HOW PROGRAMMING WORKS

PROGRAMMING PHILOSOPHY, DATA STRUCTURES & ALGORITHMS FOR NON-PROGRAMMERS

GUY W. LECKY-THOMPSON, BSC.

Preface

It feels odd writing something at the beginning of a book that is actually being written at the end of a long writing process, but in crafting this volume it became apparent that to keep it broadly accessible and interesting, a few concessions to practicality had to be made.

So, to prevent disappointment, there's a few items that we need to get out of the way before you delve in. And, since this bit is usually part of your decision process in buying a book, consider this also a fair warning as to what it contains!

1. This book scratches the surface of 3 major families of programming languages;
2. Do not read this book with any expectations of using it as a briefing text for an interview at, say, Google or Microsoft;
3. You **will** be able to pick up a piece of code and understand it in almost any language except *functional languages*, and COBOL;
4. This book is **not** an in-depth guide to the art of program design, but we do touch on some best practices;
5. Nor is this book an in-depth guide to any one of the languages we use as examples.

Use this book to understand programming and programming languages, and as a precursor to deciding whether programming is for you. Use it as a way to "get into" programming, and to have a handle on what your programmers are doing.

It's an overview of programming through the eyes of someone who has done a lot of trivial programming in a variety of languages, and enjoyed every minute!

Introduction

Welcome to the wonderful world of computer programming. I'd like to start with a quote from Douglas Adams' book *Mostly Harmless*:

> "A computer terminal … is an interface where the mind and body can connect with the Universe and move bits of it about."

By extension, computer programming is merely creating a representation of an existing or imagined Universe, and then moving bits of it about.

I was once with a group of programmers, talking about video game design and development, when one leaned forward and said: "Essentially, all games programming boils down to is database design."

What he meant was that all the pretty bits have very little to do do with programming; all that graphical stuff that makes a game look nice is graphic design work. The programmers move data around, which eventually gets represented on the screen as pretty graphics, but under the hood it's all about data structures and algorithms.

There are a set of concepts that you will learn in this book that are common across all languages, and all revolve around data manipulation in one form or another.

It's actually a lot more fun than it sounds!

Who Are You?

I'm a Computer Studies graduate, professional programmer, and author of a collection of books about games design, programming, and software engineering.

Here's the thing: although I'm immensely proud of having written those books (even if the Amazon reviews don't reflect all that well on them), I don't think that I did the topics justice first time around.

This may have been for a number of reasons, but I believe that the principal reason was that I was trying to do too much with each book.

I love to teach. I enjoy coaching; helping others to understand complex topics and then achieve competence within them.

The trouble is, there's always a point where my enthusiasm overtakes my ability to communicate, and we have to take a few steps back.

This book is me taking a few steps back to explain programming properly, this time round.

Why Did You Write This Book?

Now, I could have stuck these ideas up on a blog. There are some pretty good programming blogs out there; but they all assume too much when it comes to basic knowledge.

Equally, I could have set up an online course to teach these concepts; and who knows, maybe that will be an extension of this book.

However, I'm a firm believer that just as some people learn better from the structure of a proper book -- be it electronic or paper -- some people write better if they're forced into the discipline of a proper book.

There's a lot to learn, even in generic terms, and I just feel that it is better expressed in book form.

Every single day, people think to themselves “I wish I knew how to program”, then try to find a book to teach them.

Each book tends to cover a specific language, or a specific platform, and there really aren’t any entry-level programming books that aren’t aimed at computer studies undergraduates.

That’s why this book exists. To fill that gap. To help expand minds, and hopefully even entertain a little bit along the way.

Who is This Book For?

This is a book for beginners.

It’s written for people who may have heard about some of the ideas and concepts; and who want to learn how to read, and even write, computer programs.

Computer programs come in many forms these days. The most common for everyday computer users is probably JavaScript. After all, it’s present on most web pages, sitting there, behind the scenes, modifying what visitors are seeing.

If you’ve ever seen a web page address (URL) that ends in “.php”, then the page you’ll see has been created by a program written in a language called PHP.

I draw the line at calling HTML, XHTML and XML ‘programming’ languages, however, as they deal principally with data definition and presentation. So, if you just want to learn how to write web pages in HTML, then this book *isn’t* for you.

However, there are some concepts that are useful even for web page designers, especially those who want to extend their static code with JavaScript!

Language Families

This book isn't really aimed at teaching a specific language.

Rather, I will talk at length about different families of language. Like all families, they have one or two common ancestors. As far as I can make out -- and this is from an academic background -- the current crop of languages have C, Pascal or BASIC as their ancestors.

But what *are* C, Pascal and BASIC?

These are languages that were created to give humans a more intuitive and natural way to instruct computers.

They are needed because computers don't understand human languages. In the early days of programming, humans had to program computers using the lowest form of communication available, something called assembler.

Assembler was so close to machine code that it didn't need a powerful computer to turn the instructions into something a human could read into something a machine could read, and understand.

As computers became more powerful, and as our understanding of the new field of computer languages, compiler design, and as our requirements for ever more complex applications grew, languages started to spring up that crossed the gap between the abstract and the concrete.

There are several families of languages, and this book concentrates more or less exclusively on the so-called procedural programming languages. That is, programs that are evaluated, and executed, in a top-down fashion.

More or less.

Within this family, there are several individual strains of language, and I tend to view all languages as fitting into one of three of these strains: those based on a language called C, those based on Pascal, and those that are derived from BASIC.

If you were to put these on a scale of verbosity, user-friendliness, and resemblance to natural language, machine code would be way over on the left hand side, next to zero, and BASIC would be nearer the right hand side.

Somewhere to the right of machine code, there is C, and somewhere to the left of BASIC, there is Pascal.

There's not a lot of middle ground: languages tend to go from a mixture of short mnemonics and an oddly compact and elegant structure, to something that is unnaturally close to natural language without much in between.

We'll do some examples later on.

Pseudocode & Other Languages

Some of this book is expressed in terms of something called *pseudocode*, which isn't a language in particular, but is designed to mimic the way that code is different to natural language and expose the features and innately elegant application of computer programming to problem solving.

In short, on our scale of understandability, pseudocode is somewhere in the middle. Fancy that.

Code (C, PHP, BASIC or pseudocode) will be in a fixed font, like this:

```
// This is an example of a comment
```

When I do deliver actual examples of real code it will be presented and formatted according to the rules of the following three language families:

- BASIC languages, including VisualBasic;
- Pascal languages, including Modula-2;
- C style languages, such as C/C++, Javascript and PHP.

My aim in doing so is to give you a flavour of what code looks like, so that it doesn't surprise you too much when you first confront it.

With that philosophy in mind, the first thing we need to look at is how programs are structured, and how programming languages are constructed to help us implement our ideas in an abstract environment.

Chapter 1 Code Structure

Notwithstanding all of the above, there are some aspects of programming that are shared between languages, and worth taking note of.

Just in case readers haven't met them before.

Brackets, Parentheses & Braces

By convention, I'm sticking with the mathematical definition of *parentheses* as an expression that is to be evaluated, and evaluated first. I will not expect readers to second guess parameter evaluation order, either, because that always annoyed me -- it's not like electrons are in short supply.

So, rather than writing...

```
3 x 2 + 4 / 6 - 1
```

...and expecting you to work out which bit gets evaluated first, I'll put parentheses in. Mainly because it's just possible that I'll get it wrong, for example:

```
(3 x ((2 + 4) / 6)) - 1
```

I'll leave it to the mathematicians to work out which of the above parentheses is in the wrong (or right) place. Meanwhile...

Brackets, in the context of this book, shall mean square brackets:

[and]

What we use them for is often something called indexing, but we'll come to that later. For now, just imagine that we wanted to express, within a book, that a word was in the index, we might write:

> ...imagine that we wanted to express, within a [book] that a word...

By writing [book], I'm telling the reader that, in the index, there's an entry [book] with some information associated with it. The index will likely contain all manner of entries, like [apple] and [zebra] which can all be referenced.

Computer programming has a similar mechanism, as we'll see later on.

The final of these three sets of containing characters is the *brace*, or, in plain English (UK), curly bracket. They look like this:

> { and }

Again, these things likely have mathematical and/or linguistic meaning, but in programming they group related statements together, as something I like to call a code block.

Defining Code Blocks

This is the first real programming concept that we're going to look at, so I'm going to do it reasonably slowly. A code block, in the scope of this book, is a collection of statements that share a common purpose and have the same *scope*.

They can be parts of a definition, or executable code statements.

A program is the biggest kind of code block. It contains everything, and subject to a few rules relating to how its constituent parts are evaluated, everything within it is *in scope*.

Most programs have a section that contains some definitions of things that the code will use to hold, process, or pass on information (which we'll call *data*).

Thinking back to our book index example, we might decide that the best way of defining each entry so that the computer can process them is as a kind of record, like one might find in a card file, or on a Rolodex.

Each card might have a tab at the top where the indexed word is written, followed by some text that gives a list of page numbers where the word appears in the book.

As far as the computer is concerned, this data has no meaning.

It's up to us, as programmers, to tell the computer how we wish the data to be organised; in this case we decide to split each record into two parts -- the word, and its list of references.

So, within the code block that defines such an index entry, the word and the list of references are in scope. That's an example of a definition code block.

Execution code blocks, as I like to think of them, contain lines of code that we want the computer to actually evaluate (execute).

Let's assume we want to leave some instructions for the reader on how to use the index. Now most of us wouldn't need instructions, but if you think back to when you first discovered the context of an index, there were some things that needed a bit of explanation:

- Why are some pages i, ii, iii and others 1, 2, 3?
- What's the ordering of words, and why are there sometimes sub-lists?
- Why is a specific word not in the index?

Some of these questions have an impact on the way that the index is used, and as such, need breaking down into steps which lead the reader to the right place in the book.

