Introduction to Programming

Table of Contents

How to Use this Document

The ***Introduction to Programming*** course is a basic introduction to the art and science of computer programming. Although it is assumed that you know what a computer is, where the keyboard and monitor are, and other basic skills, you do not need any knowledge of programming to read and understand this document.

Chapters should be read in the order that they are presented, since later sections of the course rely on the fact that you have completed earlier ones. If you don't fully understand any section, ask your instructor to go through it with you before you proceed to the next section.

When you have completed a study of this document and fully understand its contents, you will then be in a position to embark on a computer programming course in any of the popular programming languages.

What is Covered in this Document?

In the first chapter, “Computer Programming” you discover what a computer program is, and be introduced to the concept of programming languages.

In Chapter 2, “Numbering Systems” you will learn about different numbering systems and their relevance to computers. If you are to embark upon a career as a computer programmer, you will need to be fully conversant with the different numbering systems as it is inevitable that you will come across them. To make this chapter more palatable, it starts with a revision of the decimal numbering system, which should be familiar to all of you.

Chapter 3, “Character Sets” explains what a character set is, and describes the most common character set of the personal computer and of the larger computers. Again, as a computer programmer, you cannot avoid character sets, as these are used to create the computer programs typed in at the keyboard.

In Chapter 4, “Data Types” you will learn about the main types of data that can be handled by a computer program. You will come across terms which are common to almost all computer programming languages. Being familiar with data types will ease your study of computer programming.

Chapter 5, “Binary Logic” introduces you to the way computers ‘think’. All computer programs use binary logic, but fortunately, it’s simple!

Chapter 6, “Programming Languages” describes some of the popular programming languages available today. The difference between low and high level languages is explained, as are compilers and interpreters. High level languages are listed according to their purpose for example for business, scientific or teaching applications.

In Chapter 7, “Data Storage” the concepts of a variable and a constant are explained. All computer languages use variables and constants, and a firm understanding of these data storage items is required of all computer programmers.

Chapter 8, “Data Processing” introduces the processing of data by a computer program, from input of the data, to processing of the data and eventual output of results.

In Chapter 9, “Program Flow Control” describes different methods of controlling the flow of a computer program using the **for** and **while** statements.

Chapter 10, “Program Design Methodologies” introduces the concepts of pseudocode and structured programming are introduced.

“And finally. . .” summarises and concludes this **Introduction to Programming** course.

Document Conventions

To help you to interpret information easily, this guide uses consistent format. These conventions are explained as follows.

This	**Represents**
while	Bold type indicates programming language keywords and operators, which are normally used exactly as shown
expression	Words in italics indicate placeholders for information to be supplied. The word in italics only represents the text. Italic type also signals a new term. An explanation precedes or follows the italicized term.

Computer Programming

What is a Computer Program?

A computer program is simply a set of instructions used by the computer to perform a specific task. The task could be anything from the solution to a mathematical problem to the production of a company payroll. Computer programs are written using programming languages, which are described in more detail in the following sections.

What is a Programming Language?

Programming languages are made up of programming codes, which are entered into the computer to perform a task. There are basically three kinds of programming language - low-level *machine* languages, intermediate *assembly* languages, and finally, *high-level* languages such as BASIC, COBOL, Pascal, C and C++.

Every computer has a built-in set of instructions. These allow it to perform the most basic tasks like adding two numbers together, comparing numbers, and storing the answers in the computer's memory. This special set of instructions is known as the *instruction set*. Machine language (or machine code) uses these instructions which are represented by a number (from 0 to 256) defined by a group of eight binary digits (bits). Hence machine language is composed entirely of a long list of eight bit binary numbers. The binary numbering system is explained in the next section.

Numbering Systems

Most people are used to the decimal system of numbering, which is based on ten numbers:

0,1,2,3,4,5,6,7,8,9

Let's consider decimal numbers to remind ourselves how they work.

Decimal numbers

The number 125 (one hundred twenty five) represents one hundred and two tens and five ones. Let's show that another way:

Hundreds	Tens	Ones
1	2	5

```
1 x 100 = 100
2  x  10 =   20
5  x    1 =     5
              -----
                125
              -----
```

If we look at a larger number, for example two thousand and thirty six (2036), this number can be similarly represented:

Thousands	Hundreds	Tens	Ones
2	0	3	6

```
2  x  1000  =  2000
0  x   100 =      0
3  x    10 =     30
6  x     1 =      6
               ------
               2036
               ------
```

Notice that in decimal numbers, the right most digit represents the ones, the next represents the tens, the next the hundreds, then the thousands and so on. In other words, starting at one, the others digits are multiplied by ten as you move over to the left, as the following series shows:

10000, 1000, 100, 10, 1

The binary system of numbering operates in exactly the same manner as the decimal system except that instead of having ten numbers, there are only two, and instead of multiplying by ten from right to left, numbers are multiplied by two.

