1

First Steps In Perl

Vitually all programming languages have certain things in common. The fundamental concepts of programming are the same, no matter what language you do them in. In this chapter, we'll investigate what you need to know before you start writing any programs at all. For instance:

- What is programming anyway? What does it mean to program?
- What happens to the program that we write?
- How do we structure programs and make them easy to understand?
- How do computers see numbers and letters?
- How do we find and eliminate errors in our programs?

Of course, we'll be looking at these from a Perl perspective, and we'll look at a couple of basic Perl programs, and see how they're constructed and what they do. At the end of this chapter, I'm going to ask you to write a couple of simple Perl programs of your own.

Programming Languages

The first question I suppose we really should ask ourselves when we're learning programming is, 'What is programming?' That may sound particularly philosophical, but the answer is easy. Programming is telling a computer what you want it to do. The only trick, then, is to make sure that the program is written in a way the computer can understand. and to do this, we need to write it in a language that it can comprehend – a programming language, such as Perl.

Writing a program does not require special skills, but it does call for a particular way of thinking. When giving instructions to humans, you take certain things for granted.

- Humans can ask questions if we don't understand instructions.
- We can break up complex tasks into manageable ones.
- We can draw parallels between the current task and ones we have completed in the past.
- Perhaps most importantly, we can learn from demonstrations and from our own mistakes.

Computers can't yet do any of these things very well – it's still much easier to explain to someone how to tie their shoelaces than it is to set the clock on the video machine.

The most important thing you need to keep in mind, though, is that you're never going to be able to express a task to a computer if you can't express it to yourself. Computer programming leaves little room for vague specifications and hand waving. If you want to write a program to, say, remove useless files from your computer, you need to be able to explain how to determine whether a file is useless or not. You need to examine and break down your own mental processes when carrying out the task for yourself: Should you delete a file that hasn't been accessed for a long time? How long, precisely? Do you delete it immediately, or do you examine it? If you examine it, how much of it? And what are you examining it for?

The first step in programming is to stop thinking in terms of 'I want a program that removes useless files,' but instead thinking 'I want a program that looks at each file on the computer in turn and deletes the file if it is over six months old and if the first five lines do not contain any of the words 'Simon', 'Perl' or 'important'. In other words, you have to specify your task precisely.

When you're able to restructure your question, you need to translate that into the programming language you're using. Unfortunately, the programming language may not have a direct equivalent for what you're trying to say. So, you have to get your meaning across using what parts of the language are available to you, and this may well mean breaking down your task yet further. For instance, there's no way of saying 'if the first five lines do not contain any of the following words' in Perl. However, there is a way of saying 'if a line contains this word', a way of saying 'get another line', and 'do this five times'. Programming is the art of putting those elements together to get them to do what you want.

So much for what you have to do – what does the computer have to do? Once we have specified the task in our programming language, the computer takes our instructions and performs them. We call this **running** or **executing** the program. Usually, we'll specify the instructions in a file, which we edit with an ordinary text editor; sometimes, if we have a small program, we can get away with typing the whole thing in at the command line. Either way, the instructions that we give to the computer – in our case, written in Perl – are collectively called the **source code** (or sometimes just **code**) to our program.

Interpreted vs. Compiled Source Code

What exactly does the computer do with our source code, then? Traditionally, there were two ways to describe what computer languages did with their code: You could say they were **compiled**, or that they were **interpreted**.

An interpreted language, such as Basic, needs another program called an **interpreter** to process the source code every time you want to run the program. This translates the source code down to a lower level for the computer's consumption as it goes along. We call the lower-level language **machine code**, because it's for machines to read, whereas **source code** is for humans. While the latter can look relatively like English, for example, (`"do_this() if $that"`), machine code looks a lot more like what you'd expect computers to be happier with, for example, `"4576616E67656C6961"`, and that's the easy-to-read version! The exact machine code produced depends on the processor of the computer and the operating system it runs, the translation would be very different for an x86 computer running Windows NT compared to a Sun or Digital computer running Unix.

A compiled language, on the other hand, such as C, uses a compiler to do all this processing one time only before the code is ever run. After that, you can run the machine code directly, without needing the

compiler any more. Because you don't need to process the source code every time you run it, compiled code will usually run faster than an interpreted equivalent. You can also give the compiled code to people who don't have a compiler themselves. This will prevent other people from reading your source code – handy if you're using a proprietary algorithm or if your code is particularly embarrassing. However, because you're distributing machine code that not all types of computers can understand, this isn't necessarily portable.