If you have a whole stack of these index cards, there are many ways that you can look through them to get to the right one, for example. We will look at searching and sorting later on in the Data Structures & Algorithms chapter of the book, but for now it's just important to remember that manipulating the definition blocks (data) requires execution blocks (code).

Some are named blocks, some are parts of other programming constructs, such as decision statements, or repetition statements, but all need to be delimited (contained) in some way.

Some languages, such as Pascal, do this in a way that is close to natural language:

```
BEGIN
  // Code
END
```

Other languages, such as those derived from C, use braces:

```
{
  // Code
}
```

For the sake of clarity, pseudocode tends to be a bit more verbose, so we'll stick to as close to natural language as possible.

Top-Down Execution

Most code blocks are evaluated in a top-down fashion, in procedural programming; at least within code blocks.

However, sometimes control is passed from one code block to another, or a code block is executed several times in succession (repetition, or looping), or selected code blocks are ignored as the result of evaluating a piece of data.

We refer to these modifications as changing the *flow* of a program, and flow control is very useful in programming. Occasionally, it goes awry, and can become a headache to test, and rectify, but that's all part of the fun.

Global Variables & Scope

The final part of the discussion of code blocks refers to the scope of access to information: generally speaking a code block only has access to the information that has been declared as being within its scope.

If information is available to all code blocks in a program -- the program in this case being the largest code block possible -- then we call that a *global variable*.

PHP programmers will also come across something known as a *superglobal*, which is available across programs (global variables being restricted in scope to the program currently being evaluated.)

There is some discussion as to whether using global variables is good or bad programming practice. The community seems to generally agree that globals are often a very convenient, if supremely lazy way of organising data, with hidden dangers that can make finding errors difficult.

(The reason is that every code block that is in scope at some point has access to all global variables and subsequently can change their values.)

We've also snuck in a new word to add to you growing lexicon of programmer-speak: *variable*.

A variable is just a pigeon hole, that has a name, and can store data whose value can change. At one moment, it might be a 1, but then we go and change it to a 7; or what was once a letter A becomes a Z, and so on.

We'll look in quite some detail at how to define and store data in a later chapter, but for now this slightly loose definition will suffice.

Code Organisation

Different programming languages have different names for the various aspects of code organisation that they support. Just so we're clear on this point: a computer program isn't (necessarily) a big text document with every aspect of its operation defined in one place.

Usually, a program is made up of code that you write, code that you borrow from other people, and code that is usually sitting there on your computer, and has been supplied with a specific function in mind.

Anything that isn't explicitly created by the programmer is usually referred to as an external library, or module.

External libraries

These will come in the form of separate code files that can be combined with your own code so you don't spend 99% of your programming time re-inventing the wheel.

Sometimes they will be *source code*, but a lot of the time they will be libraries that are expressed in a language that the computer understands, but which you can't either read, or change.

And, you'd better not: these are libraries that are shared across many programs, all of which expect them to work in the same way!

Main functions

We've made a few glancing references to named code blocks now, and just to reiterate: these are usually called functions (or sometimes *procedures*, or even *subroutines*).

I'm sticking with *function,* just because most of the code that a lay programmer is likely to come across is likely to be written in C/C++, JavaScript, or PHP. All these languages refer to functions rather than procedures or anything else, so it makes sense to use their nomenclature.

All programs need to have an entry point, in C that's called a 'main' function; and that's its actual name as well as a kind of definition.

Some programming languages don't have an obvious entry point. PHP, for example, just has things that are defined inside functions, and code that sits outside of any defined function that will be executed by the web server.

To get a bit technical for a moment, I've noticed that anything that is designed to run by itself, as an app or application tends to have an entry point. Interpreted languages like PHP and JavaScript, running in a sandbox -- i.e. a browser or web server environment -- tend not to have explicit entry points.

Naming Conventions & Keywords

As a programmer, you don't have total freedom over naming stuff.

We'll look at variables in detail in the next chapter, but for now just remember that a variable is a named storage area, like a box, with a label on. The label is supposed to convey what the box contains.

In the real world, we don't really have any constraints on what we can write on that label, as long as it makes sense to whomever reads it. In programming, there are some quite strict no-no's. While they differ between languages, here are some common ones:

- Identifiers starting with a number;
- Identifiers containing spaces;
- Identifiers containing special characters, with the exception of underscores;
- Identifiers that are the same as language keywords.

Within these constraints, and respecting the conventions of the individual languages, you are free, as a programmer, to name your stuff as you wish.

Keywords

The 'main' function is an example of something called a *keyword*.

There can only be one `main` function. If you start calling other functions by the same name, not only are you going to get confused, but the computer will too!

Most languages have a list of so-called *reserved words* or keywords, and these cannot be used by any programmer for any other purpose, including:

- Function names;
- Variable names;
- Data type names;
- Classes and objects;

- Etc.

Now, with that little list above, we've gone a bit outside the ideas that have already been presented. I haven't really defined what any of these things are; some have been introduced conceptually, but now it's time to describe them in detail.

Chapter 2 Data Types & Variables

Previously, we've said that a variable is basically a named unit of data storage; like a box with a name on it, and no real way of knowing what's inside until you open it.

Some programming languages don't really care what you put into the box.

It could be a number, a letter, a word or a bunch of words. You can even take a number out of the box, and then replace it with a letter. For some languages, notably PHP, it really isn't all that important.

These are called *loosely typed* languages, because they don't test the type of data before you store it in a variable previously defined as containing data of a certain type.

The opposite are strongly typed languages, where not only can you **not** mix data types in the same variable, but you need to *declare* what data you'll be putting in a variable ahead of time.

As a programmer, I tend to prefer this approach because it catches coding errors. Note also that many weakly typed languages also have an option that you can activate that makes them a **lot** more concerned with the data types, declarations and variable usage than they would be by default.

So, what is a *data type*, then?

Primitive Data Types

These are built-in data types that can be used to store data, usually:

- Integers, like 42, 3, 17 and 65,536 or -128
- Floating point numbers like 3.151, 7.47 and -938.444444

- Characters like “q” or “*”
- Strings like “qwerty”
- Special values like TRUE or FALSE
- Etc.

Now, there are a few caveats, as always.

The first is that numerical data is usually bound by type to upper and lower limits, and for floating point numbers, accuracy.

All language families tend to be different some way, which can be a bit confusing, but it’s important to remember that traditionally, languages like C, Modula-2, Pascal and some variants of BASIC have strict limits on numerical data storage.

Integers tend to be short or long, signed or unsigned. So, a short signed integer in C can contain values between 0 and 127. A long unsigned integer can traditionally go from about -4 million to +4 million or thereabouts.

However, as the number of addressable bytes of memory, sizes of screens, hard drives and such like increases, so the demand for every bigger numerical data types also increases, so it’s highly likely that the definition of a ‘long’ data type is out of date by the time you read this!

Likewise, floating point number can be regular (float) or double-precision (double). And the exact level of precision that you can get out of a *double* is likely to change for much the same reasons as the upper limit of a long unsigned integer is constantly having to be updated.

The second caveat is on special values like `TRUE` and `FALSE`. While Modula-2 (for example) has a specific `BOOLEAN` data type to deal with these, for a lot of languages, there is either a rough equation with the values 0 and 1 (but don’t

count on it) or a built-in special type that is only valid when you try to test a value for its inherent 'trueness'.

The final caveat is in the treatment of characters and strings as data types.

For some languages, a string is a primitive data type, for others it's something built up from characters represented by a primitive data type.

To understand the subtleties of this, we need to look at the difference between a *scalar* and *vector* variable.

Scalar Variables

Scalar variables contain a single piece of information: a single unit of a primitive data type. You can't usually assign a scalar variable to contain something other than a primitive data type -- the exception is with something called user-defined data types, which we'll cover later on.

Meanwhile, this is a good point to look at how different languages assign values to variables. They all use something called an assignment operator.

Assignment Operators

Here are some examples of assignment operators:

Language	**Operator**	**Example**
BASIC	=	`LET X% = 1` Although, some BASIC implementations don't need the 'LET' keyword. The % sign denotes numerical data in this case.
C/C++ & Javascript	=	`int x = 1;`

		The int keyword denotes numerical data.
PHP	=	`$x = 1;` PHP is (by default) weakly typed; all variables need to be prefixed by the '$' sign.

Modula-2, Pascal, etc.	:=	`VAR x AS INTEGER;`
		`x := 1;`
		
		This family of languages have a specific way of declaring a variable before use, also shared by some BASIC variants, using the VAR keyword.

Pseudo code, functional languages	<-	`X <- 1` Literally, place the value 1 into X.

(Naturally, it's also possible to assign the value of one variable to another, so long as they are of the same data type…)

Following the rules for scalar variables, you can't, for example try to force a bunch of numbers into one place. The variable contains 1 or 3, or if it's a floating point variable 1.0 or 3.0 (or something else). It can't contain 1 and 3 at the same time.

This means that for languages without a string as a primitive data type, you run into an interesting situation, that you could solve by using multiple variables:

```
Letter_1 = 'h'
Letter_2 = 'e'
Letter_3 = 'l'
Letter_4 = 'l'
```

```
Letter_5 = 'o'
```

The first issue with this approach is that you have just limited yourself to words containing 5 letters. The other issue is that if you want to manipulate words, it will quite quickly get very complex.

In a language like PHP, you can just do something like the following:

```
$myWord = 'hello';
```

However, in the vast majority of languages, you need to use something called an array.

Arrays & Strings

An array is just a named collection of boxes which can contain various values of the same primitive data type. It's an indexed variable: rather than naming a collection of variables such as `Letter_1`, `Letter_2` and so on, you can just tell the computer that the variable is called `myWord`, and has a number of letters in it.

Here's something to watch out for: you still need to tell the computer **how many** items are in the array, at most.

