Chapter 2. Scalar Data

Perl's datatypes are simple. A *scalar* is a single thing. That's it. You may know the term *scalar* from physics or mathematics or some other discipline, but Perl's definition of the term is its own. It's so important that we'll say it again. A scalar is a single thing, and we use the word *thing* because we don't have a better way to describe what Perl considers 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.

If you have been using other programming languages, you're probably used to the idea of several different sorts of single items. C has `char`, `int`, and so on. Perl doesn't make distinctions among single things, which is something some people find hard to adjust to. However, as we'll see in this book, that allows us quite a bit of flexibility in dealing with data.

In this chapter, we show both scalar *data*, which are the values themselves, and scalar *variables*, which can store a scalar value. The distinction between these two is important. The value itself is fixed and we can't change it. We can, however, change what we store in a variable (hence its name). Sometimes programmers are a bit sloppy with this and simply say "scalar." We'll be a bit sloppy too, except when it matters. This will be more important in Chapter 3.

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.

All Numbers Have the Same Format Internally

Perl relies on the underlying C libraries for its numbers and uses a double-precision floating-point value to store the numbers. You don't have to know much about that, but it does mean Perl has some limitations in precision and magnitude of the numbers, which are related to how you compiled and installed the *perl* interpreter instead of a limitation of the language itself. Perl does this to be as fast as possible by using the optimizations of the local platform and libraries to do its math.

As you'll see in the next few paragraphs, you can specify both integers (whole numbers, like 255 or 2,001) and floating-point numbers (real numbers with decimal points, like 3.14159, or 1.35 × 1,025). 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. In Perl, a number is just a number, unlike some other languages that ask you to decide what magnitude and type of number it is.

Integer Literals

A *literal* is how you represent a value in your source code. A literal is not the result of a calculation or an I/O operation; it's data that you type directly into your program. Integer literals are straightforward, as in:

```
0
2001
-40
137
61298040283768
```

That last one is a little hard to read. Perl allows you to add underscores for clarity within integer literals, so you can also write that number with embedded underscores to make it easier to read:

```
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 you'll see in Chapter 3). Even then, not everyone uses commas to separate numbers.

Nondecimal Integer Literals

Like many other programming languages, Perl allows you to specify numbers in ways other than base 10 (decimal). Octal (base 8) literals start with a leading `0` and use the digits from 0 to 7:

```
0377       # same as 255 decimal
```

Starting with v5.34, you can also start octal numbers with `0o`, which brings octal numbers in line with the other bases you are about to see:

```
0o377       # same as 255 decimal
```

Hexadecimal (base 16) literals start with a leading `0x` and use the digits 0 to 9 and the letters `A` through `F` (or `a` through `f`) to represent the values from 0 to 15:

```
0xff       # FF hex, also 255 decimal
```

Binary (base 2) literals start with a leading `0b` and use only the digits 0 and 1:

```
0b11111111 # also 255 decimal
```

Although these values look different to us humans, all three are the same number to Perl. It makes no difference to Perl whether you write `0377`, `0xFF`, or `255`, so choose the representation that makes the most sense for your task. Many shell commands in the Unix world assume octal numbers, for instance, so octal values make sense for the Perl equivalents you'll see in Chapters 12 and 13.

NOTE

The “leading zero” indicator works only for literals—not for automatic string-to-number conversions, which you’ll see in “Automatic Conversion Between Numbers and Strings”.

When a nondecimal literal is more than about four characters long, it may be hard to read, so underscores are handy:

```
0x1377_0B77
0x50_65_72_7C
```

Floating-Point Literals

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 with E notation (also known as *exponential 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
```

Perl v5.22 added hexadecimal floating-point literals. Instead of an `e` to mark the exponent, you use a `p` for the power-of-2 exponent. Just like the hexadecimal integers, these start with `0x`:

```
0x1f.0p3
```

A hexadecimal floating-point literal is an exact representation of a number in the storage format that Perl uses. There is no ambiguity in its value. With a decimal floating-point number, Perl (or C or anything else that uses doubles) can't represent the number exactly if it's not a power of two. Most people don't even notice this, and those who do see a very slight round-off error.