Decimal - Base Ten

The following series of numbers illustrates base 10:

$$10^3 \qquad 10^2 \qquad 10^1 \qquad 10^0$$

10^3 is verbalized as ten to the power of three, and is calculated as 10 x 10 x 10 (1000).

10^2 is verbalized as ten to the power of two, and is calculated as 10 x 10 (100).

10^1 is verbalized as ten to the power of one, and is calculated as 10 x 1 (10).

Any value to the power of zero has a value of one:

10^0 is verbalized as ten to the power of zero, and has a value of one.

Binary numbers

The binary numbering system is based on two numbers:

0 and 1

A binary number is also called a *bit.* The word 'bit' is short for the two words 'binary' and 'digit'. There are only two bits, 0 and 1.

The three bit binary number 101 (verbalized as one zero one) represents the decimal number 5. This is explained below:

```
Fours   Twos   Ones

  1       0      1


1 x  4 =   4  (decimal)
0 x  2 =   0  (decimal)
1 x  1 =   1  (decimal)
          -----
           5  (decimal)
          -----
```

If we look at a larger number, for example one zero one one (1011), this number can be similarly represented:

```
Eights   Fours   Twos   Ones

  1        0       1      1


  1  x  8   =   8  (decimal)
  0  x  4   =   0  (decimal)
  1  x  2   =   2  (decimal)
  1  x  1   =   1  (decimal)
              -----
               11 (decimal)
              -----
```

A group of eight bits is known as a *byte*. The following is an example of a byte which represents the decimal number 64: 01000000

A group of four bits (half a byte) is known as a *nibble*. The following is an example of a nibble: 1010

The byte:	00000000	represents decimal 0
	00000001	represents decimal 1
	00000010	represents decimal 2
	00000011	represents decimal 3
	00000100	represents decimal 4

The largest binary number of a byte is 11111111. This represents the decimal number 255. Hence one byte can represent 256 decimal numbers, from 0, through 1 to 255.

Notice that in binary numbers, the right most digit represents the ones, the next represents the twos, the next the fours, then the eights and so on. In other words, starting at one, the others digits are multiplied by two as you move over to the left, as the following series shows:

16, 8, 4, 2, 1

The binary numbers are said to be to the base 2 as compared to decimal numbers which are base ten.

Binary - Base two

The following series of numbers illustrates base 2:

$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

2^3 is verbalized as two to the power of three, and is calculated as 2 x 2 x 2 (8).

2^2 is verbalized as two to the power of two, and is calculated as 2 x 2 (4).

2^1 is verbalized as two to the power of one, and is calculated as 2 x 1 (2).

Any value to the power of zero has a value of one:

2^0 is verbalized as two to the power of zero, and has a value of one.

Octal Numbers

Because binary numbers are quite unwieldly, for instance seven digits are required to represent the decimal number 64, computer programmers often use the octal (base 8) or hexadecimal (base 16) numbering systems because they require fewer digits to represent the same decimal number. The following sections show how the decimal number 64 (01000000 binary) is represented in the octal and hexadecimal systems.

Octal numbers only allow the digits 0 through 7. The following table shows the octal numbering system and the equivalent decimal value.

8^2	8^1	8^0	(Decimal)						
(64)	(8)	(1)							
	0	0	(0)						
	0	1	(1)						
	0	2	(2)						
	0	3	(3)						
	0	4	(4)						
	0	5	(5)						
	0	6	(6)						
	0	7	(7)						
	1	0	(8)						
	1	1	(9)						
1	0	0	(64)	i.e.	1 x 64	+	0 x 8	+	0 x 1

The octal number which represents decimal 64 is 100.

Hexadecimal numbers

Hexadecimal numbers only allow sixteen digits. The digits 0 through 9 are used to represent the first ten digits. The remaining six digits are represented by using the letters A to F. The following table shows the hexadecimal numbering system and the equivalent decimal value.