BASIC languages actually have a keyword for this -- `DIM`, short for DIMension -- but most of the time you can merely put the size in square brackets after the variable name:

```
int numberList[200]; // This will store 200 numbers
```

The above fragment is in C, but it's the same syntax for JavaScript and similar in most other languages. Again, the exception is PHP, where you can infer an array

simply by the way you refer to it, and arrays can also grow magically thanks to the sophistication of the language.

In fact, arrays are a special kind of data type in PHP (as well as other languages) in that they are stored as indexed 'maps' involving a *key* and a *value*.

Let's look at a C style array and a PHP array by way of example. The `numberList` above can contain up to 200 elements. These will be numbered (watch out!) from **0** to **199**.

Once you've declared the array, the values may or may not be initialised to 0. They may be something else. Never rely on the computer to do something that you haven't explicitly told it to!

We can set a value in the array as follows:

```
int numberList[200];
numberList[3] = 42;
```

Or, if need be, we can index it with another variable:

```
int numberList[200];
int numberListIndex = 3;
numberList[numberListIndex] = 42;
```

So far, so good. BASIC languages will have similar mechanisms, as will anything in the Pascal family. However, PHP and other, arguably more modern languages, don't require any of the above.

You can simply come right out and use code like:

```
$numberList[3] = 42;
```

PHP will, behind the scenes understand that you wish to use an array, with a numerical index, and that the element you wish to insert is also a number, with the value 42.

Where PHP comes into its own, though, is if you want to do this:

```
$numberList[“three”] = 42;
```

Or:

```
$numberList[3] = “forty-two”;
```

While PHP can’t add one to “forty-two” to make “forty-three”, it does allow you to index an adrar using strings, **because strings are a primitive data type in PHP**.

In other languages, they’re treated as...

...an array of characters:

```
char myString[128];
```

As such, you can’t even assign a string to the array. This won’t work in many languages:

```
char myString[128] = “hello”;
```

Instead, most languages provide a lot of supporting functions for processing strings, and it’s worth familiarising yourself with them.

(As a footnote, Pascal also has a string primitive, as does VisualBasic, and even Javascript, but these are usually also treated as arrays of characters behind the scenes.)

Complex Data Types

Most languages also have a way of defining data types that contain more than one element, called *complex data types*. Examples of these include the `struct` (C/C++ for example) and `RECORD` in Pascal style languages.

The elements are usually accessed using dotted notation. So, in C, a struct can be defined as follows:

```
struct myAddress {
  int houseNumber;
  char streetName[128];
};
```

Then, we declare a variable:

```
struct myAddress anAddress;
```

At this point, we can start to set values in `anAddress`:

```
anAddress.houseNumber = 42;
```

The story doesn't quite end there, though, because it's also possible to have arrays of complex data types, which makes a virtual address book (a common illustration in many programming courses) an relatively easy thing to construct!

The easiest way to use complex data types such as `RECORDs` and `structs` is to define them as *user-defined data types*. In fact, many languages (for example Modula-2 and Pascal) have a specific `TYPE` keyword used to declare them.

C style languages don't insist on it, and BASIC implementations such as BBC BASIC allow a kind of in-line notation anyway:

```
DIM anAddress{houseNumber%, streetName$}
```

However, by and large, where it is supported, proper use of user-defined data types makes programming clearer and more efficient.

User Defined Data Types

The best way to think of a user-defined data type is something that becomes part of the set of primitive data types available to the programmer. It's a simplistic view, but pragmatic and correct most of the time.

Generally speaking, programing languages that are *statically typed* (like Pascal, C, Java and so on) have a type definition keyword, whereas those that are *dynamically typed* (PHP, Javascript, etc.) do not.

(Statically typed languages also tend to be compiled languages, too, as well as being strongly typed, whereas Javascript, PHP and others are interpreted languages...)

In C, defining a type is easy:

```
typedef struct {
  int houseName;
  char streetName[128];
} myAddress;
```

Once done, you are then free to declare a variable:

```
myAddress anAddress;
```

For comparison, here's the same code, defined in Modula-2:

```
TYPE myAddress = RECORD
   houseName: INTEGER;
   streetName: ARRAY [0..127] OF CHAR;
END;
```

```
VAR anAddress : myAddress;
```

Minus a bit of syntactic difference, and differing capitalisation rules, Pascal is roughly similar to the Modula-2.

There's one final data type that I'm including here because it's a bit special, and that's the *pointer*. Try to imagine that the computer's memory is stuffed full of data and that some of it belongs to your program.

Usually you access that data through named variables, but sometimes it's convenient to use a pointer to the data as a reference.

We'll come back to what those circumstances might be in the Data Structures and Algorithms chapter, but for now just bear in mind that a pointer points to memory that contains a value.

Objects

There's a final caveat in this discussion, and that's the fact that object-oriented languages such as Javascript and PHP don't have a real equivalent of the `struct` or `RECORD`.

Instead, the programmer can define their own objects that contain both properties and executable code to initialize and manipulate those properties.

This isn't the time for a deep dive into what object oriented (and object based) languages offer for programmers, but bear in mind that complex variable types in Javascript, PHP and others are usually constructed by making an object, and where those properties are usually able to be accessed in much the same way as you would access the members of a `struct`, unless they're marked as private.

Chapter 3 Decision Making

Not much gets done in programming without taking some decisions based on the data that is being manipulated. All programming languages have a basic decision making mechanism known as the `IF` statement:

> "If A is equal to B then do C"

Naturally, implementations range from the verbose to the terse, but all languages have the same core building blocks:

```
IF ... THEN ...
IF ... THEN ... ELSE ...
IF ... THEN ... ELSEIF ... THEN [... ELSE ...]
```

Anything derived from C will use { and } to enclose the code that is to be executed. BASIC, along with languages from the Pascal family will use some form of `IF` and `ENDIF` markers, possibly with a `BEGIN` thrown in for good measure.

But they all follow the same basic ideology: compare two (or more) values, and then do something. In the event that the comparison fails, you might also want to do something *else*.

So, how are the comparisons constructed?

Comparison Operators

The comparison operators used are remarkably similar to mathematical operators:

Equal to... == in C style languages,

	or = in others
Not equal to...	!= in C style languages,
	or <> in others
Greater than	>
Less than	<
Greater than/equal to	>=
Less than/equal to	<=

These can all be used to compare two primitive data types. They cannot be used (usually) to compare arrays -- strings are a possible exception, depending on the language -- or complex data types.

Some examples (all of the following are equivalent, ... denotes multiple lines of code):

```
IF x = 5 THEN ... END
```
Modula-2

```
if ( x == 5 ) { ... }
```
C/C++ & Javascript/Java etc.

```
if ($x == 5) { ... }
```
PHP

```
IF x% = 5 THEN ... END
```
BASIC

Most languages also allow the programmer to skip the enclosing code ({ and }, or END statements) in cases where the code to execute is a single statement following the condition test.

What all the languages also have in common is that the decision to execute the code is based on testing the veracity of the comparison. In other words, the result can be `TRUE` or `FALSE`.

These are values named after the mathematician, George Boole, called Boolean values, and can be combined and tested using something called Boolean logic.

Boolean Logic

Programmers are very fond of Boolean logic. While there are many different ways that Boolean logic can be used in programming, we're only going to look at how it applies to decision making.

In the following table, the BASIC / Pascal style notation is on the left, and the C/Java/Javascript/PHP notation is in the middle. The plain English description is on the right:

`AND`	`&&`	the output is `TRUE` if both inputs are `TRUE`
`OR`	`||`	the output is `TRUE` if either input is `TRUE`, or both
`NOT`	`!`	the output is the opposite of the input

Mathematically inclined readers will notice two things. Firstly, we're treating the operators as binary (i.e. having two inputs) except for the `NOT` operator, which takes only one input.

Secondly, we've missed out one of the most mathematically interesting operators, `XOR`.

`XOR` is a funny beast: it stands for eXclusive-OR, and is `TRUE` if, and only if, one of the inputs are `TRUE`. Put another way, it's `TRUE` when the inputs are not the same.

This makes it functionally equivalent to the `NOT` operator when used in the context of an `IF` statement, and as such isn't terribly useful here. However,

XOR does have a multitude of uses, including graphics processing and cryptography, but readers will have to research those for themselves.

There's an awful lot more to Boolean algebra than these three (or four) operators, but for now, they're all we need to create complex `IF` statements.

Complex If Statements

So far, we've only seen statements with a single condition. However, it's also handy to be able to give alternatives:

```
IF A = 1 THEN
  Do action relating to A
ELSE
 Do action relating to !A
ENDIF
```

C style languages have similar constructs:

```
if (A == 1) {
  Do A
} else {
  Do !A
}
```

And, as we implied at the start of the chapter, it's also possible, in most languages to have several options in a row. For example, using the `ELSE IF` (or `elseif` in C, and `ELIF` in some languages) construction.

What happens if we have two independent, but connected conditions to test? That's when we go back to Boolean algebra.

Consider, for example, the following:

```
IF A = 1 AND B = 2 THEN C := 3 ELSE C := 4 ENDIF
```

For those readers getting a bit confused, here's a truth table for the AND operator:

Input 1	Input 2	Output
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Now, if you remember that each element if the IF statement is tested for veracity, and therefore results in a TRUE or FALSE value, then the statement begins to make more sense.

If A is equal to 1 and B is equal to 2 then C will be set to 3 (TRUE AND TRUE is TRUE, so the IF test succeeds). However, should A not be 1, or B not be 2, or both of them be values other than the ones being tested, then C will be set to 4, as the IF statement will fail, and be set to FALSE.

The key point is that IF tests for a TRUE value.

The equivalent in C, and with all brackets and white space put in to make it as verbosely readable as possible, would look like this:

```
if ( (A == 1) && (B == 2) ) {
  C = 3;
} else {
  C = 4;
}
```

Naturally, you can add ANDs, and ORs into the IF tests to model all manner of conditions, and even flip the result using the NOT operator. At the end of the day,

though, to keep things reasonable, readability has to triumph over trying to get every condition tested in one line.