Recent languages have blurred the compiled/interpreted distinction. Java and Perl both class as 'byte-compiled' languages so they have been particularly blurry. In the case of Perl, where the interpreter (which we'll always call **perl** with a small 'p') reads your source code, it actually compiles the whole program at once. However, instead of compiling into the machine code spoken by the computer you happen to be on, it compiles into a special **virtual machine** code for a fictitious computer. Java's 'virtual machine' is quite like a normal computer's processor in terms of what it can do, and people have tried building processors that can speak the Java virtual machine code 'natively'. In comparison, Perl's virtual machine doesn't much resemble any existing computer processor and is far less likely to be built.

Once you've got this machine code, which we call **bytecode**, you can do a number of things with it. You can:

- Save it away to be run later.
- Translate it to the native machine code of your computer, and run that instead.
- Run it through a program that pretends to be the virtual machine and steps through the bytecode, and performs the appropriate actions.

We don't really do the first of these in Perl, although Java does. The 'Perl compiler' tries to do the second, but it's a very tricky job, and hasn't quite accomplished it. Normally, however, we do the third, and so after perl has finished compiling the source into bytecode, it then takes on the role of interpreter, translating the virtual machine code into real code. Hence Perl isn't strictly a compiled language or an interpreted one.

What some people will say is that Perl is a 'scripting' language, by which they probably mean an interpreted language. As we've seen, that's not actually true. However, be aware that you might hear the word 'script' where you might expect 'program'.

Libraries, Modules, and Packages

A lot of people use Perl. One consequence of this is that, unsurprisingly, a lot of Perl code has been written. In fact, a lot of the Perl code that you will ever need to write has probably already been written before. To avoid wasting time reinventing the wheel, Perl programmers package up the reusable elements of their code and distribute it, notably on CPAN – the Comprehensive Perl Archive Network – which you can find online at `http://www.perl.com/CPAN/`.

The biggest section of CPAN deals with Perl **modules**. A module is a file or a bundle of files that helps accomplish a task. There is a module for laying out text in paragraphs, one for drawing graphs, and even one for downloading and installing other modules. Your programs can use these modules and acquire their functionality. Later on, we'll devote the whole of Chapter 10 to using, downloading, and writing modules.

Closely linked to the idea of a module is the concept of a **package**, which is another way to divide up a program. By using packages, you can be sure that what you do in one section of your program does not affect another section. Whereas a module works with a file or bunch of files on your disk, a package is purely part of the source code. A single file, for instance, can contain several packages. Conversely, a package can be spread over several files. A module typically lives in its own package, to keep it distinct from the code that you write and to keep it from interfering. Again, we'll come to this later on in Chapter 10.

Every Perl installation comes with a collection of 'core modules'. The **core**, unsurprisingly, is the collective term for the files that are installed with your Perl distribution. At times, they're also referred to as the 'module library', although this could cause confusion if you intend to look back at older Perl code: 'library files' were used in Perl in versions 4 and earlier until replaced by modules in Perl 5. They are the same thing – pieces of code that you can use in your program to do a job that's been done before. However, they didn't have a package of their own, and so they put themselves in the same package as the rest of your program. It's also fairly simple to spot which file is a library and which is a module – the extension for a library file is usually `.pl`, whereas the extension for a module is `.pm`.

The result of this is that the module library contains library files as well as modules, and so it's hopelessly unclear what 'library' refers to any more. From now on, if we talk about a 'library', we're referring to the collection of files distributed with Perl, rather than Perl 4 library files; we won't be doing any work with library files (while library files have more or less been replaced by modules, they can still be useful) but will use the new-style modules instead.

Why Is Perl Such A Great Language?

Perl is in use on millions of computers, and it's one of the fastest-growing programming languages available. Why is this? We've already seen a number of reasons for this in the introduction, but I think it's worth restating them briefly here.

It's Really Easy

Perl is not a difficult language to learn. It's a language that tries to shape itself around the way humans think about problems and provides nothing contrary to their expectations. Perls' designers believe that Perl is a populist language – and not just for the mathematicians and computer scientists of this world. I know plenty of people with scientific and non-scientific backgrounds alike who successfully use Perl.

Flexibility Is Our Watchword

Perl doesn't want you to see things the way the computer does – that's not what it's for. Instead, Perl allows you to develop your personal approach to programming. It doesn't say that there's one right or wrong way to get a job done. In fact, it's quite the opposite – the Perl motto is "There's more than one way to do it", and Perl allows you to program whichever way makes most sense to you.

