2

Working with Simple Values

The essence of programming is computation – we want the computer to do some work with the input (the data we give it). Very rarely do we write programs that tell us something we already know. Even more rarely do we write programs that do nothing interesting with our data at all. So, if we're going to write programs that do more than say "hello" to us, we're going to need to know how to perform computations, or operations, on our data. The things that perform these operations are called **operators**, and the second part of this chapter will be dedicated to looking at some common operators in Perl.

Variables are another key topic we'll introduce in this chapter. Variables give us somewhere to store a value while we're doing calculations on it, allowing us to do long computations with intermediary stages. As their name suggests, they also allow us to change their contents at will. Variables are the basis for all serious programming, so we need to meet them sooner rather than later.

Finally in the chapter, we'll see one way of getting data from the user, and we'll use that to build our first 'useful' program.

Types of Data

A lot of programming jargon is about familiar words in an unfamiliar context. We've already seen a string, which was a series of characters. We could also describe that string as a **scalar literal constant**. What does that mean?

It's a **literal**, because it's something that means what it says, as opposed to a variable. A variable is more like a pigeonhole for data; the important thing is to look inside it and see what it contains. A variable, such as `$fish`, is probably not going to stand for the word 'fish' preceded by a dollar sign, it's more likely to contain 'trout', 42, or –10. A literal, on the other hand, such as the string `"Hello, world"` is the piece of paper that goes into a pigeonhole – it doesn't stand for something else. It represents literally those twelve characters.

It's also a **constant**, because it can't change. Variables, as their name implies, may change their contents, but constants are written into the text of your program once and for all, and the program can't change that. Another way of expressing this is that the data is **hard-coded** into the program. We will see later how it's almost always better to avoid hard-coding information.

By calling a variable a **scalar**, we're describing the type of data it contains. If you remember your math (and even if you don't) a **scalar** is a plain, simple, one-dimensional value. In math, the word is used to distinguish it from a vector, which is expressed as several numbers. Velocity, for example, has a pair of co-ordinates (speed and direction), and so must be a vector. In Perl, a scalar is the fundamental, basic unit of data of which there are two kinds – numbers and strings.

> *We use the term 'scalar' to distinguish it from aggregates, like* ***lists*** *or* ***hashes****, which are single entities made up of several scalars. We'll look at what we can do with these two data types and how to manipulate them in the next chapter.*

Numbers

Numbers are…well, they're numbers. Now there are two types of number that we're interested in as Perl programmers: integers and floating-point numbers. The latter we'll come to in a minute, but let's work with integers right now. **Integers** are whole numbers with no numbers after the decimal point like 42, -1, or 10. The following program prints a couple of integer literals in Perl.

```
#!/usr/bin/perl
#number1.plx
use warnings;
print 25, -4;
```

```
> perl number1.plx
25-4>
```

Well, that's what you see, but it's not exactly what we want. Our program has a bug. Fortunately, this is a pretty easy bug to understand and fix. First, we didn't tell perl to separate the numbers with a space, and second, we didn't tell it to insert a new line at the end. Let's change the program so it does that:

```
#!/usr/bin/perl
#number2.plx
use warnings;
print 25, " ", -4, "\n";
```

This will do what we were thinking of:

```
> perl number2.plx
25 -4
>
```

For very large integers, we might find it easier to split the number up. So when we write out ten million, we're likely to split up the thousands with commas, like this: 10,000,000. We can also do this in Perl, but with an underscore (_) instead of a comma. Note that this is only to help us make our code clearer – perl ignores it. Change your program to look like the following, and then save it.

```
#!/usr/bin/perl
#number3.plx
use warnings;
print 25_000_000, " ", -4, "\n";
```

Notice, that those underscores don't appear in the output:

```
> perl number3.plx
25000000 –4
>
```

As well as integers, there's another class of number – **floating-point numbers**. These contain everything else, like 0.5, -0.01333, and 1.1. Now, floating-point numbers have a big problem. Take what happens when you divide 1 by 7, you get a number that starts off 0.14285714285714… and keeps going. It's an infinite sequence, and you can't possibly write out all of it. You have to stop somewhere, and this means you lose accuracy.

We've seen that computers represent numbers internally in binary form, and this is true for fractional numbers too. 0.1 is equivalent to a 1/2, or what we would call 0.5 in decimal; 0.01 is 1/4, or 0.25 in decimal; 0.001 is 1/8, and so on. The upshot of all this is that numbers we can express perfectly accurately in decimal, such as one-fifth (0.2), cannot be accurately expressed by a computer, as its binary representation is 0.001100110011…. Because of this, you need to be careful when working with floating-point numbers. While perl does try to provide sensible looking answers whenever possible, you may get the odd occasion where you end up with a number like 24.999999999999, instead of 25, which is what you should see. There's an old programming adage that goes 'don't compare floating-point numbers solely for equality' – allow for a bit of 'fudge factor'. We'll see how this is done when we get to comparisons.

The other potential inaccuracy is that Perl, by default, only uses a set number of bits to store each of your numbers in. To see how much storage your computer allows, change your program again to this:

```
#!/usr/bin/perl
#number4.plx
use warnings;
print 25_000_000, " ", 3.14159265358979323846264338327 9, "\n";
```

Here's what happens on my computer:

```
> perl number4.plx
25000000 3.14159265358979
>
```

As you can see, what we put in is only good to 14 decimal places. Some computers may have more than that, but those that don't may emulate arbitrarily long storage with the core `Math::BigFloat` module. Integers are also limited by the computer's storage, the maximum available size for storing a single integer is typically 32 bits, or 4294967295, and everything above that gets stored as a floating-point number. There's also a `Math::BigInt` module, included in the standard Perl distribution, for allowing larger integers than this. We will see more of modules in Chapter 10.

Binary, Hexadecimal, and Octal Numbers

As we saw in the previous chapter, we can express numbers as binary, hexadecimal, or octal numbers in our programs. We can mix the various representations in our program at will.

Try it out – Number systems

Here we'll create a simple program to demonstrate how we use the various number systems. Type in the following code, and save it as `goodnums.plx`:

```
#!/usr/bin/perl
#goodnums.plx
use warnings;
print 255,         "\n";
print 0377,        "\n";
print 0b11111111, "\n";
print 0xFF,        "\n";
```

All of these are representations of the number 255, and accordingly, we get the following output:

```
> perl goodnums.plx
255
255
255
255
>
```

How It Works

When perl reads your program, it reads and understands numbers in any of the allowed number systems: `0` for octal, `0b` for binary, and `0x` for hex.

What happens, you might ask, if you specify a number in the wrong system? Well, let's try it out. Edit `goodnums.plx` to give you a new program `badnums.plx` that looks like this:

```
#!/usr/bin/perl
#badnums.plx
use warnings;
print 255,          "\n";
print 0378,        "\n";
print 0b11111112, "\n";
print 0xFG,        "\n";
```

Since octal digits only run from 0 to 7, binary digits from 0 to 1, and hex digits from 0 to F, none of the last three lines make any sense. Let's see what perl makes of it:

```
> perl badnums.plx
Illegal octal digit '8' at badnums.plx line 5, at end of line
Illegal binary digit '2' at badnums.plx line 6, at end of line
Bareword found where operator expected at badnums.plx line 7, near "0xFG"
        (Missing operator before G?)
syntax error at badnums.plx line 7, near "0xFG"
Execution of badnums.plx aborted due to compilation errors.
>
```

Now, let's match those errors up with the relevant lines:

```
Illegal octal digit '8' at badnums.plx line 5, at end of line
```

And line 5 is:

```
print 0378,       "\n";
```

As you can see, perl thought it was dealing with an octal number, but then along came an 8, which stopped it from making sense, so perl quite rightly complained. The same thing happened on the next line:

```
Illegal binary digit '2' at badnums.plx line 6, at end of line
```

And line 4 is:

```
print 0b11111112, "\n";
```

The problem with the next line is even bigger:

```
Bareword found where operator expected at badnums.plx line 7, near "0xFG"
      (Missing operator before G?)
syntax error at badnums.plx line 7, near "0xFG"
```

'What's a bareword?' I hear you asking. A **bareword** is a series of characters outside of a string that perl doesn't recognize. The word could mean a number of things, and Perl can usually understand what you mean. In this case, the bareword was 'G': perl had understood `0xF`, but couldn't see how the 'G' fitted in. We might have wanted an operator do something with it, but there was no operator there. In the end, perl gave us a 'syntax error', which is the equivalent of it giving up in disgust saying, 'How do you expect me to understand this?'

