x1377_0b77
0x50_65_72_7C
```

2.2.5. Numeric Operators

Perl provides the typical ordinary addition, subtraction, multiplication, and division operators, and so on. For example:

```
2 + 3      # 2 plus 3, or 5
5.1 - 2.4  # 5.1 minus 2.4, or 2.7
3 * 12     # 3 times 12 = 36
14 / 2     # 14 divided by 2, or 7
10.2 / 0.3 # 10.2 divided by 0.3, or 34
10 / 3     # always floating-point divide, so 3.3333333...
```

Perl also supports a *modulus* operator (`%`). The value of the expression `10 % 3` is the remainder when ten is divided by three, which is one. Both values are first reduced to their integer values, so `10.5 % 3.2` is computed as `10 % 3`.

Additionally, Perl provides the FORTRAN-like *exponentiation* operator, which many have yearned for in Pascal and C. The operator is represented by the double asterisk, such as `2**3`, which is two to the third power, or eight.

In addition, there are other numeric operators, which we'll introduce as we need them.

2.3. Strings

Strings are sequences of characters (like `hello`). Strings may contain any combination of any characters.

The shortest possible string has no characters. The longest string fills all of your available memory (although you wouldn't be able to do much with that). This is in accordance with the principle of "no built-in limits" that Perl follows at every opportunity. Typical strings are printable sequences of letters and digits and punctuation in the ASCII 32 to ASCII 126 range. However, the ability to have any character in a string means you can create, scan, and manipulate raw binary data as strings -- something with which many other utilities would have great difficulty. For example, you could update a graphical image or compiled program by reading it into a Perl string, making the change, and writing the result back out.

Like numbers, strings have a literal representation, which is the way you represent the string in a Perl program. Literal strings come in two different flavors: *single-quoted string literals* and *double-quoted string literals*.

2.3.1. Single-Quoted String Literals

A *single-quoted string literal* is a sequence of characters enclosed in single quotes. The single quotes are not part of the string itself -- they're just there to let Perl identify the beginning and the ending of the string. Any character other than a single quote or a backslash between the quote marks (including newline characters, if the string continues onto successive lines) stands for itself inside a string. To get a backslash, put two backslashes in a row, and to get a single quote, put a backslash followed by a single quote. In other words:

```
'fred'    # those four characters: f, r, e, and d
'barney'  # those six characters
''        # the null string (no characters)
'Don\'t let an apostrophe end this string prematurely!'
'the last character of this string is a backslash: \\'
'hello\n' # hello followed by backslash followed by n
'hello
there'    # hello, newline, there (11 characters total)
'\'\\'    # single quote followed by backslash
```

Note that the `\n` within a single-quoted string is not interpreted as a newline, but as the two characters backslash and `n`. Only when the backslash is followed by another backslash or a single quote does it have special meaning.

2.3.2. Double-Quoted String Literals

A *double-quoted string literal* is similar to the strings you may have seen in other languages. Once again, it's a sequence of characters, although this time enclosed in double quotes. But now the backslash takes on its full power to specify certain control characters, or even any character at all through octal and hex representations. Here are some double-quoted strings:

```
"barney"        # just the same as 'barney'
"hello world\n" # hello world, and a newline
"The last character of this string is a quote mark: \""
"coke\tsprite"  # coke, a tab, and sprite
```

Note that the double-quoted literal string `"barney"` means the same six-character string to Perl as does the single-quoted literal string `'barney'`. It's like what we saw with numeric literals, where we saw that `0377` was another way to write `255.0`. Perl lets you write the literal in the way that makes more sense to you. Of course, if you wish to use a backslash escape (like `\n` to mean a newline character), you'll need to use the double quotes.

The backslash can precede many different characters to mean different things (generally called a *backslash escape*). The nearly complete list of double-quoted string escapes is given in Table 2-1.

Table 2-1. Double-quoted string backslash escapes

Construct	Meaning
`\n`	Newline
`\r`	Return
`\t`	Tab
`\f`	Formfeed
`\b`	Backspace
`\a`	Bell
`\e`	Escape (ASCII escape character)
`\007`	Any octal ASCII value (here, `007` = bell)
`\x7f`	Any hex ASCII value (here, `7f` = delete)
`\cC`	A "control" character (here, Ctrl-C)
`\\`	Backslash
`\"`	Double quote
`\l`	Lowercase next letter
`\L`	Lowercase all following letters until `\E`
`\u`	Uppercase next letter

`\U`	Uppercase all following letters until `\E`
`\Q`	Quote non-word characters by adding a backslash until `\E`
`\E`	Terminate `\L`, `\U`, or `\Q`

Another feature of double-quoted strings is that they are *variable interpolated,* meaning that some variable names within the string are replaced with their current values when the strings are used. We haven't formally been introduced to what a variable looks like yet, so we'll get back to this later in this chapter.

2.3.3. String Operators

String values can be concatenated with the `.` operator. (Yes, that's a single period.) This does not alter either string, any more than `2+3` alters either `2` or `3`. The resulting (longer) string is then available for further computation or to be stored into a variable. For example:

```
"hello" . "world"       # same as "helloworld"
"hello" . ' ' . "world" # same as 'hello world'
'hello world' . "\n"    # same as "hello world\n"
```

Note that the concatenation must be explicitly requested with the `.` operator, unlike in some other languages where you merely have to stick the two values next to each other.

A special string operator is the *string repetition* operator, consisting of the single lowercase letter `x`. This operator takes its left operand (a string) and makes as many concatenated copies of that string as indicated by its right operand (a number). For example:

```
"fred" x 3       # is "fredfredfred"
"barney" x (4+1) # is "barney" x 5, or "barneybarneybarneybarneybarney"
5 x 4            # is really "5" x 4, which is "5555"
```

That last example is worth spelling out slowly. The string repetition operator wants a string for a left operand, so the number `5` is converted to the string `"5"` (using rules described in detail later), giving a one-character string. This new string is then copied four times, yielding the four-character string `5555`. Note that if we had reversed the order of the operands, as `4 x 5`, we would have made five copies of the string `4`, yielding `44444`. This shows that string repetition is not commutative.

The copy count (the right operand) is first truncated to an integer value (4.8 becomes 4) before being used. A copy count of less than one results in an empty (zero-length) string.

2.3.4. Automatic Conversion Between Numbers and Strings

For the most part, Perl automatically converts between numbers to strings as needed. How does it know whether a number or a string is needed? It all depends upon the operator being used on the scalar value. If an operator expects a number (like `+` does), Perl will see the value as a number. If an operator expects a string (like `.` does), Perl will see the value as a string. So you don't need to worry about the difference between numbers and strings; just use the proper operators, and Perl will make it all work.

When a string value is used where an operator needs a number (say, for multiplication), Perl automatically converts the string to its equivalent numeric value, as if it had been entered as a decimal floating-point value. So `"12" * "3"` gives the value `36`. Trailing nonnumber stuff and leading whitespace are discarded, so `"12fred34" * " 3"` will also give `36` without any complaints. At the extreme end of this, something that isn't a number at all converts to zero. This would happen if you used the string `"fred"` as a number.

Likewise, if a numeric value is given when a string value is needed (say, for string concatenation), the numeric value is expanded into whatever string would have been printed for that number. For example, if you want to concatenate the string `Z` followed by the result of 5 multiplied by 7, you can say this simply as:

```
"Z" . 5 * 7 # same as "Z" . 35, or "Z35"
```

In other words, you don't really have to worry about whether you have a number or a string (most of the time). Perl performs all the conversions for you. And if you're worried about efficiency, don't be. Perl generally remembers the result of a conversion so that it's done only once.

2.5. Scalar Variables

A *variable* is a name for a container that holds one or more values. The name of the variable stays the same throughout the program, but the value or values contained in that variable typically change over and over again throughout the execution of the program.

A scalar variable holds a single scalar value, as you'd expect. Scalar variable names begin with a dollar sign followed by what we'll call a *Perl identifier:* a letter or underscore, and then possibly more letters, or digits, or underscores. Another way to think of it is that it's made up of alphanumerics and underscores, but can't start with a digit. Uppercase and lowercase letters are distinct: the variable `$Fred` is a different variable from `$fred`. And all of the letters, digits, and underscores are significant, so:

```
    $a_very_long_variable_that_ends_in_1
```

is different from:

```
    $a_very_long_variable_that_ends_in_2
```

Scalar variables in Perl are always referenced with the leading `$`. In the shell, you use `$` to get the value, but leave the `$` off to assign a new value. In *awk* or C, you leave the `$` off entirely. If you bounce back and forth a lot, you'll find yourself typing the wrong things occasionally. This is expected. (Most Perl programmers would recommend that you stop writing shell, *awk*, and C programs, but that may not work for you.)

2.5.1. Choosing Good Variable Names

You should generally select variable names that mean something regarding the purpose of the variable. For example, `$r` is probably not very descriptive but `$line_length` is. A variable used for only two or three lines close together may be called something simple, like `$n`, but a variable used throughout a program should probably have a more descriptive name.

Similarly, properly placed underscores can make a name easier to read and understand, especially if your maintenance programmer has a different spoken language background than you have. For example, `$super_bowl` is a better name than `$superbowl`, since that last one might look like `$superb_owl`. Does `$stopid` mean `$sto_pid` (storing a process-ID of some kind?) or `$s_to_pid` (converting something to a process-ID?) or `$stop_id` (the ID for some kind of "stop" object?) or is it just a stopid mispelling?

Most variable names in our Perl programs are all lowercase, like most of the ones we'll see in this book. In a few special cases, capitalization is used. Using all-caps (like `$ARGV`) generally indicates that there's something special about that variable. (But you can get into an all-out brawl if you choose sides on the `$underscores_are_cool` versus the `$giveMeInitialCaps` argument. So be careful.)

Of course, choosing good or poor names makes no difference to Perl. You *could* name your program's three most-important variables `$OOO000OOO`, `$OO00OO00`, and `$O0O0O0O0O` and Perl wouldn't be bothered -- but in that case, please, don't ask us to maintain your code.

2.5.2. Scalar Assignment

The most common operation on a scalar variable is *assignment*, which is the way to give a value to a variable. The Perl assignment operator is the equals sign (much like other languages), which takes a variable name on the left side, and gives it the value of the expression on the right. For example:

```
    $fred = 17;            # give $fred the value of 17
    $barney = 'hello';     # give $barney the five-character string 'hello'
    $barney = $fred + 3;   # give $barney the current value of $fred plus 3 (20)
    $barney = $barney * 2; # $barney is now $barney multiplied by 2 (40)
```

Notice that last line uses the `$barney` variable twice: once to get its value (on the right side of the equals sign), and once to define where to put the computed expression (on the left side of the equals sign). This is legal, safe, and in fact, rather common. In fact, it's so common that we can write it using a convenient shorthand, as we'll see in the next section.

2.5.3. Binary Assignment Operators

Expressions like `$fred = $fred + 5` (where the same variable appears on both sides of an assignment) occur frequently enough that Perl (like C and Java) has a shorthand for the operation of altering a variable -- the *binary assignment operator*. Nearly all binary operators that compute a value have a corresponding binary assignment form with an appended equals sign. For example, the following two lines are equivalent:

```
$fred = $fred + 5; # without the binary assignment operator
$fred += 5;        # with the binary assignment operator
```

These are also equivalent:

```
$barney = $barney * 3;
$barney *= 3;
```

In each case, the operator causes the existing value of the variable to be altered in some way, rather than simply overwriting the value with the result of some new expression.

Another common assignment operator is the string concatenate operator (`.`); this gives us an append operator (`.=`):

```
$str = $str . " "; # append a space to $str
$str .= " ";       # same thing with assignment operator
```

Nearly all binary operators are valid this way. For example, a *raise to the power of operator* is written as `**=`. So, `$fred **= 3` means "raise the number in `$fred` to the third power, placing the result back in `$fred`".

2.6. Output with print

It’s generally a good idea to have your program produce some output; otherwise, someone may think it didn’t do anything. The `print( )` operator makes this possible. It takes a scalar argument and puts it out without any embellishment onto standard output. Unless you’ve done something odd, this will be your terminal display. For example:

```
print "hello world\n"; # say hello world, followed by a newline

print "The answer is ";
print 6 * 7;
print ".\n";
```

You can actually give `print` a series of values, separated by commas.

```
print "The answer is ", 6 * 7, ".\n";
```

This is actually a *list*, but we haven’t talked about lists yet, so we’ll put that off for later.

2.6.1. Interpolation of Scalar Variables into Strings

When a string literal is double-quoted, it is subject to *variable interpolation* (besides being checked for backslash escapes). This means that any scalar variable name in the string is replaced with its current value. For example:

```
$meal = "brontosaurus steak";
$barney = "fred ate a $meal";    # $barney is now "fred ate a brontosaurus steak"
$barney = ’fred ate a ’ . $meal; # another way to write that
```

As you see on the last line above, you can get the same results without the double quotes. But the double-quoted string is often the more convenient way to write it.

If the scalar variable has never been given a value, the empty string is used instead:

```
$barney = "fred ate a $meat"; # $barney is now "fred ate a "
```

Don't bother with interpolating if you have just the one lone variable:

```
print "$fred"; # unneeded quote marks
print $fred;   # better style
```

There's nothing really *wrong* with putting quote marks around a lone variable, but the other programmers will laugh at you behind your back.

Variable interpolation is also known as *double-quote interpolation*, because it happens when double-quote marks (but not single quotes) are used. It happens for some other strings in Perl, which we'll mention as we get to them.

To put a real dollar sign into a double-quoted string, precede the dollar sign with a backslash, which turns off the dollar sign's special significance:

```
$fred = 'hello';
print "The name is \$fred.\n";    # prints a dollar sign
print 'The name is $fred' . "\n"; # so does this
```

The variable name will be the longest possible variable name that makes sense at that part of the string. This can be a problem if you want to follow the replaced value immediately with some constant text that begins with a letter, digit, or underscore. As Perl scans for variable names, it would consider those characters to be additional name characters, which is not what you want. Perl provides a delimiter for the variable name in a manner similar to the shell. Simply enclose the *name* of the variable in a pair of curly braces. Or, you can end that part of the string and start another part of the string with a concatenation operator:

```
$what = "brontosaurus steak";
$n = 3;
print "fred ate $n $whats.\n";          # not the steaks, but the value of $whats
print "fred ate $n ${what}s.\n";        # now uses $what
print "fred ate $n $what" . "s.\n";     # another way to do it
print 'fred ate ' . $n . ' ' . $what . "s.\n"; # an especially difficult way
```

2.6.2. Operator Precedence and Associativity

Operator precedence determines which operations in a complex group of operations happen first. For example, in the expression `2+3*4`, do we perform the addition first or the multiplication first? If we did the addition first, we'd get `5*4`, or `20`. But if we did the multiplication first (as we were taught in math class), we'd get `2+12`, or `14`. Fortunately, Perl chooses the common mathematical definition, performing the multiplication first. Because of this, we say multiplication has a *higher precedence* than addition.

You can override the default precedence order by using parentheses. Anything in parentheses is completely computed before the operator outside of the parentheses is applied (just like you learned in math class). So if I really want the addition before the multiplication, I can say `(2+3)*4`, yielding `20`. Also, if I wanted to demonstrate that multiplication is performed before addition, I could add a decorative but unnecessary set of parentheses, as in `2+(3*4)`.

While precedence is simple for addition and multiplication, we start running into problems when faced with, say, string concatenation compared with exponentiation. The proper way to resolve this is to consult the official, accept-no-substitutes Perl operator precedence chart, shown in Table 2-1. (Note that some of the operators have not yet been described, and in fact, may not even appear anywhere in this book, but don't let that scare you from reading about them in the *perlop* manpage.)

Table 2-2. Associativity and precedence of operators (highest to lowest)

Associativity	Operators
left	parentheses and arguments to list operators
left	`->`
	`++ --` (autoincrement and autodecrement)
right	`**`
right	`\ ! ~ + -` (unary operators)
left	`=~ !~`
left	`* / % x`

left	+ - . (binary operators)
left	<< >>
	named unary operators (-X filetests, rand)
	< <= > >= lt le gt ge (the "unequal" ones)
	== != <=> eq ne cmp (the "equal" ones)
left	&
left	| ^
left	&&
left	||
	.. ...
right	?: (ternary)
right	= += -= .= (and similar assignment operators)
left	, =>
	list operators (rightward)
right	not
left	and
left	or xor

In the chart, any given operator has higher precedence than all of the operators listed below it, and lower precedence than all of the operators listed above it. Operators at the same precedence level resolve according to rules of *associativity* instead.

Just like precedence, associativity resolves the order of operations when two operators of the same precedence compete for three operands:

```
    4 ** 3 ** 2 # 4 ** (3 ** 2), or 4 ** 9 (right associative)
    72 / 12 / 3 # (72 / 12) / 3, or 6/3, or 2 (left associative)
    36 / 6 * 3  # (36/6)*3, or 18
```

In the first case, the ** operator has right associativity, so the parentheses are implied on the right. Comparatively, the * and / operators have left associativity, yielding a set of implied parentheses on the left.

So should you just memorize the precedence chart? No! Nobody actually does that. Instead, just use parentheses when you don't remember the order of operations, or when you're too busy to look in the chart. After all, if you can't remember it without the parentheses, your maintenance programmer is going to have the same trouble. So be nice to your maintenance programmer.

2.6.3. Comparison Operators

For comparing numbers, Perl has the logical comparison operators that remind you of algebra: < <= == >= > !=. Each of these returns a *true* or *false* value. We'll find out more about those return values in the next section. Some of these may be different than you'd use in other languages. For example, == is used for equality, not a single = sign, because that's used for another purpose in Perl. And != is used for inequality testing, because <> is used for another purpose in Perl. And you'll need >= and not => for "greater than or equal to", because the latter is used for another purpose in Perl. In fact, nearly every sequence of punctuation is used for something in Perl.

For comparing strings, Perl has an equivalent set of string comparison operators which look like funny little words: lt le eq ge gt ne. These compare two strings character by character to see whether they're the same, or whether one comes first in standard string sorting order. (In ASCII, the capital letters come before the lowercase letters, so beware.)

The comparison operators (for both numbers and strings) are given in Table 2-3.

Table 2-3. Numeric and string comparison operators

Comparison	Numeric	String
Equal	==	eq

Not equal	`!=`	`ne`
Less than	`<`	`lt`
Greater than	`>`	`gt`
Less than or equal to	`<=`	`le`
Greater than or equal to	`>=`	`ge`

Here are some example expressions using these comparison operators:

```
35 != 30 + 5          # false
35 == 35.0            # true
'35' eq '35.0'        # false (comparing as strings)
'fred' lt 'barney'    # false
'fred' lt 'free'      # true
'fred' eq "fred"      # true
'fred' eq 'Fred'      # false
' ' gt ''             # true
```

2.7. The if Control Structure

Once you can compare two values, you'll probably want your program to make decisions based upon that comparison. Like all similar languages, Perl has an `if` control structure:

```
if ($name gt 'fred') {
  print "'$name' comes after 'fred' in sorted order.\n";
}
```

If you need an alternative choice, the `else` keyword provides that as well:

```
if ($name gt 'fred') {
  print "'$name' comes after 'fred' in sorted order.\n";
} else {
  print "'$name' does not come after 'fred'.\n";
  print "Maybe it's the same string, in fact.\n";
}
```

Unlike in C, those block curly braces are required around the conditional code. It's a good idea to indent the contents of the blocks of code as we show here; that makes it easier to see what's going on. If you're using a programmers' text editor, it'll do most of the work for you.

2.7.1. Boolean Values

You may actually use any scalar value as the conditional of the `if` control structure. That's handy if you want to store a true or false value into a variable, like this:

```
$is_bigger = $name gt 'fred';
if ($is_bigger) { ... }
```

But how does Perl decide whether a given value is true or false? Perl doesn't have a separate Boolean data type, like some languages have. Instead, it uses a few simple rules:

1. The special value `undef` is false. (We'll see this a little later in this section.)

2. Zero is false; all other numbers are true.

3. The empty string (`''`) is false; all other strings are normally true.

4. The one exception: since numbers and strings are equivalent, the string form of zero, `'0'`, has the same value as its numeric form: false.

So, if your scalar value is `undef`, `0`, `''`, or `'0'`, it's false. All other scalars are true -- including all of the types of scalars that we haven't told you about yet.

If you need to get the opposite of any Boolean value, use the unary *not* operator, `!`. If what follows it is a true value, it returns false; if what follows is false, it returns true:

```
if (! $is_bigger) {
  # Do something when $is_bigger is not true
}
```

2.8. Getting User Input

At this point, you're probably wondering how to get a value from the keyboard into a Perl program. Here's the simplest way: use the line-input operator, `<STDIN>` . Each time you use `<STDIN>` in a place where a scalar value is expected, Perl reads the next complete text line from *standard input* (up to the first newline), and uses that string as the value of `<STDIN>`. Standard input can mean many things, but unless you do something uncommon, it means the keyboard of the user who invoked your program (probably you). If there's nothing waiting to be read (typically the case, unless you type ahead a complete line), the Perl program will stop and wait for you to enter some characters followed by a newline (return).

The string value of `<STDIN>` typically has a newline character on the end of it. So you *could* do something like this:

```
$line = <STDIN>;
if ($line eq "\n") {
  print "That was just a blank line!\n";
} else {
  print "That line of input was: $line";
}
```

But in practice, you don't often want to keep the newline, so you need the `chomp` operator.

2.9. The chomp Operator

The first time you read about the `chomp` operator, it seems terribly overspecialized. It works on a variable. The variable has to hold a string. And if the string ends in a newline character, `chomp` can get rid of the newline. That's (nearly) all it does. For example:

```
$text = "a line of text\n"; # Or the same thing from <STDIN>
chomp($text);               # Gets rid of the newline character
```

But it turns out to be so useful, you'll put it into nearly every program you write. As you see, it's the best way to remove a trailing newline from a string in a variable. In fact, there's an easier way to use `chomp`, because of a simple rule: any time that you need a variable in Perl, you can use an assignment instead. First, Perl does the assignment. Then it uses the variable in whatever way you requested. So the most common use of `chomp` looks like this:

```
chomp($text = <STDIN>); # Read the text, without the newline character

$text = <STDIN>;        # Do the same thing...
chomp($text);           # ...but in two steps
```

At first glance, the combined `chomp` may not seem to be the easy way, especially if it seems more complex! If you think of it as two operations -- read a line, then `chomp` it -- then it's more natural to write it as two statements. But if you think of it as one operation -- read just the text, not the newline -- it's more natural to write the one statement. And since most other Perl programmers are going to write it that way, you may as well get used to it now.

`chomp` is actually a function. As a function, it has a return value, which is the number of characters removed. This number is hardly ever useful:

```
$food = <STDIN>;
```

```
$betty = chomp $food; # gets the value 1 - but we knew that!
```

As you see, you may write chomp with or without the parentheses. This is another general rule in Perl: except in cases where it changes the meaning to remove them, parentheses are always optional.

If a line ends with two or more newlines, chomp removes only one. If there's no newline, it does nothing, and returns zero.

If you work with older Perl programs, you may run across the chop operator. It's similar, but removes *any* trailing character, not just a trailing newline. Since that could accidentally turn pebbles into pebble, it's usually not what you want.

2.10. The while Control Structure

Like most algorithmic programming languages, Perl has a number of looping structures. The while loop repeats a block of code as long as a condition is true:

```
$count = 0;
while ($count < 10) {
  $count += 1;
  print "count is now $count\n"; # Gives values from 1 to 10
}
```

As always in Perl, the truth value here works like the truth value in the if test. Also like the if control structure, the block curly braces are required. The conditional expression is evaluated before the first iteration, so the loop may be skipped completely, if the condition is initially false.

2.11. The undef Value

What happens if you use a scalar variable before you give it a value? Nothing serious, and definitely nothing fatal. Variables have the special undef value before they are first assigned, which is just Perl's way of saying "nothing here to look at -- move along, move along." If you try to use this "nothing" as a "numeric something," it acts like 0. If you try to use it as a "string something," it acts like the empty string. But undef is neither a number nor a string; it's an entirely separate kind of scalar value.

Because undef automatically acts like zero when used as a number, it's easy to make an numeric accumulator that starts out empty:

```
# Add up some odd numbers
$n = 1;
while ($n < 10) {
  $sum += $n;
  $n += 2; # On to the next odd number
}
print "The total was $sum.\n";
```

This works properly when $sum was undef before the loop started. The first time through the loop, $n is one, so the first line inside the loop adds one to $sum. That's like adding one to a variable that already holds zero (because we're using undef as if it were a number). So now it has the value 1. After that, since it's been initialized, adding works in the traditional way.

Similarly, you could have a string accumulator that starts out empty:

```
$string .= "more text\n";
```

If $string is undef, this will act as if it already held the empty string, putting "more text\n" into that variable. But if it already holds a string, the new text is simply appended.

Perl programmers frequently use a new variable in this way, letting it act as either zero or the empty string as needed.

Many operators return `undef` when the arguments are out of range or don't make sense. If you don't do anything special, you'll get a zero or a null string without major consequences. In practice, this is hardly a problem. In fact, most programmers will rely upon this behavior. But you should know that when warnings are turned on, Perl will typically warn about unusual uses of the undefined value, since that may indicate a bug. For example, simply copying `undef` from one variable into another isn't a problem, but trying to `print` it would generally cause a warning.

2.12. The defined Function

One operator that can return `undef` is the line-input operator, `<STDIN>`. Normally, it will return a line of text. But if there is no more input, such as at end-of-file, it returns `undef` to signal this. To tell whether a value is `undef` and not the empty string, use the `defined` function, which returns false for `undef`, and true for everything else:

```
$madonna = <STDIN>;
if ( defined($madonna) ) {
  print "The input was $madonna";
} else {
  print "No input available!\n";
}
```

If you'd like to make your own `undef` values, you can use the obscurely named `undef` operator:

```
$madonna = undef; # As if it had never been touched
```

Chapter 3. Lists and Arrays

If a scalar was the "singular" in Perl, as we described them at the beginning of Chapter 2, "Scalar Data", the "plural" in Perl is represented by lists and arrays.

A *list* is an ordered collection of scalars. An *array* is a variable that contains a list. In Perl, the two terms are often used as if they're interchangeable. But, to be accurate, the list is the data, and the array is the variable. You can have a list value that isn't in an array, but every array variable holds a list (although that list may be empty). Figure 3-1 represents a list, whether it's stored in an array or not.

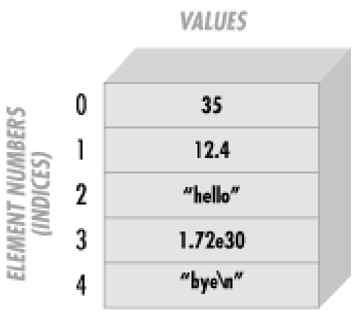

Figure 3-1. A list with five elements

Each *element* of an array or list is a separate scalar variable with an independent scalar value. These values are ordered -- that is, they have a particular sequence from the first to the last element. The elements of an array or list are *indexed* by small integers starting at zero and counting by ones, so the first element of any array or list is always element zero.

Since each element is an independent scalar value, a list or array may hold numbers, strings, `undef` values, or any mixture of different scalar values. Nevertheless, it's most common to have all elements of the same type, such as a list of book titles (all strings) or a list of cosines (all numbers).

Arrays and lists can have any number of elements. The smallest one has no elements, while the largest can fill all of available memory. Once again, this is in keeping with Perl's philosophy of "no unnecessary limits."

3.1. Accessing Elements of an Array

If you’ve used arrays in another language, you won’t be surprised to find that Perl provides a way to subscript an array in order to refer to an element by a numeric index.

The array elements are numbered using sequential integers, beginning at zero and increasing by one for each element, like this:

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";
```

The array name itself (in this case, `"fred"`) is from a completely separate namespace than scalars use; you could have a scalar variable named `$fred` in the same program, and Perl will treat them as different things, and wouldn’t be confused. (Your maintenance programmer might be confused, though, so don’t capriciously make all of your variable names the same!)

You can use an array element like `$fred[2]` in every place where you could use any other scalar variable like `$fred`. For example, you can get the value from an array element or change that value by the same sorts of expressions we used in the previous chapter:

```
print $fred[0];
$fred[2] = "diddley";
$fred[1] .= "whatsis";
```

Of course, the subscript may be any expression that gives a numeric value. If it’s not an integer already, it’ll automatically be truncated to the next lower integer:

```
$number = 2.71828;
print $fred[$number - 1]; # Same as printing $fred[1]
```

If the subscript indicates an element that would be beyond the end of the array, the corresponding value will be `undef`. This is just as with ordinary scalars; if you’ve never stored a value into the variable, it’s `undef`.

```
$blank = $fred[ 142_857 ]; # unused array element gives undef
$blanc = $mel;             # unused scalar $mel also gives undef
```

3.2. Special Array Indices

If you store into an array element that is beyond the end of the array, the array is automatically extended as needed -- there’s no limit on its length, as long as there’s available memory for Perl to use. If intervening elements need to be created, they’ll be created as `undef` values.

```
$rocks[0] = 'bedrock';       # One element...
$rocks[1] = 'slate';         # another...
$rocks[2] = 'lava';          # and another...
$rocks[3] = 'crushed rock'; # and another...
$rocks[99] = 'schist';       # now there are 95 undef elements
```

Sometimes, you need to find out the last element index in an array. For the array of `rocks` that we’ve just been using, the last element index is `$#rocks`. That’s not the same as the number of elements, though, because there’s an element number zero. As seen in the code snippet below, it’s actually possible to assign to this value to change the size of the array, although this is rare in practice.

```
$end = $#rocks;                  # 99, which is the last element's index
$number_of_rocks = $end + 1;     # okay, but we'll see a better way later
$#rocks = 2;                     # Forget all rocks after 'lava'
$#rocks = 99;                    # add 97 undef elements (the forgotten rocks are
                                 # gone forever)
$rocks[ $#rocks ] = 'hard rock'; # the last rock
```

Using the `$#name` value as an index, like that last example, happens often enough that Larry has provided a shortcut:

negative array indices count from the end of the array. But don't get the idea that these indices "wrap around." If you've got three elements in the array, the valid negative indices are `-1` (the last element), `-2` (the middle element), and `-3` (the first element). In the real world, nobody seems to use any of these except `-1`, though.

```
$rocks[ -1 ] = 'hard rock'; # easier way to do that last example above
$dead_rock = $rocks[-100];  # gets 'bedrock'
$rocks[ -200 ] = 'crystal'; # fatal error!
```

3.3. List Literals

A *list literal* (the way you represent a list value within your program) is a list of comma-separated values enclosed in parentheses. These values form the elements of the list. For example:

```
(1, 2, 3)      # list of three values 1, 2, and 3
(1, 2, 3,)     # the same three values (the trailing comma is ignored)
("fred", 4.5)  # two values, "fred" and 4.5
( )             # empty list - zero elements
(1..100)       # list of 100 integers
```

That last one uses the .. *range operator,* which is seen here for the first time. That operator creates a list of values by counting from the left scalar up to the right scalar by ones. For example:

```
(1..5)            # same as (1, 2, 3, 4, 5)
(1.7..5.7)        # same thing - both values are truncated
(5..1)            # empty list - .. only counts "uphill"
(0, 2..6, 10, 12) # same as (0, 2, 3, 4, 5, 6, 10, 12)
($a..$b)          # range determined by current values of $a and $b
(0..$#rocks)      # the indices of the rocks array from the previous section
```

As you can see from those last two items, the elements of an array are not necessarily constants -- they can be expressions that will be newly evaluated each time the literal is used. For example:

```
($a, 17)       # two values: the current value of $a, and 17
($b+$c, $d+$e) # two values
```

Of course, a list may have any scalar values, like this typical list of strings:

```
("fred", "barney", "betty", "wilma", "dino")
```

3.3.1. The qw Shortcut

It turns out that lists of simple words (like the previous example) are frequently needed in Perl programs. The `qw` shortcut makes it easy to generate them without typing a lot of extra quote marks:

```
qw/ fred barney betty wilma dino / # same as above, but less typing
```

`qw` stands for "quoted words" or "quoted by whitespace," depending upon whom you ask. Either way, Perl treats it like a single-quoted string (so, you can't use `\n` or `$fred` inside a `qw` list as you would in a double-quoted string). The whitespace (characters like spaces, tabs, and newlines) will be discarded, and whatever is left becomes the list of items. Since whitespace is discarded, here's another (but unusual) way to write that same list:

```
qw/fred
  barney     betty
wilma dino/   # same as above, but pretty strange whitespace
```

Since `qw` is a form of quoting, though, you can't put comments inside a `qw` list.

The previous two examples have used forward slashes as the delimiter, but Perl actually lets you choose any punctuation character as the delimiter. Here are some of the common ones:

```
qw! fred barney betty wilma dino !
```

```
qw# fred barney betty wilma dino #   # like in a comment!
qw( fred barney betty wilma dino )
qw{ fred barney betty wilma dino }
qw[ fred barney betty wilma dino ]
qw< fred barney betty wilma dino >
```

As those last four show, sometimes the two delimiters can be different. If the opening delimiter is one of those "left" characters, the corresponding "right" character is the proper closing delimiter. Other delimiters use the same character for start and finish.

If you need to include the closing delimiter *within* the string as one of the characters, you probably picked the wrong delimeter. But even if you can't or don't want to change the delimiter, you can still include the character using the backslash:

```
qw! yahoo\! google excite lycos ! # include yahoo! as an element
```

As in single-quoted strings, two consecutive backslashes contribute one single backslash to the item.

Now, although the Perl motto is "There's More Than One Way To Do It," you may well wonder why anyone would need all of those different ways! Well, we'll see later that there are other kinds of quoting where Perl uses this same rule, and it can come in handy in many of those. But even here, it could be useful if you were to need a list of Unix filenames:

```
qw{
  /usr/dict/words
  /home/rootbeer/.ispell_english
}
```

That list would be quite inconvenient to read, write, and maintain if the slash were the only delimiter available.

3.4. List Assignment

In much the same way as scalar values may be assigned to variables, list values may also be assigned to variables:

```
($fred, $barney, $dino) = ("flintstone", "rubble", undef);
```

All three variables in the list on the left get new values, just as if we did three separate assignments. Since the list is built up before the assignment starts, this makes it easy to swap two variables' values in Perl:

```
($fred, $barney) = ($barney, $fred); # swap those values
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

But what happens if the number of variables (on the left side of the equals sign) isn't the same as the number of values (from the right side)? In a list assignment, extra values are silently ignored -- Perl figures that if you wanted those values stored somewhere, you would have told it where to store them. Alternatively, if you have too many variables, the extras get the value `undef`.

```
($fred, $barney) = qw< flintstone rubble slate granite >; # two ignored items
($wilma, $dino) = qw[flintstone];                         # $dino gets undef
```

Now that we can assign lists, you *could* build up an array of strings with a line of code like this:

```
($rocks[0], $rocks[1], $rocks[2], $rocks[3]) = qw/talc mica feldspar quartz/;
```

But when you wish to refer to an entire array, Perl has a simpler notation. Just use the at-sign (`@`) before the name of the array (and no index brackets after it) to refer to the entire array at once. You can read this as "all of the," so `@rocks` is "all of the rocks." This works on either side of the assignment operator:

```
@rocks = qw/ bedrock slate lava /;
@tiny = ( );                        # the empty list
@giant = 1..1e5;                   # a list with 100,000 elements
@stuff = (@giant, undef, @giant); # a list with 200,001 elements
$dino = "granite";
```

```
    @quarry = (@rocks, "crushed rock", @tiny, $dino);
```

That last assignment gives `@quarry` the five-element list `(bedrock, slate, lava, crushed rock, granite)`, since `@tiny` contributes zero elements to the list. (In particular, it doesn't put an `undef` item into the list -- but we could do that explicitly, as we did with `@stuff` earlier.) It's also worth noting that an array name is replaced by the list it contains. An array doesn't become an element in the list, because these arrays can contain only scalars, not other arrays.

The value of an array variable that has not yet been assigned is `( )`, the empty list. Just as new, empty scalars start out with `undef`, new, empty arrays start out with the empty list.

It's worth noting that when an array is copied to another array, it's still a list assignment. The lists are simply stored in arrays. For example:

```
    @copy = @quarry; # copy a list from one array to another
```

3.4.1. The pop and push Operators

You *could* add new items to the end of an array by simply storing them into elements with new, larger indices. But real Perl programmers don't use indices. So in the next few sections, we'll present some ways to work with an array without using indices.

One common use of an array is as a stack of information, where new values are added to and removed from the right-hand side of the list. (This is the end with the "last" items in the array, the end with the highest index values.) These operations occur often enough to have their own special functions.

The `pop` operator takes the last element off of an array, and returns it:

```
    @array = 5..9;
    $fred = pop(@array);  # $fred gets 9, @array now has (5, 6, 7, 8)
    $barney = pop @array; # $barney gets 8, @array now has (5, 6, 7)
    pop @array;           # @array now has (5, 6). (The 7 is discarded.)
```

That last example uses `pop` "in a void context," which is merely a fancy way of saying the return value isn't going anywhere. There's nothing wrong with using `pop` in this way, if that's what you want.

If the array is empty, `pop` will leave it alone (since there is no element to remove), and it will return `undef`.

You may have noticed that `pop` may be used with or without parentheses. This is a general rule in Perl: as long as the meaning isn't changed by removing the parentheses, they're optional.

The converse operation is `push`, which adds an element (or a list of elements) to the end of an array:

```
    push(@array, 0);      # @array now has (5, 6, 0)
    push @array, 8;       # @array now has (5, 6, 0, 8)
    push @array, 1..10;   # @array now has those ten new elements
    @others = qw/ 9 0 2 1 0 /;
    push @array, @others; # @array now has those five new elements (19 total)
```

Note that the first argument to `push` or the only argument for `pop` must be an array variable -- pushing and popping would not make sense on a literal list.

3.4.2. The shift and unshift Operators

The `push` and `pop` operators do things to the end of an array (or the right side of an array, or the portion with the highest subscripts, depending upon how you like to think of it). Similarly, the `unshift` and `shift` operators perform the corresponding actions on the "start" of the array (or the "left" side of an array, or the portion with the lowest subscripts). Here are a few examples:

```
    @array = qw# dino fred barney #;
    $a = shift(@array);      # $a gets "dino", @array now has ("fred", "barney")
    $b = shift @array;       # $b gets "fred", @array now has ("barney")
    shift @array;            # @array is now empty
    $c = shift @array;       # $c gets undef, @array is still empty
```

```
unshift(@array, 5);      # @array now has the one-element list (5)
unshift @array, 4;       # @array now has (4, 5);
@others = 1..3;
unshift @array, @others; # @array now has (1, 2, 3, 4, 5)
```

Analogous to `pop`, `shift` returns `undef` if given an empty array variable.

3.5. Interpolating Arrays into Strings

Like scalars, array values may be interpolated into a double-quoted string. Elements of an array are automatically separated by spaces upon interpolation:

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";  # prints five rocks separated by spaces
```

There are no extra spaces added before or after an interpolated array; if you want those, you'll have to put them in yourself:

```
print "Three rocks are: @rocks.\n";
print "There's nothing in the parens (@empty) here.\n";
```

If you forget that arrays interpolate like this, you'll be surprised when you put an email address into a double-quoted string. For historical reasons, this is a fatal error at compile time:

```
$email = "fred@bedrock.edu";  # WRONG! Tries to interpolate @bedrock
$email = "fred\@bedrock.edu"; # Correct
$email = 'fred@bedrock.edu';  # Another way to do that
```

A single element of an array will be replaced by its value, just as you'd expect:

```
@fred = qw(hello dolly);
$y = 2;
$x = "This is $fred[1]'s place";    # "This is dolly's place"
$x = "This is $fred[$y-1]'s place"; # same thing
```

Note that the index expression is evaluated as an ordinary expression, as if it were outside a string. It is *not* variable-interpolated first. In other words, if `$y` contains the string `"2*4"`, we're still talking about element 1, not element 7, because `"2*4"` as a number (the value of `$y` used in a numeric expression) is just plain 2.

If you want to follow a simple scalar variable with a left square bracket, you need to delimit the square bracket so that it isn't considered part of an array reference, as follows:

```
@fred = qw(eating rocks is wrong);
$fred = "right";               # we are trying to say "this is right[3]"
print "this is $fred[3]\n";    # prints "wrong" using $fred[3]
print "this is ${fred}[3]\n";  # prints "right" (protected by braces)
print "this is $fred"."[3]\n"; # right again (different string)
print "this is $fred\[3]\n";   # right again (backslash hides it)
```

3.6. The foreach Control Structure

It's handy to be able to process an entire array or list, so Perl provides a control structure to do just that. The `foreach` loop steps through a list of values, executing one iteration (time through the loop) for each value:

```
foreach $rock (qw/ bedrock slate lava /) {
  print "One rock is $rock.\n";  # Prints names of three rocks
}
```

The control variable (`$rock` in that example) takes on a new value from the list for each iteration. The first time through the loop, it’s "bedrock"; the third time, it’s "lava".

The control variable is not a copy of the list element -- it actually *is* the list element. That is, if you modify the control variable inside the loop, you’ll be modifying the element itself, as shown in the following code snippet. This is useful, and supported, but it would surprise you if you weren’t expecting it.

```
@rocks = qw/ bedrock slate lava /;
foreach $rock (@rocks) {
  $rock = "\t$rock";              # put a tab in front of each element of @rocks
  $rock .= "\n";                  # put a newline on the end of each
}
print "The rocks are:\n", @rocks; # Each one is indented, on its own line
```

What is the value of the control variable after the loop has finished? It’s the same as it was before the loop started. The value of the control variable of a foreach loop is automatically saved and restored by Perl. While the loop is running, there’s no way to access or alter that saved value. So after the loop is done, the variable has the value it had before the loop, or undef if it hadn’t had a value. That means that if you want to name your loop control variable "$rock", you don’t have to worry that maybe you’ve already used that name for another variable.