To help with this, you can *nest* conditions.

Nested IF Statements

Here's the previous example, rewritten as a nested `IF` statement:

```
if ( A == 1 ) {
  if ( B == 2 ) {
    C = 3;
  } else {
    C = 4;
  }
} else {
  C = 4;
}
```

With all these sub-conditions and brackets flying around, you might have missed the obvious drawback **in this case** of using nested `IF` statements. To work out what I mean, try leaving out one of the `ELSE` clauses and see what happens to the value of `C`. The code would look like this:

```
if ( A == 1 ) {
  if ( B == 2 ) {
    C = 3;
  } else {
    C = 4;
  }
}
```

At first glance, it doesn't look too problematic, but if we hand-execute a few test cases, holes start to appear.

Let's say we leave out the second `ELSE` clause. If `A` and `B` are the correct values (i.e. 1 and 2, respectively) all is well, and `C` becomes set to the value 3. If `A` is 1 but `B` is, say, 7, then `C` is set to 4, because the nested test fails.

However, what happens when `A` is 3 and `B` is 2?

The answer is that `C` never gets set to anything, which could be a disaster!

Of course, the easy way round that is to set the value of `C` to be 4 at the outset, and then just execute a nested `IF` statement with no `ELSE` clauses at all. But, there will be times when that approach isn't feasible, and it's worth taking the time to understand the potential pitfalls of nested `IF`s.

Now, this is all fine for doing simple evaluations based on discrete values. Using the comparison operators that we mentioned at the outset, you can also check for ranges of values:

```
IF A > 1 AND A < 5 THEN ...
```

(This will execute the code for values of A equal to 2, 3 and 4, naturally.)

But, what do you do if you want to execute one action for a value of 2, another one for a value of 3 and a third action for a value of 4? You could end up with this:

```
IF A > 1 AND > < 5 THEN
  IF A = 2 THEN ... ENDIF
  IF A = 3 THEN ... ENDIF
  IF A = 4 THEN ... ENDIF
ELSE ... ENDIF
```

The above doesn't look too bad for a few options, but what happens if you want to trigger actions based on, say, a whole keyboard of letters?

Luckily, most languages have a way of dealing with this, called the switch statement.

Multiple Choice Selectors

The good news is that the reserved words associated with multiple choice selectors are the same between BASIC, C and Pascal style languages. The bad news is that they are used in different ways.

Here's a quick run-down: BASIC and Pascal style languages use a `CASE OF ... WHEN ... OTHERWISE` structure rather than an `IF ... THEN ... ELSE` structure, and C style languages use a `switch(...) { case ... }` structure.

The following table gives a loose comparison:

BASIC-ish	C-ish	Pascal-ish

`CASE <variable> OF`
`  WHEN <value>:`
`  WHEN <value>:`
`  .`
`  .`
`  .`
`OTHERWISE:`
`ENDCASE`
	`switch (<variable>) {`
	`  case <value>:`
	`  break;`
	
	`  case <value>:`
	`  break;`
	
	`  .`
	`  .`
	`  .`
	`  default:`
	`}`
		`case <variable> of`
		`  <value>:`
		`  <value>:`
		`  .`
		`  .`
		`  .`
		`else`
		`end;`

In most C style languages, only integer numbers and characters can be tested, and not floating point numbers, strings or Booleans. The exception is Javascript, which allows strings to be used.

The others, by and large allow anything except Boolean values and floating point numbers.

However, there's a caveat: in some cases it's not clear whether "orange" is the same as "Orange", so case-sensitivity is something that will need to be tested on a language by language basis.

Members of the C language family also require the use of the `break` keyword to prevent fall-through. If the `break` keyword were not there, the computer would just go on to the next line, and keep going until it hits a `break` statement or the end of the set of `case`s.

This enables programmers to write code such as:

```
switch ( myNumber ) {
  case 0: // Do something with zero
  case 1: // ...or 1
  case 2: // ...or 2
  break;
  case 3: // Do something different with three
  break;
  default: // Nothing matched, so...
}
```

Note, however, that the default clause is optional.

In the other languages, this kind of code isn't supported. Instead, Pascal style languages use ranges and comma-separated lists, such as:

```
case myNum of
  0..2 : // Do something;
       3 : // Do something else
end;
```

Or:

```
case myChar of
  'A', 'q', 'M' : // Do something;
           'Z' : // Do something else
end;
```

Of course, you're not limited to a single statement of code in the instruction block following the case label, however be sure to follow your chosen language's conventions for follow-on code statements and correct statement separation with ':', ';' or '|' characters as appropriate.

Where switch/case statements really come into their own is in processing inside loops, as we'll see in the next chapter.

Chapter 4 Loops

Up until now we've seen:

- How to store data;
- How to test data against various values;
- The conditional execution of code based on evaluating data.

This is all very useful, but considering the fact the we need to tell computers how to do everything, and that they typically don't learn, there's a bit missing: repetition.

Imagine a human baking a cake. If you tell them to break 8 eggs, and show them how to break one egg, you can be fairly secure in the knowledge that the other 7 will get safely broken into a bowl.

There might be some errors, and the odd bit of shell, but humans tend to learn pretty quickly from their own experiences.

Which is why we complain when we have to tell someone the same thing over and over again; however computers need to be told how to do something every time you want them to do it.

And, in the same way that it gets tedious to have to show someone how to break an egg 8-10 times, it would get tedious if you had to program in the same way. Also, what happens when you want to do something a *varying* number of times that you don't necessarily know in advance?

Welcome to the world of loops...

Uncounted Loops

All loops need three things:

- A beginning;
- An end;
- An exit condition.

My absolute favorite loop is in Modula-2:

```
LOOP
  IF <condition> THEN EXIT END;
END;
```

You can do anything with this kind of loop that you can do with all the other kinds of loop; which often leads me to say that any programming language only needs one kind of loop.

These are called uncounted loops because the exit condition set by the programmer means that it isn't necessary to know the number of *iterations* that the loop will go through at the time the loop is defined.

Subsequently, the computer doesn't need to keep track of the number of iterations. The corollary to that is the counted loop, which we'll look at in a moment.

Modula-2 and Pascal, as well as many types of BASIC also have a REPEAT ... UNTIL loop:

```
REPEAT
  // Do stuff
UNTIL <condition>;
```

The `<conditions>` that we refer to are constructed in exactly the same way as an `IF` condition. `REPEAT` loops all evaluate their exit condition at the end, whereas, `WHILE` loops do it the other way around.

Here's one in C:

```
while (<condition>) {
  // Do something
}
```

Most languages have WHILE loops, and also allow you to construct them like a REPEAT loop:

```
do {
 // something
} while (<condition>);
```

In C, Javascript and PHP you don't get a choice because they only have `while` and `do-while` loops. However, in many Pascal style and BASIC languages, you could use a `REPEAT ... UNTIL` or `DO ... WHILE` loop to achieve the same thing.

Here's an exercise: what code would you write if you wanted to execute a block of code 10 times (and only 10 times?)

Think about the various ways you could do it with a loop-exit, do-while, or even repeat-until loops; and then read the next section.

Counted Loops

A counted loop in any language isn't strictly necessary in the sense that it is only an elegant re-statement of one of the other types of loop. Known in most languages as for-loops, the construction varies depending on the family.

Referring to the last section, where I asked readers to reflect on a counted loop that would execute 10 times, here's the solution, in C, using a counted loop rather than an uncounted one:

```
int myCount;
for (myCount = 0; myCount < 10; myCount++) {
  // Do something
}
```

Everything in the above will be familiar, except one little bit of code:

```
myCount++;
```

In C, this is functionally equivalent to:

```
myCount = myCount + 1;
```

Otherwise, the for-loop code fragment should more or less make sense. And, I'll agree that it is more elegant than an uncounted loop would have been. For completeness, I should also mention that the following is functionally equivalent:

```
int myCount;
for (myCount = 0; myCount < 20; myCount = myCount + 2) {
  // Do something
}
```

There are also other ways of representing the conditions and operations, but they follow the precise rules for C programming and are so a bit outside the scope of this book.

BASIC and Pascal based languages also have for loops. Here's the same loop as above, but expressed in Pascal notation:

```
for myCount := 0 to 9 do
begin
```

```
  // do stuff
end;
```

Eagle eyed readers will note that I have delimited the loop to 0 to 9, but I could equally as well have used 1 to 10. In the C loop, I could also have changed the condition to a test of equality (i.e. `myCount == 9`): it all depends which you prefer!

BASIC languages are only a little different:

```
FOR myCount% = 0 TO 9
  REM Do stuff
NEXT myCount%
```

In order to emulate the `myCount = myCount + 2` code in the second C example, we need an additional keyword. BASIC has the `STEP` keyword, and the Pascal family have the `BY` keyword.

So, in BASIC, the for statement would read:

```
FOR myCount% = 1 TO 20 STEP 2
```

And, in Pascal:

```
for myCount := 1 to 20 by 2 do
```

All of the above can also be turned on its head to construct a count *down* loop rather than a count *up* loop, **and** all of the various numbers can be replaced by variables.

So, even if you weren't sure at design time how many iterations you would need, as long as you can calculate a start, end, and step value, you could construct code such as:

```
for (myCount = myStart; myCount < myEnd; myCount = myCount + myStep)
```

So, even if you argue that counted loops aren't strictly necessary, we can all agree that they're a pretty elegant way to achieve results that would take a few more lines of code to achieve otherwise.

Iterative Loops

Some programming languages, such as Javascript and PHP have loops that can be made to *iterate* over sets of values. For example, suppose we have defined an array of arbitrary length in Javascript as:

```
var myArray = [ 10, 9, 4, 1 ];
```

We could easily iterate over the array with a `for` loop, assuming we knew how many elements there were in the array, or were willing to count them beforehand.