Numeric Operators

Operators are Perl's verbs. They decide how to treat the nouns. Perl provides the typical ordinary addition, subtraction, multiplication, and division operators. These numeric operators always treat their operands as numbers, and are denoted by symbolic characters:

```
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's numeric operators return what you'd expect from doing the same operation on a calculator. Perl only has single values and it doesn't distinguish numbers that are integers, fractions, or floating-point numbers. This sometimes annoys people who are used to minutely specifying these in other languages. For instance, someone used to integer-only math might expect `10/3` to be another integer (`3`).

Perl also supports a *modulus* operator (`%`). The value of the expression `10 % 3` is the remainder when 10 is divided by 3, which is 1. Both values are first reduced to their integer values, so `10.5 % 3.2` is computed as `10 % 3`.

The result of the modulus operator when either or both numbers are negative can vary between Perl implementations because the underlying library does it differently because people disagree about rounding numbers. For `-10 % 3`, is the remainder `2` because it's two places above `-12` or `-1` because it's one place from `-9`? It's best to avoid finding out accidentally.

Additionally, Perl provides the FORTRAN-like *exponentiation* operator, represented by the double asterisk. So, `2**3` is two to the third power, or eight. In addition, there are other numeric operators, which we'll introduce as we need them.

Strings

Strings are sequences of characters, such as `hello` or `☃★๛`. Strings may contain any combination of any characters. The shortest possible string has no characters and is called the *empty string*. 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, digits, punctuation, and whitespace. 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.

Perl has full support for Unicode, and your string can contain any of the valid Unicode characters. However, because of Perl's history, it doesn't automatically interpret your source as Unicode. If you want to use Unicode literally in your program, you need to add the `utf8` pragma. It's probably a good practice to always include this in your program unless you know why you wouldn't want to:

```
use utf8;
```

For the rest of this book, we assume you're using that pragma. In some cases it won't matter, but if you see characters outside the ASCII range in the source, you'll need it. Also, you should ensure that you save your files with the UTF-8 encoding. If you skipped our advice about Unicode from Chapter 1, you might want to go through Appendix C to learn more.

NOTE

A *pragma* is something that tells the Perl compiler how to act.

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*.

Single-Quoted String Literals

A *single-quoted string literal* is a sequence of characters enclosed in single quotes, the `'` character. The single quotes are not part of the string itself—they're just there to let Perl identify the beginning and end of the string:

```
'fred'    # those four characters: f, r, e, and d
'barney'  # those six characters
''        # the null string (no characters)
'⅚∞☃☠'    # Some "wide" Unicode characters
```

Any character other than a single quote or a backslash between the quote marks stands for itself inside a string. If you want a literal single quote or backslash inside your string, you need to *escape* it with a backslash:

```
'Don\'t let an apostrophe end this string prematurely!'
'the last character is a backslash: \\'
'\'\\'    # single quote followed by backslash
```

You can spread your string out over two (or more) lines. A newline between the single quotes is a newline in your string:

```
'hello
there'    # hello, newline, there (11 characters total)
```

Note that Perl does not interpret the `\n` within a single-quoted string as a newline but as the two characters backslash and `n`:

```
'hello\nthere'    # hello\nthere
```

Only when the backslash is followed by another backslash or a single quote does it have special meaning.

Double-Quoted String Literals

A *double-quoted string literal* is 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
"\x{2668}"      # Unicode HOT SPRINGS character code point
"\N{SNOWMAN}"   # Unicode Snowman by name
```

Note that the double-quoted literal string `"barney"` means the same six-character string to Perl as does the single-quoted literal string `'barney'`.