3.7. Perl’s Favorite Default: $_

If you omit the control variable from the beginning of the foreach loop, Perl uses its favorite default variable, $_. This is (mostly) just like any other scalar variable, except for its unusual name. For example:

```
foreach (1..10) {  # Uses $_ by default
  print "I can count to $_!\n";
}
```

Although this isn’t Perl’s only default by a long shot, it’s Perl’s most common default. We’ll see many other cases in which Perl will automatically use $_ when you don’t tell it to use some other variable or value, thereby saving the programmer from the heavy labor of having to think up and type a new variable name. So as not to keep you in suspense, one of those cases is print, which will print $_ if given no other argument:

```
$_ = "Yabba dabba doo\n";
print;  # prints $_ by default
```

3.7.1. The reverse Operator

The reverse operator takes a list of values (which may come from an array) and returns the list in the opposite order. So if you were disappointed that the range operator, .., only counts upwards, this is the way to fix it:

```
@fred = 6..10;
@barney = reverse(@fred); # gets 10, 9, 8, 7, 6
@wilma = reverse 6..10;   # gets the same thing, without the other array
@fred = reverse @fred;    # puts the result back into the original array
```

The last line is noteworthy because it uses @fred twice. Perl always calculates the value being assigned (on the right) before it begins the actual assignment.

Remember that reverse returns the reversed list; it doesn’t affect its arguments. If the return value isn’t assigned anywhere, it’s useless:

```
reverse @fred;         # WRONG - doesn’t change @fred
@fred = reverse @fred; # that’s better
```

3.7.2. The sort Operator

The sort operator takes a list of values (which may come from an array) and sorts them in the internal character ordering. For ASCII strings, that would be ASCIIbetical order. Of course, ASCII is a strange place where all of the capital letters come before all of the lowercase letters, where the numbers come before the letters, and the punctuation marks -- well, those are here, there, and everywhere. But sorting in ASCII order is just the *default* behavior; we’ll see in Chapter 15, "Strings and

Sorting", *Strings and Sorting*, how to sort in whatever order you'd like:

```
@rocks = qw/ bedrock slate rubble granite /;
@sorted = sort(@rocks);      # gets bedrock, granite, rubble, slate
@back = reverse sort @rocks; # these go from slate to bedrock
@rocks = sort @rocks;        # puts sorted result back into @rocks
@numbers = sort 97..102;     # gets 100, 101, 102, 97, 98, 99
```

As you can see from that last example, sorting numbers as if they were strings may not give useful results. But, of course, any string that starts with `1` has to sort before any string that starts with `9`, according to the default sorting rules. And like what happened with `reverse`, the arguments themselves aren't affected. If you want to sort an array, you must store the result back into that array:

```
sort @rocks;           # WRONG, doesn't modify @rocks
@rocks = sort @rocks; # Now the rock collection is in order
```

3.8. Scalar and List Context

This is the most important section in this chapter. In fact, it's the most important section in the entire book. In fact, it wouldn't be an exaggeration to say that your entire career in using Perl will depend upon understanding this section. So if you've gotten away with skimming the text up to this point, this is where you should really pay attention.

That's not to say that this section is in any way difficult to understand. It's actually a simple idea: a given expression may mean different things depending upon where it appears. This is nothing new to you; it happens all the time in natural languages. For example, in English, suppose someone asked you what the word "read" means. It has different meanings depending on how it's used. You can't identify the meaning, until you know the *context*.

The *context* refers to where an expression is found. As Perl is parsing your expressions, it always expects either a scalar value or a list value. What Perl expects is called the context of the expression.

```
5 + something  # The something must be a scalar
sort something # The something must be a list
```

Even if *something* is the exact same sequence of characters, in one case it may give a single, scalar value, while in the other, it may give a list.

Expressions in Perl always return the appropriate value for their context. For example, how about the "name" of an array. In a list context, it gives the list of elements. But in a scalar context, it returns the number of elements in the array:

```
@people = qw( fred barney betty );
@sorted = sort @people; # list context: barney, betty, fred
$number = 5 + @people;  # scalar context: 5 + 3 gives 8
```

Even ordinary assignment (to a scalar or a list) causes different contexts:

```
@list = @people; # a list of three people
$n = @people;    # the number 3
```

But please don't jump to the conclusion that scalar context always gives the number of elements that would have been returned in list context. Most list-producing expressions return something *much* more interesting than that.

3.8.1. Using List-Producing Expressions in Scalar Context

There are many expressions that would typically be used to produce a list. If you use one in a scalar context, what do you get? See what the author of that operation says about it. Usually, that person is Larry, and usually the documentation gives the whole story. In fact, a big part of learning Perl is actually learning how Larry thinks. Therefore, once you can think like Larry does, you know what Perl should do. But while you're learning, you'll probably need to look into the documentation.

Some expressions don't have a scalar-context value at all. For example, what should `sort` return in a scalar context? You wouldn't need to sort a list to count its elements, so until someone implements something else, `sort` in a scalar context always returns `undef`.

Another example is `reverse`. In a list context, it gives a reversed list. In a scalar context, it returns a reversed string (or reversing the result of concatenating all the strings of a list, if given one):

```
@backwards = reverse qw/ yabba dabba doo /;
   # gives doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /;
   # gives oodabbadabbay
```

At first, it's not always obvious whether an expression is being used in a scalar or a list context. But, trust us, it *will* get to be second nature for you eventually.

Here are some common contexts to start you off:

```
$fred = something;             # scalar context
@pebbles = something;          # list context
($wilma, $betty) = something; # list context
($dino) = something;           # still list context!
```

Don't be fooled by the one-element list; that last one is a list context, not a scalar one. If you're assigning to a list (no matter the number of elements), it's a list context. If you're assigning to an array, it's a list context.

Here are some other expressions we've seen, and the contexts they provide. First, some that provide scalar context to *something*:

```
$fred = something;
$fred[3] = something;
123 + something
something + 654
if (something) { ... }
while (something) { ... }
$fred[something] = something;
```

And here are some that provide a list context:

```
@fred = something;
($fred, $barney) = something;
($fred) = something;
push @fred, something;
foreach $fred (something) { ... }
sort something
reverse something
print something
```

3.8.2. Using Scalar-Producing Expressions in List Context

Going this direction is straightforward: if an expression doesn't normally have a list value, the scalar value is automatically promoted to make a one-element list:

```
@fred = 6 * 7; # gets the one-element list (42)
@barney = "hello" . ' ' . "world";
```

Well, there's one possible catch:

```
@wilma = undef; # OOPS! Gets the one-element list (undef)
  # which is not the same as this:
@betty = ( );    # A correct way to empty an array
```

Since `undef` is a scalar value, assigning `undef` to an array doesn't clear the array. The better way to do that is to assign an empty list.

3.8.3. Forcing Scalar Context

On occasion, you may need to force scalar context where Perl is expecting a list. In that case, you can use the fake function `scalar`. It's not a true function, because it just tells Perl to provide a scalar context:

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
print "I have ", @rocks, " rocks!\n";        # WRONG, prints names of rocks
print "I have ", scalar @rocks, " rocks!\n"; # Correct, gives a number
```

Oddly enough, there's no corresponding function to force list context. It turns out never to be needed. Trust us on this, too.

3.9. <STDIN> in List Context

One previously seen operator that returns a different value in an array context is the line-input operator, `<STDIN>`. As described earlier, `<STDIN>` returns the next line of input in a scalar context. Now, in list context, this operator returns *all* of the remaining lines up to the end of file. Each line is returned as a separate element of the list. For example:

```
@lines = <STDIN>; # read standard input in list context
```

When the input is coming from a file, this will read the rest of the file. But how can there be an end-of-file when the input comes from the keyboard? On Unix and similar systems, including Linux and Mac OS X, you'll normally type a Control-D to indicate to the system that there's no more input; the special character itself is never seen by Perl, even though it may be echoed to the screen. On DOS/Windows systems, use Ctrl-Z instead. You'll need to check the documentation for your system or ask your local expert, if it's different from these.

If the person running the program types three lines, then presses the proper keys needed to indicate end-of-file, the array ends up with three elements. Each element will be a string that ends in a newline, corresponding to the three newline-terminated lines entered.

Wouldn't it be nice if, having read those lines, you could `chomp` the newlines all at once? It turns out that if you give `chomp` a list of lines, it will remove the newlines from each item in the list. For example:

```
@lines = <STDIN>; # Read all the lines
chomp(@lines);    # discard all the newline characters
```

But the more common way to write that is with code similar to what we used earlier:

```
chomp(@lines = <STDIN>); # Read the lines, not the newlines
```

Although you're welcome to write your code either way in the privacy of your own cubicle, most Perl programmers will expect the second, more compact, notation.

It may be obvious to you (but it's not obvious to everyone) that once these lines of input have been read, they can't be re-read. Once you've reached end-of-file, there's no more input out there to read.

And what happens if the input is coming from a 400MB log file? The line input operator reads all of the lines, gobbling up lots of memory. Perl tries not to limit you in what you can do, but the other users of your system (not to mention your system administrator) are likely to object. If the input data is large, you should generally find a way to deal with it without reading it all into memory at once.

Chapter 5. Hashes

In this chapter, we will see one of Perl's features that makes Perl one of the world's truly great programming languages -- *hashes.* Although hashes are a powerful and useful feature, you may have used other powerful languages for years without ever hearing of hashes. But you'll use hashes in nearly every Perl program you'll write from now on; they're that important.

5.1. What Is a Hash?

A hash is a data structure, not unlike an array in that it can hold any number of values and retrieve them at will. But instead of indexing the values by *number*, as we did with arrays, we'll look up the values by *name*. That is, the *indices* (here, we'll call them *keys*) aren't numbers, but instead they are arbitrary unique strings (see Figure 5-1).

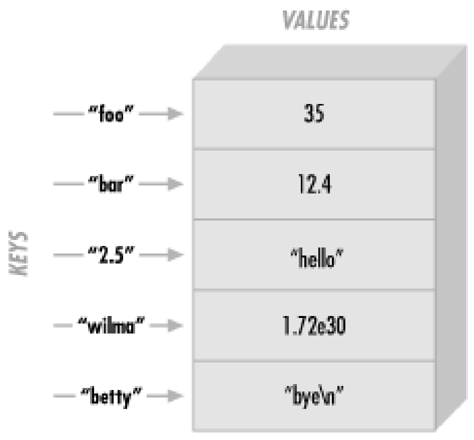

Figure 5-1. Hash keys and values

The keys are *strings*, first of all, so instead of getting element number `3` from an array, we'll be accessing the hash element named `wilma`.

These keys are arbitrary strings -- you can use any string expression for a hash key. And they are unique strings -- just as there's only one array element numbered `3`, there's only one hash element named `wilma`.

Another way to think of a hash is that it's like a barrel of data, where each piece of data has a tag attached. You can reach into the barrel and pull out any tag and see what piece of data is attached. But there's no "first" item in the barrel; it's just a jumble. In an array, we'd start with element `0`, then element `1`, then element `2`, and so on. But in a hash, there's no fixed order, no first element. It's just a collection of key-value pairs.

The keys and values are both arbitrary scalars, but the keys are always converted to strings. So, if you used the numeric expression `50/20` as the key, it would be turned into the three-character string `"2.5"`, which is one of the keys shown in the diagram above.

As usual, Perl's no-unnecessary-limits philosophy applies: a hash may be of any size, from an empty hash with zero key-value pairs, up to whatever fills up your memory.

Some implementations of hashes (such as in the original *awk* language, from where Larry borrowed the idea) slow down as the hashes get larger and larger. This is not the case in Perl -- it has a good, efficient, scalable algorithm. So, if a hash has only three key-value pairs, it's very quick to "reach into the barrel" and pull out any one of those. If the hash has three *million* key-value pairs, it should be just about as quick to pull out any one of those. A big hash is nothing to fear.

It's worth mentioning again that the keys are always unique, although the values may be duplicated. The values of a hash may be all numbers, all strings, `undef` values, or a mixture. But the keys are all arbitrary, unique strings.

5.1.1. Why Use a Hash?

When you first hear about hashes, especially if you've lived a long and productive life as a programmer using other languages that don't have hashes, you may wonder why anyone would want one of these strange beasts. Well, the general idea is that you'll have one set of data "related to" another set of data. For example, here are some hashes you might find in typical applications of Perl:

Given name, family name
: The given name (first name) is the key, and the family name is the value. This requires unique given names, of course; if there were two people named `randal`, this wouldn't work. With this hash, you can look up anyone's given name, and find the corresponding family name. If you use the key `tom`, you get the value `phoenix`.

Host name, IP address
: You may know that each computer on the Internet has both a host name (like www.stonehenge.com) and an IP address number (like 123.45.67.89). That's because machines like working with the numbers, but we humans have an easier time remembering the names. The host names are unique strings, so they can be used to make this hash. With this hash, you could look up a host name and find the corresponding IP address number.

IP address, host name

Or you could go in the opposite direction. We generally think of the IP address as a number, but it can also be a unique string, so it's suitable for use as a hash key. In this hash, we can look up the IP address number to determine the corresponding host name. Note that this is *not* the same hash as the previous example: hashes are a one-way street, running from key to value; there's no way to look up a value in a hash and find the corresponding key! So these two are a *pair* of hashes, one for storing IP addresses, one for host names. It's easy enough to create one of these given the other, though, as we'll see below.

Word, count of number of times that word appears

The idea here is that you want to know how often each word appears in a given document. Perhaps you're building an index to a number of documents, so that when a user searches for `fred`, you'll know that a certain document mentions `fred` five times, another mentions `fred` seven times, and yet another doesn't mention `fred` at all -- so you'll know which documents the user is likely to want. As the index-making program reads through a given document, each time it sees a mention of `fred`, it adds one to the value filed under the key of `fred`. That is, if we had seen `fred` twice already in this document, the value would be `2`, but now we'll increment it to `3`. If we had never seen `fred` before, we'd change the value from `undef` (the implicit, default value) to `1`.

Username, number of disk blocks they are using [wasting]

System administrators like this one: the usernames on a given system are all unique strings, so they can be used as keys in a hash to look up information about that user.

Driver's license number, name

There may be many, many people named John Smith, but we hope that each one has a different driver's license number. That number makes for a unique key, and the person's name is the value.

So, yet another way to think of a hash is as a *very* simple database, in which just one piece of data may be filed under each key. In fact, if your task description includes phrases like "finding duplicates," "unique," "cross-reference," or "lookup table," it's likely that a hash will be useful in the implementation.

5.2. Hash Element Access

To access an element of a hash, use syntax that looks like this:

```
$hash{$some_key}
```

This is similar to what we used for array access, but here we use curly braces instead of square brackets around the subscript (key). And that key expression is now a string, rather than a number:

```
$family_name{"fred"} = "flintstone";
$family_name{"barney"} = "rubble";
```

Figure 5-2 shows how the resulting hash keys are assigned.

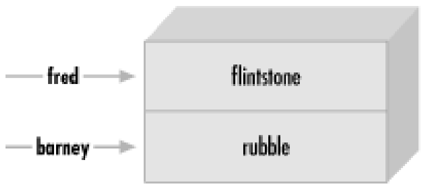

Figure 5-2. Assigned hash keys

This lets us use code like this:

```
foreach $person (qw< barney fred >) {
  print "I've heard of $person $family_name{$person}.\n";
}
```

The name of the hash is like any other Perl identifier (letters, digits, and underscores, but can't start with a digit). And it's from a separate namespace; that is, there's no connection between the hash element `$family_name{"fred"}` and a subroutine `&family_name`, for example. Of course, there's no reason to confuse everyone by giving everything the same name. But Perl won't mind if you also have a scalar called `$family_name` and array elements like `$family_name[5]`. We humans will have to do as Perl does; that is, we'll have to look to see what punctuation appears before and after the identifier to see what it means. When there is a dollar sign in front of the name and curly braces afterwards, it's a hash element that's being accessed.

When choosing the name of a hash, it's often nice to think of the word "of" between the name of the hash and the key. As in, "the `family_name` of `fred` is `flintstone`". So the hash is named `family_name`. Then it becomes clear what the relationship is between the keys and their values.

Of course, the hash key may be any expression, not just the literal strings and simple scalar variables that we're showing here:

```
$foo = "bar";
print $family_name{ $foo . "ney" };  # prints "rubble"
```

When you store something into an existing hash element, that overwrites the previous value:

```
$family_name{"fred"} = "astaire";  # gives new value to existing element
$bedrock = $family_name{"fred"};   # gets "astaire"; old value is lost
```

That's analogous to what happens with arrays and scalars; if you store something new into `$pebbles[17]` or `$dino`, the old value is replaced. If you store something new into `$family_name{"fred"}`, the old value is replaced as well.

Hash elements will spring into existence by assignment:

```
$family_name{"wilma"} = "flintstone";             # adds a new key (and value)
$family_name{"betty"} .= $family_name{"barney"};  # creates the element if needed
```

That's also just like what happens with arrays and scalars; if you didn't have `$pebbles[17]` or `$dino` before, you will have it after you assign to it. If you didn't have `$family_name{"betty"}` before, you do now.

And accessing outside the hash gives `undef`:

```
$granite = $family_name{"larry"};  # No larry here: undef
```

Once again, this is just like what happens with arrays and scalars; if there's nothing yet stored in `$pebbles[17]` or `$dino`, accessing them will yield `undef`. If there's nothing yet stored in `$family_name{"larry"}`, accessing it will yield `undef`.

5.2.1. The Hash as a Whole

To refer to the entire hash, use the percent sign ("`%`") as a prefix. So, the hash we've been using for the last few pages is actually called `%family_name`.

For convenience, a hash may be converted into a list, and back again. Assigning to a hash (in this case, the one from Figure 5-1) is a list-context assignment, where the list is made of key-value pairs:

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
      "wilma", 1.72e30, "betty", "bye\n");
```

The value of the hash (in a list context) is a simple list of key-value pairs:

```
@any_array = %some_hash;
```

We call this *unwinding* the hash; turning it back into a list of key-value pairs. Of course, the pairs won't necessarily be in the same order as the original list:

```
print "@any_array\n";
  # might give something like this:
  #  betty bye (and a newline) wilma 1.72e+30 foo 35 2.5 hello bar 12.4
```

The order is jumbled because Perl keeps the key-value pairs in an order that's convenient for Perl, so that it can look up any item quickly. So you'd normally use a hash either when you don't care what order the items are in, or when you have an easy way to put them into the order you want.

Of course, even though the order of the key-value pairs is jumbled, each key "sticks" with its corresponding value in the resulting list. So, even though we don't know where the key `foo` will appear in the list, we know that its value, `35`, will be right after it.

5.2.2. Hash Assignment

It's rare to do so, but a hash may be copied using the obvious syntax:

```
%new_hash = %old_hash;
```

This is actually more work for Perl than meets the eye. Unlike what happens in languages like Pascal or C, where such an operation would be a simple matter of copying a block of memory, Perl's data structures are more complex. So, that line of code tells Perl to unwind the `%old_hash` into a list of key-value pairs, then assign those to `%new_hash`, building it up one key-value pair at a time.

It's more common to transform the hash in some way, though. For example, we could make an inverse hash:

```
%inverse_hash = reverse %any_hash;
```

This takes `%any_hash` and unwinds it into a list of key-value pairs, making a list like `(`*key*, *value*, *key*, *value*, *key*, *value*, ...`)`. Then `reverse` turns that list end-for-end, making a list like `(`*value*, *key*, *value*, *key*, *value*, *key*, ...`)`. Now the keys are where the values used to be, and the values are where the keys used to be. When that's stored into `%inverse_hash`, we'll be able to look up a string that was a value in `%any_hash` -- it's now a key of `%inverse_hash`. And the value we'll find is one that was one of the keys from `%any_hash`. So, we have a way to look up a "value" (now a key), and find a "key" (now a value).

Of course, you might guess (or determine from scientific principles, if you're clever) that this will work properly only if the values in the original hash were unique -- otherwise we'd have duplicate keys in the new hash, and keys are always unique. Here's the rule that Perl uses: the last one in wins. That is, the later items in the list overwrite any earlier ones. Of course, we don't know what order the key-value pairs will have in this list, so there's no telling which ones would win. You'd use this technique only if you know there are no duplicates among the original values. But that's the case for the IP address and host name examples given earlier:

```
%ip_address = reverse %host_name;
```

Now we can look up a host name or IP address with equal ease, to find the corresponding IP address or host name.