However, with the for-of loop, life becomes much simpler:

```
for (var myItem of myArray) {
  // Do something with varying values of myItem
}
```

The loop will run through each item, setting the value of the named variable to the value previously stored in the array, and in the order that they appear.

If the object isn't an array, but a single object with properties, there is also a `for-in` loop:

```
  var myAddress = { houseNumber: 27, streetName: "main street" };
  for (var myAddressItem in myAddress) {
    // Do something with myAddress[myAddressItem]
  }
```

Note that you can also iterate over an array with the `for-in` construct, but the order of elements returned is not guaranteed as it is with a `for-of` loop.

In PHP, the `foreach` loop fills a similar role, and can only be used with arrays or objects. If you're working with an array, then the following form is used:

```
foreach (<$array> as <$value>) {
  // Do something
}
```

All the types in the array should be the same, and each element should be initialised. If you're working with a keyed array (rather than an indexed array), then you can also retrieve the key values with the following form:

```
foreach (<$array> as <$key> => <$value>) {
  // Do something
}
```

This enables programmers to iterate over an array of properties, as if it were like a Javascript object, or a C struct. Note that C does not offer any native facility to iterate over `struct` members in this way, which is a shame, because it doesn't support keyed arrays, either, only indexed ones.

'Fake' Loops

This section is more a historical curiosity than anything else; but it does help to understand the roots of several languages. For the curious, read on, for the others, you might like to skip to the next section, which introduces recursion.

The early BASIC languages had line numbers, and these were often used to create something I call 'fake' loops. Consider the following extract:

```
10 PRINT "HELLO"
20 GOTO 10
```

By now you've read enough code to know roughly what this will do, and that it will keep doing it forever. It's an example of something called an *infinite loop* (see below.)

However, using your new-found skills, you'll also realise that if we apply the loop qualities from the opening part of this chapter (namely that there should be an exit clause), you'll also see quite quickly, that there's a way out:

```
10 PRINT "HELLO"
20 IF <key pressed> THEN GOTO 40
30 GOTO 10
40 <rest of program continues here>
```

I've mixed in some descriptive phrases in < and > rather than actual code, but you get the idea: line 20 contains the exit clause, making the above equivalent to a Modula-2 `LOOP ... EXIT ... END` loop, or one of various equivalents in C (I'll leave discovering those as an exercise).

Another fake loop construct is the `GOSUB` statement.

Again, something that I've only seen in early BASIC languages, but worth knowing about, it is often paired with a `RETURN` statement:

```
10 PRINT "Going to subroutine"
20 GOSUB 100
30 PRINT "Back from subroutine"
40 ... <other program lines> ...
100 PRINT "Inside subroutine"
110 ... <other program lines>
200 RETURN
```

One of the issues with something like this, is that if you forget to terminate the program between line 40 and 100, then an error will be produced at the first `RETURN` statement, as it is encountered with no pairing `GOSUB`.

Subroutines are, for the most part, to be avoided, and aren't supported in many modern languages, *per se*. However, as we'll see in the next chapter, the idea of breaking a problem up into component parts, which can be developed separately -- the main goal of subroutines in the first place -- is alive and well under the guise of named code blocks.

(Footnote: before anyone complains, I realise that most languages also have a `label:` construct which acts like a `GOTO` statement would, although without the accompanying `RETURN` to pair with a `GOSUB` equivalent. Although undeniably useful, they do tend to lead to poor programming practices in bigger projects and are usually best avoided.)

Infinite Loops

An infinite loop is one that *has the potential* to go on forever.

Sometimes this is required (like in a videogame, where the computer tests for player input, then moves the game world around) but frequently it's the result of an error in the design or flawed implementation of a correct design.

One of the most difficult places to catch an infinite loop, but by far the easiest way to inadvertently create one, is in the use of recursion.

Recursion is an odd beast when you encounter it for the first time, but has a lot in common with creating loops: i.e. **it needs a definite exit condition**.

Without one, it will turn into an unwanted infinite loop.

Usually an infinite loop is just an inconvenience, and a difficult thing to test and correct. However, when encountered in recursive code -- code that essentially calls itself -- it can become something more sinister: a system killer.

Because recursive code calls itself, it takes up space in the system for each call: memory has to be reserved for each *iteration* of the code's execution. All of the variables that were being used at the time must be suspended, and new copies made for the new iteration.

This uses something called a *stack*. We'll look at how stacks work in the Data Structures and Algorithms chapter, but for now just bear in mind that the program stack in most computers serves as (temporary) storage for program data.

If a piece of code waltzes off into infinity, calling itself over and over, that stack will invariably become full, and eventually get resized, stealing resources from other processes.

These days, we'd hope that a modern operating system would detect that this is happening, and shut down the rogue process, but it isn't always easy -- read: it's never "easy" -- to detect!

So, while an infinite loop is bad, it's rarely terminal, whereas a rogue recursive function can bring down the whole house of cards...

...so make sure all your loops have definite exit clauses.

The Case for Switches Inside Loops

Since a loop is designed to have a different value (or at least a varying value) each time it is run, decision making inside would be cumbersome if you had to rely on a collection of IF statements.

As we saw in the previous chapter, using switch/case statements is a far more elegant way to conditionally execute code based on a value that is varying constantly within a loop.

It avoids cascading nested `IF` statements, and forces the programmer to think in a certain way that may even lead to slightly better, more efficient code. I certainly find it easier to read and figure out what someone's code is doing when it's expressed as `switch` and `case` statements rather than a block of `IF`s.

Chapter 5 Named Code Blocks

We looked at line numbering and the use of subroutines and the oft-berated `GOTO` statement in the last Chapter on loops, and I mentioned that there was an almost universally accepted “better way”.

Before we look at it in detail, let’s just consider why there should even *be* a better way.

Programs are just lists of instructions, like a recipe.

And, like a recipe, they can be broken down into steps. These steps can be simple -- “add a teaspoonful of flour” -- or complex -- “make a white sauce”.

In the first case, the imaginary robot chef would probably be able to carry out the instruction right away: providing it knew what four was, and what a teaspoon was, and had the right context for the action.

For the second case, which is a lot more complex, the robot would probably benefit from an additional list of instructions to enable it to carry out the request.

We might call that set of steps the “White Sauce Subroutine”.

As a human chef -- or home cook -- we’d probably need to look this up in a recipe book somewhere. There would be an index (remember those?) which would lead us to a page where the instructions would be laid out for the preparation of a white sauce.

So, our White Sauce Subroutine is part of a Library of cookery *functions*.

That’s all a *function*, often also called a *procedure* in BASIC languages, really is.

Procedures & Functions

Time to get one thing out of the way: in BASIC programming, historically, a `PROCEDURE` differed from a `FUNCTION` in that a `FUNCTION` returned a result.

This convention somehow also made its way into language such as Modula-2, but *only* as a convention. The reserved word for a named code block is `PROCEDURE` whether it returns a value, or not.

Just to muddy the waters, in C and languages derived from C, we refer **only** to functions, and the keyword is... `function`.

So, in Visual BASIC, a simple procedure is defined as follows:

```
Sub <name> (<parms>)
.
.
.
End Sub
```

The <name> can be any legal identifier name -- i.e. not a keyword, nor one that starts with a number -- and the <parms> section contains a list of variables, **and their types**.

Procedures don't need parameters, but some BASICs will require that the empty parentheses are included anyway.

While a procedure comes to a natural end at the `End` statement, some implementations will allow the programmer to use an exit statement -- usually one of `Break`, `Exit` or `Return` -- to return control to the calling code.

However, not all BASIC variants support this, so it's wise to check.

To return a value from a procedure, you need to use a function, which is what the `Return` keyword is usually reserved for:

```
Function <name> (parms) As <type>
```

```
  .
  .
  Return <value>
End Function
```

Obviously, the type of the returned value needs to match the declared type of the function, or an error will be produced. Some BASICs also allow you to write:

```
  =<return_value>
```

Or:

```
  <function_name>=<return_value>
```

The difference between these is usually that `Return` will immediately quit the function and return control to the calling code, whereas setting the return value with the = sign merely causes the return value to be associated with the function, and is not formally returned until the `End` statement is encountered.

In C and other languages from this family, and function is a procedure is a subroutine is a function. If you wanted to make a kind of template for all possible uses, it might look something like this:

```
[<type>] function <name> (parms) {
 .
 .
 .
 [return <value>;]
}
```

All the items in [and] are optional, otherwise the various parts as are you would expect -- `type` is the return type, `parms` are the parameters, and the return `value` (if there is one) has to match the `type`.

Pascal is another language that make the distinction between a procedure and a function by dint of the presence of a returned value. The keywords are, for Pascal, `function` and `procedure`.

Modula-2 uses `PROCEDURE` as the keyword for both procedures that return a value, and those that don't. The return type is cited *after* the procedure name, thus:

```
PROCEDURE <name> (param_1 : type_1, ... param_n : type_n) : type
BEGIN
  .
  .
  RETURN <value>;
End <name>.
```

For a procedure that doesn't return a value, just leave out the return type and return statement. Easy!

Finally, a slight oddity, BBC BASIC. This is the language that I grew up with -- along with a lot of UK school children in the 1980s! -- and is one of those that insists on the use of a `DEF` keyword.

In addition, whereas most languages allow the programmer to call the function just by writing its name (and parameters), BBC BASIC (and probably others) insist on using the PROC keyword, followed by the name.

Luckily, for the sake of readability, the naming conventions allow the procedure name (or function name -- yes it also makes *that* distinction!) to start with an underscore. So, a procedure can be defined as follows:

```
DEF PROC<_name> (param_1, param_2... param_n)
  .
  .
  .
ENDPROC
```

The procedure is then called by writing:

```
PROC<_name>[(parameters)]
```

Note that the parameters don't have types associated with them, however, the usual `$` and `%` notation can be used for letters (strings) and numbers.

Functions follow a similar pattern:

```
DEF FN_name[(parameters)]
  .
  .
  .
=<value>
```

BBC BASIC also allows a one line function, which makes sense for some mathematical operations, such as square root definitions.