Strings

The other type of scalar available to us is the string, and we've already seen a few examples of them. In the last chapter, we met the string `"Hello, world\n"` and I mentioned that a string was a series of characters surrounded by some sort of quotation marks. Strings can contain ASCII (or Unicode) data and escape sequences such as the `\n` of our example, and there is no maximum length restriction on a string imposed by Perl. Practically speaking, there is a limit imposed by the amount of memory in your computer, but it's quite hard to hit.

Single- vs Double-Quoted Strings

The quotation marks you choose for your string are significant. So far we've only seen **double-quoted** strings, like this: `"Hello, world\n"`. There is another type of string – one which has been **single-quoted**. Predictably, they are surrounded by single quotes: `''`. The important difference is that no processing is done within single quoted strings, except on `\\` and `\'` . We'll also see later that variable names inside double-quoted strings are replaced by their contents, whereas single-quoted strings treat them as ordinary text. We call both these types of processing **interpolation**, and say that single-quoted strings are not interpolated.

Consider the following program, bearing in mind that `\t` is the escape sequence that represents a tab.

```
#!/usr/bin/perl
#quotes.plx
use warnings;
print'\tThis is a single quoted string.\n';
print "\tThis is a double quoted string.\n";
```

The double-quoted string will have its escape sequences processed, and the single-quoted string will not. The output we get is:

```
> perl quotes.plx
\tThis is a single quoted string.\n          This is a double quoted string.
>
```

What do we do if we want to have a backslash in a string? This is a common concern for Windows users, as a Windows path looks something like this: `C:\WINNT\Profiles\...`. In a double-quoted string, a backslash will start an escape sequence, which is not what we want it to do.

Well, there is, of course, more than one way to do it. We can either use a single-quoted string, as above, or we can **escape** the backslash. One principle that we'll see often in Perl, and especially when we get to regular expressions, is that we can use a backslash to turn off any special effect a character may have. For example, a full stop in a regular expression denotes 'any character'. If you escape the full stop by placing a backslash in front of it, like so `\.` you get the ordinary meaning of 'a full stop'. This operation is called escaping, or more commonly, **backwhacking**.

In this case, we want to turn off the special effect a backslash has, and so we escape it:

```
#!/usr/bin/perl
#quotes2.plx
use warnings;
print"C:\\WINNT\\Profiles\\\n";
print 'C:\WINNT\Profiles\ ', "\n";
```

This prints:

```
> perl quotes2.plx
C:\WINNT\Profiles\
C:\WINNT\Profiles\
>
```

Aha! Some of you may have got this message instead:

```
Can't find string terminator " ' " anywhere before EOF at quotes2.plx line 5.
```

The reason for this is that you have probably left out the space character in line 5 before the second single quote. Remember that `\'` tells perl to escape the single quote, and so it merrily heads off to look for the next quote, which of course is not there. Try this program to see how perl treats these special cases:

```
#!/usr/bin/perl
#aside1.plx
use warnings;
print 'ex\\ er\\' , ' ci\' se\'' , "\n";
```

The output you get this time is:

```
> perl aside1.plx
ex\ er\ ci' se'
>
```

Can you see how perl did this? Well, we simply escaped the backslashes and single quotes. It will help you to sort out what is happening if you look at each element individually. Remember, there are three arguments in this example. Don't let all the quotes confuse you.

Actually, there's an altogether sneakier way of doing it. Internally, Windows allows you to separate paths in the Unix style with a forward slash, instead of a backslash. If you're referring to directories in Perl on Windows, you may find it easier to say `C:/WINNT/Profiles/` instead. This allows you to get the variable interpolation of double-quoted strings without the 'Leaning Toothpick Syndrome' of multiple backslashes.

So much for backslashes, what about quotation marks? The trick is making sure perl knows where the end of the string is. Naturally, there's no problem with putting single quotes inside a double-quoted string, or vice versa:

```
#!/usr/bin/perl
#quotes3.plx
use warnings;
print"It's as easy as that.\n";
print '"Stop," he cried.', "\n";
```

This will produce the quotation marks in the right places:

```
> perl quotes3.plx
It's as easy as that.
"Stop," he cried.
>
```

The trick comes when we want to have double quotes inside a double-quoted string or single quotes inside a single-quoted string. As you might have guessed, though, the solution is to escape the quotes on the inside. Suppose we want to print out the following quote, including both sets of quotation marks:

```
'"Hi," said Jack. "Have you read Slashdot today?"'
```

Here's a way of doing it with a double-quoted string:

```
#!/usr/bin/perl
#quotes4.plx
use warnings;
print"'\"Hi,\" said Jack. \"Have you read Slashdot today?\"'\n";
```

Now see if you can modify this to make it a single-quoted string – don't forget that `\n` needs to go in separate double quotes to make it interpolate.

Alternative Delimiters

Of course, it would be nicer if you could select a completely different set of quotes so that there would be no ambiguity and no need to escape any quotes inside the text. The first operators we're going to meet are the **quote-like operators** that do this for us. They're written as `q//` and `qq//`, the first acting like a single-quoted string and the second, like a double-quoted string. Now instead of the above, we can write:

```
#!/usr/bin/perl
#quotes5.plx
use warnings;
print qq/'"Hi," said Jack. "Have you read Slashdot today?"'\n/;
```

That's all very well, of course, until we want a `/` in the string. Suppose we want to replace 'Slashdot' with '/.' – now we're back where we started, having to escape things again. Thankfully, Perl allows us to choose our own delimiters so we don't have to stick with `//`. Any non-alphanumeric (that is, non-alphabetic and non-numeric) character can be used as a delimiter, provided it's the same on both sides of the text. Furthermore, you can use `{}`, `[]`, `()` and `<>` as left and right delimiters. Here are a few ways of doing the above, all of which have the same effect:

```
#!/usr/bin/perl
#quotes6.plx
use warnings;
print qq|'"Hi," said Jack. "Have you read /. today?"'\n|;
print qq#'"Hi," said Jack. "Have you read /. today?"'\n#;
print qq('"Hi," said Jack. "Have you read /. today?"'\n);
print qq<'"Hi," said Jack. "Have you read /. today?"'\n>;
```

We'll see more of these alternative delimiters when we start working with regular expressions.

Here-Documents

There's one final way of specifying a string – by using a **here-document**. This idea was taken from the Unix shell, and works on any platform. Effectively, it means that you can write a large amount of text within your program, and it will be treated as a string provided it is identified correctly. Here's an example.

```
#!/usr/bin/perl
#heredoc.plx
use warnings;
print<<EOF;

This is a here-document. It starts on the line after the two arrows,
and it ends when the text following the arrows is found at the beginning
of a line, like this:

EOF
```

A here-document must start with << and then a label. The label can be anything you choose, but is traditionally `EOF` (End Of File). The label must follow directly after the arrows with no spaces between, unless the same number of spaces precedes the end marker. It ends when the label is found at the beginning of
a line. In our case, the semicolon does not form part of the label, because it marks the end of the `print` statement.