Perl on the Web

Perl's influence is not felt among the shell scripters of the world alone. Not only can it be used for rooting around in directories or renaming files, it also has massive importance in the world of **CGI**

scripting out on the World Wide Web. You'll find lots of Perl automating communication between servers and browsers world-wide and in more than one form. **Perlscript** is a (relatively new) derivation of Perl into a proper scripting language that can run both client- and server-side web routines, just as Javascript can. As we've said however, Perl's main function on the web is as a way to script CGI routines.

For a while, CGI was the standard way for a web server to communicate with other programs on the server, allowing the programs to do the hard work of generating content in a web page while the server dedicated itself to pass that content onto browsers as fast as it could. Of course, web pages are completely text-based and, thanks to its excellent text-handling abilities, PerlCGI set the standard for web server automation in the past. It's CGI (and Perl) that we have to thank for the wonderfully dynamic web pages we have become accustomed to on the Internet.

Later on in Chapter 12, we will explore the world of CGI in some detail, and among other things, we'll also see how to write CGI scripts using Perl. For the moment, however, let's get back to learning about Perl itself. If you would like to take a look, more information on PerlCGI and PerlScript is available at `www.fastnetltd.ndirect.co.uk/Perl/index.html`.

The Open Source Effort

Perl is free. It belongs to the world. It's Larry Wall's creation language, of course, but anyone in the world can download it, use it, copy it, and help make improvements. Roughly six hundred people are named in the changes files for assisting in the evolution from Perl 4.0 to Perl 5.0 to Perl 5.6, and that doesn't include the people who took the time to fill in helpful bug reports and help us fix the problems they had.

When I say 'anyone can help', I don't mean anyone who can understand the whole of the Perl source code. Of course, people who can knuckle down and attack the source files are useful, but equally useful work is done by the army of volunteers who offer their services as testers, documenters, proofreaders and so on. Anyone who can take the time to check the spelling or grammar of some of the core documentation can help, as can anyone who can think of a new way of explaining a concept, or anyone who can come up with a more helpful example for a function.

Perl development is done in the open, on the **perl5-porters** mailing list. The `perlbug` program, shipped with Perl, can be used to report problems to the list, but it's a good idea to check to make sure that it really is a problem and that it isn't fixed in a later or development release of Perl.

Developers Releases and Topaz

Perl is a living language, and it continues to evolve. The development happens on two fronts:

Stable releases of Perl, intended for the general public, have a version number x.y.z where z is less than 50. Currently, we're at 5.6.0; the next major stable release is going to be 5.8.0. Cases where z is more than 0 are maintenance releases issued to fix any overwhelming bugs. This happens extremely infrequently. For example, the 5.5 series had three maintenance releases in approximately one year of service.

Meanwhile between stable releases, the porters work on the development track, (where y is odd). When 5.6.0 was released, work began on 5.7.0 (the development track) to eventually become 5.8.0. Naturally, releases on the development track happen much more frequently than those on the stable track, but don't think that you should be using a development Perl to get the latest and greatest features or just because your stable version of last year seems old in comparison to the bright and shiny Perl released last week. No guarantees whatsoever are made about a development release of Perl.

Releases are coordinated by a 'patch pumpkin holder', or 'pumpking' – a quality controller who, with help from Larry, decides which contributions make the grade and when and bears the heavy responsibility of releasing a new Perl. He or she maintains the most current and official source to Perl, which they sometimes make available to the public: Gurusamy Sarathy is the current pumpkin, and keeps the very latest Perl at `ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/perl-current/`

> *Why a pumpkin? To allow people to work on various areas of Perl at the same time and to avoid two people changing the same area in different ways, one person has to take responsibility for bits of development, and all changes are to go through them. Hence, the person who has the patch pumpkin is the only person who is allowed to make the change. Chip Salzenburg explains:*
>
> *'David Croy once told me once that at a previous job, there was one tape drive and multiple systems that used it for backups. But instead of some high-tech exclusion software, they used a low-tech method to prevent multiple simultaneous backups: a stuffed pumpkin. No one was allowed to make backups unless they had the "backup pumpkin".'*

So what development happens? As well as bug fixes, the main thrust of development is to allow Perl to build more easily on a wider range of computers and to make better use of what the operating system and the hardware provides for example support for 64-bit processors. (The Perl compiler, mentioned above, is steadily getting more useful but still has a way to go.) There's also a range of optimizations to be done, to make Perl faster and more efficient, and work progresses to provide more helpful and more accurate documentation. Finally, there are a few enhancements to Perl syntax that are being debated – the 'Todo' file in the Perl source kit explains what's currently on the table.