5.2.3. The Big Arrow

When assigning a list to a hash, sometimes it's not obvious which elements are keys and which are values. For example, in this assignment (which we saw earlier), we humans have to count through the list, saying, "key, value, key, value...", in order to determine whether `2.5` is a key or a value:

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
      "wilma", 1.72e30, "betty", "bye\n");
```

Wouldn't it be nice if Perl gave us a way to pair up keys and values in that kind of a list, so that it would be easy to see which ones were which? Larry thought so, too, which is why he invented the big arrow, (`=>`). To Perl, it's just a different way to "spell" a comma. That is, in the Perl grammar, any time that you need a comma (`,`), you can use the big arrow instead; it's all the same to Perl. So here's another way to set up the hash of last names:

```
my %last_name = (  # a hash may be a lexical variable
  "fred" => "flintstone",
  "dino" => undef,
  "barney" => "rubble",
  "betty" => "rubble",
);
```

Here, it's easy (or perhaps at least easier) to see whose name pairs with which value, even if we end up putting many pairs on one line. And notice that there's an extra comma at the end of the list. As we saw earlier, this is harmless, but convenient; if we need to add additional people to this hash, we'll simply make sure that each line has a key-value pair and a trailing comma. Perl will see that there is a comma between each item and the next, and one extra (harmless) comma at the end of the list.

5.3. Hash Functions

Naturally, there are some useful functions that can work on an entire hash at once.

5.3.1. The keys and values Functions

The `keys` function yields a list of all the current keys in a hash, while the `values` function gives the corresponding values. If there are no elements to the hash, then either function returns an empty list:

```
my %hash = ("a" => 1, "b" => 2, "c" => 3);
my @k = keys %hash;
my @v = values %hash;
```

So, `@k` will contain `"a"`, `"b"`, and `"c"`, and `@v` will contain `1`, `2`, and `3` -- in *some* order. Remember, Perl doesn't maintain the order of elements in a hash. But, whatever order the keys are in, the values will be in the corresponding order: If `"b"` is last in the keys, `2` will be last in the values; if `"c"` is the first key, `3` will be the first value. That's true as long as you don't modify the hash between the request for the keys and the one for the values. If you add elements to the hash, Perl reserves the right to rearrange it as needed, to keep the access quick.

In a scalar context, these functions give the number of elements (key-value pairs) in the hash. They do this quite efficiently, without having to visit each element of the hash:

```
my $count = keys %hash;  # gets 3, meaning three key-value pairs
```

Once in a long while, you'll see that someone has used a hash as a Boolean (true/false) expression, something like this:

```
if (%hash) {
  print "That was a true value!\n";
}
```

That will be true if (and only if) the hash has at least one key-value pair. So, it's just saying, "if the hash is not empty...". But this is a pretty rare construct, as such things go.

5.3.2. The each Function

If you wish to iterate over (that is, examine every element of) an entire hash, one of the usual ways is to use the `each` function, which returns a key-value pair as a two-element list. On each evaluation of this function for the same array, the next successive key-value pair is returned, until all the elements have been accessed. When there are no more pairs, `each` returns an empty list.

In practice, the only way to use `each` is in a `while` loop, something like this:

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

There's a lot going on here. First, `each %hash` returns a key-value pair from the hash, as a two-element list; let's say that the key is `"c"` and the value is `3`, so the list is `("c", 3)`. That list is assigned to the list `($key, $value)`, so `$key` becomes `"c"`, and `$value` becomes `3`.

But that list assignment is happening in the conditional expression of the `while` loop, which is a scalar context. (Specifically, it's a Boolean context, looking for a true/false value; and a Boolean context is a particular kind of scalar context.) The value of a list assignment in a scalar context is the number of elements in the source list -- `2`, in this case. Since `2` is a true value, we enter the body of the loop and print the message `c => 3`.

The next time through the loop, `each %hash` gives a new key-value pair; let's say it's `("a", 1)` this time. (It knows to return a different pair than previously because it keeps track of where it is; in technical jargon, there's an iterator stored in with each hash.) Those two items are stored into `($key, $value)`. Since the number of elements in the source list was again `2`, a true value, the `while` condition is true, and the loop body runs again, telling us `a => 1`.

We go one more time through the loop, and by now we know what to expect, so it's no surprise to see `b => 2` appear in the output.

But we knew it couldn't go on forever. Now, when Perl evaluates `each %hash`, there are no more key-value pairs available. So, `each` has to return an empty list. The empty list is assigned to `($key, $value)`, so `$key` gets `undef`, and `$value` also gets `undef`.

But that hardly matters, because the whole thing is being evaluated in the conditional expression of the `while` loop. The value of a list assignment in a scalar context is the number of elements in the source list -- in this case, that's `0`. Since `0` is a false value, the `while` loop is done, and execution continues with the rest of the program.

Of course, `each` returns the key-value pairs in a jumbled order. (It's the same order as `keys` and `values` would give, incidentally; the "natural" order of the hash.) If you need to go through the hash in order, simply sort the keys, perhaps something like this:

```
foreach $key (sort keys %hash) {
  $value = $hash{$key};
  print "$key => $value\n";
  # Or, we could have avoided the extra $value variable:
  #  print "$key => $hash{$key}\n";
}
```

We'll see more about sorting hashes in Chapter 15, "Strings and Sorting".

5.4. Typical Use of a Hash

At this point, you may find it helpful to see a more concrete example.

The Bedrock library uses a Perl program in which a hash keeps track of how many books each person has checked out, among other information:

```
$books{"fred"} = 3;
$books{"wilma"} = 1;
```

It's easy to see whether an element of the hash is true or false, do this:

```
if ($books{$someone}) {
  print "$someone has at least one book checked out.\n";
}
```

But there are some elements of the hash that aren't true:

```
$books{"barney"} = 0;       # no books currently checked out
$books{"pebbles"} = undef;  # no books EVER checked out - a new library card
```

Since Pebbles has never checked out any books, her entry has the value of `undef`, rather than `0`.

There's a key in the hash for everyone who has a library card. For each key (that is, for each library patron), there's a value that is either a number of books checked out, or `undef` if that person's library card has never been used.

5.4.1. The exists Function

To see whether a key exists in the hash, (that is, whether someone has a library card or not), use the `exists` function, which returns a true value if the given key exists in the hash, whether the corresponding value is true or not:

```
if (exists $books{"dino"}) {
  print "Hey, there's a library card for dino!\n";
```

```
      }
```

That is to say, `exists $books{"dino"}` will return a true value if (and only if) `dino` is found in the list of keys from `keys %books`.

5.4.2. The delete Function

The `delete` function removes the given key (and its corresponding value) from the hash. (If there's no such key, its work is done; there's no warning or error in that case.)

```
      my $person = "betty";
      delete $books{$person};  # Revoke the library card for $person
```

Note that this is *not* the same as storing `undef` into that hash element -- in fact, it's precisely the opposite! Checking `exists ($books{"betty"})` will give opposite results in these two cases; after a `delete`, the key *can't* exist in the hash, but after storing `undef`, the key *must* exist.

5.4.3. Hash Element Interpolation

You can interpolate a single hash element into a double-quoted string just as you'd expect:

```
      foreach $person (sort keys %books) {            # for each library patron,in order
        if ($books{$person}) {
          print "$person has $books{$person} items\n";# fred has 3 items
        }
      }
```

But there's no support for entire hash interpolation; `"%books"` is just the six chararcters of (literally) `%books`. So we've seen all of the magical characters that need backslashing in double quotes: `$` and `@`, because they introduce a variable to be interpolated; `"`, since that's the quoting character that would otherwise end the double-quoted string; and `\`, the backslash itself. Any other characters in a double-quoted string are non-magical and should simply stand for themselves.

Chapter 6. I/O Basics

We've already seen how to do some input/output (I/O), in order to make some of the earlier exercises possible. But now we'll learn a little more about those operations. As the title of this chapter implies, there will be more about Perl's I/O operations in Chapter 11, "Filehandles".

6.1. Input from Standard Input

Reading from the standard input stream is easy. We've been doing it already with the `<STDIN>` operator. Evaluating this operator in a scalar context gives you the next line of input:

```
      $line = <STDIN>;                 # read the next line
      chomp($line);                    # and chomp it

      chomp($line = <STDIN>);          # same thing, more idiomatically
```

Since the line-input operator will return `undef` when you reach end-of-file, this is handy for dropping out of loops:

```
      while (defined($line = <STDIN>)) {
        print "I saw $line";
      {
```

There's a lot going on in that first line: we're reading the input into a variable, checking that it's defined, and if it is (meaning that we haven't reached the end of the input) we're running the body of the `while` loop. So, inside the body of the loop, we'll see each line, one after another, in `$line`. This is something you'll want to do fairly often, so naturally Perl has a shortcut for it. The shortcut looks like this:

```
while (<STDIN>) {
  print "I saw $_";
}
```

Now, to make this shortcut, Larry chose some useless syntax. That is, this is *literally* saying, "Read a line of input, and see if it's true. (Normally it is.) And if it is true, enter the `while` loop, but *throw away that line of input!*" Larry knew that it was a useless thing to do; nobody should ever need to do that in a real Perl program. So, Larry took this useless syntax and made it useful.

What this is *actually* saying is that Perl should do the same thing as we saw in our earlier loop: it tells Perl to read the input into a variable, and (as long as the result was defined, so we haven't reached end-of file) then enter the `while` loop. However, instead of storing the input into `$line`, Perl will use its favorite default variable, `$_`, just as if you had written this:

```
while (defined($_ = <STDIN>)) {
  print "I saw $_";
}
```

Now, before we go any further, we must be very clear about something: this shortcut works *only* if you write it just as we did. If you put a line-input operator anywhere else (in particular, as a statement all on its own) it won't read a line into `$_` by default. It works *only* if there's nothing but the line-input operator in the conditional of a `while` loop. If you put anything else into the conditional expression, this shortcut won't apply.

There's no connection between the line-input operator (`<STDIN>`) and Perl's favorite default variable (`$_`). In this case, though, it just happens that the input is being stored in that variable.

On the other hand, evaluating the line-input operator in a list context gives you all of the (remaining) lines of input as a list -- each element of the list is one line:

```
foreach (<STDIN>) {
  print "I saw $_";
}
```

Once again, there's no connection between the line-input operator and Perl's favorite default variable. In this case, though, the default control variable for `foreach` is `$_`. So in this loop, we'll see each line of input in `$_`, one after the other.

That may sound familiar, and for good reason: That's the same behavior as the `while` loop would do. Isn't it?

The difference is under the hood. In the `while` loop, Perl reads a line of input, puts it into a variable, and runs the body of the loop. Then, it goes back to find another line of input. But in the `foreach` loop, the line-input operator is being used in a list context (since `foreach` needs a list to iterate through). So it has to read all of the input before the loop can start running. That difference will become apparent when the input is coming from your 400MB web server log file! It's generally best to use code like the `while` loop's shortcut, which will process input a line at a time, whenever possible.

6.2. Input from the Diamond Operator

Another way to read input is with the diamond operator: <>. This is useful for making programs that work like standard Unix utilities, with respect to the invocation arguments (which we'll see in a moment). If you want to make a Perl program that can be used like the utilities *cat*, *sed*, *awk*, *sort*, *grep*, *lpr*, and many others, the diamond operator will be your friend. If you want to make anything else, the diamond operator probably won't help.

The *invocation arguments* to a program are normally a number of "words" on the command line after the name of the program. In this case, they give the names of a number of files to be processed in sequence:

```
$ ./my_program fred barney betty
```

That command means to run the command *my_program* (which will be found in the current directory), and that it should process file *fred*, followed by file *barney*, followed by file *betty*.

If you give no invocation arguments, the program should process the standard input stream. Or, as a special case, if you give just a hyphen as one of the arguments, that means standard input as well. So, if the invocation arguments had been `fred -`

`betty`, that would have meant that the program should process file *fred*, followed by the standard input stream, followed by file *betty*.

The benefit of making your programs work like this is that you may choose where the program gets its input at run time; for example, you won't have to rewrite the program to use it in a pipeline (which we'll discuss more later). Larry put this feature into Perl because he wanted to make it easy for you to write your own programs that work like standard Unix utilities -- even on non-Unix machines. Actually, he did it so he could make his *own* programs work like standard Unix utilities; since some vendors' utilities don't work just like others', Larry could make his own utilities, deploy them on a number of machines, and know that they'd all have the same behavior. Of course, this meant porting Perl to every machine he could find.

The diamond operator is actually a special kind of line-input operator. But instead of getting the input from the keyboard, it comes from the user's choice of input:

```
while (defined($line = <>)) {
  chomp($line);
  print "It was $line that I saw!\n";
}
```

So, if we run this program with the invocation arguments `fred`, `barney`, and `betty`, it will say something like: "It was [a line from file *fred*] that I saw!", "It was [another line from file *fred*] that I saw!", on and on until it reaches the end of file `fred`. Then, it will automatically go on to file `barney`, printing out one line after another, and then on to file `betty`. Note that there's no break when we go from one file to another; when you use the diamond, it's as if the input files have been merged into one big file. The diamond will return `undef` (and we'll drop out of the `while` loop) only at the end of all of the input.

In fact, since this is just a special kind of line-input operator, we may use the same shortcut we saw earlier, to read the input into `$_` by default:

```
while (<>) {
  chomp;
  print "It was $_ that I saw!\n";
}
```

This works like the loop above, but with less typing. And you may have noticed that we're using the default for `chomp`; without an argument, `chomp` will work on `$_`. Every little bit of saved typing helps!

Since the diamond operator is generally being used to process all of the input, it's typically a mistake to use it in more than one place in your program. If you find yourself putting two diamonds into the same program, especially using the second diamond inside the `while` loop that is reading from the first one, it's almost certainly not going to do what you would like. In our experience, when beginners put a second diamond into a program, they meant to use `$_` instead. Remember, the diamond operator *reads* the input, but the input itself is (generally, by default) found in `$_`.

If the diamond operator can't open one of the files and read from it, it'll print an allegedly helpful diagnostic message, such as:

```
can't open wimla: No such file or directory
```

The diamond operator will then go on to the next file automatically, much like what you'd expect from *cat* or another standard utility.

6.3. The Invocation Arguments

Technically, the diamond operator isn't looking literally at the invocation arguments -- it works from the `@ARGV` array. This array is a special array that is preset by the Perl interpreter to be a list of the invocation arguments. In other words, this is just like any other array, (except for its funny, all-caps name), but when your program starts, `@ARGV` is already stuffed full of the list of invocation arguments.

You can use `@ARGV` just like any other array; you could `shift` items off of it, perhaps, or use `foreach` to iterate over it. You could even check to see if any arguments start with a hyphen, so that you could process them as invocation options (like Perl does with its own `-w` option).

This is how the diamond operator knows what filenames it should use: it looks in `@ARGV`. If it finds an empty list, it uses the standard input stream; otherwise it uses the list of files that it finds. This means that after your program starts and before you start using the diamond, you've got a chance to tinker with `@ARGV`. For example, here we can process three specific files, regardless of what the user chose on the command line:

```
@ARGV = qw# larry moe curly #;  # force these three files to be read
while (<>) {
  chomp;
  print "It was $_ that I saw in some stooge-like file!\n";
}
```

In Chapter 11, "Filehandles", we'll see how to open and close specific filenames at specific times. But this technique will suffice for the next few chapters.

6.4. Output to Standard Output

The `print` operator takes a list of values and sends each item (as a string, of course) to standard output in turn, one after another. It doesn't add any extra characters before, after, or in between the items; if you want spaces between items and a newline at the end, you have to say so:

```
$name = "Larry Wall";
print "Hello there, $name, did you know that 3+4 is ", 3+4, "?\n";
```

Of course, that means that there's a difference between printing an array and interpolating an array:

```
print @array;      # print a list of items
print "@array";    # print a string (containing an interpolated array)
```

That first `print` statement will print a list of items, one after another, with no spaces in between. The second one will print exactly one item, which is the string you get by interpolating `@array` into the empty string -- that is, it prints the contents of `@array`, separated by spaces. So, if `@array` holds `qw/ fred barney betty /`, the first one prints `fredbarneybetty`, while the second prints `fred barney betty` separated by spaces.

But before you decide to always use the second form, imagine that `@array` is a list of unchomped lines of input. That is, imagine that each of its strings has a trailing newline character. Now, the first `print` statement prints `fred`, `barney`, and `betty` on three separate lines. But the second one prints this:

```
fred
 barney
 betty
```

Do you see where the spaces come from? Perl is interpolating an array, so it puts spaces between the elements. So, we get the first element of the array (`fred` and a newline character), then a space, then the next element of the array (`barney` and a newline character), then a space, then the last element of the array (`betty` and a newline character). The result is that the lines seem to have become indented, except for the first one. Every week or two, a message appears on the newsgroup *comp.lang.perl.misc* with a subject line something like:

> Perl indents everything after the first line

Without even reading the message, we can immediately see that the program used double quotes around an array containing unchomped strings. "Did you perhaps put an array of unchomped strings inside double quotes?" we ask, and the answer is always yes.

Generally, if your strings contain newlines, you simply want to print them, after all:

```
print @array;
```

But if they don't contain newlines, you'll generally want to add one at the end:

```
print "@array\n";
```

So, if you're using the quote marks, you'll be (generally) adding the `\n` at the end of the string anyway; this should help you to remember which is which.

It's normal for your program's output to be *buffered* . That is, instead of sending out every little bit of output at once, it'll be saved until there's enough to bother with. That's because if (for example) the output were going to be saved on disk, it would be (relatively) slow and inefficient to spin the disk every time that one or two characters need to be added to the file. Generally, then, the output will go into a buffer that is *flushed* (that is, actually written to disk, or wherever) only when the buffer gets full, or when the output is otherwise finished (such as at the end of runtime). Usually, that's what you want.

But if you (or a program) may be waiting impatiently for the output, you may wish to take that performance hit and flush the output buffer each time you `print`. See the Perl manpages for more information on controlling buffering in that case.

Since `print` is looking for a list of strings to print, its arguments are evaluated in list context. Since the diamond operator (as a special kind of line-input operator) will return a list of lines in a list context, these can work well together:

```
print <>;          # source code for 'cat'

print sort <>;  # source code for 'sort'
```

Well, to be fair, the standard Unix commands *cat* and *sort* do have some additional functionality that these replacements lack. But you can't beat them for the price! You can now re-implement all of your standard Unix utilities in Perl, and painlessly port them to any machine that has Perl, whether that machine is running Unix or not. And you can be sure that the programs on every different type of machine will nevertheless have the same behavior.

What might not be obvious is that `print` has optional parentheses, which can sometimes cause confusion. Remember the rule that parentheses in Perl may always be omitted, except when doing so would change the meaning of a statement. So, here are two ways to print the same thing:

```
print("Hello, world!\n");
print "Hello, world!\n";
```

So far, so good. But another rule in Perl is that if the invocation of `print` *looks* like a function call, then it *is* a function call. It's a simple rule, but what does it mean for something to look like a function call?

In a function call, there's a function name immediately followed by parentheses around the function's arguments, like this:

```
print (2+3);
```

That looks like a function call, so it is a function call. It prints `5`, but then it returns a value like any other function. The return value of `print` is a true or false value, indicating the success of the print. It nearly always succeeds, unless you get some I/O error, so the `$result` in the following statement will normally be `1`:

```
$result = print("hello world!\n");
```

But what if you used the result in some other way? Let's suppose you decide to multiply the return value times four:

```
print (2+3)*4;  # Oops!
```

When Perl sees this line of code, it prints `5`, just as you asked. Then it takes the return value from `print`, which is `1`, and multiplies that times `4`. It then throws away the product, wondering why you didn't tell it to do something else with it. And at this point, someone looking over your shoulder says, "Hey, Perl can't do math! That should have printed `20`, rather than `5`!"

This is the problem with allowing the parentheses to be optional; sometimes we humans forget where the parentheses really belong. When there are no parentheses, `print` is a list operator, printing all of the items in the following list; that's generally what you'd expect. But when the first thing after `print` is a left parenthesis, `print` is a function call, and it will print only what's found inside the parentheses. Since that line had parentheses, it's the same to Perl as if you'd said this:

```
( print(2+3) ) * 4;  # Oops!
```

Fortunately, Perl itself can almost always help you with this, if you ask for warnings -- so use `-w`, at least during program development and debugging.

Actually, this rule -- "If it looks like a function call, it is a function call" -- applies to all list functions in Perl, not just to `print`. It's just that you're most likely to notice it with `print`. If `print` (or another function name) is followed by an open parenthesis, make sure that the corresponding close parenthesis comes after *all* of the arguments to that function.

6.5. Formatted Output with printf

You may wish to have a little more control with your output than `print` provides. In fact, you may be accustomed to the formatted output of C's `printf` function. Fear not -- Perl provides a comparable operation with the same name.

The `printf` operator takes a format string followed by a list of things to print. The format string is a fill-in-the-blanks template showing the desired form of the output:

```
printf "Hello, %s; your password expires in %d days!\n",
  $user, $days_to_die;
```

The format string holds a number of so-called *conversions* ; each conversion begins with a percent sign (`%`) and ends with a letter. (As we'll see in a moment, there may be significant extra characters between these two symbols.) There should be the same number of items in the following list as there are conversions; if these don't match up, it won't work correctly. In the example above, there are two items and two conversions, so the output might look something like this:

```
Hello, merlyn; your password expires in 3 days!
```

There are many possible `printf` conversions, so we'll take time here to describe just the most common ones. Of course, the full details are available in the `perlfunc` manpage.

To print a number in what's generally a good way, use `%g` , which automatically chooses floating-point, integer, or even exponential notation as needed:

```
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;  # 2.5 3 1.0683e+29
```

The `%d` format means a decimal integer, truncated as needed:

```
printf "in %d days!\n", 17.85;  # in 17 days!
```

Note that this is truncated, not rounded; we'll see how to round off a number in a moment.

In Perl, `printf` is most often used for columnar data, since most formats accept a field width. If the data won't fit, the field will generally be expanded as needed:

```
printf "%6d\n", 42;  # output like ````42 (the ` symbol stands for a space)
printf "%2d\n", 2e3 + 1.95;  # 2001
```

The `%s` conversion means a string, so it effectively interpolates the given value as a string, but with a given field width:

```
printf "%10s\n", "wilma";  # looks like `````wilma
```

A negative field width is left-justified (in any of these conversions):

```
printf "%-15s\n", "flintstone";  # looks like flintstone `````
```

The `%f` conversion (floating-point) rounds off its output as needed, and even lets you request a certain number of digits after the decimal point:

```
printf "%12f\n", 6 * 7 + 2/3;    # looks like ```42.666667
printf "%12.3f\n", 6 * 7 + 2/3;  # looks like ``````42.667
printf "%12.0f\n", 6 * 7 + 2/3;  # looks like ``````````43
```

To print a real percent sign, use `%%`, which is special in that it uses no element from the list:

```
printf "Monthly interest rate: %.2f%%\n",
  5.25/12;  # the value looks like "0.44%"
```

6.5.1. Arrays and printf

Generally, you won't use an array as an argument to `printf`. That's because an array may hold any number of items, and a given format string will work with only a certain fixed number of items: if there are three conversions in the format, there must be exactly three items.

But there's no reason you can't whip up a format string on the fly, since it may be any expression. This can be tricky to get right, though, so it may be handy (especially when debugging) to store the format into a variable:

```
my @items = qw( wilma dino pebbles );
my $format = "The items are:\n" . ("%10s\n" x @items);
## print "the format is <<$format>>\n"; # for debugging
printf $format, @items;
```

This uses the `x` operator (which we learned about in Chapter 2, "Scalar Data") to replicate the given string a number of times given by `@items` (which is being used in a scalar context). In this case, that's `3`, since there are three items, so the resulting format string is the same as if we had written it as `"The items are:\n%10s\n%10s\n%10s\n."` And the output prints each item on its own line, right-justified in a ten-character column, under a heading line. Pretty cool, huh? But not cool enough, because you can even combine these:

```
printf "The items are:\n".("%10s\n" x @items), @items;
```

Note that here we have `@items` being used once in a scalar context, to get its length, and once in a list context, to get its contents. Context is important.

Chapter 10. More Control Structures

In this chapter, we'll see some alternative ways to write Perl code. For the most part, these techniques don't make the language more powerful, but they make it easier or more convenient to get the job done. You don't have to use these techniques in your own code, but don't be tempted to skip this chapter -- you're certain to see these control structures in other people's code, sooner or later (in fact, you're absolutely certain to see these things in use by the time you finish reading this book).

10.1. The unless Control Structure

In an `if` control structure, the block of code is executed only when the conditional expression is true. If you want a block of code to be executed only when the conditional is false, change `if` to `unless` :

```
unless ($fred =~ /^[A-Z_]\w*$/i) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

Using `unless` says to run the block of code *unless* this condition is true. It's just like using an `if` test with the opposite condition. Another way to say it is that it's like having the `else` clause on its own. That is, whenever you see an `unless` that you don't understand, you can rewrite it (either in your head or in reality) to be an `if` test:

```
if ($fred =~ /^[A-Z_]\w*$/i) {
  # Do nothing
} else {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

It's no more or less efficient, and it should compile to the same internal byte codes. Or, another way to rewrite it would be to negate the conditional expression by using the negation operator (`!`):

```
if ( ! ($fred =~ /^[A-Z_]\w*$/i) ) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

Generally, you should pick the way of writing code that makes the most sense to you, since that will probably make the most sense to your maintenance programmer. If it makes the most sense to write `if` with a negation, do that. More often, however, you'll probably find it natural to use `unless`.