The backslash can precede many characters to mean something other than their literal representation (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
\007	Any octal ASCII value (here, `007` = bell)
\a	Bell
\b	Backspace
\cC	A “control” character (here, Ctrl-C)
\e	Escape (ASCII escape character)
\E	End `\F`, `\L`, `\U`, or `\Q`
\f	Formfeed
\F	Unicode case-fold all following letters until `\E`
\l	Lowercase next letter
\L	Lowercase all following letters until `\E`
\n	Newline
\N{CHARACTER NAME}	Any Unicode code point, by name
\Q	Quote nonword characters by adding a backslash until `\E`
\r	Return
\t	Tab
\u	Uppercase next letter

Construct	Meaning
\U	Uppercase all following letters until `\E`
\x7f	Any two-digit, hex ASCII value (here, `7f` `=` delete)
\x{2744}	Any hex Unicode code point (here, `U+2744` `=` snowflake)
\\	Backslash
\“	Double quote

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. You haven’t formally been introduced to what a variable looks like yet, so we’ll get back to that later in this chapter.

String Operators

You can concatenate, or join, string values 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 assignment to 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 you must explicitly use the concatenation 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 4.8          # is really "5" x 4, which is "5555"
```

That last example is worth noting carefully. 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. The `x` copies the new string four times, yielding the four-character string `5555`. Note that if you had reversed the order of the operands, as `4 x 5`, you 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 `1` results in an empty (zero-length) string.

Automatic Conversion Between Numbers and Strings

For the most part, Perl automatically converts between numbers and strings as needed. How does it know which it should use? It all depends on the operator that you apply to 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 you use a string value where an operator needs a number (say, for multiplication), Perl automatically converts the string to its equivalent numeric value, as if you had entered it as a decimal floating-point value. So `"12" * "3"` gives the value `36`. Trailing nonnumerical stuff and leading whitespace are discarded, so `"12fred34" *  " 3"` will also give `36` without any complaints (until you turn on warnings, which we'll show in a moment). 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.

The trick of using a leading zero to mean an octal value only works for literals and never for automatic conversion, which is always base 10:

```
0377    # that's octal for 255 decimal
'0377'  # that's 377 decimal
```

Later we'll show you `oct` to convert that string value as an octal number.

Likewise, if a numeric value is given where 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. It even remembers what conversions it's already done so it can be faster next time.

Perl's Built-in Warnings

Perl can be told to warn you when it sees something suspicious going on in your program. With Perl 5.6 and later, you can turn on warnings with a pragma (but be careful because it won't work for people with ancient versions of Perl):

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

You can use the `-w` option on the command line, which turns on warnings everywhere in your program, including modules that you use but didn’t write yourself, so you might see warnings from other people’s code:

```
$ perl -w my_program
```

You can also specify the command-line switches on the shebang line:

```
#!/usr/bin/perl -w
```

Now, Perl will warn you if you use `'12fred34'` as if it were a number:

```
Argument "12fred34" isn't numeric
```

NOTE

The advantage of `warnings` over `-w` is that you only turn on warnings for the file in which you use the pragma, whereas `-w` turns on warnings for the entire program.

Perl still turns the nonnumeric `'12fred34'` into `12` using its normal rules even though you get the warning.

Of course, warnings are generally meant for programmers, not for end users. If the warning won’t be seen by a programmer, it probably won’t do you any good. And warnings won’t change the behavior of your program, except that now it gripes once in a while. If you get a warning message you don’t understand, you can get a longer description of the problem with the `diagnostics` pragma. The perldiag documentation has both the short warning and the longer diagnostic description, and is the source of `diagnostics`’s helpfulness:

```
use diagnostics;
```

When you add the `use diagnostics` pragma to your program, it may seem to you that your program now pauses for a moment whenever you launch it. That’s because your program has to do a lot of work (and gobble a chunk of memory) just in case you want to read the documentation as soon as Perl notices your mistakes, if any. This leads to a nifty optimization that can speed up your program's launch (and memory footprint) with no adverse impact on users: once you no longer need to read the documentation about the warning messages produced by your program, remove the `use diagnostics` pragma. It's even better if you fix your program to avoid causing the warnings. But it's sufficient merely to finish reading the output.

A further optimization can be had by using one of Perl's command-line options, `-M`, to load the pragma only when needed instead of editing the source code each time to enable and disable `diagnostics`:

```
$ perl -Mdiagnostics ./my_program
Argument "12fred34" isn't numeric in addition (+) at ./my_program line 17 (#1)
    (W numeric) The indicated string was fed as an argument to
    an operator that expected a numeric value instead.  If you're
    fortunate the message will identify which operator was so unfortunate.
```

Note the `(W numeric)` in the message. The `W` says that the message is a warning and the `numeric` is the class of warning. In this case, you know to look for something dealing with a number.

As we run across situations in which Perl will usually be able to warn us about a mistake in our code, we'll point them out. But you shouldn't count on the text or behavior of any warning staying exactly the same in future Perl releases.

Interpreting Nondecimal Numerals

If you have a string that represents a number as another base, you can use the `hex()` or `oct()` function to interpret those numbers correctly. Curiously, the `oct()` function is smart enough to recognize the correct base if you use prefix characters to specify hex or binary, but the only valid prefix for hex is `0x`:

```
hex('DEADBEEF')     # 3_735_928_559 decimal
hex('OxDEADBEEF')   # 3_735_928_559 decimal
```

```
oct('0377')        # 255 decimal
oct('0o377')       # 255 decimal, new in v5.34, saw leading 0o
oct('377')         # 255 decimal
oct('0xDEADBEEF')  # 3_735_928_559 decimal, saw leading 0x
oct('0b1101')      # 13 decimal, saw leading 0b
oct("0b$bits")     # convert $bits from binary
```

Those string representations are for us; the computer doesn't care how we want to think about the number. Specifying the same number in decimal or hexadecimal is all the same to Perl. As long as we correctly identify the *radix* of the number, Perl will translate it into its internal format.

Remember that Perl's automatic conversion only works for base-10 numbers, and that these routines only work on strings. Giving any of these a literal number, which Perl will have already converted to its internal format, will likely give the wrong results. Perl will turn the number back into a string, which it interprets as a hexadecimal number string, then into a Perl number:

```
hex( 10 )   # decimal 10, converted to "10", then decimal 16
hex( 0x10 ) # hex 10,     converted to "16", then decimal 22
```

We'll show you more about printing numbers in different bases in Chapter 5.

Scalar Variables

A *variable* is a name for a container that holds one or more values. As you'll see, a scalar variable holds exactly one value, and in upcoming chapters you'll see other types of variables, such as arrays and hashes, that can hold many values. The name of the variable stays the same throughout your program, but the value or values in that variable can change over and over again.

A scalar variable holds a single scalar value, as you'd expect. Scalar variable names begin with a dollar sign, called the *sigil*, followed by 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 all of these refer to different variables:

```
$name
$Name
$NAME

$a_very_long_variable_that_ends_in_1
$a_very_long_variable_that_ends_in_2
$A_very_long_variable_that_ends_in_2
$AVeryLongVariableThatEndsIn2
```

Perl doesn't restrict itself to ASCII for variable names, either. If you enable the `utf8` pragma, you can use a much wider range of alphabetic or numeric characters in your identifiers:

```
$résumé
$coördinate
```

Perl uses the sigils to distinguish things that are variables from anything else that you might type in the program. You don't have to know the names of all the Perl functions and operators to choose your variable name.

Furthermore, Perl uses the sigil to denote what you're doing with that variable. The `$` sigil really means "single item" or "scalar." Since a scalar variable is always a single item, it always gets the "single item" sigil. In Chapter 3, you'll see the "single item" sigil used with another type of variable, the array. This is a very important concept in Perl. The sigil isn't telling you the variable type; it's telling you how you are accessing that variable.

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. If you are using a variable for only two or three lines close together, you might name it something simple, such as `$n`. A variable you use throughout a program should probably have a more descriptive name to not only remind you what it does, but let other people know what it does. Most of your program will make sense to you because you're the one who invented it. However, someone else isn't going to know why a name like `$srly` makes sense to you.

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. `$super_bowl` is a better name than `$superbowl`, for example, as 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 misspelling?

Most variable names in our Perl programs are all lowercase, like most of the ones you'll see in this book. In a few special cases, uppercase letters are used. Using all caps (like `$ARGV`) generally indicates that there's something special about that variable.

When a variable's name has more than one word, some say `$underscores_are_cool`, while others say `$giveMeInitialCaps`. Just be consistent. You can name your variables with all uppercase, but you might end up using a special variable reserved for Perl. If you avoid all uppercase names, you won't have that problem.

NOTE

The perlvar documentation lists all of Perl's special variable names, and perlstyle has general programming style advice.

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.

Scalar Assignment

The most common operation on a scalar variable is *assignment*, which is the way to give a value to that 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 rather common. In fact, it's so common that you can write it using a convenient shorthand, as you'll see in the next section.

Compound 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 *compound assignment operator*. Nearly all binary operators that compute a value have a corresponding compound assignment form with an appended equals sign. For example, the following two lines are equivalent:

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

These are also equivalent:

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

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

Another common assignment operator is made with the string concatenation operator (`.`). This gives us an append operator (`.=`):

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

Nearly all compound 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` ."

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 give `print` a series of values separated by commas:

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

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

Perl v5.10 adds a slightly better `print` that it calls `say` . It automatically puts a newline on the end for you:

```
use v5.10;
say "The answer is ", 6 * 7, '.';
```

If you can, use `say`. In this book, we tend to stick to `print` because we want most examples to work for the people stuck on v5.8.

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 a 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 steal
$barney = 'fred ate a ' . $meal; # another way to write that
```

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

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 "
```

You'll see more about this later in this chapter when we introduce the `undef` value.

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 you're not constructing a larger string, so you don't need the interpolation step.

To put a literal 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
```

Alternatively, you could avoid using double quotes around the problematic part of the string:

```
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 considers those characters as 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 $wha
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
```

NOTE

If you need a left square bracket or a left curly brace just after a scalar variable's name, precede it with a backslash. You may also do that if the variable's name is followed by an apostrophe or a pair of colons, or you could use the curly brace method.

Creating Characters by Code Point

Sometimes you want to create strings with characters that may not appear on your keyboard, such as *é*, *å*, *α*, or *א*. How you get these characters into your program depends on your system and the editor you're using, but sometimes, instead of typing them out, it's easier to create them by their code point with the `chr()` function:

```
$alef  = chr( 0x05D0 );
$alpha = chr( hex('03B1') );
$omega = chr( 0x03C9 );
```

NOTE

We'll use "code point" throughout the book because we're assuming Unicode. In ASCII, we might have just said *ordinal value* to denote the numeric position in ASCII. To pick up anything you might have missed about Unicode, see Appendix C.

You can go the other way with the `ord()` function, which turns a character into its code point:

```
$code_point = ord( 'א' );
```

You can interpolate these into double-quoted strings just like any other variable:

```
"$alpha$omega"
```

That might be more work than interpolating them directly by putting the hexadecimal representation in `\x{}`:

```
"\x{03B1}\x{03C9}"
```

Operator Precedence and Associativity

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

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

While precedence is simple for addition and multiplication, you 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 in the perlop documentation, which we partially show in Table 2-2.

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	`|| //`
	`.. ...`

Associativity	**Operators**
right	`?:` (conditional operator)
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 problem. So be nice to your maintenance programmer: you may be that person one day.

Comparison Operators

To compare numbers, Perl has logical comparison operators that may remind you of algebra: `<` `<=` `==` `>=` `>` `!=`. Each of these returns a *true* or *false* value. You'll find out more about those return values in the next section. Some of these may be different than what you'd use in other languages. For example, `==` is used for equality, not a single `=`, because that's used for assignment. 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. So, if you get writer's block, just let the cat walk across the keyboard, and debug the result.