The other line of development that's going on is the **Topaz** project, led by Chip Salzenburg, an attempt to rewrite the entirety of Perl in C++. Compared to the main development track, this is going slowly but steadily. Topaz is by no means ready for use; currently, it can merely emulate some of the Perl internals; there is no interpreter or compiler yet and probably will not be for some time. However, it's expected that Topaz development will speed up in the near future. The homepage of the Topaz project is `http://topaz.sourceforge.net/`.

Our First Perl Program

I'm assuming that by now you've got a copy of Perl installed on your machine after following the instructions in the introduction. If so, you're ready to go. If not, go back and follow the instructions. What we'll do now is set up a directory for all the examples we'll use in the rest of the book and write our first Perl program.

Here's what it'll look like:

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

print "Hello, world.\n";
```

The 'Hello World' example is the traditional incantation to the programming gods and will ensure your quick mastery of the language, so please make sure you actually do this exercise, instead of just reading about it.

Before we go any further however, a quick note on editors. Perl source code is just plain text and should be written with a plain text editor, rather than a word processor. If you're using Windows, you really will want to investigate getting hold of a good programmer's editor. Notepad may be fine for this example, despite its annoying tendency to want to rename file extensions to `.plx.txt` for you, but I wouldn't recommend its use beyond that. WordPad also renames file extensions for you, and additionally, you must remember to save as plain text, not Word format. Edit was bearable, but no longer ships with Windows versions after 95.

A decent editor will help you with bracket matching indentation and may even use different colors to point out different parts of your code. You will almost certainly want to view and edit your code in a fixed-width font. The usual Unix editors, vi, emacs, and so on are perfectly suitable, and versions ('ports') of these are available for Windows – I personally use a port of vim a vi-like editor – available at `http://shareware.cnet.com` –, when programming on Windows.

Right then, back to the code.

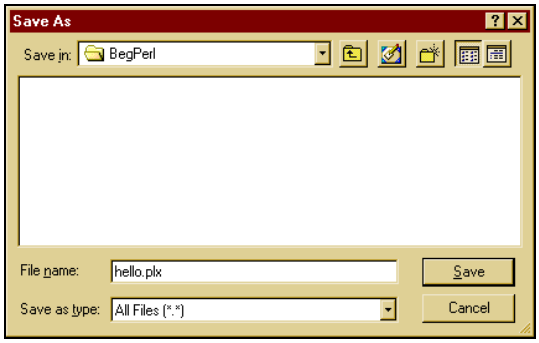
If You are a Windows User

1. Open Windows Explorer. Left click on the icon for your C: drive and choose New | Folder from the File menu.

2. Give the folder the name 'BegPerl' and press `Return`.

3. Open Notepad, which you'll find in the Programs | Accessories menu under the Start button, and type in the two lines of code as shown above.

4. Choose Save As from the File menu and change the menu option in Save as type to All Files (*.*). Find your `BegPerl` folder, and save the file as `hello1.plx`. The caption box should look this.



5. Click `Save` and then exit Notepad.

6. It's possible that Notepad will have renamed your file `hello1.plx.txt`, so in Windows Explorer, go to the `BegPerl` folder. If it has been renamed, right-click on the file and select Rename. Rename the file back to `hello1.plx`

7. The icon should change to a picture of a pearl – double click on it, and you'll see a window appear briefly and disappear before you have time to read it. This is your first lesson about clicking on Perl programs – a window will open to run them in, run them, and then close as soon as they are finished. In order to actually keep the results of our program on screen, we need to open an MS-DOS Prompt window first. So let's do that.

8. Click Start and select MS-DOS Prompt from the Programs menu. Type `cd c:\BegPerl` and press *Return*.

   Type `perl hello1.plx` – If Perl is in your path and all is well, this is what you should see on screen:

```
>perl hello1.plx
Hello, World.