10.1.1. The else Clause with unless

You could even have an `else` clause with an `unless`. While this syntax is supported, it's potentially confusing:

```
unless ($mon =~ /^(Feb)/) {
  print "This month has at least thirty days.\n";
} else {
  print "Do you see what's going on here?\n";
}
```

Some people may wish to use this, especially when the first clause is very short (perhaps only one line) and the second is several lines of code. But we'd make this one a negated `if`, or maybe simply swap the clauses to make a normal `if`:

```
if ($mon =~ /^(Feb)/) {
  print "Do you see what's going on here?\n";
} else {
  print "This month has at least thirty days.\n";
}
```

It's important to remember that you're always writing code for two readers: the computer that will run the code and the human being who has to keep the code working. If the human can't understand what you've written, pretty soon the computer won't be doing the right thing either.

10.2. The until Control Structure

Sometimes you'll want to reverse the condition of a `while` loop. To do that, just use `until`:

```
until ($j > $i) {
  $j *= 2;
}
```

This loop runs until the conditional expression returns true. But it's really just a `while` loop in disguise, except that this one repeats as long as the conditional is false, rather than true. The conditional expression is evaluated before the first iteration, so this is still a zero-or-more-times loop, just like the `while` loop.

As with `if` and `unless`, you could rewrite any `until` loop to become a `while` loop by negating the condition. But generally, you'll find it simple and natural to use `until` from time to time.

10.3. Expression Modifiers

In order to have a more compact notation, an expression may be followed by a modifier that controls it. For example, the `if` modifier works in a way analogous to an `if` block:

```
print "$n is a negative number.\n" if $n < 0;
```

That gives exactly the same result as if we had used this code, except that we saved some typing by leaving out the parentheses and curly braces:

```
if ($n < 0) {
  print "$n is a negative number.\n";
}
```

As we've said, Perl folks generally like to avoid typing. And the shorter form reads like in English: print this message, if `$n` is less than zero.

Notice that the conditional expression is still evaluated first, even though it's written at the end. This is backwards from the usual left-to-right ordering; in understanding Perl code, we'll have to do as Perl's internal compiler does, and read to the end of the statement before we can tell what it's really doing.

There are other modifiers as well:

```
&error("Invalid input") unless &valid($input);
$i *= 2 until $i > $j;
print " ", ($n += 2) while $n < 10;
&greet($_) foreach @person;
```

These all work just as (we hope) you would expect. That is, each one could be rewritten in a similar way to rewriting the `if`-modifier example earlier. Here is one:

```
while ($n < 10) {
  print " ", ($n += 2);
}
```

The expression in parentheses inside the `print` argument list is noteworthy because it adds two to `$n`, storing the result back into `$n`. Then it returns that new value, which will be printed.

These shorter forms read almost like a natural language: call the `&greet` subroutine for each `@person` in the list. Double `$i` until it's larger than `$j`.

One of the common uses of these modifiers is in a statement like this one:

```
print "fred is '$fred', barney is '$barney'\n"           if $I_am_curious;
```

By writing the code "in reverse" like this, you can put the important part of the statement at the beginning. The point of that statement is to monitor some variables; the point is not to check whether you're curious. Some people prefer to write the whole statement on one line, perhaps with some tab characters before the `if`, to move it over toward the right margin, as we showed in the previous example, while others put the `if` modifier indented on a new line:

```
print "fred is '$fred', barney is '$barney'\n"
    if $I_am_curious;
```

Although you can rewrite any of these expressions with modifiers as a block (the "old-fashioned" way), the converse isn't necessarily true. Only a single expression is allowed on either side of the modifier. So you can't write something `if` something `while` something `until` something `unless` something `foreach` something, which would just be too confusing. And you can't put multiple statements on the left of the modifier. If you need more than just a simple expression on each side, just write the code the old-fashioned way, with the parentheses and curly braces.

As we mentioned in relation to the `if` modifier, the control expression (on the right) is always evaluated first, just as it would be in the old-fashioned form.

With the `foreach` modifier, there's no way to choose a different control variable -- it's always `$_`. Usually, that's no problem, but if you want to use a different variable, you'll need to rewrite it as a traditional `foreach` loop.

10.4. The Naked Block Control Structure

The so-called "naked" block is one without a keyword or condition. That is, suppose you start with a `while` loop, which looks something like this:

```
while (condition) {
  body;
  body;
  body;
}
```

Now, take away the `while` keyword and the conditional expression, and you'll have a naked block:

```
{
  body;
  body;
  body;
}
```

The naked block is like a `while` or `foreach` loop, except that it doesn't loop; it just executes the body of the loop once, and it's done. It's an un-loop!

We'll see in a while that there are other uses for the naked block, but one of its features is that it provides a scope for temporary lexical variables:

```
{
  print "Please enter a number: ";
  chomp(my $n = <STDIN>);
  my $root = sqrt $n;  # calculate the square root
  print "The square root of $n is $root.\n";
}
```

In this block, `$n` and `$root` are temporary variables scoped to the block. As a general guideline, all variables should be declared in the smallest scope available. If you need a variable for just a few lines of code, you can put those lines into a naked block and declare the variable inside that block. Of course, if we would need the value of either `$n` or `$root` later, we would need to declare them in a larger scope.

You may have noticed the `sqrt` function in that code and wondered about it -- yes, it's a function we haven't shown before. Perl has many builtin functions that are beyond the scope of this book. When you're ready, check the *perlfunc* manpage to learn about more of them.

10.5. The elsif Clause

Every so often, you may need to check a number of conditional expressions, one after another, to see which one of them is true. This can be done with the `if` control structure's `elsif` clause, as in this example:

```
if ( ! defined $dino) {
  print "The value is undef.\n";
} elsif ($dino =~ /^-?\d+\.?$/) {
  print "The value is an integer.\n";
} elsif ($dino =~ /^-?\d*\.\d+$/) {
  print "The value is a _simple_ floating-point number.\n";
} elsif ($dino eq '') {
  print "The value is the empty string.\n";
} else {
  print "The value is the string '$dino'.\n";
}
```

Perl will test the conditional expressions one after another. When one succeeds, the corresponding block of code is executed, and then the whole control structure is done, and execution goes on to the rest of the program. If none has succeeded, the `else` block at the end is executed. (Of course, the `else` clause is still optional, although in this case it's often a good idea to include it.)

There's no limit to the number of `elsif` clauses, but remember that Perl has to evaluate the first ninety-nine tests before it can get to the hundredth. If you'll have more than half a dozen `elsif`s, you should consider whether there's a more efficient way to write it. The Perl FAQ (see the `perlfaq` manpage) has a number of suggestions for emulating the "case" or "switch" statements of other languages.

You may have noticed by this point that the keyword is spelled `elsif`, with only one `e`. If you write it as "elseif", with a second `e`, Perl will tell you that it is not the correct spelling. Why not? Because Larry says so.

10.6. Autoincrement and Autodecrement

You'll often want a scalar variable to count up or down by one. Since these are frequent constructs, there are shortcuts for them, like nearly everything else we do frequently.

The autoincrement operator ("++") adds one to a scalar variable, like the same operator in C and similar languages:

```
my $bedrock = 42;
$bedrock++;  # add one to $bedrock; it's now 43
```

Just like other ways of adding one to a variable, the scalar will be created if necessary:

```
my @people = qw{ fred barney fred wilma dino barney fred pebbles };
my %count;                      # new empty hash
$count{$_}++ foreach @people;  # creates new keys and values as needed
```

The first time through that `foreach` loop, `$count{$_}` is incremented. That's `$count{"fred"}`, which thus goes from `undef` (since it didn't previously exist in the hash) up to `1`. The next time through the loop, `$count{"barney"}` becomes `1`; after that, `$count{"fred"}` becomes `2`. Each time through the loop, one element in `%count` is incremented, and possibly created as well. After that loop is done, `$count{"fred"}` is `3`. This provides a quick and easy way to see which items are in a list and how many times each one appears.

Similarly, the autodecrement operator ("--") subtracts one from a scalar variable:

```
$bedrock--;  # subtract one from $bedrock; it's 42 again
```

10.6.1. The Value of Autoincrement

You can fetch the value of a variable and change that value at the same time. Put the `++` operator in front of the variable name to increment the variable first and then fetch its value. This is a *preincrement*:

```
my $a = 5;
my $b = ++$a;  # increment $a to 6, and put that value into $b
```

Or put the `--` operator in front to decrement the variable first and then fetch its value. This is a *predecrement*:

```
my $c = --$a;  # decrement $a to 5, and put that value into $c
```

Here's the tricky part. Put the variable name first to fetch the value first, and then do the increment or decrement. This is called a *postincrement* or *postdecrement*:

```
my $d = $a++;  # $d gets the old value (5), then increment $a to 6
my $e = $a--;  # $e gets the old value (6), then decrement $a to 5
```

It's tricky because we're doing two things at once. We're fetching the value, and we're changing it in the same expression. If the operator is first, we increment (or decrement) first, then use the new value. If the variable is first, we return its (old) value first, then do the increment or decrement. Another way to say it is that these operators return a value, but they also have the side effect of modifying the variable's value.

If you write these in an expression of their own, not using the value but only the side effect, there's no difference whether you put the operator before or after the variable:

```
$bedrock++;  # adds one to $bedrock
++$bedrock;  # just the same; adds one to $bedrock
```

A common use of these operators is in connection with a hash, to identify when an item has been seen before:

```
my @people = qw{ fred barney bamm-bamm wilma dino barney betty pebbles };
my %seen;
```

```
foreach (@people) {
  print "I've seen you somewhere before, $_!\n"
    if $seen{$_}++;
}
```

When `barney` shows up for the first time, the value of `$seen{$_}++` is false, since it's the value of `$seen{$_}`, which is `$seen{"barney"}`, which is `undef`. But that expression has the side effect of incrementing `$seen{"barney"}`. When `barney` shows up again, `$seen{"barney"}` is now a true value, so the message is printed.

10.7. The for Control Structure

Perl's `for` control structure is like the common `for` control structure you may have seen in other languages such as C. It looks like this:

```
for (initialization; test; increment) {
  body;
  body;
}
```

To Perl, though, this kind of loop is really a `while` loop in disguise, something like this:

```
initialization;
while (test) {
  body;
  body;
  increment;
}
```

The most common use of the `for` loop, by far, is for making computed iterations:

```
for ($i = 1; $i <= 10; $i++) {  # count from 1 to 10
  print "I can count to $i!\n";
}
```

When you've seen these before, you'll know what the first line is saying even before you read the comment. Before the loop starts, the control variable, `$i`, is set to `1`. Then, the loop is really a `while` loop in disguise, looping while `$i` is less than or equal to `10`. Between each iteration and the next is the increment, which here is a literal increment, adding one to the control variable, which is `$i`.

So, the first time through this loop, `$i` is `1`. Since that's less than or equal to `10`, we see the message. Although the increment is written at the top of the loop, it logically happens at the bottom of the loop, after printing the message. So, `$i` becomes `2`, which is less than or equal to `10`, so we print the message again, and `$i` is incremented to `3`, which is less than or equal to `10`, and so on.

Eventually, we print the message that our program can count to `9`. Then `$i` is incremented to `10`, which is less than or *equal* to `10`, so we run the loop one last time and print that our program can count to `10`. Finally, `$i` is incremented for the last time, to `11`, which is not less than or equal to `10`. So control drops out of the loop, and we're on to the rest of the program.

All three parts are together at the top of the loop so that it's easy for an experienced programmer to read that first line and say, "Ah, it's a loop that counts `$i` from one to ten."

Note that after the loop is done, the control variable has a value "after" the loop. That is, in this case, the control variable has gone all the way to `11`.

This loop is a very versatile loop, since you can make it count in all sorts of ways. This loop counts from `-150` to `1000` by threes:

```
for ($i = -150; $i <= 1000; $i += 3) {
  print "$i\n";
}
```

In fact, you could make any of the three control parts (initialization, test, or increment) empty, if you wish, but you still need the two semicolons. In this (quite unusual) example, the test is a substitution, and the increment is empty:

```
for ($_ = "bedrock"; s/(.)//; ) {  # loops while the s/// is successful
  print "One character is: $1\n";
}
```

The test expression (in the implied `while` loop) is the substitution, which will return a true value if it succeeded. In this case, the first time through the loop, the substitution will remove the `b` from `bedrock`. Each iteration will remove another letter. When the string is empty, the substitution will fail, and the loop is done.

If the test expression (the one between the two semicolons) is empty, it's automatically true, making an infinite loop. But don't make an infinite loop like this until you see how to break out of such a loop, which we'll discuss later in this chapter:

```
for (;;) {
  print "It's an infinite loop!\n";
}
```

A more Perl-like way to write an intentional infinite loop, when you really want one, is with `while`:

```
while (1) {
  print "It's another infinite loop!\n";
}
```

Although C programmers are familiar with the first way, even a beginning Perl programmer should recognize that `1` is always true, making an intentional infinite loop, so the second is generally a better way to write it. Perl is smart enough to recognize a constant expression like that and optimize it away, so there's no difference in efficiency.

10.7.1. The Secret Connection Between foreach and for

It turns out that, inside the Perl grammar, the keyword `foreach` is exactly equivalent to the keyword `for`. That is, any time Perl sees one of them, it's the same as if you had typed the other. Perl can tell which you meant by looking inside the parentheses. If you've got the two semicolons, it's a computed `for` loop (like we've just been talking about). If you don't have the semicolons, it's really a `foreach` loop:

```
for (1..10) {  # Really a foreach loop from 1 to 10
  print "I can count to $_!\n";
}
```

That's really a `foreach` loop, but it's written `for`. Except for that one example, all through this book, we'll spell out `foreach` wherever it appears. But in the real world, do you think that Perl folks will type those extra four letters? Excepting only beginners' code, it's always written `for`, and you'll have to do as Perl does and look for the semicolons to tell which kind of loop it is.

In Perl, the true `foreach` loop is almost always a better choice. In the `foreach` loop (written `for`) in that previous example, it's easy to see at a glance that the loop will go from `1` to `10`. But do you see what's wrong with this computed loop that's trying to do the same thing?

```
for ($i = 1; $i < 10; $i++) {  # Oops! Something is wrong here!
  print "I can count to $_!\n";
}
```

There are two and one-half bugs. First, the conditional uses a less-than sign, so the loop will run nine times, instead of ten. Second, the control variable is `$i`, but the loop body is using `$_`. And second and a half, it's a lot more work to read, write, maintain, and debug this type of loop, which is why we say that the true `foreach` is generally a better choice in Perl.

10.8. Loop Controls

As you've surely noticed by now, Perl is one of the so-called "structured" programming languages. In particular, there's just one entrance to any block of code, which is at the top of that block. But there are times when you may need more control or versatility than what we've shown so far. For example, you may need to make a loop like a `while` loop, but one that always runs at least once. Or maybe you need to occasionally exit a block of code early. Perl has three loop-control operators you can use in loop blocks to make the loop do all sorts of tricks.

10.8.1. The last Operator

The `last` operator immediately ends execution of the loop. (If you've used the "break" operator in C or a similar language, it's like that.) It's the "emergency exit" for loop blocks. When you hit `last`, the loop is done. For example:

```
# Print all input lines mentioning fred, until the __END__ marker
while (<STDIN>) {
  if (/__END__/) {
    # No more input on or after this marker line
    last;
  } elsif (/fred/) {
    print;
  }
}
## last comes here ##
```

Once an input line has the `__END__` marker, that loop is done. Of course, that comment line at the end is merely a comment -- it's not required in any way. We just threw that in to make it clearer what's happening.

There are five kinds of loop blocks in Perl. These are the blocks of `for`, `foreach`, `while`, `until`, or the naked block. The curly braces of an `if` block or subroutine don't qualify. As you may have noticed in the example above, the `last` operator applied to the entire loop block.

The `last` operator will apply to the innermost currently running loop block. To jump out of outer blocks, stay tuned; that's coming up in a little bit.

10.8.2. The next Operator

Sometimes you're not ready for the loop to finish, but you're done with the current iteration. That's what the `next` operator is good for. It jumps to the *inside* of the bottom of the current loop block. After `next`, control continues with the next iteration of the loop (much like the "continue" operator in C or a similar language):

```
# Analyze words in the input file or files
while (<>) {
  foreach (split) {  # break $_ into words, assign each to $_ in turn
    $total++;
    next if /\W/;    # strange words skip the remainder of the loop
    $valid++;
    $count{$_}++;    # count each separate word
    ## next comes here ##
  }
}

print "total things = $total, valid words = $valid\n";
foreach $word (sort keys %count) {
  print "$word was seen $count{$word} times.\n";
}
```

This one is a little more complex than most of our examples up to this point, so let's take it step by step. The `while` loop is reading lines of input from the diamond operator, one after another, into `$_`; we've seen that before. Each time through that loop, another line of input will be in `$_`.

Inside that loop, the `foreach` loop is iterating over the return value `split`. Do you remember the default for `split` with no arguments? That splits `$_` on whitespace, in effect breaking `$_` into a list of words. Since the `foreach` loop doesn't mention some other control variable, the control variable will be `$_`. So, we'll see one word after another in `$_`.

But didn't we just say that `$_` holds one line of input after another? Well, in the outer loop, that's what it is. But inside the `foreach` loop, it holds one word after another. It's no problem for Perl to reuse `$_` for a new purpose; this happens all the time.

Now, inside the `foreach` loop, we're seeing one word at a time in `$_`. `$total` is incremented, so it must be the total number of words. But the next line (which is the point of this example) checks to see whether the word has any nonword characters -- anything but letters, digits, and underscores. So, if the word is `Tom's`, or if it is `full-sized`, or if it has an adjoining comma, quote mark, or any other strange character, it will match that pattern and we'll skip the rest of the loop, going on to the next word.

But let's say that it's an ordinary word, like `fred`. In that case, we count `$valid` up by one, and also `$count{$_}`, keeping a count for each different word. So, when we finish the two loops, we've counted every word in every line of input from every file the user wanted us to use.

We're not going to explain the last few lines. By now, we hope you've got stuff like that down already.

Like `last`, `next` may be used in any of the five kinds of loop blocks: `for`, `foreach`, `while`, `until`, or the naked block. Also, if loop blocks are nested, `next` works with the innermost one. We'll see how to change that at the end of this section.