16^2	16^1	16^0	(Decimal)
(256)	(16)	(1)	
	0	0	(0)
	0	1	(1)
	0	2	(2)
	0	3	(3)
	0	4	(4)
	0	5	(5)
	0	6	(6)
	0	7	(7)
	0	8	(8)
	0	9	(9)
	0	A	(10)
	0	B	(11)
	0	C	(12)
	0	D	(13)
	0	E	(14)
	0	F	(15)
	1	0	(16)
	1	1	(17)
	1	2	(18)
	1	3	(19)
	1	4	(20)
	4	0	(64)

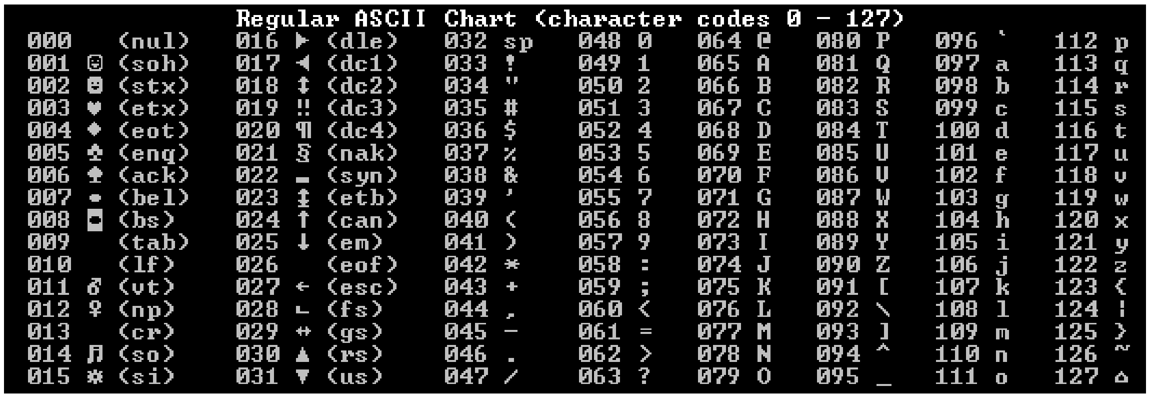
The hexadecimal number which represents decimal 64 is 40.

Although computers ultimately require all data in binary format, data supplied in octal or hexadecimal format is easily converted to binary by the computer because eight and sixteen are multiples of two. The advantage for the programmer of using octal or hexadecimal numbers is that fewer digits are required.

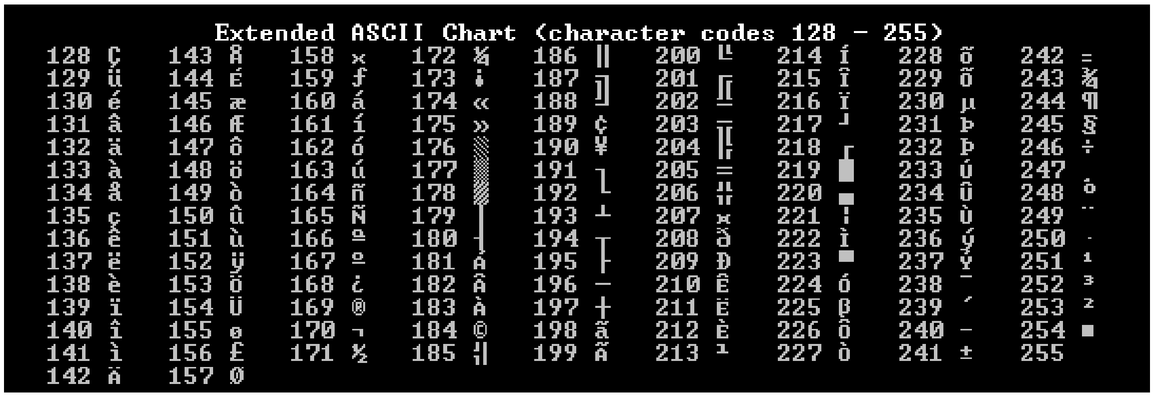
Character sets

The character set of a computer are all the characters A through Z, a though z, the digits 0 through 9, punctuation marks and other special characters. The character set is used to generate computer programs and any other text processed by the computer. A single character typed in at the keyboard is converted to a byte of information within the central processing unit of the computer. One byte (a group of eight bits) can represent 256 different characters. Certain standards have been set up to decide which byte refers to which character. The standard which is universally accepted on most personal computers is known as ASCII (pronounced asskey). ASCII is an acronym of American Standard Code for Information Interchange.