By default, a here-document works like a double-quoted string. In order for it to work like a single-quoted string, surround the label in single quotes. This will become important when variable interpolation comes into play, as we'll see later on.

Converting between Numbers and Strings

The perl interpreter treats numbers and strings on an equal footing, and where necessary, perl converts between strings, integers, and floating-point numbers behind the scenes. This means that you don't have to worry about making the conversions yourself, like you do in other languages. If you have a string literal `"0.25"`, and multiply it by four, perl treats it as a number and gives you the expected answer, `1`.

There is, however, one area where this doesn't take place. Octal, hex, and binary numbers in string literals or strings stored in variables don't get converted automatically:

```
#!/usr/bin/perl
#octhex1.plx
use warnings;
print"0x30\n";
print "030\n";
```

gives you

```
> perl octhex1.plx
0x30
030
>
```

If you ever find yourself with a string containing a hex or octal value that you need to convert into a number, you can use the `hex()` or `oct()` functions accordingly:

```
#!/usr/bin/perl
#octhex2.plx
use warnings;
print hex("0x30"), "\n";
print oct("030"), "\n";
```

This will now produce the expected answers, 48 and 24. Note that for `hex()` or `oct()`, the prefix `0x` or `0`, respectively, is not required. If you know that what you have is definitely supposed to be a hex or oct number, then `hex(30)` and `oct(30)` will produce the results above. As you can see from that, the string `"30"` and the number `30` are treated as the same.

Furthermore, these functions will stop reading when they get to a digit that doesn't make sense in that number system:

```
#!/usr/bin/perl
#octhex3.plx
use warnings;
print hex("FFG"), "\n";
print oct("178"), "\n";
```

These will stop at `FF` and `17`, respectively, and convert to 255 and 15.

What about binary numbers? Well, there's no corresponding `bin()` function, but there is actually a little trick here. If you have the correct prefix in place for any of the number systems, (`0`, `0b`, or `0x`) you can use `oct()` to convert it to decimal. For example `print oct("0b11010")` prints 26.

Operators

Now we know how to specify our strings and numbers, let's see what we can do with them. The majority of the things we'll be looking at here are numeric operators (operators that act on and produce numbers) like plus and minus, which take two numbers as 'arguments' and add or subtract them. There aren't as many string operators, but there are plenty of string functions. Perl doesn't draw a very strong distinction between functions and operators, but the main difference between the two is that operators tend to go in the middle of their arguments – for example: 2 + 2. Functions go before their arguments and have them separated by commas. Both of them take arguments, do something with them, and produce a new value. We generally say they **return** a value. Let's take a look:

Numeric Operators

The numeric operators take at least one number as an argument and return another number. Of course, because perl automatically converts between strings and numbers, the arguments may appear as string literals or come from strings in variables. We'll group these operators into three types: ordinary arithmetic operators, bitwise operators, and logic operators.

Arithmetic Operators

The arithmetic operators are those that deal with basic mathematics like adding, subtracting, multiplying, dividing, and so on. To add two numbers together, we would write something like this:

```
#!/usr/bin/perl
#arithop1.plx
use warnings;
print 69 + 118;
```

And, of course, we would see the answer 187. Subtracting numbers is easy, too, and we can subtract at the same time:

```
#!/usr/bin/perl
#arithop2.plx
use warnings;
print "21 from 25 is: ", 25 - 21, "\n";
print "4 + 13 - 7 is: ", 4 + 13 - 7, "\n";
```

```
>perl arithop2.plx
21 from 25 is: 4
4 + 13 - 7 is: 10
>
```

Our next set of operators (multiplying and dividing) is where it gets interesting. We use the `*` and `/` operators to multiply and divide, respectively.

```
#!/usr/bin/perl
#arithop3.plx
use warnings;
print"7 times 15 is ", 7 * 15, "\n";
print "249 over 3 is ", 249 / 3, "\n";
```

The fun comes when you want to multiply something and then add something, or add then divide. Here's an example of the problem:

```
#!/usr/bin/perl
#arithop4.plx
use warnings;
print 3 + 7 * 15, "\n";
```

Now this could mean one of two things: either perl must add the three and the seven and then multiply by fifteen, or it must multiply seven and fifteen first, then add. Which does Perl do? Try it and see.

So, perl should have given you 108, as it did the multiplication first. The order in which perl performs operations is called **precedence**. Multiply and divide have a higher precedence than add and subtract, and so they get performed first. We can start to draw up a table of precedence as follows:

* /
+ -

To force perl to perform an operation of lower precedence first, we need to use brackets, like so:

```
#!/usr/bin/perl
#arithop5.plx
use warnings;
print(3 + 7) * 15;
```

Unfortunately, if you run that, you'll get a warning and 10 is returned. What happened? The problem is that `print` is itself an operator as well, and the precedence of operators like `print` is highest of all.

`print` as an operator takes a list of arguments and performs an operation (printing them to the screen). It returns a 1 if it succeeds or no value if it does not. Perl calculated 3 plus 7, printed the result, and then multiplied the result of the returned value (1) by 15, throwing away the final result of 15.

To get what we actually want then, we need another set of brackets:

```
#!/usr/bin/perl
#arithop6.plx
use warnings;
print((3 + 7) * 15);
```

This now gives us the correct answer, 150, and we can put another entry in our table of precedence:

List operators
* /
+ -

Next we have the exponentiation operator, `**`, which simply raises one number to the power of another – squaring, cubing, and so on. Here's an example of some exponentiation:

```
#!/usr/bin/perl
#arithop7.plx
use warnings;
print 2**4, " ", 3**5, " ", -2**4, "\n";
```

That's 2*2*2*2, 3*3*3*3*3, and -2*-2*-2*-2. Or is it?

The output we get is:

```
>perl arithop7.plx
16 243 -16
>
```

Hmm, the first two look OK, but the last one's a bit wrong. –2 to the 4th power should be positive. Again, it's a precedence issue. Turning a number into a negative number requires an operator, the 'unary minus' operator. It's called 'unary' because unlike the ordinary minus operator, it only takes one argument. Although unary minus has a higher precedence than times and divide, it has a lower precedence than exponentiation. What's actually happening, then, is `-(2**4)` instead of `(-2)**4`. Let's put these two operators in the table as well:

List operators
**
Unary minus
* /
+ -

The last arithmetic operator is `%`, the remainder, or 'modulo' operator. This calculates the remainder when one number divides another. For example, six divides into fifteen twice, with a remainder of three, as our next program will confirm:

```
#!/usr/bin/perl
#arithop8.plx
use warnings;
print"15 divided by 6 is exactly ", 15 / 6, "\n";
print "That's a remainder of ", 15 % 6, "\n";
```

```
>perl arithop8.plx
15 divided by 6 is exactly 2.5
That's a remainder of 3
>
```

The modulo operator has the same precedence as multiply and divide.

Bitwise Operators

Those operators worked on numbers in the way we think of them. However, as we already know, computers don't see numbers the same as we do; they see them as a string of bits. These next few operators perform operations on numbers one bit at a time – that's why we call them bitwise. These aren't used quite so much in Perl as in other languages, but we'll see them when dealing with things like low-level file access.