To compare strings, Perl has an equivalent set of string comparison operators that look like funny little words: `lt`, `le`, `eq`, `ge`, `gt`, and `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. Note that the order of characters in ASCII or Unicode is not an order that might make sense to you. You'll see how to fix that in Chapter 14.

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
```

The if Control Structure

Once you can compare two values, you'll probably want your program to make decisions based on that comparison. Like all similar languages, Perl has an `if` control structure that only executes if its condition returns a true value:

```
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";
}
```

You must have those block curly braces around the conditional code, unlike C (whether or not you know C). 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 programmer's text editor (as we show in Chapter 1), it should do most of that work for you.

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 in 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 datatype like some languages have. Instead, it uses a few simple rules:

- If the value is a number, `0` means false; all other numbers mean true.
- Otherwise, if the value is a string, the empty string (`''`) and the string `'0'` mean false; all other strings mean true.
- If the variable doesn't have a value yet, it's false.

If you need to get the opposite of any Boolean value, use the unary *not* operator, `!`. If what follows 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
}
```

Here's a handy trick. Since Perl doesn't have a separate Boolean type, the `!` has to return some scalar to represent true and false. It turns out that `1` and `0` are good enough values, so some people like to normalize their data to just those values. To do that, they double up the `!` to turn true into false into true again (or the other way around):

```
$still_true  = !! 'Fred';
$still_false = !! '0';
```

However, this idiom isn't documented to always return exactly the value `1` or `0`, and we don't think that behavior will change anytime soon.

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>`.

NOTE

The `<STDIN>` is actually a line-input operator working on the filehandle `STDIN`, but we can't tell you about that until we get to filehandles (in Chapter 5).

Each time you use `<STDIN>` in a place where Perl expects a scalar value, Perl reads the next complete text line from *standard input* (up to and including 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 for `<STDIN>` to 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>` usually 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.

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()` removes that 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: anytime 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—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 you 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 string ends with two or more newlines, `chomp()` removes only one. If there's no newline, it does nothing, and returns zero. For the most part, you don't care what `chomp()` returns.

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 += 2;
  print "count is now $count\n"; # Gives values 2 4 6 8 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.

NOTE

Eventually you'll create an infinite loop by accident. You can stop it in the same way you'd stop any other program. Often, typing Ctrl-C will stop a runaway program; check with your system's documentation to be sure.

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 zero. 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 a numeric accumulator that starts out empty. You don’t do anything with `$sum` before you use it:

```
# 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 1, so the first line inside the loop adds 1 to `$sum`. That’s like adding 1 to a variable that already holds 0 (because you’re using `undef` as if it were a number). So now it has the value `1`. After that, since it’s been initialized, addition 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 on 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 generally causes a warning.

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:

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

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

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

Exercises

See "Answers to Chapter 2 Exercises" for answers to these exercises:

1. [5] Write a program that computes the circumference of a circle with a radius of 12.5. Circumference is 2π times the radius (approximately 2 times 3.141592654). The answer you get should be about 78.5.
2. [4] Modify the program from the previous exercise to prompt for and accept a radius from the person running the program. So, if the user enters 12.5 for the radius, they should get the same number as in the previous exercise.
3. [4] Modify the program from the previous exercise so that, if the user enters a number less than zero, the reported circumference will be zero rather than negative.
4. [8] Write a program that prompts for and reads two numbers (on separate lines of input) and prints out the product of the two numbers multiplied together.

5. [8] Write a program that prompts for and reads a string and a number (on separate lines of input) and prints out the string the number of times indicated by the number on separate lines. (Hint: use the `x` operator.) If the user enters "fred" and "3," the output should be three lines, each saying "fred." If the user enters "fred" and "299792," there may be a lot of output.