A final note: in all the language discussed here, return values can be either constants or variables. However, care must be taken when deciding whether a function should return a variable or modify one of the variables supplied as a parameter.

The extent to which the programmer is free to choose is governed by the *variable scope* and whether parameters have been passed by *reference* or by *value*.

Variable Scope

In ~~most~~ all programming languages, a variable defined *within* a function is accessible to code within that function **only**:

```
Function Do_Something ( number )
  Var another_number;
  // We can do things with number
  // We can do things with another_nmber
End Function.
Do_Something(42); // Perfectly fine
```

```
Print another_number; // Error > another_number out of scope
```

Global variables -- ones that are defined outside any other code block, including any `main` functions -- are accessible to all code. Parameters that are variables outside of the function and passed to the function, are available as a value.

Any variables instantiated within a function will be void when the function exits. This also means that they will be reinitiated when the function is called again, and will not hold their value between calls.

With these items out of the way, it's time to look a a few ways around them.

Passing by Value vs. Passing by Reference

Naturally, across languages, passing variables is done by *value*. In other words, a copy of the variable's value is passed as a *constant* to the function. The exception is when a parameter is passed *by reference*.

When a parameter is passed by reference, the actual reference to the memory that contains the variable is passed to the function. This means that the function can change the value of that variable.

In the following pseudocode extract, we're going to use a C style indirection operator (`&`) to show a parameter that has been passed by reference:

```
Function add_one_to (&by_ref)
  by_ref := by_ref + 1
End Function.
```

Now, if we didn't want to pass the parameter by reference, we could use a function to return the incremented value:

```
Function add_one_to (by_val)
  return by_val + 1;
End Function.
```

What we *can't* do is change the value of `by_val` in the code, and then return the updated value, because it cannot be accessed as only a copy of the actual value has been passed and not a variable.

That's the difference between passing by reference and by value.

Prototypes

Generally speaking, it's a good idea to declare a function (or procedure, hereafter I'll just call everything a function) before use. Many languages insist on it, and a few rare ones don't support prototyping at all, but for those that do, it's considered good practice.

In C programs (and some variations of BASIC), you can actually declare the functions, along with their various inputs and outputs at the top of the source code, and write the code for the functions themselves separately.

The advantage of doing this is that the entry point for the code is then *above* the functions, making the code easier to maintain.

PHP, for example, doesn't really care where the functions are defined, meaning that lazy programmers can create quite unreadable -- and hard to maintain code -- that is a mixture of functions that are only executed when needed, and code that is to be executed in line.

Javascript, due to the fact that it is loaded at the same time as the web page, can also contain functions that are defined after they're used.

But, declaring functions before they are used is better practice, and helps debugging! Add to this the fact that some languages will require it, it's better to assume they all do, so you're never caught out.

Earlier BASICs, like BBC BASIC *will* only execute procedures out of sequence, however, so they need to be put them in a so-called *unreachable* code block, i.e. after the END statement that terminates the program.

(Note also that the word *prototype* is subjective: Javascript documentation uses it for another purpose, albeit related, and is discussed in the Chapter on Classes and Objects.)

Chapter 6 Classes & Objects

Time for a little bit of back covering: this isn't a full discussion about object oriented design and programming. And, while useful for any programmer, some of this Chapter is a bit on the technical side.

It's useful because there's a lot of object oriented philosophy in many languages, and the arguments *for* it are strong. On the other hand, C has survived quite happily since its inception around 1972 without more than a nod towards true object oriented principles…

That being said, it's a good idea to understand the concepts, even if your chosen language has no support or use for them.

Modula-2, for example, in many implementations, relies on a `MODULE` system, where functions are `IMPORT`ed. The system of modules has some of the characteristics of object oriented programming, but leaves out others.

The base unit of programming is known as the class, and is used as a kind of template for defining objects that share the classes underlying functionality.

Constructors & Destructors

A constructor is code that is called at the time that the programmer creates an *instance* of a class (as an object). Usually, the constructor is a good place to set the default values for any *properties* that the class might support.

These properties are essentially variables that are local in scope to the object. They can only be modified by the object, unless they are declared as public. However, it is the opinion of many programmers that public properties are not a good thing.

The destructor is called when the object is disposed of.

It should clean up any memory that has been allocated, although so-called garbage collection has become a lot more intelligent in modern computing, and since the primary function of a destructor is usually to release memory, some languages under-emphasize its use.

The following is some basic C++ code for defining a class, and creating an object:

```
class example_class { // An example class
  private: // properties
    char * memory_blob;

  public: // access functions
    // Constructor
    example_class() { memory_blob = new char [256]; }

    // Destructor
    ~example_class() { delete [] memory_blob; }
};

void main () { // Entry point
  // Create the object
  example_class oObject;
  // Do stuff with the object
  delete oObject(); // Destroy the object
}
```

There's quite a lot of code in this example, but the comments ought to make most of it self-explanatory. In C++, as in many languages, constructors and destructors carry the name of the class, with a prefix of ~ (in C++) for the destructor.

The `*` in front of the `memory_blob` property indicates that it is a *pointer* to a value (i.e. a location in memory that is of type `char`, but might be of any size) which is then instantiated with the `new` keyword.

Since we have allocated a 256 byte block of `char` memory, and given that C++ has limited garbage collection facilities, it's only polite to destroy the blob when we've finished with it, which is done in the destructor, using the `delete` keyword.

While other languages will do this differently, the facilities and features are basically the same:

- Constructor and destructor;
- Keywords like new and delete to create and destroy;
- Properties hidden (private), access functions exposed (public);
- Created as objects or pointers to objects.

There are also some other things we can do with classes, but before we look at theml, we should discuss the obvious exceptions.

Prototype-Based Languages

Javascript is a good example of a prototype based language; it has no classes, just a kind of templating system, using prototypes. The way that a prototype is created is simple:

```
  var example_prototype = function () { };
```

This creates a prototype that does absolutely nothing. We should really add some code to the constructor, though, even if only to set up the properties:

```
var example_prototype = function () {
  this.example_property = 0;
```

```
};
```

Now, we can instantiate some objects:

```
var example_object = new example_prototype();
```

Where Javascript can get interesting (and confusing) is in its ability to accept new additions in line with other code. So, for example, there's nothing to stop the programmer suddenly deciding that the prototype needs an extra method (function):

```
example_object.prototype.example_method = function () {
  // do something
}
```

This can be done any time after the prototype has been created, whether there is intervening code or not, but it is ill-advised, at best. The process should be:

- Instantiate the prototype, and properties;
- Create the access methods;
- Create the objects for use in the program.

Something that is very handy is the possibility to use the prototype of the object as the prototype for another set of prototypes. These will have the same properties and methods as the parent, but will also serve as a starting point for new features.

This is called inheritance, and is an important part of both object oriented programming and the concept of classes and objects.

Classes vs. Object Oriented Programming

Firstly, I want to reiterate something: C with classes is *not* object oriented programming. For a full discussion of this, search online for the keyword phrase "c with classes"...

That said, class based programming, prototype based languages and pure object oriented programming do share three really key philosophies which make programming an awful lot more efficient.

However, arguably for the beginning programmer, they make life a bit difficult and increase the need for correct design before development starts.

This is no bad thing.

Data Hiding

The principle of *data hiding* is that only the instantiated object should have access to its own data, and no other code should be able to access it.

Were this principle to be embraced by C++, you might argue that there would be no need for the `private` and `public` keywords at all, but that's not an assertion I'm willing to make.

However, I do believe that all properties should be hidden inside the object, and that only access functions and interfaces that exchange data should be made public.

(That's not so say that, on occasion, you can't expose other methods, but I would say that properties should always be hidden!)

Inheritance

This is another great feature of object oriented programming that is adopted by prototype based languages and class based languages alike.

The way to think of it is like classification of flora and fauna. There is a starting definition, which is assumed to cover all possible cases. Then, however, a new example is found in the wild, but it is noted to have a difference that makes it a new family on its own, but with some shared characteristics.

Thus, a new class is created, that *inherits* from the parent, and then specializes.

Of course, we know that the story rarely ends there, and that new examples that are subtly different will be discovered. Thus, the biologists in charge of classification have to decide where to create a new class, or just extend the definition of the old one.

Programmers come up against exactly this dilemma.

Here's a hint: it's much easier (read: cheaper) to come up against this challenge -- and solve it -- at design time, as opposed to once you've started developing.

This is why I mentioned at the outset that, while powerful, adopting even a C with classes approach to object oriented programming often requires a greater investment in the design philosophy than regular procedural and non-object based programming.

At least, until you get the hang of it!

Polymorphism

The final OO principle that is shared with other non-OO but object based programming is the idea of polymorphism: the theory is that a class can have one method that does two different things, depending on implementation.

Polymorphism allows us to do two things : specialise and share.

Specialisation allows the programmer to inherit a definition, and then change part of it to fit a new object type. Sharing behaviours allows objects of different types to share a parent that has behavior that is useful to each.

It's a big topic, and this is just a guiding principle; if you really want to get into it, search for polymorphism online and be prepared to drink a *lot* of coffee (or tea, or even water) whilst you go through the arguments for, and against...

Chapter 7 Data Structures & Algorithms

This is the fun bit.

This is where we take all of the concepts, and roll them into some examples of code (and pseudocode) that will actually *do something*.

No matter that all of what we're about to see has already been done before -- they are interesting thought experiments, if nothing else, for budding programmers.

In fact, a lot of modern programming languages actually implement some of these data structures and algorithms; and if they don't most have libraries of code that support them.

Take, for example, arrays, and the way that Javascript implements support for processing data stored in them.

Arrays

In PHP and Javascript, support for arrays is quite vast. In C, BASIC and others, less so. As a programmer, though, it is useful to understand that an array is simply an area of memory that is segmented into units of storage just big enough to contain one element at a time.