First, let's have a look at the kind of numbers we're going to use in this section, just so we get used to them:

0 in binary is 0, but let's write it as 8 bits:	00000000
51 in binary is	00110011
85 in binary is	01010101
170 in binary is	10101010
204 in binary is	11001100
255 in binary is	11111111

Does it surprise you that 10101010 (170) is twice as much as 01010101 (85)? It shouldn't, when we multiply a number by 10 in base 10, all we do is slap a zero on the end, so 21 becomes 210. Similarly, to multiply a number by 2 in base 2, we do exactly the same.

Bitwise operators work from right to left. The rightmost bit is called the 'least significant bit', and the leftmost is called the 'most significant bit'.

The 'and' Operator

The easiest operator to fathom is called the '`and`' operator and is written `&`. This compares pairs of bits as follows:

1 and 1 gives 1

1 and 0 gives 0

0 and 1 gives 0

0 and 0 gives 0

For example, `51 & 85` looks like this:

```
51   00110011
85   01010101
17   00010001
```

Sure enough, if we ask Perl:

```
#!/usr/bin/perl
#bitop1.plx
use warnings;
print"51 ANDed with 85 gives us", 51 & 85, "\n";
```

It'll tell us the answer is 17. Notice that since we're comparing one pair of bits at a time, it doesn't really matter which way around the arguments go, `51 & 85` is exactly the same as `85 & 51`. Operators with this property are called **associative** operators.

Here's another example, look at the bits, and see what you get:

```
51    00110011
170   10101010
34    00100010
```

The 'or' Operator

As well as checking whether the first **and** the second bits are 1, we can check whether one **or** another is 1. The '**or**' operator in Perl is |, and this is how we would calculate `204 | 85`

```
204   11001100
85    01010101
221   11011101
```

Now we produce zeros only if both the bits are zero, if either or both are one, we produce a one. As a quick rule of thumb, `X & Y` will always be smaller or equal to the smallest value of `X` and `Y`, and `X | Y` will be bigger than or equal to the largest value of `X` or `Y`.

The 'exclusive or' Operator

What if you really want to know if one or the other, but not both, are set to one? For this, you need the '**exclusive or**' operator, written as the ^ operator:

```
204   11001100
170   10101010
102   01100110
```

The 'not' Operator

Finally, you can flip the number completely, and replace all the ones by zeros and vice versa. This is done with the '**not**', or ~ operator:

```
85    01010101
170   10101010
```

Let's see, however, what happens when we try this in Perl:

```
#!/usr/bin/perl
#bitop2.plx
use warnings;
print"NOT 85 is", ~85, "\n";
```

On my computer, I get:

```
NOT 85 is 4294967210
>
```

Your answer might be different, and I'll explain why in a second.

Why is it so big? Well, let's look at that number in binary to see if we can find a clue as to what's going on:

4294697210 11111111111111111111111110101010

Aha! The last part is right, but it's a lot wider than we're used to. That's because in the examples, I've only used 8 bits across, whereas my computer stores integers as 32 bits across, what's actually happened is this:

85 00000000000000000000000001010101
4294697210 11111111111111111111111110101010

If you get a much bigger number, it's because your computer represents numbers internally with 64 bits instead of 32, and Perl has been configured to take advantage of this.

Truth and Falsehood

"What is truth?" If we had asked that of a Perl programmer, we could be sure that he would have replied something like this: "Truth is anything that is not zero, an empty string, an undefined value, or an empty list."

Later, we will want to perform actions based on whether something is true or false, for example if one number is bigger than another, or, unless a problem has occurred, or, while there is data left to examine. We will use **comparison operators** to evaluate whether these things are true or false so that we can make decisions based on them.

It's customary to represent false as 0 and true as 1. This allows us to use operators very similar to those bitwise operators we've just met to combine our comparisons, to say 'if this *or* this is true', 'if this is *not* true', and so on. The idea of combining values that represent truth and falsehood is called **Boolean logic**, after George Boole, who invented the concept in 1847, and we call the operators that do the combining '**Boolean operators**'.

Comparing Numbers for Equality

The first simple comparison operator is ==. Two equals signs tells perl to 'return true if the two numeric arguments are equal.' If they're not equal, return false. Boolean values of truth and falsehood aren't very exciting to look at, but let's see them anyway:

```
#!/usr/bin/perl
#bool1.plx
use warnings;
print"Is two equal to four? ",          2 == 4, "\n";
print "OK, then, is six equal to six? ", 6 == 6, "\n";
```

This will produce:

```
>perl bool1.plx
Is two equal to four?
OK, then, is six equal to six? 1
>
```

The second line is definitely true, and as we'd expect, we get a one back from the operator. But what happened in the first line? Well, there's a special value in Perl that is conspicuous by its absence. Can you guess what it is? You might have noticed before that I mentioned "… an undefined value or an empty list." This next paragraph will help you work it out.

The undefined value isn't simply a string with nothing in it – it's nothing at all. In a very Zen-like way, a string with no characters **is** still a string. The undefined value isn't zero either, although it gets converted to zero if you use it as a number in the same way that an empty string does. The undefined value represents nothing, empty, void.

The obvious counterpart to test whether things are equal is to test whether they're not equal. The way we do this is with the `!=` operator. Note that there's only one `=` this time. We'll find out later why there had to be two before.

```
#!/usr/bin/perl
#bool2.plx
use warnings;
print"So, two isn't equal to four? ", 2 != 4, "\n";
```

```
>perl bool2.plx
So, two isn't equal to four? 1
>
```

There you have it – irrefutable proof that two actually isn't four. Good.

Comparing Numbers for Inequality

So much for equality, let's check if one thing is bigger than another. Just like in mathematics, we use the greater-than and less-than signs to do this: < and >.

```
#!/usr/bin/perl
#bool3.plx
use warnings;
print"Five is more than six? ",      5 >  6, "\n";
print "Seven is less than sixteen? ", 7 < 16, "\n";
print "Two is equal to two? ",        2 == 2, "\n";
print "One is more than one? ",       1 >  1, "\n";
print "Six is not equal to seven? ",  6 != 7, "\n";
```

The results should hopefully not be very new to you:

```
>perl bool3.plx
Five is more than six?
Seven is less than sixteen? 1
Two is equal to two? 1
One is more than one?
Six is not equal to seven? 1
>
```

Let's have a look at one last pair of comparisons. We can check greater-than-or-equal-to and less-than-or-equal-to with the >= and <= operators, respectively.

```
#!/usr/bin/perl
#bool4.plx
use warnings;
print"Seven is less than or equal to sixteen? ", 7 <= 16, "\n";
print "Two is more than or equal to two? ", 2 >= 2,  "\n";
```

As expected, perl faithfully prints out:

```
>perl bool4.plx
Seven is less than or equal to sixteen? 1
Two is more than or equal to two? 1
>
```

There's also a special operator that isn't really a Boolean comparison because it doesn't give us a true-or-false value. Instead it returns 0 if the two are equal, -1 if the right hand side is bigger, and 1 if the left-hand side is bigger. It is denoted by `<=>`. Think of it as a balance, pointing towards the lower number:

```
#!/usr/bin/perl
#bool5.plx
use warnings;
print"Compare six and nine? ",   6 <=> 9, "\n";
print "Compare seven and seven? ",7 <=> 7, "\n";
print "Compare eight and four? ", 8 <=> 4, "\n";
```

Gives us:

```
>perl bool5.plx
Compare six and nine? -1
Compare seven and seven? 0
Compare eight and four? 1
>
```

We'll see this in more detail when we look at sorting things, where we have to know whether something goes before, after, or in the same place as something else.

Boolean Operators

As well as being able to evaluate the truth and falsehood of some statements, we can also combine such statements. For example, we may want to do something if one number is bigger than another and another two numbers are the same. The combining is done in a very similar manner to the bitwise operators we saw earlier. We can ask if one value **and** another value are both true, or if one value **or** another value are true, and so on.