10.8.3. The redo Operator

The third member of the loop control triad is `redo`. It says to go back to the top of the current loop block, without testing any conditional expression or advancing to the next iteration. (If you've used C or a similar language, you've never seen this one before. Those languages don't have this kind of operator.) Here's an example:

```
# Typing test
my @words = qw{ fred barney pebbles dino wilma betty };
my $errors = 0;

foreach (@words) {
  ## redo comes here ##
  print "Type the word '$_': ";
  chomp(my $try = <STDIN>);
  if ($try ne $_) {
    print "Sorry - That's not right.\n\n";
    $errors++;
    redo;  # jump back up to the top of the loop
  }
}
print "You've completed the test, with $errors errors.\n";
```

Like the other two operators, `redo` will work with any of the five kinds of loop blocks, and it will work with the innermost loop block when they're nested.

The big difference between `next` and `redo` is that `next` will advance to the next iteration, but `redo` will redo the current iteration. Here's an example program that you can play with to get a feel for how these three operators work:

```
foreach (1..10) {
  print "Iteration number $_.\n\n";
  print "Please choose: last, next, redo, or none of the above? ";
  chomp(my $choice = <STDIN>);
  print "\n";
  last if $choice =~ /last/i;
  next if $choice =~ /next/i;
  redo if $choice =~ /redo/i;
  print "That wasn't any of the choices... onward!\n\n";
}
print "That's all, folks!\n";
```

If you just press return without typing anything (try it two or three times), the loop counts along from one number to the next. If you choose `last` when you get to number four, the loop is done, and you won't go on to number five. If you choose `next` when you're on four, you're on to number five without printing the "onward" message. And if you choose `redo` when you're on four, you're back to doing number four all over again.

10.8.4. Labeled Blocks

When you need to work with a loop block that's not the innermost one, use a label. Labels in Perl are like other identifiers -- made of letters, digits, and underscores, but they can't start with a digit -- however, since they have no prefix character, labels could be confused with the names of builtin function names, or even with your own subroutines' names. So, it would be a poor choice to make a label called `print` or `if`. Because of that, Larry recommends that they be all uppercase. That not only ensures that the label won't conflict with another identifier but it also makes it easy to spot the label in the code. In any case, labels are rare, only showing up in a small percentage of Perl programs.

To label a loop block, just put the label and a colon in front of the loop. Then, inside the loop, you may use the label after `last`, `next`, or `redo` as needed:

```
LINE: while (<>) {
  foreach (split) {
    last LINE if /_ _END_  _/;  # bail out of the LINE loop
    ...;
  }
}
```

For readability, it's generally nice to put the label at the left margin, even if the current code is at a higher indentation. Notice that the label names the entire block; it's not marking a target point in the code.

In that previous snippet of sample code, the special __`END`__ token marks the end of all input. Once that token shows up, the program will ignore any remaining lines (even from other files).

It often makes sense to choose a noun as the name of the loop. That is, the outer loop is processing a line at a time, so we called it `LINE`. If we had to name the inner loop, we would have called it `WORD`, since it processes a word at a time. That makes it convenient to say things like "(move on to the) `next WORD`" or "`redo` (the current) `LINE`".

10.9. Logical Operators

As you might expect, Perl has all of the necessary logical operators needed to work with Boolean (true/false) values. For example, it's often useful to combine logical tests by using the logical AND operator (`&&`) and the logical OR operator (`||`):

```
if ($dessert{'cake'} && $dessert{'ice cream'}) {
  # Both are true
  print "Hooray! Cake and ice cream!\n";
} elsif ($dessert{'cake'} || $dessert{'ice cream'}) {
  # At least one is true
  print "That's still good...\n";
} else {
  # Neither is true - do nothing (we're sad)
}
```

There may be a shortcut. If the left side of a logical AND operation is false, the whole thing is false, since logical AND needs both sides to be true in order to return true. In that case, there's no reason to check the right side, so it will not even be evaluated. Consider what happens in this example if `$hour` is `3`:

```
if ( (9 <= $hour) && ($hour < 17) ) {
  print "Aren't you supposed to be at work...?\n";
}
```

Similarly, if the left side of a logical OR operation is *true*, the right side will not be evaluated. Consider what happens here if `$name` is `fred`:

```
if ( ($name eq 'fred') || ($name eq 'barney') ) {
  print "You're my kind of guy!\n";
}
```

Because of this behavior, these operators are called "short-circuit" logical operators. They take a short circuit to the result whenever they can. In fact, it's fairly common to rely upon this short-circuit behavior. Suppose you need to calculate an average:

```
if ( ($n != 0) && ($total/$n < 5) ) {
  print "The average is below five.\n";
}
```

In that example, the right side will be evaluated only if the left side is true, so we can't accidentally divide by zero and crash the program.

10.9.1. The Value of a Short-Circuit Operator

Unlike what happens in C (and similar languages), the value of a short-circuit logical operator is the last part evaluated, not just a Boolean value. This provides the same result, in that the last part evaluated is always true when the whole thing should be true, and it's always false when the whole thing should be false.

But it's a much more useful return value. Among other things, the logical OR operator is quite handy for selecting a default value:

```
my $last_name = $last_name{$someone} || '(No last name)';
```

If `$someone` is not listed in the hash, the left side will be `undef`, which is false. So, the logical OR will have to look to the right side for the value, making the right side the default. We'll see other uses for this behavior later.

10.9.2. The Ternary Operator, ?:

When Larry was deciding which operators to make available in Perl, he didn't want former C programmers to be left wishing for something that C had and Perl didn't, so he brought over all of C's operators to Perl. That meant bringing over C's most confusing operator: the ternary `?:` operator. While it may be confusing, it can also be quite useful.

The ternary operator is like an if-then-else test, all rolled into an expression. It is called a "ternary" operator because it takes three operands. It looks like this:

```
expression ? if_true_expr : if_false_expr
```

First, the expression is evaluated to see whether it's true or false. If it's true, the second expression is used; otherwise, the third expression is used. Every time, one of the two expressions on the right is evaluated, and one is ignored. That is, if the first expression is true, then the second expression is evaluated, and the third is ignored. If the first expression is false, then the second is ignored, and the third is evaluated as the value of the whole thing.

In this example, the result of the subroutine `&is_weekend` determines which string expression will be assigned to the variable:

```
my $location = &is_weekend($day) ? "home" : "work";
```

And here, we calculate and print out an average -- or just a placeholder line of hyphens, if there's no average available:

```
my $average = $n ? ($total/$n) : "-----";
print "Average: $average\n";
```

You could always rewrite any use of the `?:` operator as an `if` structure, often much less conveniently and less concisely:

```
my $average;
if ($n) {
  $average = $total / $n;
} else {
  $average = "-----";
}
print "Average: $average\n";
```

Here's a trick you might see, used to code up a nice multiway branch:

```
my $size =
  ($width < 10) ? "small"  :
  ($width < 20) ? "medium" :
  ($width < 50) ? "large"  :
                  "extra-large"; # default
```

That is really just three nested `?:` operators, and it works quite well, once you get the hang of it.

Of course, you're not obliged to use this operator. Beginners may wish to avoid it. But you'll see it in others' code, sooner or later, and we hope that one day you'll find a good reason to use it in your own programs.

10.9.3. Control Structures Using Partial-Evaluation Operators

These three operators that we've just seen -- `&&`, `||`, and `?:` -- all share a peculiar property: depending upon whether the value on the left side is true or false, they may or may not evaluate an expression. Sometimes the expression is evaluated, and sometimes it isn't. For that reason, these are sometimes called *partial-evaluation* operators, since they may not evaluate all of the expressions around them. And partial-evaluation operators are automatically control structures.

It's not as if Larry felt a burning need to add more control structures to Perl. But once he had decided to put these partial-evaluation operators into Perl, they automatically became control structures as well. After all, anything that can activate and deactivate a chunk of code is a control structure.

Fortunately, you'll notice this only when the controlled expression has side effects, like altering a variable's value or causing some output. For example, suppose you ran across this line of code:

```
    ($a < $b) && ($a = $b);
```

Right away, you should notice that the result of the logical AND isn't being assigned anywhere. Why not?

If `$a` is really less than `$b`, the left side is true, so the right side will be evaluated, thereby doing the assignment. But if `$a` is not less than `$b`, the left side will be false, and thus the right side would be skipped. So that line of code would do essentially the same thing as this one, which is easier to understand:

```
    if ($a < $b) { $a = $b; }
```

Or maybe you'll be maintaining a program, and you'll see a line like this one:

```
    ($a > 10) || print "why is it not greater?\n";
```

If `$a` is really greater than ten, the left side is true, and the logical OR is done. But if it's not, the left side is false, and this will go on to print the message. Once again, this could (and probably should) be written in the traditional way, probably with `if` or `unless`.

If you have a particularly twisted brain, you might even learn to read these lines as if they were written in English. For example: check that `$a` is less than `$b`, *and if it is*, then do the assignment. Check that `$a` is more than ten, *or if it's not*, then print the message.

It's generally former C programmers or old-time Perl programmers who most often use these ways of writing control structures. Why do they do it? Some have the mistaken idea that these are more efficient. Some think these tricks make their code cooler. Some are merely copying what they saw someone else do.

In the same way, the ternary operator may be used for control. In this case, we want to assign `$c` to the smaller of two variables:

```
    ($a < $b) ? ($a = $c) : ($b = $c);
```

If `$a` is smaller, it gets `$c`. Otherwise, `$b` does.

There is another way to write the logical AND and logical OR operators. You may wish to write them out as words: `and` and `or`. These word-operators have the same behaviors as the ones written with punctuation, but the words are much lower on the precedence chart. Since the words don't "stick" so tightly to the nearby parts of the expression, they may need fewer parentheses:

```
    $a < $b and $a = $b;  # but better written as the corresponding if
```

Then again, you may need *more* parentheses. Precedence is a bugaboo. Be sure to use parentheses to say what you mean, unless you're sure of the precedence. Nevertheless, since the word forms are very low precedence, you can generally understand that they cut the expression into big pieces, doing everything on the left first, and then (if needed) everything on the right.

Despite the fact that using logical operators as control structures can be confusing, sometimes they're the accepted way to write code. We'll see a common use of the `or` operator starting in the next chapter.

So, using these operators as control structures is part of idiomatic Perl -- Perl as she is spoken. Used properly, they can make your code more powerful; otherwise they can make your code unmaintainable. Don't overuse them.

Chapter 11. Filehandles

11.1. What Is a Filehandle?

A filehandle is the name in a Perl program for an I/O connection between your Perl process and the outside world. That is, it's the name of a *connection*, not necessarily the name of a file.

Filehandles are named like other Perl identifiers (letters, digits, and underscores, but they can't start with a digit), but since they don't have any prefix character, they might be confused with present or future reserved words, as we saw with labels. Once again, as with labels, the recommendation from Larry is that you use all uppercase letters in the name of your filehandle -- not only will it stand out better, but it will also guarantee that your program won't fail when a future (lowercase) reserved word is introduced.

But there are also six special filehandle names that Perl already uses for its own purposes: `STDIN`, `STDOUT`, `STDERR`, `DATA`, `ARGV`, and `ARGVOUT`. Although you may choose any filehandle name you'd like, you shouldn't choose one of those six unless you intend to use that one's special properties.

Maybe you recognized some of those names already. When your program starts, `STDIN` is the filehandle naming the connection between the Perl process and wherever the program should get its input, known as the *standard input stream*. This is generally the user's keyboard unless the user asked for something else to be the source of input, such as reading the input from a file or reading the output of another program through a pipe.

There's also the *standard output stream*, which is `STDOUT`. By default, this one goes to the user's display screen, but the user may send the output to a file or to another program, as we'll see shortly. These standard streams come to us from the Unix "standard I/O" library, but they work in much the same way on most modern operating systems. The general idea is that your program should blindly read from `STDIN` and blindly write to `STDOUT`, trusting in the user (or generally whichever program is starting your program) to have set those up. In that way, the user can type a command like this one at the shell prompt:

```
$ ./your_program <dino >wilma
```

That command tells the shell that the program's input should be read from the file *dino*, and the output should go to the file *wilma*. As long as the program blindly reads its input from `STDIN`, processes it (in whatever way we need), and blindly writes its output to `STDOUT`, this will work just fine.

And at no extra charge, the program will work in a *pipeline*. This is another concept from Unix, which lets us write command lines like this one:

```
$ cat fred barney | sort | ./your_program | grep something | lpr
```

Now, if you're not familiar with these Unix commands, that's okay. This line says that the *cat* command should print out all of the lines of file *fred* followed by all of the lines of file *barney*. Then that output should be the input of the *sort* command, which sorts those lines and passes them on to *your_program*. After it has done its processing, *your_program* will send the data on to *grep*, which discards certain lines in the data, sending the others on to the *lpr* command, which should print everything that it gets on a printer. Whew!

But pipelines like that are common in Unix and many other systems today because they let you put together a powerful, complex command out of simple, standard building blocks.

There's one more standard I/O stream. If (in the previous example) *your_program* had to emit any warnings or other diagnostic messages, those shouldn't go down the pipeline. The *grep* command is set to discard anything that it hasn't specifically been told to look for, and so it will most likely discard the warnings. Even if it did keep the warnings, we probably don't want those to be passed downstream to the other programs in the pipeline. So that's why there's also the *standard error stream*: `STDERR`. Even if the standard output is going to another program or file, the errors will go to wherever the user desires. By default, the errors will generally go to the user's display screen, but the user may send the errors to a file with a shell command like this one:

```
$ netstat | ./your_program 2>/tmp/my_errors
```

11.2. Opening a Filehandle

So we see that Perl provides three filehandles -- `STDIN`, `STDOUT`, and `STDERR` -- which are automatically open to files or devices established by the program's parent process (probably the shell). When you need other filehandles, use the `open` operator to tell Perl to ask the operating system to open the connection between your program and the outside world. Here are some examples:

```
open CONFIG, "dino";
open CONFIG, "<dino";
open BEDROCK, ">fred";
open LOG, ">>logfile";
```

The first one opens a filehandle called `CONFIG` to a file called *dino*. That is, the (existing) file *dino* will be opened and whatever it holds will come into our program through the filehandle named `CONFIG`. This is similar to the way that data from a file could come in through `STDIN` if the command line had a shell redirection like `<dino`. In fact, the second example uses exactly that sequence. The second does the same as the first, but the less-than sign explicitly says "this filename is to be used for input," even though that's the default.

Although you don't have to use the less-than sign to open a file for input, we include that because, as you can see in the third example, a greater-than sign means to create a new file for output. This opens the filehandle `BEDROCK` for output to the new file *fred*. Just as when the greater-then sign is used in shell redirection, we're sending the output to a *new* file called *fred*. If there's already a file of that name, we're asking to wipe it out and replace it with this new one.

The fourth example shows how two greater-than signs may be used (again, as the shell does) to open a file for appending. That is, if the file already exists, we will add new data at the end. If it doesn't exist, it will be created in much the same way as if we had used just one greater-than sign. This is handy for log files; your program could write a few lines to the end of a log file each time it's run. So that's why the fourth example names the filehandle `LOG` and the file *logfile*.

You can use any scalar expression in place of the filename specifier, although typically you'll want to be explicit about the direction specification:

```
my $selected_output = "my_output";
open LOG, "> $selected_output";
```

Note the space after the greater-than. Perl ignores this, but it keeps unexpected things from happening if `$selected_output` were `">passwd"` for example (which would make an append instead of a write).

We'll see how to use these filehandles later in this chapter.

11.2.1. Closing a Filehandle

When you are finished with a filehandle, you may close it with the `close` operator like this:

```
close BEDROCK;
```

Closing a filehandle tells Perl to inform the operating system that we're all done with the given data stream, so any last output data should be written to disk in case someone is waiting for it.

Perl will automatically close a filehandle if you reopen it (that is, if you reuse the filehandle name in a new `open`) or if you exit the program. Because of this, many simple Perl programs don't bother with `close`. But it's there if you want to be tidy, with one `close` for every `open`. In general, it's best to close each filehandle soon after you're done with it, though the end of the program often arrives soon enough.

11.2.2. Bad Filehandles

Perl can't actually open a file all by itself. Like any other programming language, Perl can merely ask the operating system to let us open a file. Of course, the operating system may refuse, because of permission settings, an incorrect filename, or other reasons.

If you try to read from a bad filehandle (that is, a filehandle that isn't properly open), you'll see an immediate end-of-file. (With the I/O methods we'll see in this chapter, end-of-file will be indicated by `undef` in a scalar context or an empty list in a list context.) If you try to write to a bad filehandle, the data is silently discarded.

Fortunately, these dire consequences are easy to avoid. First of all, if we ask for warnings with `-w`, Perl will generally be able to tell us with a warning when it sees that we're using a bad filehandle. But even before that, `open` always tells us if it succeeded or failed, by returning true for success or false for failure. So you could write code like this:

```
my $success = open LOG, ">>logfile";  # capture the return value
unless ($success) {
  # The open failed
  ...
}
```

Well, you *could* do it like that, but there's another way that we'll see in the next section.

11.3. Fatal Errors with die

Let's step aside for a moment. We need some stuff that isn't directly related to (or limited to) filehandles, but is more about getting out of a program earlier than normal.

When a fatal error happens inside Perl (for example, if you divide by zero, use an invalid regular expression, or call a subroutine that hasn't been declared) your program stops with an error message telling why. But this functionality is available to us with the `die` function, so we can make our own fatal errors.

The `die` function prints out the message you give it (to the standard error stream, where such messages should go) and makes sure that your program exits with a nonzero exit status.

You may not have known it, but every program that runs on Unix (and many other modern operating systems) has an exit status, telling whether it was successful or not. Programs that run other programs (like the *make* utility program) look at that exit status to see that everything is running correctly. The exit status is just a single byte, so it can't say much; traditionally, it is zero for success and a nonzero value for failure. Perhaps one means a syntax error in the command arguments, while two means that something went wrong during processing and three means the configuration file couldn't be found; the details differ from one command to the next. But zero always means that everything worked. When the exit status shows failure, a program like *make* knows not to go on to the next step.

So we could rewrite the previous example, perhaps something like this:

```
unless (open LOG, ">>logfile") {
  die "Cannot create logfile: $!";
}
```

If the `open` fails, `die` will terminate the program and tell us that it cannot create the logfile. But what's that `$!` in the message? That's the human-readable complaint from the system. In general, when the system refuses to do something we've requested (like opening a file), it will give us a reason (perhaps "permission denied" or "file not found," in this case). This is the string that you may have obtained with `perror` in C or a similar language. This human-readable complaint message will be available in Perl's special variable `$!`. It's a good idea to include `$!` in the message when it could help the user to figure out what he or she did wrong. But if you use `die` to indicate an error that is not the failure of a system request, don't include `$!`, since it will generally hold an unrelated message left over from something Perl did internally. It will hold a useful value only immediately after a *failed* system request. A successful request won't leave anything useful there.

There's one more thing that `die` will do for you: it will automatically append the Perl program name and line number to the end of the message, so you can easily identify which `die` in your program is responsible for the untimely exit. The error message from the previous code might look like this, if `$!` contained the message `permission denied`:

```
Cannot create logfile: permission denied at your_program line 1234.
```

That's pretty helpful -- in fact, we always seem to want more information in our error messages than we put in the first time around. If you don't want the line number and file revealed, make sure that the dying words have a newline on the end. That is, another way you could use `die` is in a line like this, with a trailing newline:

```
die "Not enough arguments\n" if @ARGV < 2;
```

If there aren't at least two command-line arguments, that program will say so and quit. It won't include the program name and line number, since the line number is of no use to the user; this is the user's error, after all. As a rule of thumb, put the newline on messages that indicate a usage error and leave it off when it the error might be something you want to track down during debugging.

When opening a file fails, though, there's an easier and more common way instead of the `unless` block:

```
open LOG, ">>logfile"
  or die "Cannot create logfile: $!";
```

This uses the low-precedence short-circuit `or` operator that we saw in Chapter 10, "More Control Structures". If the `open` succeeds, it returns true, and the `or` is done. If the `open` fails, it returns false, and the short-circuit `or` goes on to the right side and dies with the message. You can read this as if it were English: "Open this file, or die!" It may not be the battle cry that will win a war, but it's a good way to write code.

You should always check the return value of `open`, since the rest of the program is relying upon its success. That's why we say that this is really the only way to write `open` -- with `or die` after it. Until you're ready to be extra tricky, you should simply think of this as the syntax for `open`. Typing `or die` and a message takes only a moment when you're writing the program, but it can save hours, or possibly days of debugging time when something goes wrong.

11.3.1. Warning Messages with warn

Just as `die` can indicate a fatal error that acts like one of Perl's builtin errors (like dividing by zero), you can use the `warn` function to cause a warning that acts like one of Perl's builtin warnings (like using an `undef` value as if it were defined, when warnings are enabled).

The `warn` function works just like `die` does, except for that last step -- it doesn't actually quit the program. But it adds the program name and line number if needed, and it prints the message to standard error, just as `die` would.

And having talked about death and dire warnings, we now return you to your regularly scheduled filehandle instructional material. Read on.

11.4. Using Filehandles

Once a filehandle is open for reading, you can read lines from it just like you can read from standard input with `STDIN`. So, for example, to read lines from the Unix password file:

```
open PASSWD, "/etc/passwd"
  or die "How did you get logged in? ($!)";

while (<PASSWD>) {
  chomp;
  if (/^root:/) {  # found root entry...
    ...;
  }
}
```

In this example, the `die` message uses parentheses around `$!`. Those are merely parentheses around the message in the output. (Sometimes a punctuation mark is just a punctuation mark.) As you can see, what we've been calling the "line-input operator" is really made of two components; the angle brackets (the *real* line-input operator) are around an input filehandle. Each line of input is then tested to see if it begins with `root` followed by a colon, triggering unseen actions.