Fixed Length Arrays

Accessing elements in the array uses something called an index (we covered these earlier) which the computer translates into an *offset* from the place at which the array starts:

0	1	2	3	4
<- element size ->	<- 1 x size	<- 2 x size	<- 3 x size	<- 4 x size

The way I think of an array is that the array name *points to* the memory location at index 0, and that the *type* of each element gives the computer the offset that is needed for each indexed element.

So, if you have a user defined data type that contains two integers, you could expect that it take sup twice as much space as a single integer. Thus, the offsets for the struct members will be 0, 2, 4, … and for the integer 0, 1, 2, 3 …

The indexes, however, remain the same for both.

When using fixed length arrays, this is an easy proposition: the computer knows by your definition of the array how much memory it will need.

Incidentally, a lot of C programming documentation refers to arrays and strings almost interchangeably, as if the only function of an array is to provide access to a string.

While it's true that a string in C is usually just an array of characters (in C99, C++ and other modern versions there are additional definitions) this approach

leaves to one side a lot of array processing that Javascript and PHP programmers take for granted.

These languages define arrays as being variable length (at run-time).

Variable Length Arrays

Modern programming languages tend to separate arrays from strings and provide native classes (or prototype objects, in the case of Javascript) to process both data types.

C++, for example, added an `array` class, but even then, a lot of the methods that you would otherwise expect appear to be left out, because in C++ **arrays are static at run-time**. To cope with this, the `vector` template class was introduced, as well as the type `string`.

So, now, it is possible to define an array of strings as:

```
vector<string> strList;
```

In C programming prior to C99, which introduced variable length arrays, defining an array of variable length strings involved a lot of calls to the memory allocation and deallocation library, and was fairly messy.

This is one area where, arguably, modern languages have the edge as far as simplicity for the programmer, and flexibility goes. For example, let's look at the Javascript methods `push`, `pop` and `splice`.

Array Push

The typical array push method places a new element of the correct type -- although Javascript remains quite loose in this respect -- on to the end of the array, increasing the size by one *element*.

(Bear in mind that if you push a 5 character string onto the array, then the actual size grows by an offset of the size of the string. However, unlike in languages where memory allocation and deallocation isn't automatic, in Javascript, we don't need to care how much space is being used.)

Here's some valid Javascript code that illustrates an array push:

```
var testArray = new Array(); // This is how I like to do it...
testArray.push("hello"); // A string
testArray.push(42); // An integer
```

Javascript also maintains the `length` property, which stores the number of elements in the array. So, in the above example, `length` would end up as 2.

The opposite of push is called pop.

Array Pop

The `pop` method is interesting in that it returns the end-most array element, removes it from the array, and decreases the `length` property accordingly.

As it is a method, the correct Javascript syntax is:

```
testArray.pop();
```

The above code will run, but the value returned is lost forever! Instead, it is more usual to see:

```
arrayElement = testArray.pop();
```

This is all fine if you just want to operate on the *end* of the array. But, usually, it's nice to be able to insert and remove elements from arbitrary positions, and for that, we have `splice`.

Array Splice

To understand how this is defined, it is best to look at the PHP definition of `array_splice`:

> "Remove a portion of the array and replace it with something else."
>
> PHP Manual definition of array_splice

Based on this definition, we can say several things:

- We may decide to remove a zero length portion (insert);
- We may decide not to replace it with something else (remove);
- We may decide to replace it with one or more items (splice).

This helps to explain why most implementations of arrays (or vectors) don't have an 'insert' or 'remove' function, but an index-based splice.

So, here's the `splice` equivalent of `push`, in Javascript:

```
testArray.splice(testArray.length, 0, elementToPush);
```

The first parameter tells splice where to start working, the next indicates how many elements to remove, and the last n parameters denote a list of elements to add. Thus, there may be more than 3 parameters.

There may also be only two, as in the `splice` equivalent to the `pop` method:

```
arrayElement = testArray.splice(testArray.length-1, 1);
```

There are two important things to note. The first is that the `arrayElement` returned is, itself an array. This is a necessity because the splice operation *may* return more than one element, depending on the value of parameter 2.

Thus, it is possible to pop off more than one element at a time.

The second point to note is that the first parameter has to be set to n less than the array length since it is the last *n* elements that are to be popped. In this case, we're removing one element, so the offset adjustment is -1.

Arrays Without Array Objects

Support for these methods is more or less uniform across Javascript, PHP (although PHP doesn't really use classes or prototypes, but operates directly on objects, using functions `array_push`, `array_pop` etc.) and C++ `vector` templates.

However, let's briefly look at the steps needed to achieve these functions, without using array objects.

To push an arbitrarily sized element onto an array, the following are necessary:

- Determine the size of the element;
- Determine the starting offset of the new element;
- Reserve enough memory for the element;
- Copy the data from the element to the array end.

Clearly this is not a trivial exercise, but most languages have support for determining the memory footprint of a variable, if they don't support things like variable length arrays. These languages (like C) will also have reasonable support for memory allocation, and deallocation. The deallocation is needed for the implementation of, for example, a pop function:

- Determine the offset of the last element;
- Copy the data into a temporary variable that is big enough;
- Deallocate the memory from the end of the array.

Immediately, though, there's a problem. If each element is the same size, then as long as the type is known, the offset of the last element can be determined by simple subtraction from the size of memory pointed to by the array variable.

What happens if the array contains strings, as arrays of characters?

Without some way of knowing the length of each array (string) it becomes impossible to allocate memory correctly. And, what happens when you want to splice the array, and insert an element in the middle?

Clearly, the basic array has its limitations, so it's time to move on to more sophisticated data structures and algorithms.

Pointers Revisited

The array example above is being treated like a data structure known as a stack. Specifically, a LIFO stack: last in first out.

While these are actually quite hard to do with arrays, if we dispense with the whole idea of an indexed list -- which is what an array is -- and start with the idea of an object that exists dynamically in memory, then things actually get easier.

Let's consider for a moment that all a pointer is is a reference to an object that has been instantiated in the system by using some variation of the `new` statement:

```
ptrObject := new Object;
```

Behind the scenes, the `new` statement is responsible for allocating enough memory and calling any object constructors. Similarly, if we have a `delete` statement, then that will be responsible for deallocating memory and calling destructors.

A Trip Down Memory Lane...

At this point, I want to take a look at what actually goes on in a language that provides for the use of pointers as detailed above. We'll use C as the example language, and in particular three very important functions:

`malloc` - allocate memory of a given size;

`realloc` - re-allcoate a memory block to a new size;

`free` - release the memory pointed to.

To fully appreciate just how good we've got it with languages like Javascript, we're also going to take a non-trivial example of allocating an object that contains a string and a number:

```
struct {
  int houseNumber;
  char * streetName;
} streetAddress;
```

To mimic everything that a `new` operator might do, we need to:

- Determine the size of the struct (using `sizeof`);
- Allocate memory (using `malloc`).

At this point, we can store an integer, `houseNumber`, and a *pointer* to an area of memory, as yet unallocated, to store the `streetName` variable. Presumably, part of the constructor code would be to allocate a string:

- Determine the memory required;
- Allocate memory (using malloc);
- Copy the string.

All of the above needs to be performed for each new `streetAddress` object that we want to create. If we need to change the `streetName`, then we can use `realloc` to change the memory pointed to by the variable.

To get rid of the object, we then need to first free the memory used to store the `streetName` string, and then the memory used for the `streetAddress` struct itself.

All of which is quite a lot of work.

Let's take it one step further: if we wanted to store a collection of these `structs`, we might use an array of pointers to the `streetAddress` objects created by our `new` operator.

However, one common data structure used for this purpose is called the *linked list*.

Linked Lists

To illustrate our linked list, we are going to make a very rudimentary and abstract use of classes and objects. This is just an illustration of principle, and not actually usable code; different languages will handle the following in different ways.

First, we create a class (prototype object, or whatever is supported by the chosen language):

```
class listNode has
  pointer ptrData
  pointer ptrNext
end
```

The `ptrData` points to the payload, and `ptrNext` points to another, as yet unallocated, `listNode`, both to be allocated at run-time. The idea is that when the list is created, we allocate a pointer to the first element:

```
ptrHead := new listNode(ptrData)
```

It is assumed that `ptrData` has been allocated via a call to `new` with an object that represents the information to be stored in each element of the list. Note that, by using pointers that are essentially abstract, different objects can be stored in each element.

Part of the work of the constructor should also be to set any unused pointers -- in this case `ptrNext` -- to a null value.

To add a node to the head of the list, we can use code such as:

```
ptrNewNode := new listNode(ptrNewData)
ptrNewNode->ptrNext := ptrHead
ptrHead := ptrNewNode
```

Subsequent calls to the above will grow the list from the head. If we want to add a node to the end of an arbitrarily long linked list, starting from the head, we can execute a simple loop:

```
ptrCurrent := ptrHead
while ptrCurrent != null
  ptrTail := ptrCurrent
  ptrCurrent := ptrCurrent->ptrNext
end while
ptrTail->ptrNext := new listNode(ptrNewData)
```

This is a stepwise approach that might not be the most efficient or elegant, but it is at least, I hope, **clear**. What it also illustrates is the possibility to also insert a

node, by temporarily disconnecting the list, and reconnecting it with the new node in the middle.

Of course, in the above pseudocode we have violated most of the principles of object oriented programming such as data hiding (encapsulation) in order to achieve this clarity.

Lists vs. Stacks

What the code also shows is that lists and stacks are very closely related. In act, a stack is just a list which only allows operations at each end.

But a stack is an abstraction, and a list (in this case a linked list, but it could also be an array) is an implementation. We say that the stack is an abstract data type, and one in which the operation always happens on the last node added (LIFO).

A queue is a special kind of list in which the FIFO principle holds. The list grows from the tail, and items are removed from the head. While it's perfectly possible to implement this as a single linked list, it's much easier with a doubly linked list.