>
```

Congratulations. You've successfully run your first piece of code!

If You're A Unix User

1. Open up a terminal window if you haven't already got one open, and `cd` to your home directory.

2. Type `mkdir begperl; cd begperl`

3. Open your favorite editor and edit `hello1.plx` – for example, `vi hello1.plx`

4. Confirm that your Perl distribution has been installed in `/usr/bin/perl` as the first line suggests, by typing `which perl` – if this doesn't give you anything, try `whence perl`. If the result is not `/usr/bin/perl`, be prepared to make appropriate changes.

5. Type in the two lines of code as shown above, save, and exit.

6. Run the file by typing `perl hello1.plx` – you should get similar output to the Windows users:

```
>perl hello1.plx
Hello, World.
>
```

Note that from this point on, we'll not run through these steps again. Instead, the name we've given the file will be shown as a comment on the second line of the program.

You may also have noticed that the output for hello1.plx on Windows and Unix differs in that Windows adds a silent print \n to all its perl programs. From now on, we'll only print the Unix output that is more strict. Windows users please be aware of this.

How It Works

So, all being well, your Perl program has greeted the light of day. Let's see how it was done, by going through it a line at a time. The first line is:

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

Now normally, Perl treats a line starting with # as a comment and ignores it. However, the # and ! characters together at the start of the first line tell Unix how the file should be run. In this case the file should be passed to the perl interpreter, which lives in `/usr/bin/perl`.

Perl also reads this line, regardless of whether you are on Unix, Windows, or any other system. This is done to see if there are any special options it should turn on. In this case, `-w` is present, and it instructs perl to turn on additional warning reporting. Using this flag, or its alternative, is a very good habit to get into, and we shall see why in just a moment. But first, let's have a look at the second line of our program:

```
print "Hello, world.\n";
```

The `print` function tells perl to display the given text without the quotation marks. The text inside the quotes is not interpreted as code (except for some 'special cases') and is called a **string**. As we'll see later, strings start and end with some sort of quotation mark. The `\n` at the end of the quote is one of these 'special cases' – it's a type of escape sequence, which stands for 'new line'. This instructs perl to finish the current line and take the prompt to the start of a new one.

You may be wondering why `-w` is so helpful. Well, suppose we altered our program to demonstrate this and made two mistakes by leaving out `-w` and by typing `printx` instead of `print`. Then `hello1.plx` would look like this:

```
#!/usr/bin/perl

printx "Hello, world.\n";
```

Remember to save these changes in `Hello2.plx` before exiting your file. Now let's get back to the command prompt, and type:

```
>perl Hello2.plx
```

Instead of getting the expected

```
Hello, world.
>
```

the output we get has a plethora of rather-nasty looking statements like this:

```
String found where operator expected at hello.plx line 2, near "printx "hello, world. \n""
(Do you need to predeclare printx?)
syntax error at hello.plx line 2, near "printx "Hello, world. \n""
Execution of Hello.plx aborted due to compilation errors.
>
```

If we now correct one of our mistakes by including `-w` in our program, then `Hello2.plx` looks like this:

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

printx "Hello, world. \n";
```

Once we have saved this new change into the program, we can run it again. The output that we get now contains a warning as well as the error message, so the screen looks like this:

```
>perl hello2.plx

Unquoted string "printx" may clash with future reserved word at hello2.plx line 3.
String found where operator expected at hello.plx line 2, near "printx "hello, world. \n""
(Do you need to predeclare printx?)
Syntax error at hello2.plx line 2, near "printx "Hello, world. \n""
Execution of Hello2.plx aborted due to compilation errors.
```

On the surface of things, it may seem that we have just given ourselves more nasty-looking lines to deal with. But bear in mind that the first line is now a **warning** message and is informing us that perl has picked something up that may (or may not) cause problems later on in our program. Don't worry if you don't understand everything in the error message at the moment, just so long as you are beginning to see the usefulness of having an early-warning system in place.

For versions of Perl 5.6.x and higher, the `-w` switch *should be replaced with a* `use warnings` directive, which follows **after** the shebang line. Although `-w` will still be recognized by perl, it has been deprecated, and for arguments sake we will assume from now on that you have Perl 5.6.x or higher. The resulting "en vogue" (and correct) version of `hello.plx` then, will look like this:

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

print "Hello, world. \n";
```

Program Structure

One of the things we want to develop throughout this book is a sense of good programming practice. Obviously this will not only benefit you while using Perl, but in almost every other programming language, too. The most fundamental notion is how to structure and lay out the code in your source files. By keeping this tidy and easy to understand, you'll make your own life as a programmer easier.

Documenting Your Programs

As we saw earlier, a line starting with a sharp (#) is treated as a comment and ignored. This allows you to provide comments on what your program is doing, something that'll become extremely useful to you when working on long programs or when someone else is looking over your code. For instance, you could make it quite clear what the program above was doing by saying something like this:

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

# Print a short message
print "Hello, world.\n";
```