The operators even resemble the bitwise operators. To ask if both truth-values are true, we would use `&&` instead of `&`.

> **In many cases, `&` and the other bitwise operators will work just fine, if you are sure that the values are either one or zero. But as we know, truth is anything that is not zero, an empty string, an undefined value, or an empty list, rather than just one or zero. For example, `-2` is a true value. However, `~-2` is also a true value. When testing truths, always use the Boolean rather than the bitwise operators.**

So, to test whether six is more than three **and** twelve is more than four, we can write:

```
6 > 3 && 12 > 4
```

To test if nine is more than seven **or** eight is less than six, we use the doubled form of the | operator, ||:

```
9 > 7 || 6 > 8
```

To negate the sense of a test, however, use the slightly different operator !. This has a higher precedence than the comparison operators, so use brackets. For example, this tests whether two is not more than three,

```
!(2>3)
```

while this one tests whether !2 is more than three:

```
!2>3
```

2 is a true value. !2 is therefore a false value, the undefined value, which gets converted to zero when we do a numeric comparison. We're actually testing if zero is more than three, which has the opposite effect to what we wanted.

Instead of those forms, &&, ||, and !, we can also use the slightly easier-to-read versions, and, or, and not. There's also xor, for exclusive or (one or the other but not both are true) which doesn't have a symbolic form. However, you need to be careful about precedence again:

```
#!/usr/bin/perl
#bool6.plx
use warnings;
print"Test one: ", 6 > 3 && 3 > 4, "\n";
print "Test two: ", 6 > 3 and 3 > 4, "\n";
```

This prints, somewhat surprisingly:

```
> perl bool6.plx
Test one:
Test two: 1>
```

Well, we can tell from the position of the prompt (or least Unix users can – Windows users need to be a bit more alert) that something is amiss because the second newline did not get printed. The trouble is that and has a lower precedence than &&. What has actually happened is this:

```
print ("Test two: ", 6 > 3) and 3 > 4, "\n";
```

Now, six is more than three, so that returned 1, `print` then returned one, and the rest was irrelevant. However, we can use this fact to our advantage.

Perl uses a technique called **lazy evaluation**. As soon as it knows the answer to the question, it stops working. If you ask if *x* and *y* are both true, and it finds that *x* isn't, it doesn't need to look at *y*. No matter whether *y* is true or not, it can't make them both true, so there's no point testing. Similarly, if you ask whether *x* or *y* is true, you can stop if you find that *x* is true. Whether *y* is true or not will not affect matters at all. So, we can write something like this:

```
4 >= 2 and print "Four is more than or equal to two\n";
```

If the first test is true, perl has to check if the other side is true as well, and that means printing our message. If the first test is false, there's no need to check, so the message doesn't get printed. It's a crude way of saving time if a condition is met. We won't use that for the moment, until we've seen a less crude way to do it.

String Operators

After that lot, there are surprisingly few string operators. Actually, for the moment, we're only going to look at two.

The first one is the **concatenation operator**, which glues two strings together into one. Instead of saying:

```
print "Print ", "several ", "strings ", "here", "\n";
```

we could say:

```
print "Print " . "one ". "string " . "here" . "\n";
```

As it happens, printing several strings is slightly more efficient, but there will be times you really do need to combine strings together, especially if you're putting them into variables.

What happens if we try and join a number to a string? The number is evaluated and then converted:

```
#!/usr/bin/perl
#string1.plx
use warnings;
print"Four sevens are ". 4*7 ."\n";
```

which tells us, reassuringly, that:

```
> perl string1.plx
Four sevens are 28
>
```

The other string operator is the **repetition operator**, marked with an `x`. This repeats a string a given number of times:

```
#!/usr/bin/perl
#string2.plx
use warnings;
print "GO! "x3, "\n";
```

will print:

```
> perl string2.plx
GO! GO! GO!
>
```

We can, of course, use it in conjunction with concatenation. Its precedence is higher than the concatenation operator's, as we can easily see for ourselves:

```
#!/usr/bin/perl
#string3.plx
use warnings;
print "Ba". "na"x4 ,"\n";
```

On running this, we'll get:

```
> perl string3.plx
Bananananana
>
```

In this case, the repetition is done first ("nananana") and is then concatenated with the "Ba". The precedence of the repetition operator is the same as the arithmetic operators, so if you're working out how many times to repeat something, you're going to need brackets:

```
#!/usr/bin/perl
#string4.plx
use warnings;
print"Ba". "na"x4*3 ,"\n";
print "Ba". "na"x(4*3) ,"\n";
```

Compare:

```
>perl string4.plx
Ba0
Bananananananananananananananana
>
```

Why was the first one `Ba0`? Well, think what happened. The first thing was the repetition, giving us "nananana". Then the multiplication – What's nananana times three? When perl converts a string to a number, it takes any spaces, an optional minus sign, and then as many digits as it can from the beginning of the string, and ignores everything else. Since there were no digits here, the number value of nananana was zero.

That zero was then multiplied by three, to give zero. Finally, the zero was turned back into a string to be concatenated onto the Ba.

Try it out – Converting Strings to Numbers

You can see how other strings convert to numbers by adding zero to them:

```
#!/usr/bin/perl
#str2num.plx
use warnings;
print"12 monkeys"     + 0,  "\n";
print "Eleven to fly" + 0,  "\n";
print "UB40"          + 0,  "\n";
print "-20 10"        + 0,  "\n";
print "0x30"          + 0,  "\n";
```

You get a warning for each line saying that the strings aren't 'numeric in addition (+)', but what can be converted is. Ignoring the warnings then, here's what they come out as:

```
>perl str2num.plx
12
0
0
-20
0
>
```

How It Works

Our first string, `"12 monkeys"`, did pretty well. Perl understood the 12 and stopped after that. The next one was not handled so well – English words don't get converted to numbers. Our third string was also a non-starter, as perl only looks for a number at the beginning of the string. If something other than a number is there, it's evaluated as a zero. Similarly, perl only looks for the first number in the string. Any numbers after that are discarded. Finally, perl doesn't convert binary, hex. or octal to decimal when it's stringifying a number, so you have to use the `hex()` or `oct()` functions to do that. On our last effort, perl stopped at the `x`, returning 0. If we had an octal number, such as `030`, that would be treated as the decimal number 30.

String Comparison

As well as comparing the value of numbers, we can compare the value of strings. By this, I don't mean we convert a string to a number, although if you say something like `"12" > "30"`, perl will convert to numbers for you. What I mean is, we can compare the strings alphabetically: "Bravo" comes after "Alpha" but before "Charlie", for instance.

In fact, it's more than alphabetical order: The computer is using either ASCII or Unicode internally to represent the string and has converted it to a series of numbers in the relevant sequence. This means, for example, "Fowl" comes before "fish", because a capital F has a smaller ASCII value (70) than a lower case f (102). See Appendix F for the full ASCII table.

We can find the character's value by using the `ord()` function, which tells us where in the (ASCII) **ord**er it comes. Let's see which comes first, a `#` or a `*`?

```
#!/usr/bin/perl
#ascii.plx
use warnings;
print"A # has ASCII value ", ord("#"),"\n";
print "A * has ASCII value ", ord("*"),"\n";
```

This should say:

```
>perl ascii.plx
A # has ASCII value 35
A * has ASCII value 42
>
```

I suppose if we're only concerned with one character at a time we can compare the return values of `ord()` using the `<` and `>` operators. However, when comparing entire strings, it may get a bit tedious. If the first character of each string is the same, you would move onto the next character in each string, and then the next, and so on.