A filehandle open for writing or appending may be used with `print` or `printf`, appearing immediately after the keyword but before the list of arguments:

```
print LOG "Captain’s log, stardate 3.14159\n";  # output goes to LOG
printf STDERR "%d percent complete.\n", $done/$total * 100;
```

Did you notice that there’s no comma between the filehandle and the items to be printed? This looks especially weird if you use parentheses. Either of these forms is correct:

```
printf (STDERR "%d percent complete.\n", $done/$total * 100);
printf STDERR ("%d percent complete.\n", $done/$total * 100);
```

11.4.1. Changing the Default Output Filehandle

By default, if you don’t give a filehandle to `print` (or to `printf`, as everything we say here about one applies equally well to the other), the output will go to `STDOUT`. But that default may be changed with the `select` operator. Here we’ll send some output lines to `BEDROCK`:

```
select BEDROCK;
print "I hope Mr. Slate doesn’t find out about this.\n";
print "Wilma!\n";
```

Once you’ve selected a filehandle as the default for output, it will stay that way. But it’s generally a bad idea to confuse the rest of the program, so you should generally set it back to `STDOUT` when you’re done.

Also by default, the output to each filehandle is buffered. Setting the special `$|` variable to `1` will set the currently selected filehandle (that is, the one selected at the time that the variable is modified) to always flush the buffer after each output operation. So if you wanted to be sure that the logfile gets its entries at once, in case you might be reading the log to monitor progress of your long-running program, you could use something like this:

```
select LOG;
$| = 1;  # don’t keep LOG entries sitting in the buffer
select STDOUT;
# ... time passes, babies learn to walk, tectonic plates shift, and then...
print LOG "This gets written to the LOG at once!\n";
```

14.4. Using Backquotes to Capture Output

With both `system` and `exec`, the output of the launched command ends up wherever Perl’s standard output is going. Sometimes, it’s interesting to capture that output as a string value to perform further processing. And that’s done simply by creating a string using backquotes instead of single or double quotes:

```
my $now = ‘date‘;             # grab the output of date
print "The time is now $now"; # newline already present
```

Normally, this *date* command spits out a string approximately 30 characters long to its standard output, giving the current date and time followed by a newline. When we’ve placed *date* between backquotes, Perl executes the *date* command, arranging for its standard output to be captured as a string value, and in this case assigned to the `$now` variable.

This is very similar to the Unix shell’s meaning for backquotes. However, the shell also performs the additional job of ripping off the final end-of-line to make it easier to use the value as part of other things. Perl is honest; it gives the real output. To get the same result in Perl, we can simply add an additional `chomp` operation on the result:

```
chomp(my $no_newline_now = ‘date‘);
print "A moment ago, it was $no_newline_now, I think.\n";
```

The value beween backquotes is just like the single-argument form of system, and is interpreted as a double-quoted string, meaning that backslash-escapes and variables are expanded appropriately. For example, to fetch the Perl documentation on a list of Perl functions, we might invoke the *perldoc* command repeatedly, each time with a different argument:

```
my @functions = qw{ int rand sleep length hex eof not exit sqrt umask };
my %about;

foreach (@functions) {
  $about{$_} = `perldoc -t -f $_`;
}
```

Note that $_ will be a different value for each invocation, letting us grab the output of a different command varying only in one of its parameters. Also note that if you haven't seen some of these functions yet, it might be useful to look them up in the documentation to see what they do!

There's no easy equivalent of single quotes for backquotes ; variable references and backslash items are always expanded. Also, there's no easy equivalent of the multiple-argument version of `system` (where a shell is never involved). If the command inside the backquotes is complex enough, a Unix Bourne Shell (or whatever your system uses instead) is invoked to interpret the command automatically.

At the risk of actually introducing the behavior by demonstrating how *not* to do it, we'd also like to suggest that you avoid using backquotes in a place where the value isn't being captured. For example:

```
print "Starting the frobnitzigator:\n";
`frobnitz -enable`; # please don't do this!
print "Done!\n";
```

The problem is that Perl has to work a bit harder to capture the output of this command, even when you're just throwing it away, and then you also lose the option to use multiple arguments to `system` to precisely control the argument list. So from both a security standpoint and an efficiency viewpoint, just use `system` instead, please.

Standard error of a backquoted command is inherited from Perl's current standard error output. If the command spits out error messages to standard error, you'll probably see them on the terminal, which could be confusing to the user who hasn't personally invoked the *frobnitz* command. If you want to capture error messages with standard output, you can use the shell's normal "merge standard error to the current standard output," which is spelled `2>&1` in the normal Unix shell:

```
my $output_with_errors = `frobnitz -enable 2>&1`;
```

Note that this will make the standard error output intermingled with the standard output, much as it appears on the terminal (although possibly in a slightly different sequence because of buffering). If you need the output and the error output separated, there are many harder-to-type solutions.

Similarly, standard input is inherited from Perl's current standard input. Most commands we typically use with backquotes do not read standard input, so that's rarely a problem. However, let's say the *date* command asked which time zone (as we imagined earlier). That'll be a problem, because the prompt for "which time zone" will be sent to standard output, which is being captured as part of the value, and then the *date* command will start trying to read from standard input. But since the user has never seen the prompt, he or she doesn't know to be typing anything! Pretty soon, the user calls you up and tells you that your program is stuck.

So, stay away from commands that read standard input. If you're not sure whether something reads from standard input, then add a redirection from */dev/null* for input, like this:

```
my $result = `some_questionable_command arg arg argh </dev/null`;
```

Then the child shell will redirect input from */dev/null*, and the grandchild questionable command will at worst try to read and immediately get an end of file.

14.4.1. Using Backquotes in a List Context

If the output from a command has multiple lines, the scalar use of backquotes returns it as a single long string containing newline characters. However, using the same backquoted string in a list context yields a list containing one line of output per element.

For example, the Unix *who* command normally spits out a line of text for each current login on the system as follows:

```
merlyn     tty/42     Dec 7  19:41
rootbeer   console    Dec 2  14:15
rootbeer   tty/12     Dec 6  23:00
```

The left column is the username, the middle column is the tty name (that is, the name of the user's connection to the machine), and the rest of the line is the date and time of login (and possibly remote login information, but not in this example). In a scalar context, we get all that at once, which we would then need to split up:

```
my $who_text = `who`;
```

But in a list context, we automatically get the data broken up by lines:

```
my @who_lines = `who`;
```

We'll have a number of separate elements in `@who_lines`, each one terminated by a newline. Of course, adding a `chomp` around the outside of that will rip off those newlines, but let's go a different direction. If we put that as part of the value for a `foreach`, we'll iterate over the lines automatically, placing each one in `$_`:

```
foreach (`who`) {
  my($user, $tty, $date) = /(\S+)\s+(\S+)\s+(.*)/;
  $ttys{$user} .= "$tty at $date\n";
}
```

This loop will iterate three times for the data above. (Your system will probably have more than three active logins at any given time.) Notice that we've got a regular expression match, and in the absence of the binding operator ("`=~`"), that's matching against `$_`, which is good because that's where the data is.

Also notice the regular expression is looking for a nonblank word, some whitespace, a nonblank word, some whitespace, and then the rest of the line up to, but not including, the newline (since dot doesn't match newline by default). That's also good, because that's what the data looks like each time in `$_`. That'll make `$1` be "`merlyn`", `$2` be "`tty/42`", and `$3` be "`Dec 7 19:41`", as a successful match on the first time through the loop.

However, this regular expression match is in a list context, so instead of returning back a true/false value (as when you have a regular expression match in a scalar context), we take the memory variables and bundle them up in sequence as a list. In this case, the right side of that assignment is thus a three-element list, which happens to correspond to the three elements of the literal list on the left, and we get those nice corresponding assignments. So, `$user` ends up being `"merlyn"`, and so on.

The second statement inside the loop simply stores away the tty and date information, appending to a (possibly `undef`) value in the hash, because a user might be logged in more than once, as user `"rootbeer"` was in our example.

Chapter 15. Strings and Sorting

As we mentioned near the beginning of this book, Perl is designed to be good at solving programming problems that are about 90% working with text and 10% everything else. So it's no surprise that Perl has strong text processing abilities, including all that we've done with regular expressions. But sometimes the regular expression engine is too fancy, and you'll need a simpler way of working with a string, as we'll see in this chapter.

15.1. Finding a Substring with index

Finding a substring depends on where you have lost it. If you happen to have lost it within a bigger string, you're in luck, because the `index` function can help you out. Here's how it looks:

```
$where = index($big, $small);
```

Perl locates the first occurrence of the small string within the big string, returning an integer location of the first character. The character position returned is a zero-based value -- if the substring is found at the very beginning of the string, `index` returns `0`. If it's one character later, the return value is `1`, and so on. If the substring can't be found at all, the return value is `-1` to indicate that. In this example, `$where` gets `6`:

```
my $stuff = "Howdy world!";
my $where = index($stuff, "wor");
```

Another way you could think of the position number is the number of characters to skip over before getting to the substring. Since `$where` is `6`, we know that we have to skip over the first six characters of `$stuff` before we find `wor`.

The `index` function will always report the location of the *first found* occurrence of the substring. But you can tell it to start searching at a later point than the start of the string by using the optional third parameter, which tells `index` to start at that position:

```
my $stuff  = "Howdy world!";
my $where1 = index($stuff, "w");               # $where1 gets 2
my $where2 = index($stuff, "w", $where1 + 1);  # $where2 gets 6
my $where3 = index($stuff, "w", $where2 + 1);  # $where3 gets -1 (not found)
```

(Of course, you wouldn’t normally search repeatedly for a substring without using a loop.) That third parameter is effectively giving a minimum value for the return value; if the substring can’t be found at that position or later, the return value will be `-1`.

Once in a while, you might prefer to have the *last found* occurrence of the substring. You can get that with the `rindex` function. In this example, we can find the last slash, which turns out to be at position `4` in a string:

```
my $last_slash = rindex("/etc/passwd", "/");  # value is 4
```

The `rindex` function also has an optional third parameter, but in this case it effectively gives the *maximum* permitted return value:

```
my $fred = "Yabba dabba doo!";
my $where1 = rindex($fred, "abba");  # $where1 gets 7
my $where2 = rindex($fred, "abba", $where1 - 1);  # $where2 gets 1
my $where3 = rindex($fred, "abba", $where2 - 1);  # $where3 gets -1
```

15.2. Manipulating a Substring with substr

The `substr` operator works with only a part of a larger string. It looks like this:

```
$part = substr($string, $initial_position, $length);
```

It takes three arguments: a string value, a zero-based initial position (like the return value of `index`), and a length for the substring. The return value is the substring:

```
my $mineral = substr("Fred J. Flintstone", 8, 5);  # gets "Flint"
my $rock = substr "Fred J. Flintstone", 13, 1000;  # gets "stone"
```

As you may have noticed in the previous example, if the requested length (`1000` characters, in this case) would go past the end of the string, there’s no complaint from Perl, but you simply get a shorter string than you might have. But if you want to be sure to go to the end of the string, however long or short it may be, just omit that third parameter (the length), like this:

```
my $pebble = substr "Fred J. Flintstone", 13;  # gets "stone"
```

The initial position of the substring in the larger string can be negative, counting from the end of the string (that is, position `-1` is the last character). In this example, position `-3` is three characters from the end of the string, which is the location of the letter `i`:

```
my $out = substr("some very long string", -3, 2);  # $out gets "in"
```

As you might expect, `index` and `substr` work well together. In this example, we can extract a substring that starts at the location of the letter `l`:

```
my $long = "some very very long string";
my $right = substr($long, index($long, "l") );
```

Now here’s something really cool: The selected portion of the string can be changed if the string is a variable:

```
my $string = "Hello, world!";
substr($string, 0, 5) = "Goodbye";  # $string is now "Goodbye, world!"
```

As you see, the assigned (sub)string doesn't have to be the same length as the substring it's replacing. The string's length is adjusted to fit. Or if that wasn't cool enough to impress you, you could use the binding operator (=~) to restrict an operation to work with just part of a string. This example replaces fred with barney wherever possible within just the last twenty characters of a string:

```
substr($string, -20) =~ s/fred/barney/g;
```

To be completely honest, we've never actually needed that functionality in any of our own code, and chances are that you'll never need it either. But it's nice to know that Perl can do more than you'll ever need, isn't it?

Much of the work that substr and index do could be done with regular expressions. Use those where they're appropriate. But substr and index can often be faster, since they don't have the overhead of the regular expression engine: they're never case-insensitive, they have no metacharacters to worry about, and they don't set any of the memory variables.

Besides assigning to the substr function (which looks a little weird at first glance, perhaps), you can also use substr in a slightly more traditional manner with the four-argument version, in which the fourth argument is the replacement substring:

```
my $previous_value = substr($string, 0, 5, "Goodbye");
```

The previous value comes back as the return value, although as always, you can use this function in a void context to simply discard it.

15.3. Formatting Data with sprintf

The sprintf function takes the same arguments as printf (except for the optional filehandle, of course), but it returns the requested string instead of printing it. This is handy if you want to store a formatted string into a variable for later use, or if you want more control over the result than printf alone would provide:

```
my $date_tag = sprintf
  "%4d/%02d/%02d %2d:%02d:%02d",
  $yr, $mo, $da, $h, $m, $s;
```

In that example, $date_tag gets something like "2038/01/19 3:00:08". The format string (the first argument to sprintf) used a leading zero on some of the format number, which we didn't mention when we talked about printf formats in Chapter 6, "I/O Basics". The leading zero on the format number means to use leading zeroes as needed to make the number as wide as requested. Without a leading zero in the formats, the resulting date-and-time string would have unwanted leading spaces instead of zeroes, looking like "2038/ 1/19 3: 0: 8".

15.3.1. Using sprintf with "Money Numbers"

One popular use for sprintf is when a number needs to be rendered with a certain number of places after the decimal point, such as when an amount of money needs to be shown as 2.50 and not 2.5 -- and certainly not as 2.49997! That's easy to accomplish with the "%.2f" format:

```
my $money = sprintf "%.2f", 2.49997;
```

The full implications of rounding are numerous and subtle, but in most cases you should keep numbers in memory with all of the available accuracy, rounding off only for output.

If you have a "money number" that may be large enough to need commas to show its size, you might find it handy to use a subroutine like this one.

```
sub big_money {
  my $number = sprintf "%.2f", shift @_;
  # Add one comma each time through the do-nothing loop
  1 while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
  # Put the dollar sign in the right place
  $number =~ s/^(-?)/$1\$/;
  $number;
}
```

This subroutine uses some techniques you haven't seen yet, but they logically follow from what we've shown you. The first line of the subroutine formats the first (and only) parameter to have exactly two digits after the decimal point. That is, if the parameter were the number `12345678.9`, now our `$number` is the string `"12345678.90"`.

The next line of code uses a `while` modifier. As we mentioned when we covered that modifier in Chapter 10, "More Control Structures", that can always be rewritten as a traditional `while` loop:

```
while ($number =~ s/^(-?\d+)(\d\d\d)/$1,$2/) {
  1;
}
```

What does that say to do? It says that, as long as the substitution returns a true value (signifying success), the loop body should run. But the loop body does nothing! That's okay with Perl, but it tells us that the purpose of that statement is to do the conditional expression (the substitution), rather than the useless loop body. The value `1` is traditionally used as this kind of a placeholder, although any other value would be equally useful. This works just as well as the loop above:

```
'keep looping' while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
```

So, now we know that the substitution is the real purpose of the loop. But what is the substitution doing? Remember that `$number` will be some string like `"12345678.90"` at this point. The pattern will match the first part of the string, but it can't get past the decimal point. (Do you see why it can't?) Memory `$1` will get `"12345"`, and `$2` will get `"678"`, so the substitution will make `$number` into `"12345,678.90"` (remember, it couldn't match the decimal point, so the last part of the string is left untouched).

Do you see what the dash is doing near the start of that pattern? (Hint: The dash is allowed at only one place in the string.) We'll tell you at the end of this section, in case you haven't figured it out.

We're not done with that substitution statement yet. Since the substitution succeeded, the do-nothing loop goes back to try again. This time, the pattern can't match anything from the comma onward, so `$number` becomes `"12,345,678.90"`. The substitution thus adds a comma to the number each time through the loop.

Speaking of the loop, it's still not done. Since the previous substitution was a success, we're back around the loop to try again. But this time, the pattern can't match at all, since it has to match at least four digits at the start of the string, so now that is the end of the loop.

Why couldn't we have simply used the `/g` modifier to do a "global" search-and-replace, to save the trouble and confusion of the `1 while`? We couldn't use that because we're working backwards from the decimal point, rather than forward from the start of the string. Putting the commas in a number like this can't be done simply with the `s///g` substitution alone.

So, did you figure out the dash? It's allowing for a possible minus-sign at the start of the string. The next line of code makes the same allowance, putting the dollar-sign in the right place so that `$number` is something like `"$12,345,678.90"`, or perhaps `"-$12,345,678.90"` if it's negative. Note that the dollar sign isn't necessarily the first character in the string, or that line would be a lot simpler. Finally, the last line of code returns our nicely formatted "money number," ready to be printed in the annual report.

15.4. Advanced Sorting

Earlier, in Chapter 3, "Lists and Arrays ", we showed that you could sort a list in ascending ASCIIbetical order by using the builtin `sort` operator. What if you want a numeric sort? Or a case-insensitive sort? Or maybe you want to sort items according to information stored in a hash. Well, Perl lets you sort a list in whatever order you'd need; we'll see all of those examples by the end of the chapter.

You'll tell Perl what order you want by making a *sort-definition subroutine*, or *sort subroutine* for short. Now, when you first hear the term "sort subroutine," if you've been through any computer science courses, visions of bubble sort and shell sort and quick sort race through your head, and you say, "No, never again!" Don't worry; it's not that bad. In fact, it's pretty simple. Perl already knows how to sort a list of items; it merely doesn't know which order you want. So the sort-definition subroutine simply tells it the order.

Why is this necessary? Well, if you think about it, sorting is putting a bunch of things in order by comparing them all. Since you can't compare them all at once, you need to compare two at a time, eventually using what you find out about each pair's order to put the whole kit'n'caboodle in line. Perl already understands all of those steps *except* for the part about how you'd like to compare the items, so that's all you have to write.

This means that the sort subroutine doesn't need to sort many items after all. It merely has to be able to compare two items. If it can put two items in the proper order, Perl will be able to tell (by repeatedly consulting the sort subroutine) what order you want for your data.

The sort subroutine is defined like an ordinary subroutine (well, almost). This routine will be called repeatedly, each time checking on a pair of elements from the list to be sorted.

Now, if you were writing a subroutine that's expecting to get two parameters that need sorting, you might write something like this to start:

```
sub any_sort_sub {  # It doesn't really work this way
  my($a, $b) = @_;  # Get and name the two parameters
  # start comparing $a and $b here
  ...
}
```

But the sort subroutine will be called again and again, often hundreds or thousands of times. Declaring the variables `$a` and `$b` and assigning them values at the top of the subroutine will take just a little time, but multiply that by the thousands of times that the routine will be called, and you can see that it contributes significantly to the overall execution speed.

We don't do it like that. (In fact, if you did it that way, it wouldn't work.) Instead, it is as if Perl has done this for us, before our subroutine's code has even started. You'll really write a sort subroutine without that first line; both `$a` and `$b` have been assigned for you. When the sort subroutine starts running, `$a` and `$b` are two elements from the original list.

The subroutine returns a coded value describing how the elements compare (like C's `qsort(3)` does, but it's Perl's own internal sort implementation). If `$a` should appear before `$b` in the final list, the sort subroutine returns `-1` to say so. If `$b` should appear before `$a`, it returns `1`.

If the order of `$a` and `$b` doesn't matter, the subroutine returns `0`. Why would it not matter? Perhaps you're doing a case-insensitive sort and the two strings are `fred` and `Fred`. Or perhaps you're doing a numeric sort, and the two numbers are equal.

We could now write a numeric sort subroutine like this:

```
sub by_number {
  # a sort subroutine, expect $a and $b
  if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }
}
```

To use the sort subroutine, just put its name (without an ampersand) between the keyword `sort` and the list to be sorted. This example puts a numerically sorted list of numbers into `@result`:

```
my @result = sort by_number @some_numbers;
```

We called the subroutine `by_number` because that describes how it's sorting. But more importantly, you can read the line of code that uses it with sort as saying "sort by number," as you would in English. Many sort-subroutine names begin with `by_` to describe how they sort. Or we could have called this one `numerically`, for a similar reason, but that's more typing and more chance to mess something up.

Notice that we don't have to do anything in the sort subroutine to declare `$a` and `$b`, and to set their values -- and if we did, the subroutine wouldn't work right. We just let Perl set up `$a` and `$b` for us, and so all we need to write is the comparison.

In fact, we can make it even simpler (and more efficient). Since this kind of three-way comparison is frequent, Perl has a convenient shortcut to use to write it. In this case, we use the spaceship operator (<=>). This operator compares two numbers and returns `-1`, `0`, or `1` as needed to sort them numerically. So we could have written that sort subroutine better, like this:

```
sub by_number { $a <=> $b }
```

Since the spaceship compares numbers, you may have guessed that there's a corresponding three-way string-comparison operator: `cmp`. These two are easy to remember and keep straight. The spaceship has a family resemblance to the numeric comparison operators like >=, but it's three characters long instead of two because it has three possible return values instead of two. And `cmp` has a family resemblance to the string comparison operators like `ge`, but it's three characters long instead of two because it *also* has three possible return values instead of two.