Actually, this isn't the whole story. A line may contain some Perl code, and be followed by a comment. This means that we can document our program 'inline' like this:

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

print "Hello, world.\n"; # Print a short message
```

When we come to write more advanced programs, we'll take a look at some good and bad commenting practice.

Keywords

There are certain instructions that perl recognizes and understands. The word `print` above was one such example. On seeing `print`, perl knew it had to print out to the screen whatever followed in quotes. Words that perl is already aware of are called **keywords**, and they come in several classes. `print` is one example of the class called **functions.** These are the verbs of a programming language, and they tell perl what to do. There are also control keywords, such as `if` and `else`. These are used in context like this:

```
if  Condition;
do this;

else
do this;
```

It's a good idea to respect keywords and not reuse them as names. For example, a little later on we'll learn that you can create and name a variable, and that calling your variable `$print` is perfectly allowable. The problem with this is that it leads to confusing and uninformative statements like `print $print`. It is always a good idea to give a variable a meaningful name, one that relates to its content in a logical manner. For example `$my_name`, `$telephone_number`, `@shopping_list`, and so on, rather than `$a`, `$b` and `%c`.

Statements and Statement Blocks

If functions are the verbs of Perl, then **statements** are the sentences. Instead of a full stop, a statement in Perl usually ends with a semicolon, as we saw above:

```
print "Hello, world.\n";
```

To print something again, we can add another statement:

```
print "Hello, world.\n";
print "Goodbye, world.\n";
```

There are times when you can get away without adding the semicolon, such as when it's absolutely clear to perl that the statement has finished. However, it is good practice to put a semicolon at the end of each statement. For example, you can miss out the final semicolon in the example above, without causing a problem. Missing out the first would be incorrect.

We can also group together a bunch of statements into a **block** – which is a bit like a paragraph – by surrounding them with braces: {…}. We'll see later how blocks are used to specify a set of statements that must happen at a given time and also how they are used to limit the effects of a statement. Here's an example of a block:

```
{
     print "This is";
     print "a block";
     print "of statements.\n";
}
```

Do you notice how I've used indentation to separate the block from its surroundings? This is because, unlike paragraphs, you can put blocks inside of blocks, which makes it easier to see on what level things are happening. This:

```
print "Top level\n";
{
     print "Second level\n";
     {
          print "Third level\n";
     }
     print "Where are we?";
}
```

is easier to follow than this:

```
print "Top level\n";
{
print "Second level\n";
{
print "Third level\n";
}
print "Where are we?";
}
```

As well as braces to mark out the territory of a block of statements, you can use parentheses to mark out what you're giving a function. We call the set of things you give to a function the **arguments**, and we say that we **pass** the arguments to the function. For instance, you can pass a number of arguments to the `print` function by separating them with commas:

```
print "here ", "we ", "print ", "several ", "strings.\n";
```

The `print` function happily takes as many arguments as it can, and it gives us the expected answer:

```
here we print several strings.
```

Surrounding the arguments with brackets clears things up a bit:

```
print ("here ", "we ", "print ", "several ", "strings.\n");
```

We can also limit the amount of arguments we pass by moving the brackets:

```
print ("here ", "we ", "print "), "several ", "strings.\n";
```

We only pass three arguments, so they're the ones that get printed:

```
here we print
```

What happens to the others? Well, we didn't give perl instructions, so nothing happens.

In the cases where semicolons or brackets are optional, the important thing to do is to use your judgment. Sometimes code will look perfectly clear without the brackets, but when you've got a complicated statement and you need to be sure of which arguments belong to which function, putting in the brackets can clarify your work. Always aim to help the readers of your code, and remember that these reader will more than likely include you.

ASCII and Unicode

Computers are, effectively, lumps of sand and metal. They don't know much about the world. They don't understand words or symbols or letters. They do, however, know how to count. As far as a computer is concerned, everything is a number, and every character, albeit a letter or a symbol, is represented by a number in a sequence. This is called a 'character set', and the character set that computers predominantly use these days is called the 'ASCII' sequence. If you're interested, you can find the complete ASCII character set in Appendix F for reference.

The ASCII sequence consists of 256 characters, running from character number 0 (all computers, and plenty of computer users, start counting from zero) to character number 255. The letter 'E', for instance, is number 69 in the sequence, and a plus sign (+) is number 43. 255 is a key number for computers and computer programmers alike, because it's the largest number you can store in one 'byte'.

The big problem with ASCII is that it's American. Well, that's not entirely the problem; the real reason is that it's not particularly useful for people who don't use the Roman alphabet. What used to happen was that particular languages would stick their own alphabet in the upper range of the sequence, between 128 and 255. Of course, we then ended up with plenty of variants that weren't quite ASCII, and the whole point of standardization was lost.

Worse still, if you've got a language like Chinese or Japanese that has hundreds or thousands of characters, then you really can't fit them into a mere 256. This meant that programmers had to forget about ASCII altogether and build their own systems using pairs of numbers to refer to one character.

To fix this, **Unicode** was developed by a number of computer companies, standards organizations, and bibliographic interests. It is currently maintained and developed by the Unicode Consortium, an organization in California. They have also produced a couple of new character sets, UTF8 and UTF16. UTF8 uses two bytes instead of one, so it can contain 65536 characters, which is enough for most people. You can learn more about Unicode at `http://www.unicode.org/`

Perl 5.6 introduces Unicode support. Previously, you could print any data that you were capable of producing in your editor or from external sources. However, the functions to translate between lower and upper case wouldn't necessarily work with Greek letters without a lot of support from your operating system. Now, if you have Unicode data, you can consider a single Japanese *kana* to be one character instead of two. So, if you use a Unicode editor for your programming:

- You can write your variable names in your native alphabet.
- You can match certain classes of symbol or character regardless of language, while processing data.

Escape Sequences

So, UTF8 gives us 65536 characters, and ASCII gives us 256 characters, but on the average keyboard, there only a hundred or so keys. Even using the shift keys, there will still be some characters that you aren't going to be able to type. There'll also be some things that you don't want to stick in the middle of your program, because they would make it messy or confusing. However, you'll want to refer to some of these characters in strings that you output. Perl provides us with mechanisms called 'escape sequences' as an alternative way of getting to them. We've already seen the use of `\n` to start a new line. Here are the more common escape sequences:

Escape Sequence	Meaning
`\t`	Tab
`\n`	Start a new line (Usually called 'newline')
`\b`	Back up one character ('backspace')
`\a`	Alarm (Rings the system bell)
`\x{1F18}`	Unicode character

In the last example, 1F18 is a hexadecimal number (see 'Number Systems' just below) referring to a character in the Unicode character set, which runs from 0000-FFFF. As another example, `\x{2620}` is the Unicode character for a skull-and-crossbones!

White Space

White space is the name we give to tabs, spaces, and new lines. Perl is very flexible about where you put white space in your program. We have already seen how we're free to use indentation to help show the structure of blocks. You don't need to use any white space at all, if you don't want to. If you prefer, your programs can all look like this:

```
print"Top level\n";{print"Second level\n";{print"Third level\n";}print"Where are
we?";}
```

Personally, though, I'd call that a bad idea. White space is another tool we have to make our programs more understandable. Let's use it as such.

Number Systems

If you thought the way computers see characters is complicated, we have a surprise for you.

The way most humans count is using the decimal system, or what we call base 10; we write 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and then when we get to 10, we carry 1 in the 10s column and start from 0 again. Then when the 10s column gets to 9 and the 1s column gets to 9, we carry 1 in the 100s column and start again. Why 10? We used to think it's because we have 10 fingers, but then we discovered that the Babylonians counted up to 60, which stopped that theory.

On the other hand, computers count by registering whether or not electricity flows in a certain part of the circuit. For simplicity's sake, we'll call a flow of electricity a 1, and no flow a 0. So, we start off with 0, no flow. Then we get a flow, which represents 1. That's as much as we can do with that part of the circuit: 0 or 1, on or off. Instead of base 10, the decimal system, this is **base 2**, the **binary system**. In the binary system, one digit represents one unit of information: one **bi**nary digi**t**, or **bit**.

When we join two parts of the circuit together, things get more interesting. Look at them both in a row, when they are both off, the counter reads 00. Then one comes on, so we get 01. Then what? Well, humans get to 9 and have to carry one to the next column, but computers only get to 1. The next number, number two, is represented as 10. Then 11. And we need some more of our circuit. Number four is 100, 5 is 101, and so ad infinitum. If we got used to it, and we used the binary system naturally, we could count up to 1023 on our fingers.

This may sound like an abnormal way to count, but even stranger, counting mechanisms are all around us. As I write this, it's 7:59pm. In one minute, it'll be 8:00pm, which seems unremarkable. But that's a base 60 system. In fact, it's worse than that – time doesn't stay in base 60, because hours carry at 24 instead of 60. Anyone who's used the Imperial measurement system, a Chinese abacus, or pounds, shillings, and pence knows the full horror of mixed base systems, which are far more complicated than what we're dealing with here.

As well as binary, there are two more important sequences we need to know about when talking to computers. We don't often get to deal with binary directly, but the following two sequences have a logical relationship to base 2 counting. The first is **octal, base 8**.

Eight is an important number in computing. Bits are organized in groups of eight to form **byte**s, giving you the range of 0 to 255 that we saw earlier with ASCII. Each ASCII character can be represented by one byte. As we said in the paragraph before, octal is one way of counting bits – it has, however, fallen out of fashion these days. Octal numbers all start with 0, (that's a zero, not an oh) so we know they're octal and proceed as you'd expect: 00, 01, 02, 03, 04, 05, 06, 07, carry one, 010, 011, 012…017, carry one, 020 and so on. Perl recognizes octal numbers if you're certain to put that zero in front, like this:

```
print 01101;
```

prints out the decimal number:

```
577
```

The second is called the **hexadecimal** system, as mentioned above. Of course, programmers are lazy, so they just call it **hex**. (They like the wizard image.)

Decimal is base 10, and hexagons have six sides, so this system is base 16. As you might have guessed from the number 1F18 above, digits above 9 are represented by letters, so A is 10, B is 11, and so on, all the way through to F which is 15. We then carry one and start with 10 (which, in decimal, is 16) all the way up through 19, 1A, 1B, 1C, 1D, 1E, 1F, and carry one again to get 20 (which in decimal is 32). The magic number 255, the maximum number we can store in one byte, is FF. Two bytes next to each other can get you up to FFFF, better known as 65535. We met 65535 as the highest number in the Unicode character set, and you guessed it, a Unicode character can be stored as a pair of bytes.

To get perl to recognize hex, place `0x` in front of the digits so that:

```
print 0xBEEF;
```

gives the answer:

```
48879
```

The Perl Debugger

One thing you'll notice about programming is that you'll make mistakes; mistakes in programs are called **bugs**. Bugs are almost entirely unavoidable, and creating bugs does not mean you're a bad programmer. Windows 2000 allegedly shipped with 65,000 bugs (but then that's a special case) and even the greatest programmers in the world have problems with bugs. Donald Knuth's typesetting software TeX has been in use for 18 years, and bugs were still found until a couple of years ago.

While we will be showing you ways to avoid getting bugs in your program, Perl provides you with a tool to help find and trace the causes of bugs. Naturally, any tool for getting rid of bugs in your program is called a 'debugger'. Mundanely enough, the corresponding tool for putting bugs into your program is called a 'programmer'.

Summary

We've started on the road to programming in Perl, and programming in general. We've seen our first piece of Perl code, and hopefully, you've had it running. If you haven't, please do get through it and all the examples to come; trying everything yourself is the best way to learn.

Programming is basically telling a computer what to do in a language it comprehends. It's about breaking down problems or ideas into byte-sized chunks (as it were) and examines the task at hand in order to communicate them clearly to the machine.

Thankfully, Perl is a language that allows us a certain degree of freedom in our expression, and so long as we work within the bounds of the language, it won't enforce any particular method of expression on us. Of course, it may judge what we're saying to be wrong, because we're not speaking the language correctly, and that's how the majority of bugs are born. Generally though, if a program does what we want, that's enough - There's More Than One Way To Do It.

We've also seen a few ways of making it easy for ourselves to spot potential problems, and we know there are tools that can help us if we need it. We have examined a little bit of what goes on inside a computer, how it sees numbers, and how it sees characters, as well as what it does to our programs when and as it executes them.

I'm now going to ask you to write a simple program for yourself, nothing strenuous, and nothing harder than we've already seen. But it's important that you take that psychological step into programming right now.

Exercises

1. Look through the documentation installed with your Perl distribution.

2. Create a program newline.plx containing print "Hi Mum.\nThis is my second program. \n". Run this and then to replace \n with a space or an Enter and compare the results.

3. Download the code for this book from the wrox website at http://www.wrox.com.

4. Have a look around the Perl homepage at www.perl.com and at our Beginning_Perl mailing list at http://p2p.wrox.com.