Instead, there are string comparison operators that do this all for us. Whereas the comparison operators for numbers were mathematical symbols, the operators for strings are abbreviations. To test whether one string is less than another, use `lt`. 'Greater than' becomes `gt`, 'equal to' becomes `eq`, and 'not equal' becomes `ne`. There's also `ge` and `le` for 'greater than or equal to' and 'less than and equal to'. The three-way-comparison becomes `cmp`.

Here are a few examples of these:

```
#!/usr/bin/perl
#strcomp1.plx
use warnings;
print"Which came first, the chicken or the egg? ";
print "chicken" cmp "egg", "\n";
print "Are dogs greater than cats? ";
print "dog" gt "cat", "\n";
print "Is ^ less than + ? ";
print "^" lt "+", "\n";
```

And the results:

```
>perl strcomp1.plx
Which came first, the chicken or the egg? -1
Are dogs greater than cats? 1
Is ^ less than + ?
>
```

But watch this carefully:

```
#!/usr/bin/perl
#strcomp2.plx
use warnings;
print "Test one: ", "four" eq "six", "\n";
print "Test two: ", "four" == "six", "\n";
```

```
>perl strcomp2.plx
Test one:
Test two: 1
>
```

Is the second line really claiming that four is equal to six? No, but if you compare them as numbers, they get converted to numbers. `"four"` converts to 0, and `"six"` converts to 0. The 0s are equal, so our test returns true and we get a couple of warnings telling us that they were not numbers to begin with. The moral of this story is, compare strings with string comparison operators, and compare numbers with numeric comparison operators. Otherwise, your results may not be what you anticipate.

Operators To Be Seen Later

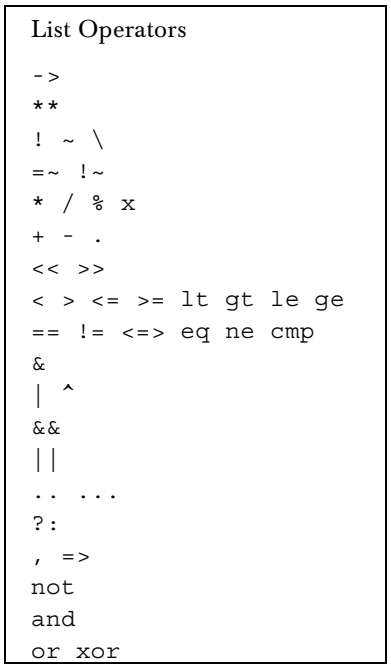
There are a few operators left that we are not going to go into in detail right now. Don't worry, we'll come across the more important ones again in time.

- The ternary hook operator looks like this: `a?b:c`. It returns `b` if `a` is true, and `c` if it is false.
- The range operators, `...` and `...`, make a range of values.
- We've seen the comma for separating arguments to functions like `print`. In fact, the comma is an operator that builds a list, and `print` works on a list of arguments. The operator `=>` works like a comma with certain additional properties.
- The `=~` and `!~` operators are used to 'apply' a regular expression to a string.
- As well as providing an escape sequence and backwhacking special characters, `\` is used to take a reference to a variable, to examine the variable itself rather than its contents.
- The `>>` and `<<` operators 'shift' a binary number right and left a given number of bits.
- `->` is hairy voodoo. We will get to it later on.

Operator Precedence

Here, finally, is a full table of precedence for all the operators we've seen so far, listed in descending order of precedence.

Remember that if you need to get things done in a different order, you will need to use brackets. Also remember that you can use brackets even when they're not strictly necessary, and you should certainly do so to help keep things readable. While perl knows full well what order to do `7+3*2/6-3+5/2&3` in, you may find it easier on yourself to spell it out, because next week you may not remember everything you have just written.



Variables

Variables! We've talked about them all the time, but what are they? As I've explained, a variable is storage for your scalars. Once you've calculated `42*7`, it's gone. If you want to know what it was, you must do the calculation again. Instead of being able to use the result as a halfway point in more complicated calculations, you've got to spell it all out in full. That's no fun.

What we need to be able to do, and what variables allow us to do, is store a scalar away and refer to it again later. As previously mentioned, there are three types of data: **scalars**, **lists**, and **hashes**. There are also three types of variable to put them in: scalar variables, arrays, and hashes. We'll look at the latter two in chapters to come and just concentrate on scalar variables for now.

A scalar variable starts with a dollar sign. Here's a simple scalar variable: `$name`. We can put certain types of data into it. Scalar variables can hold either numbers or strings and are only limited by the size of your computer's memory. To put data into our scalar, we assign the data to it, with the assignment operator =. (Incidentally, this is why numeric comparison is ==, because = was taken to mean the assignation operator.)

What we're going to do here is tell Perl that our scalar contains the string `"fred"`. Now we can get at that data by simply using the variable's name:

```
#!/usr/bin/perl
#vars1.plx
use warnings;
$name = "fred";
print "My name is ", $name, "\n";
```

Lo and behold, our computer announces to us that:

```
>perl vars1.plx
My name is fred
>
```

Now we're cut free at last from the problem of once-off data. We've got somewhere to store our data, and some way to get it back again. The next logical step is to be able to change it.

Modifying a Variable

Modifying the contents of a variable is easy, just assign something different to it. We can say:

```
#!/usr/bin/perl
#vars2.plx
use warnings;
$name = "fred";
print "My name is ",            $name, "\n";
print "It's still ",            $name, "\n";
$name = "bill";
print "Well, actually, it's ", $name, "\n";
$name = "fred";
print "No, really, it's ",      $name, "\n";
```

And watch our computer have an identity crisis:

```
>perl vars2.plx
My name is fred
It's still fred
Well, actually, it's bill
No, really, it's fred
>
```

We can also do a calculation in several stages:

```
#!/usr/bin/perl
#vars3.plx
use warnings;
$a = 6*9;
print "Six nines are ", $a, "\n";
$b = $a + 3;
print "Plus three is ", $b, "\n";
$c = $b / 3;
print "All over three is ", $c, "\n";
$d = $c + 1;
print "Add one is ", $d, "\n";
print "\nThose stages again: ", $a, " ", $b, " ", $c, " ", $d, "\n";
```

```
>perl vars3.plx
Six nines are 54
Plus three is 57
All over three is 19
Add one is 20
```

```
Those stages again: 54 57 19 20
>
```

While this works perfectly fine, it's often easier to stick with one variable and modify its value, if you don't need to know the stages you went through at the end:

```
#!/usr/bin/perl
#vars4.plx
use warnings;
$a = 6 * 9;
print "Six nines are ", $a, "\n";
$a = $a + 3;
print "Plus three is ", $a, "\n";
$a = $a / 3;
print "All over three is ", $a, "\n";
$a = $a + 1;
print "Add one is ", $a, "\n";
```

The assignment operator =, has very low precedence. This means that perl will do the calculations on the right hand side of it, including fetching the current value, before assigning the new value. To illustrate this, take a look at the sixth line of our example. perl takes the current value of `$a`, adds three to it, and then stores it back in `$a`.

Operating and Assigning at Once

Operations, like fetching a value, modifying it, or storing it, are very common, so there's a special syntax for them. Generally:

```
$a = $a <some operator> $b;
```

can be written as

```
$a <some operator>= $b;
```

For instance, we could rewrite the example above as follows:

```
#!/usr/bin/perl
#vars5.plx
use warnings;
$a = 6 * 9;
print "Six nines are ", $a, "\n";
$a += 3;
print "Plus three is ", $a, "\n";
$a /= 3;
print "All over three is ", $a, "\n";
$a += 1;
print "Add one is ", $a, "\n";
```

This works for `**=`, `*=`, `+=`, `-=`, `/=`, `.=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `&&=` and `||=`. These all have the same precedence as the assignment operator =.