Doubly-Linked Lists

To create a doubly linked list, we need:

- A pointer to the next **and** previous node (or null)
- A pointer to the head **and** tail elements

So, our new class definition might be:

```
class listNode has
  pointer ptrData
  pointer ptrNext
  pointer ptrPrevious
```

```
end
```

Adding the first node to the list, then, requires setting the head and tail to point to the new node. It's when we add a node to one of the ends that things get more interesting:

```
ptrNewNode := new listNode(ptrNewData)
ptrNewNode->ptrNext := ptrHead
ptrHead->ptrPrevious := ptrNewNode
ptrHead := ptrNewNode
```

The new code is in bold, and ensures that the double linkage holds; we can traverse the list starting from the head and going to the tail, or vice versa.

We can also implement an easy FIFO queue, by removing the tail node:

```
ptrOldNode := ptrTail
ptrTail := ptrTail->ptrPrevious
ptrTail->ptrNext = null
```

Note that we have retained the old tail node for further processing, including, it is assumed, deleting the object if there is no more use for it.

List management functions such as those for adding and removing nodes are usually put in a separate class, which might also contain functions known as iterators.

Iterators

To traverse a linked list, we can use something called an iterator function. Usually an iterator has a purpose in mind, returning a specific node (random access iterator), for example.

Assuming that we wanted to return a node by number, emulating the array indexing feature of most languages, we could use code such as:

```
for nodeCount := 1 to 5
  ptrCurrent := ptrCurrent->ptrNext;
end for
```

This could be encapsulated in an iterator function and made part of the linked list management class:

```
class linkedList has
  listNode pointer ptrHead
  function addNode (ptrNewNode)
  function nodeAt(index)
end class
```

So, assuming the constructors for this class existed, we could then write something such as:

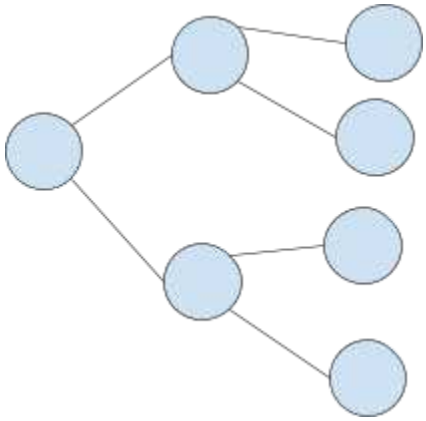
```
listObject := new linkedList
listObject->addNode(new listNode(ptrNewData))
.
.
.
ptrNodeData := listObject->nodeAt(3)->ptrData
```

Where iterators are commonly found is in the destruction of a list object, where they systematically move through the collection of nodes, deleting each one.

Trees

Linked lists come into their own when constructing a specific kind of abstract data type known as a tree. A simple binary tree contains a collection of nodes, linked together in such a way that each node has at most one parent, and at most two children.

The result is something akin to the following image:



It is not usual to implement a tree as a doubly linked list, however, so the link to the parent is inevitably lost.

A class definition for each node might look like:

```
class treeNode has
  pointer ptrData
  pointer ptrLeftChild
  pointer ptrRightChild
end class
```

There is an awful lot of theory surrounding how a tree should be traversed, or, to use the correct terminology, *walked*.

The non-trivial nature of tree walking stems from the possibility that each node can have more than one child, and the fact that one can never be entirely sure which nodes are siblings (i.e. at the same level of the tree with the same parent) without enumerating the whole tree somehow.

The first question to ponder is, in creating an iterator, how can the programmer be sure that *every* node has been visited?

Recursion

The answer, of course, is by using recursion. By using recursion, we can walk the tree with either a left hand child or right hand child bias. We can imagine a tree iterator function to achieve simple traversal:

```
function treeIterator ( treeNode )
  if (treeNode->ptrLeftChild != null) then
    treeIterator (treeNode->ptrLeftChild)
  end if
  if (treeNode->ptrRightChild != null) then
    treeIterator (treeNode->ptrRightChild)
  end if
  // Do something before stopping
end function
```

The recursive nature of this function means that it will call itself, first down the left hand side, then the right of a set of nodes, only stopping when it encounters a null node pointer on each side.

Thus, each of the nodes will be visited.

Searching & Sorting Algorithms

Iterators, lists and trees all come into their own when trying to search and sort collections of objects. The topics are often linked because they require a somewhat similar set of properties such as a comparison function, and a method of traversal.

While this isn't designed to be a complete discussion of searching and sorting algorithms, it serves as a starting point for understanding the issues involved. Firstly, we'll look at potentially unordered data in an array.

Searching Arrays

If we assume that the data in an array is unordered, then the first search algorithm is pretty easy:

```
foreach item in array
  if item == search_item then stop
end
```

Of course, the drawback with this is that the longer the array is, the more time it will take to search.

While we could use a divide and conquer methodology to reduce the search space by subdividing the array and searching one piece at a time, or multiple pieces concurrently, since the array is unordered, it is unlikely to provide much benefit.

What we need to do, is first sort the array, and to do this, we can use a technique called bubble sorting.

Bubble Sorting an Array

The idea behind bubble sorting is easy: make the nominally larger values 'bubble' to the top. To do this we need:

- A suitable comparison function;
- A swap element function;
- A way to determine when there's no more swapping left to do!

The third item also implies that we need to be sure that when there are less than 2 times, we don't bother swapping, as well as keeping track of the swaps we have done.

A bubble sort might not be very efficient, but it makes searching much more efficient, as we can now pick an element at random, and know if the target is going to be before or after the chosen element in the the array.

At a minimum we can divide the search space into two (possibly uneven) search areas. In the best case, we'll start in the middle, and find that the value we need lies above or below the chosen point each time:

Target			**Start**			
7	10	11	56	60	100	105
			Target is below			
	Target is Below		Discarded Search Space			
Hit Target!	Discarded Search Space					

It's not perfect, and would certainly benefit from an array (or list) that is sorted as it is built, but it works; and is more efficient than a simple traversal for large sets of data.

I should also mention another sorting algorithm here, called Quicksort.

The principle behind quicksort mixes the idea of swapping and partitioning to create a recursive algorithm that can sort an array of items more efficiently in many cases. The exceptions are an already sorted list, and a list where all the items tend towards equality.

The first step is to pick a pivot point, like we did above. Let's assume that we mick a point in the middle. The next step is called partitioning.

In essence, all the partitioning does is swap elements so that all those lower than the pivot are on one side of it, and all those higher are on the other side. Having done that, the low values are then recursively sorted, and then the high values.

Each time the recursive algorithm is called, the set of values to pivot-swap gets smaller, hence this is a classic example of a divide and conquer algorithm.

For more information, the reader is encouraged to search the Internet -- there are plenty of great resources detailing all the various quicksort variations!

Searching Trees

Tree searching suffers from a similar drawback as long as the items are unordered. However, this is not likely to be the case, as the binary tree actively encourages *some* classification as it is being built: why else have a left hand and right hand child?

In fact, this classification is used to good effect when applied to a sorting algorithm known as a tree sort, which is a form of insertion sort.

Insertion Sorting with Trees

It is possible to perform an insertion sort with a regular list, however, the performance is likely to be approaching that of a bubble sort. After all, at each step a comparison is performed to see *where* in the list the new node should be inserted.

Binary trees take away the need to do this for every element, and allow the programmer to just make a series of higher/lower decisions as they walk the ever-growing data structure.

As long as the resulting tree is traversed in the same way (right-left or left-right) as it was built, and recursively, the elements will come out in the right order.

If we assume that we defined our nodes as before, then the insertion function looks something like this:

```
function treeInsert ( treeChild, newNode )
  if treeChild = null then
    treeChild := newNode
    return
  endif
  if treeChild->ptrData < newNode->ptrData then
    treeInsert(treeChild->ptrLeftChild)
  else
    treeInsert(treeChild->ptrRightChild)
  endif
end function
```

Once the function returns, the resulting tree is said to be sorted, and the items can be displayed by walking the tree down the left and right hand sides respectively:

```
function treeDisplay ( treeChild )
  if treeChild = null then return
  treeDisplay ( treeChild->ptrLeftChild )
  // Display the current ptrData
  treeDisplay ( treeChild->ptrRightChild )
end function
```

Again, this might not be the most efficient implementation, but I hope that it is clear enough to give readers a flavour of the power of the tree structure as a sorting mechanism.

Once sorted, of course, searching the tree ought to become much more efficient, as each node will naturally be either above or below the target value. When neither is true, wherever you are left with must be the node you are looking for.

Or is it?

I'll leave that for you to figure out for yourself!

References

- PHP.net, 2016, Manual Page for array_splice [Online] Available from : http://php.net/manual/en/function.array-splice.php (last accessed October 2016)
- Tutorials Point, 2016, Data Structures & Algorithms (DSA) Tutorial [Online] Available from: http://www.tutorialspoint.com/data_structures_algorithms/ (last accessed October 2016)
- University of Glasgow, 2016, Stacks Queues and Lists presentation [Online] Available from: http://www.dcs.gla.ac.uk/~pat/52233/slides/StacksQueuesLists1x1.pdf (last accessed October 2016)
- Wikipedia, 2016, Iterator definition [Online] Available from: https://en.wikipedia.org/wiki/Iterator (last accessed October 2016)

Conclusion

One of the great things about electronic pûblshing is that new versions of the book can be released from time to time: and the main source of updates is, of course...

READER FEEDBACK

With that in mind, and as an incentive (of sorts), I'd like to encourage anyone who buys this book and who has a question, comment, or suggestion to send them to me via email, along with proof of purchase, **or** reference to your review of my book on Amazon, and I will send you all future updates, as Kindle compatible files by way of a thank you.

The email to send your comments to is:

`guy.leckythompson@gmail.com`

I hope you enjoyed this book, and will put your newly discovered programming skills to good use!