Of course, `cmp` by itself provides the same order as the default sort. You'd never need to write this subroutine, which yields merely the default sort order:

```
    sub ASCIIbetically { $a cmp $b }
    my @strings = sort ASCIIbetically @any_strings;
```

But you can use `cmp` to build a more complex sort order, like a case-insensitive sort:

```
    sub case_insensitive { "\L$a" cmp "\L$b" }
```

In this case, we're comparing the string from `$a` (forced to lowercase) against the string from `$b` (forced to lowercase), giving a case-insensitive sort order.

Note that we're not modifying the elements themselves; we're merely using their values. That's actually important: for efficiency reasons, `$a` and `$b` aren't copies of the data items. They're actually new, temporary aliases for elements of the original list, so if we changed them we'd be mangling the original data. Don't do that -- it's neither supported nor recommended.

When your sort subroutine is as simple as the ones we show here (and most of the time, it is), you can make the code even simpler yet, by replacing the name of the sort routine with the entire sort routine "in line," like so:

```
    my @numbers = sort { $a <=> $b } @some_numbers;
```

In fact, in modern Perl, you'll hardly ever see a separate sort subroutine; you'll frequently find sort routines written inline as we've done here.

Suppose you want to sort in descending numeric order. That's easy enough to do with the help of `reverse`:

```
    my @descending = reverse sort { $a <=> $b } @some_numbers;
```

But here's a neat trick. The comparison operators (<=> and `cmp`) are very nearsighted; that is, they can't see which operand is `$a` and which is `$b`, but only which *value* is on the left and which is on the right. So if `$a` and `$b` were to swap places, the comparison operator would get the results backwards every time. That means that this is another way to get a reversed numeric sort:

```
    my @descending = sort { $b <=> $a } @some_numbers;
```

You can (with a little practice) read this at a glance. It's a descending-order comparison (because `$b` comes before `$a`, which is descending order), and it's a numeric comparison (because it uses the spaceship instead of `cmp`). So, it's sorting numbers in reverse order.

15.4.1. Sorting a Hash by Value

Once you've been sorting lists happily for a while you'll run into a situation where you want to sort a hash by value. For example, three of our characters went out bowling last night, and we've got their bowling scores in the following hash. We want to be able to print out the list in the proper order, with the game winner at the top, so we want to sort the hash by score:

```
    my %score = ("barney" => 195, "fred" => 205, "dino" => 30);
    my @winners = sort by_score keys %score;
```

Of course, we aren't really going to be able to sort the hash by score; that's just a verbal shortcut. You can't sort a hash! But when we've used `sort` with hashes before now, we've been sorting the keys of the hash (in ASCIIbetical order). Now, we're still going to be sorting the keys of the hash, but the order is now defined by their corresponding values from the hash. In this case, the result should be a list of our three characters' names, in order according to their bowling scores.

Writing this sort subroutine is fairly easy. What we want is to use a numeric comparison on the scores, rather than the names. That is, instead of comparing `$a` and `$b` (the players' names), we want to compare `$score{$a}` and `$score{$b}` (their scores). If you think of it that way, it almost writes itself, as in:

```
sub by_score { $score{$b} <=> $score{$a} }
```

Let's step through this and see how it works. Let's imagine that the first time it's called, Perl has set `$a` to `barney` and `$b` to `fred`. So the comparison is `$score{"fred"} <=> $score{"barney"}`, which (as we can see by consulting the hash) is `205 <=> 195`. Remember, now, the spaceship is nearsighted, so when it sees `205` before `195`, it says, in effect: "No, that's not the right numeric order; `$b` should come before `$a`." So it tells Perl that `fred` should come before `barney`.

Maybe the next time the routine is called, `$a` is `barney` again but `$b` is now `dino`. The nearsighted numeric comparison sees `30 <=> 195` this time, so it reports that that they're in the right order; `$a` does indeed sort in front of `$b`. That is, `barney` comes before `dino`. At this point, Perl has enough information to put the list in order: `fred` is the winner, then `barney` in second place, then `dino`.

Why did the comparison use the `$score{$b}` before the `$score{$a}`, instead of the other way around? That's because we want bowling scores arranged in *descending* order, from the highest score of the winner down. So you can (again, after a little practice) read this one at sight as well: `$score{$b} <=> $score{$a}` means to sort according to the scores, in reversed numeric order.

15.4.2. Sorting by Multiple Keys

We forgot to mention that there was a fourth player bowling last night with the other three, so the hash really looked like this:

```
my %score = (
  "barney" => 195, "fred" => 205,
  "dino" => 30, "bamm-bamm" => 195,
);
```

Now, as you can see, `bamm-bamm` has the same score as `barney`. So which one will be first in the sorted list of players? There's no telling, because the comparison operator (seeing the same score on both sides) will have to return zero when checking those two.

Maybe that doesn't matter, but we generally prefer to have a well-defined sort. If several players have the same score, we want them to be together in the list, of course. But within that group, the names should be in ASCIIbetical order. But how can we write the sort subroutine to say that? Again, this turns out to be pretty easy:

```
my @winners = sort by_score_and_name keys %score;

sub by_score_and_name {
  $score{$b} <=> $score{$a}  # by descending numeric score
    or
  $a cmp $b                  # ASCIIbetically by name
}
```

How does this work? Well, if the spaceship sees two different scores, that's the comparison we want to use. It returns `-1` or `1`, a true value, so the low-precedence short-circuit `or` will mean that the rest of the expression will be skipped, and the comparison we want is returned. (Remember, the short-circuit `or` returns the last expression evaluated.) But if the spaceship sees two identical scores, it returns `0`, a false value, and thus the `cmp` operator gets its turn at bat, returning an appropriate ordering value considering the keys as strings. That is, if the scores are the same, the string-order comparison breaks the tie.

We know that when we use the `by_score_and_name` sort subroutine like this, it will never return `0`. (Do you see why it won't? The only way it could return `0` would be if the two strings were identical, and, since the strings are keys of a hash, we already know that they're different. Of course, if you passed a list with duplicate strings to `sort`, it would return `0` when comparing those, but we're passing a list of hash keys.) So we know that the sort order is always well-defined; that is, we know that the result today will be the same as the result with the same data tomorrow.

There's no reason that your sort subroutine has to be limited to two levels of sorting, of course. Here the Bedrock library program puts a list of patron ID numbers in order according to a five-level sort. This example sorts according to the amount of each patron's outstanding fines (as calculated by a subroutine `&fines`, not shown here), the number of items they currently have checked out (from `%items`), their name (in order by family name, then by personal name, both from hashes), and finally by the patron's ID number, in case everything else is the same:

```
@patron_IDs = sort {
  &fines($b) <=> &fines($a) or
  $items{$b} <=> $items{$a} or
  $family_name{$a} cmp $family_name{$a} or
```

```
    $personal_name{$a} cmp $family_name{$b} or
    $a <=> $b
  } @patron_IDs;
```

17.2. Picking Items from a List with grep

Sometimes you'll want only certain items from a list. Maybe it's only the odd numbers selected from a list of numbers, or maybe it's only the lines mentioning `Fred` from a file of text. As we'll see in this section, picking some items from a list can be done simply with the `grep` operator.

Let's try that first one and get the odd numbers from a large list of numbers. We don't need anything new to do that:

```
my @odd_numbers;

foreach (1..1000) {
  push @odd_numbers, $_ if $_ % 2;
}
```

That code uses the modulus operator (`%`), which we saw in Chapter 2, "Scalar Data". If a number is even, that number "mod two" gives zero, which is false. But an odd number will give one; since that's true, only the odd numbers will be pushed onto the array.

Now, there's nothing wrong with that code as it stands -- except that it's a little longer to write and slower to run than it might be, since Perl provides the `grep` operator:

```
my @odd_numbers = grep { $_ % 2 } 1..1000;
```

That line gets a list of 500 odd numbers in one quick line of code. How does it work? The first argument to `grep` is a block that uses `$_` as a placeholder for each item in the list, and returns a Boolean (true/false) value. The remaining arguments are the list of items to search through. The `grep` operator will evaluate the expression once for each item in the list, much as our original `foreach` loop did. For the ones where the last expression of the block returns a true value, that element is included in the list that results from `grep`.

While the `grep` is running, `$_` is aliased to one element of the list after another. We've seen this behavior before, in the `foreach` loop. It's generally a bad idea to modify `$_` inside the `grep` expression, because this will damage the original data.

The `grep` operator shares its name with a classic Unix utility that picks matching lines from a file by using regular expressions. We can do that with Perl's `grep`, which is much more powerful. Here we pull only the lines mentioning `fred` from a file:

```
my @matching_lines = grep { /\bfred\b/i } <FILE>;
```

There's a simpler syntax for `grep`, too. If all you need for the selector is a simple expression (rather than a whole block), you can just use that expression, followed by a comma, in place of the block. Here's the simpler way to write that latest example:

```
my @matching_lines = grep /\bfred\b/i, <FILE>;
```

17.3. Transforming Items from a List with map

Another common task is transforming items from a list. For example, suppose you have a list of numbers that should be formatted as "money numbers" for output, as with the subroutine `&big_money` (from Chapter 15, "Strings and Sorting"). But we don't want to modify the original data; we need a modified copy of the list just for output. Here's one way to do that:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
my @formatted_data;

foreach (@data) {
  push @formatted_data, &big_money($_);
}
```

That looks similar in form to the example code used at the beginning of the section on `grep`, doesn't it? So it may not surprise you that the replacement code resembles the first `grep` example:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);

my @formatted_data = map { &big_money($_) } @data;
```

The `map` operator looks much like `grep` because it has the same kind of arguments: a block that uses `$_`, and a list of items to process. And it operates in a similar way, evaluating the block once for each item in the list, with `$_` aliased to a different original list element each time. But the last expression of the block is used differently; instead of giving a Boolean value, the final value actually becomes part of the resulting list.

Any `grep` or `map` statement could be rewritten as a `foreach` loop pushing items onto a temporary array. But the shorter way is typically more efficient and more convenient. Since the result of `map` or `grep` is a list, it can be passed directly to another function. Here we can print that list of formatted "money numbers" as an indented list under a heading:

```
print "The money numbers are:\n",
  map { sprintf("%25s\n", $_) } @formatted_data;
```

Of course, we could have done that processing all at once, without even the temporary array `@formatted_data`:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
print "The money numbers are:\n",
  map { sprintf("%25s\n", &big_money($_) ) } @data;
```

As we saw with `grep`, there's also a simpler syntax for `map`. If all you need for the selector is a simple expression (rather than a whole block), you can just use that expression, followed by a comma, in place of the block:

```
print "Some powers of two are:\n",
  map "\t" . ( 2 ** $_ ) . "\n", 0..15;
```

17.4. Unquoted Hash Keys

Perl offers many shortcuts that can help the programmer. Here's a handy one: you may omit the quote marks on some hash keys.

Of course, you can't omit the quote marks on just *any* key, since a hash key may be any arbitrary string. But keys are often simple. If the hash key is made up of nothing but letters, digits, and underscores without starting with a digit, you *may* be able to omit the quote marks. This kind of simple string without quote marks is called a *bareword*, since it stands alone without quotes.

One place you are permitted to use this shortcut is the most common place a hash key appears: in the curly braces of a hash element reference. For example, instead of `$score{"fred"}`, you could write simply `$score{fred}`. Since many hash keys are simple like this, not using quotes is a real convenience. But beware; if there's anything inside the curly braces besides a bareword, Perl will interpret it as an expression.

Another place where hash keys appear is when assigning an entire hash using a list of key-value pairs. The big arrow (=>) is especially useful between a key and a value, because (again, only if the key is a bareword) the big arrow quotes it for you:

```
# Hash containing bowling scores
my %score = (
  barney   => 195,
  fred     => 205,
  dino     => 30,
);
```

This is the one important difference between the big arrow and a comma; a bareword to the left of the big arrow is implicitly quoted. (Whatever is on the right is left alone, though.) This feature of the big arrow doesn't have to be used only for hashes, although that's the most frequent use.

17.6. Slices

It often happens that we need to work with only a few elements from a given list. For example, the Bedrock Library keeps information about their patrons in a large file. Each line in the file describes one patron with six colon-separated fields: a person's name, library card number, home address, home phone number, work phone number, and number of items currently checked out. A little bit of the file looks something like this:

```
fred flintstone:2168:301 Cobblestone Way:555-1212:555-2121:3
barney rubble:709918:3128 Granite Blvd:555-3333:555-3438:0
```

One of the library's applications needs only the card numbers and number of items checked out; it doesn't use any of the other data. It could use code something like this to get only the fields it needs:

```
while (<FILE>) {
  chomp;
  my @items = split /:/;
  my($card_num, $count) = ($items[1], $items[5]);
  ...  # now work with those two variables
}
```

But the array `@items` isn't needed for anything else; it seems like a waste. Maybe it would be better to assign the result of `split` to a list of scalars, like this:

```
my($name, $card_num, $addr, $home, $work, $count) = split /:/;
```

Well, that avoids the unneeded array `@items` -- but now we have four scalar variables that we didn't really need. For this situation, some people used to make up a number of dummy variable names, like `$dummy_1`, that showed that they really didn't care about that element from the `split`. But Larry thought that that was too much trouble, so he added a special use of `undef`. If an item in a list being assigned to is `undef`, that means simply to ignore the corresponding element of the source list:

```
my(undef, $card_num, undef, undef, undef, $count) = split /:/;
```

Is this any better? Well, it has an advantage that there aren't any unneeded variables. But it has the disadvantage that you have to count `undef`s to tell which element is `$count`. And this becomes quite unwieldy if there are more elements in the list. For example, some people who wanted just the mtime value from `stat` were writing code like this:

```
my(undef, undef, undef, undef, undef, undef, undef,
  undef, undef, $mtime) = stat $some_file;
```

If you use the wrong number of `undef`s, you'll get the atime or ctime by mistake, and that's a tough one to debug. There's a better way: Perl can index into a list as if it were an array. This is a *list slice*. Here, since the mtime is item `9` in the list returned by `stat`, we can get it with a subscript:

```
my $mtime = (stat $some_file)[9];
```

Those parentheses are required around the list of items (in this case, the return value from `stat`). If you wrote it like this, it wouldn't work:

```
my $mtime = stat($some_file)[9];  # Syntax error!
```

A list slice has to have a subscript expression in square brackets after a list in parentheses. The parentheses holding the arguments to a function call don't count.

Going back to the Bedrock Library, the list we're working with is the return value from `split`. We can now use a slice to pull out item `1` and item `5` with subscripts:

```
my $card_num = (split /:/)[1];
my $count = (split /:/)[5];
```

Using a scalar-context slice like this (pulling just a single element from the list) isn't bad, but it would be more efficient and simpler if we didn't have to do the `split` twice. So let's not do it twice; let's get both values at once by using a list slice in list context:

```
    my($card_num, $count) = (split /:/)[1, 5];
```

The indices pull out element `1` and element `5` from the list, returning those as a two-element list. When that's assigned to the two `my` variables, we get exactly what we wanted. We do the `slice` just once, and we set the two variables with a simple notation.

A slice is often the simplest way to pull a few items from a list. Here, we can pull just the first and last items from a list, using the fact that index `-1` means the last element:

```
    my($first, $last) = (sort @names)[0, -1];
```

The subscripts of a slice may be in any order and may even repeat values. This example pulls five items from a list of ten:

```
    my @names = qw{ zero one two three four five six seven eight nine };
    my @numbers = ( @names )[ 9, 0, 2, 1, 0 ];
    print "Bedrock @numbers\n";  # says Bedrock nine zero two one zero
```

17.6.1. Array Slice

That previous example could be made even simpler. When slicing elements from an array (as opposed to a list), the parentheses aren't needed. So we could have done the slice like this:

```
    my @numbers = @names[ 9, 0, 2, 1, 0 ];
```

This isn't merely a matter of omitting the parentheses; this is actually a different notation for accessing array elements: an *array slice*. Earlier (in Chapter 3, "Lists and Arrays"), we said that the at-sign on `@names` meant "all of the elements." Actually, in a linguistic sense, it's more like a plural marker, much like the letter "s" in words like "cats" and "dogs." In Perl, the dollar sign means there's just one of something, but the at-sign means there's a list of items.

A slice is always a list, so the array slice notation uses an at-sign to indicate that. When you see something like `@names [ ... ]` in a Perl program, you'll need to do just as Perl does and look at the at-sign at the beginning as well as the square brackets at the end. The square brackets mean that you're indexing into an array, and the at-sign means that you're getting a whole list of elements, not just a single one (which is what the dollar sign would mean). See Figure 17-1.

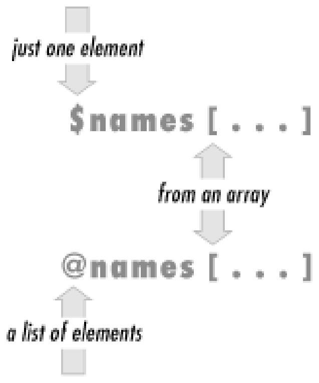

Figure 17-1. Array slices versus single elements

The punctuation mark at the front of the variable reference (either the dollar sign or at-sign) determines the context of the subscript expression. If there's a dollar sign in front, the subscript expression is evaluated in a scalar context to get an index. But if there's an at-sign in front, the subscript expression is evaluated in a list context to get a list of indices.

So we see that `@names[ 2, 5 ]` means the same list as `($names[2], $names[5])` does. If you want that list of values, you can simply use the array slice notation. Any place you might want to write the list, you can instead use the simpler array slice.

But the slice can be used in one place where the list can't: a slice may be interpolated directly into a string:

```
    my @names = qw{ zero one two three four five six seven eight nine };
    print "Bedrock @names[ 9, 0, 2, 1, 0 ]\n";
```

If we were to interpolate `@names`, that would give all of the items from the array, separated by spaces. If instead we interpolate `@names[ 9, 0, 2, 1, 0 ]`, that gives just those items from the array, separated by spaces.

Let's go back to the Bedrock Library for a moment. Maybe now our program is updating Mr. Slate's address and phone number in the patron file, because he just moved into a large new place in the Hollyrock hills. If we've got a list of information about him in `@items`, we could do something like this to update just those two elements of the array:

```
    my $new_home_phone = "555-6099";
    my $new_address = "99380 Red Rock West";
    @items[2, 3] = ($new_address, $new_home_phone);
```

Once again, the array slice makes a more compact notation for a list of elements. In this case, that last line is the same as an assignment to `($items[2], $items[3])`, but more compact and efficient.

17.6.2. Hash Slice

In a way exactly analogous to an array slice, we can also slice some elements from a hash in a *hash slice*. Remember when three of our characters went bowling, and we kept their bowling scores in the `%score` hash? We could pull those scores with a list of hash elements or with a slice. These two techniques are equivalent, although the second is more concise and efficient:

```
    my @three_scores = ($score{"barney"}, $score{"fred"}, $score{"dino"});

    my @three_scores = @score{ qw/ barney fred dino/ };
```

A slice is always a list, so the hash slice notation uses an at-sign to indicate that. When you see something like `@score { ... }` in a Perl program, you'll need to do just as Perl does and look at the at-sign at the beginning as well as the curly braces at the end. The curly braces mean that you're indexing into a hash; the at-sign means that you're getting a whole list of elements, not just a single one (which is what the dollar sign would mean). See Figure 17-2.

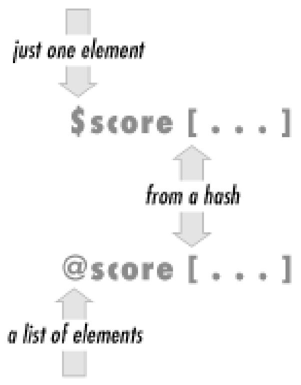

Figure 17-2. Hash slices versus single elements

As we saw with the array slice, the punctuation mark at the front of the variable reference (either the dollar sign or at-sign) determines the context of the subscript expression. If there's a dollar sign in front, the subscript expression is evaluated in a scalar context to get a single key. But if there's an at-sign in front, the subscript expression is evaluated in a list context to get a list of keys.

It's normal at this point to wonder why there's no percent sign ("`%`") here, when we're talking about a hash. That's the marker that means there's a whole hash; a hash slice (like any other slice) is always a *list*, not a hash. In Perl, the dollar sign means there's just one of something, but the at-sign means there's a list of items, and the percent sign means there's an entire hash.

As we saw with array slices, a hash slice may be used instead of the corresponding list of elements from the hash, anywhere within Perl. So we can set our friends' bowling scores in the hash (without disturbing any other elements in the hash) in this simple way:

```
    my @players = qw/ barney fred dino /;
    my @bowling_scores = (195, 205, 30);
    @score{ @players } = @bowling_scores;
```

That last line does the same thing as if we had assigned to the three-element list `($score{"barney"}, $score {"fred"}, $score{"dino"})`.

A hash slice may be interpolated, too. Here, we print out the scores for our favorite bowlers:

```
print "Tonight’s players were: @players\n";
print "Their scores were: @score{@players}\n";
```