Autoincrement and Autodecrement

There are two more operators, ++ and --. They add and subtract one from the variable, but their precedence is a little strange. When they precede a variable, they act before everything else. If they come afterwards, they act after everything else. Let's examine these:

Try it out – The autoincrement and autodecrement operators

Type in and run the following code:

```
#!/usr/bin/perl
#auto1.plx
use warnings;
$a=4;
$b=10;
print "Our variables are ", $a, " and ", $b, "\n";
$b=$a++;
print "After incrementing, we have ", $a, " and ", $b, "\n";
$b=++$a*2;
print "Now, we have ", $a, " and ", $b, "\n";
$a=--$b+4;
print "Finally, we have ", $a, " and ", $b, "\n";
```

You should see the following output:

```
>perl auto1.plx
Our variables are 4 and 10
After incrementing, we have 5 and 4
Now, we have 6 and 12
Finally, we have 15 and 11
>
```

How It Works

Let's work this through a piece at a time. First we set up our variables, giving the values `4` and `10` to `$a` and `$b`, respectively. :

```
$a=4;
$b=10;
print "Our variables are ", $a, " and ", $b, "\n";
```

Now in the following line, the assignment happens before the increment. So `$b` is set to `$a`'s current value, `4` and then `$a` is autoincremented, becoming 5.

```
$b=$a++;
print "After incrementing, we have ", $a, " and ", $b, "\n";
```

This time, however, the incrementing takes place first. `$a` is now 6, and `$b` is set to twice that, 12.

```
$b=++$a*2;
print "Now, we have ", $a, " and ", $b, "\n";
```

Finally, `$b` is decremented first and becomes 11. `$a` is set to `$b` plus 4, which is 15.

```
$a=--$b+4;
print "Finally, we have ", $a, " and ", $b, "\n";
```

The autoincrement operator actually does something interesting if the variable contains a string of only alphabetic characters, followed optionally by numeric characters. Instead of converting to a number, perl 'advances' the variable along the ranges a-z, A-Z, and 0-9. This is more easily understood from a few examples:

```
#!/usr/bin/perl
#auto2.plx
use warnings;
$a = "A9"; print ++$a, "\n";
$a = "bz"; print ++$a, "\n";
$a = "Zz"; print ++$a, "\n";
$a = "z9"; print ++$a, "\n";
$a = "9z"; print ++$a, "\n";
```

Should produce:

```
>perl auto2.plx
B0
ca
AAa
aa0
10
>
```

This shows that a 9 turns into a 0 and increments the next digit left. A 'z' turns into an 'a' and increments the next digit left. If there are no more digits to the left, either an 'a' or an 'A' is created, depending on the case of the current leftmost digit.

Multiple Assignments

We've said that = is an operator, but does that mean it returns a value? Well, actually it does. It returns whatever was assigned. This allows us to set up several variables at once. Here's a simple example of this (read it from right to left):

```
$d = $c = $b = $a = 1;
```

First we set `$a` to 1, and the result of this is 1. `$b` is set with that, the result of which is 1. And so it goes.

A slightly more complicated version occurs when you operate on the return value of the assignment. As usual, we need to pay attention to precedence. This won't work:

```
$b = 4 + $a = 1;
```

which is just as well, because it's horribly confusing. Perl complains that it 'Can't modify addition (+) in scalar assignment'. That is to say, it's trying to assign `1` to `4+$a`, and you can only assign to a variable, not to an addition. We say that addition is not a legal **lvalue**. It is not allowed on the left-hand side of an assignment.

If you wanted to do this, you'd have to say:

```
$b = 4 + ($a = 1);
```

This sets `$a` to 1 and `$b` to 5 as expected, but it's considered a bit messy. The reason for this is that setting various different variables with different values in one go is complicated to read and just the sort of thing that gives Perl a bad name.

Scoping

All the variables we've seen so far in our programs have been **global** variables, that is, they can be seen and changed from anywhere in the program. For the moment, that's not too much of a problem, since our programs are very small, and we can easily understand where things get assigned and used. However, when we start writing larger programs, this becomes a problem.

Why is this? Well, suppose one part of your program uses a variable, `$counter`. If another part of your program wants a counter, it can't call it `$counter` as well for fear of clobbering the old value. This becomes more of an issue when we get into **subroutines**, which are little sections of code we can temporarily call upon to accomplish something for us before returning to what we were previously doing. Currently, we'd have to make sure all the variables in our program had different names, and with a large program, that's not desirable. It would be easier to restrict the life of a variable to a certain area of the program. Let's see how this is done.

Try it out – Lexical variables

To achieve this, Perl provides another type of variable, called **lexical** variables. These are constrained to the enclosing block and all blocks inside it. If they're not currently inside a block, they are constrained to the current file. To tell perl that a variable is lexical, we say '`my $variable;`'. This creates a brand-new lexical variable for the current block and sets it to the undefined value. Here's an example:

```
#!/usr/bin/perl
#scope1.plx
use warnings;
$record = 4;
print "We're at record ", $record, "\n";

{
     my $record;
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
}

print "We're still at record ", $record, "\n";
```

And this should tell you:

```
>perl scope1.plx
We're at record 4
Inside the block, we're at record 7
We're still at record 4
>
```

How It Works

Firstly, we set our global variable `$record` to 4.

```
$record = 4;
print "We're at record ", $record, "\n";
```

Now we enter a new block and create a new lexical variable. Important! This is completely and utterly unrelated to the global variable `$record` as `my` creates a **new** lexical variable. This exists for the duration of the block only, and has the undefined value.

```
{
     my $record;
```

Next, the lexical varable is set to `7` and printed out. The global is unchanged.

```
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
```

Finally, the block ends, and the lexical ends with it. We say that it has gone 'out of scope'. The global remains, however, and so `$record` has the value 4.

```
}

print "We're still at record ", $record, "\n";
```

In order to make us think clearly about our programming, we will ask Perl to be strict about our variable use. The statement '`use strict;`' checks that, among other things, we've declared all our variables. We declare lexicals with `my`, and we can also declare globals in the same way with `our`. Here's what happens if we change our program to '`use strict`' format:

```
#!/usr/bin/perl
#scope2.plx
use strict;
use warnings;
$record = 4;
print "We're at record ", $record, "\n";

{
     my $record;
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
}

print "We're still at record ", $record, "\n";
```

Now, the global `$record` is not declared. So sure enough, perl complains about it, telling us that:

```
Global symbol "$record" requires explicit package name.
```

We'll see exactly what this means in later chapters, but for now it suffices to declare `$record` as either a global or a lexical. Normally, we'd try and avoid globals as much as possible, but let's make it a global this once:

```
#!/usr/bin/perl
#scope3.plx
use strict;
use warnings;
our $record;
$record = 4;
print "We're at record ", $record, "\n";

{
     my $record;
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
}

print "We're still at record ", $record, "\n";
```

Now perl is happy, and we get the same output as before. You should almost always start your programs with those two lines – turn on warnings, and then turn on strictness. Of course nobody's going to force you to use them, but they will help you avoid a lot of mistakes and will certainly give other people who have to look at your code more confidence in it.

Variable Names

We've not really examined yet what the rules are regarding what we can call our variables. We know that scalar variables have to start with a dollar sign, but what next? The next character must be a letter (uppercase or lowercase) or an underscore. After that, any combination of numbers, letters, and underscores is permissible, up to a total of 251 characters.

Be aware that Perl's variables are case-sensitive so `$user` is different from `$User`, and both are different from `$USER`.

The following are legal variable names: `$I_am_a_long_variable_name`, `$simple`, `$box56`, `$__hidden`, `$B1`

The following are not legal variable names: `$10c` (doesn't start with letter or underscore), `$mail-alias` (- is not allowed), `$your name` (spaces not allowed).

The Special Variable $_

There are certain variables, which Perl provides internally, that you either are not allowed to, or do not want to, overwrite. One which is allowed by the naming conventions above is `$_`, a very special variable indeed. `$_` is the 'default variable' that a lot of functions read from and write to if no other variable is given. We'll see plenty of examples of it throughout the book. For your reference, Appendix B lists all the special variables that perl uses and what they do.

Apart from the prefix, the same restrictions apply to arrays and hashes. Scalar variables are prefixed by a dollar sign ($), arrays begin with an at sign (@), and hashes begin with a percent sign (%).

Variable Interpolation

We said earlier that double-quoted strings interpolate variables. What does this mean? Well, if you mention a variable, sa,y `$name` in the middle of a double-quoted string, you get the value of the variable, rather than the actual characters. Interpolation happens for scalar variables and arrays but not for hashes. As an example, see what perl does to this:

```
#!/usr/bin/perl
#varint1.plx
use warnings;
use strict;
my $name = "fred";
print "My name is $name\n";
```

This is what comes out:

```
>perl varint1.plx
My name is fred
>
```

Perl interpolates the value of `$name` into the string. Note that this doesn't happen with single-quoted strings, just like escape sequence interpolation:

```
#!/usr/bin/perl
#varint2.plx
use warnings;
use strict;
my $name = "fred";
print 'My name is $name\n';
```

Here we get:

```
>perl varint2.plx
My name is $name\n
>
```

This doesn't just happen in things we print, it happens every time we construct a string:

```
#!/usr/bin/perl
#varint3.plx
use warnings;
use strict;
my $name = "fred";
my $salutation = "Dear $name,";
print $salutation, "\n";
```

This gives us:

```
>perl varint3.plx
Dear fred,
>
```

This has exactly the same effect as:

```
my $salutation = "Dear ". $name. ",";
```

but is more concise and easier to understand.

If you need to place text immediately after the variable, you can use braces to delimit the name of the variable. Take this example:

```
#!/usr/bin/perl
#varint4.plx
use warnings;
use strict;
my $times = 8;
print "This is the $timesth time.\n";
```

This won't work, because perl looks for a variable `$timesth` that hasn't been declared. In this case, we have to change the last line to this:

```
print "This is the ${times}th time.\n";
```

Now we get the right thing:

```
>perl varint4.plx
This is the 8th time.
>
```

Currency Converter

Let's begin to wind up this chapter with a real example – a program to convert between currencies. This is our very first version, so we won't make it do anything too complex. As we get more and more advanced, we'll be able to hone and refine it.

Try it out – Currency Converter

Open your editor, and type in the following program:

```
#!/usr/bin/perl
#currency1.plx
use warnings;
use strict;
my $yen = 180;
print "49518 Yen is ", (49_518/$yen), " pounds\n";
print "360 Yen is ",   (   360/$yen), " pounds\n";
print "30510 Yen is ", (30_510/$yen), " pounds\n";
```

Save this, and run it through perl. This is what you should see:

```
> perl currency1.plx
49518 Yen is 275.1 pounds
360 Yen is 2 pounds
30510 Yen is 169.5 pounds
>
```

How It Works

First, we start our program in the usual way:

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

Next, we declare the exchange rate to be a lexical variable and set it to `180`. (At the time I wrote this, there were roughly 180 Yen to the Pound.)

```
my $yen = 180;
```

Notice that we can declare and assign a variable at the same time. Now we do some calculations based on that exchange rate:

```
print "49518 Yen is ", (49_518/$yen), " pounds\n";
print "360 Yen is ",   (   360/$yen), " pounds\n";
print "30510 Yen is ", (30_510/$yen), " pounds\n";
```

And amazingly, the calculations come out to roughly round numbers!

Of course, this is currently of limited use, because the exchange rate fluctuate, and we might want to change some different amounts at times. To do either of these things, we need to be able to ask the user for additional data when we run the program.

Introducing <STDIN>

The way we do this is with the construct `<STDIN>`. We'll explain it in detail when we look at file handling in Chapter 6, but it reads a line from the file called **standard input**. Usually, the standard input is not really a file, but the user's keyboard. Similarly, the print function by default writes to the file called **standard output**, which is usually the user's screen.

So, in order to ask the user for a line of text, we say something like:

```
print "Please enter something interesting\n";
$comment = <STDIN>;
```

This will read one line from the user and assign the string that was read to the variable `$comment`. Let's use this to get the exchange rate from the user when the program is run.

Try it out - Currency Converter, Mark 2

Using your editor, change the file `currency1.plx` to `currency2.plx` as follows:

```
#!/usr/bin/perl
#currency2.plx
use warnings;
use strict;
print "Currency converter\n\nPlease enter the exchange rate: ";
my $yen = <STDIN>;
print "49518 Yen is ", (49_518/$yen), " pounds\n";
print "360 Yen is ",   (   360/$yen), " pounds\n";
print "30510 Yen is ", (30_510/$yen), " pounds\n";
```

Now when you run the program, you'll be asked for the exchange rate. The currency values will be calculated using the rate you entered:

```
> perl currency2.plx
Currency converter

Please enter the exchange rate: 100
49518 Yen is 495.18 pounds
360 Yen is 3.6 pounds
30510 Yen is 305.1 pounds
>
```

How It Works

This time we read the exchange rate from the user's keyboard, and perl converts the string to a number in order to perform the calculation.

So far, we haven't done any checking to make sure that the exchange rate given makes sense; This is something we'll need to think about in future.

Summary

Perl has three main data types – scalars, lists, and hashes. Lists and hashes are made up of scalars, which are in turn made up of integers, floating-point numbers, and strings. Perl converts between these three automatically, so we don't need to distinguish between them.

Double- and single-quoted strings differ in the way they process the text inside them. Single-quoted strings do little to no processing at all, whereas double-quoted strings interpolate escape sequences and variables. We can use alternative delimiters and here-documents as alternative ways of entering strings.

We can operate on these scalars in a number of ways – ordinary arithmetic, bitwise arithmetic, string manipulation, and logical comparison. We can also combine logical comparisons with Boolean operators. These operators vary in precedence, which is to say that some take effect before others, and as a result we must use brackets to enforce the precedence we want.

Scalar variables are a way of storing scalars so that we can get at them and change them. Scalar variables begin with a dollar sign (`$`) and are followed by one or more alphanumeric characters or underscores. There are two types of variables – lexical and global. Globals exist all the way through the program and so can be troublesome if we don't keep very good track of where they are being used. Lexicals have a life span of the current block, so we can use them safely without worrying about clobbering similarly named variables somewhere else in the program.

Finally, we've seen a way of getting input from the user, storing it into a variable, and acting upon it. Try the exercises that follow. They are a good indication of how much you have learned.

Exercises

1. Change the currency conversion program so that it asks for an exchange rate and three prices to convert.

2. Write a program that asks for a hexadecimal number and converts it to decimal. Then change it to convert an octal number to decimal.

3. Write a program that asks for a decimal number less than 256 and converts it to binary. (Hint: You may want to use the bitwise and operator, 8 times.)

4. Without the aid of the computer, work out the order in which each of the following expressions would be computed and their value. Put the appropriate parentheses in to reflect the normal precedence:

   - 2+6/4-3*5+1
   - 17+-3**3/2
   - 26+3^4*2
   - 4+3>=7||2&4*2<4