The ASCII Character set



```
                   Regular ASCII Chart (character codes 0 - 127)
000   (nul)  016 ► (dle)  032 sp   048 0   064 @   080 P   096 `   112 p
001 ☺ (soh)  017 ◄ (dc1)  033 !    049 1   065 A   081 Q   097 a   113 q
002 ☻ (stx)  018 ↕ (dc2)  034 "    050 2   066 B   082 R   098 b   114 r
003 ♥ (etx)  019 ‼ (dc3)  035 #    051 3   067 C   083 S   099 c   115 s
004 ♦ (eot)  020 ¶ (dc4)  036 $    052 4   068 D   084 T   100 d   116 t
005 ♣ (enq)  021 § (nak)  037 %    053 5   069 E   085 U   101 e   117 u
006 ♠ (ack)  022 ▬ (syn)  038 &    054 6   070 F   086 V   102 f   118 v
007 • (bel)  023 ↨ (etb)  039 '    055 7   071 G   087 W   103 g   119 w
008 ◘ (bs)   024 ↑ (can)  040 (    056 8   072 H   088 X   104 h   120 x
009   (tab)  025 ↓ (em)   041 )    057 9   073 I   089 Y   105 i   121 y
010   (lf)   026   (eof)  042 *    058 :   074 J   090 Z   106 j   122 z
011 ♂ (vt)   027 ← (esc)  043 +    059 ;   075 K   091 [   107 k   123 {
012 ♀ (np)   028 ∟ (fs)   044 ,    060 <   076 L   092 \   108 l   124 ¦
013   (cr)   029 ↔ (gs)   045 -    061 =   077 M   093 ]   109 m   125 }
014 ♫ (so)   030 ▲ (rs)   046 .    062 >   078 N   094 ^   110 n   126 ~
015 ☼ (si)   031 ▼ (us)   047 /    063 ?   079 O   095 _   111 o   127 ⌂
```



```
               Extended ASCII Chart (character codes 128 - 255)
 128 Ç   143 Å   158 ×   172 ¼   186 ║   200 ╚   214 Í   228 õ   242 ‗
 129 ü   144 É   159 ƒ   173 ¡   187 ╗   201 ╔   215 Î   229 Õ   243 ¾
 130 é   145 æ   160 á   174 «   188 ╝   202 ╩   216 Ï   230 µ   244 ¶
 131 â   146 Æ   161 í   175 »   189 ¢   203 ╦   217 ┘   231 þ   245 §
 132 ä   147 ô   162 ó   176 ░   190 ¥   204 ╠   218 ┌   232 Þ   246 ÷
 133 à   148 ö   163 ú   177 ▒   191 ┐   205 ═   219 █   233 Ú   247
 134 å   149 ò   164 ñ   178 ▓   192 └   206 ╬   220 ▄   234 Û   248 °
 135 ç   150 û   165 Ñ   179 │   193 ┴   207 ¤   221 ¦   235 Ù   249 ¨
 136 ê   151 ù   166 ª   180 ┤   194 ┬   208 ð   222 Ì   236 ý   250 ·
 137 ë   152 ÿ   167 º   181 Á   195 ├   209 Ð   223 ▀   237 Ý   251 ¹
 138 è   153 Ö   168 ¿   182 Â   196 ─   210 Ê   224 Ó   238 ¯   252 ³
 139 ï   154 Ü   169 ®   183 À   197 ┼   211 Ë   225 ß   239 ´   253 ²
 140 î   155 ø   170 ¬   184 ©   198 ã   212 È   226 Ô   240 -   254 ■
 141 ì   156 £   171 ½   185 ╣   199 Ã   213 ı   227 Ò   241 ±   255
 142 Ä   157 Ø
```

The EBCDIC Character set

Mainframe computers generally use the EBCDIC standard, which is also based on 8 bits. The term EBCDIC is an abbreviation for expanded binary coded decimal interchange code. The characters A through Z in EBCDIC have the values 193 to 233, compared to the ASCII values of 65 to 90.

Data Types

Data is information that can be used by the computer. Computer data is divided up into different types. The major data types are common to all programming languages and computer systems. This section describes each data type.

Integers

Integers are whole numbers such as 1, 2 and 3. Integers can be positive or negative or zero. The complete set therefore includes the following:

. . . -5 -4 -3 -2 -1 0 1 2 3 4 5. . .

Integers are different from other types of number because they cannot represent a fraction of a number. Integers are used where whole numbers are appropriate, for example to count the number of times an event has taken place. The largest integer value which may be stored in the computer depends on the computer hardware. Some computer languages allow you to specify the integer size by declaring a *long* or a *short* integer type.

There are certain operations which may be performed on integer data, for example add and subtract. These and other operations will be described in more detail later.

Floating Point (Real) Numbers

Floating point numbers are also known as real numbers. These numbers include fractional numbers as well as integers. Examples of real numbers are: 1.5, 2.7, 3.0, 4.0. Like integers, the size of a real number can be specified in some programming languages as *long*, *short* or *double*. There are certain operations like add, subtract and multiply that can be performed on real numbers. These operations will be described in detail later.

Characters

Characters include the letters of the alphabet, numbers, punctuation marks and special symbols which are used to generate text. The character type can store only a single character. When more than one character is to be stored, the string data type is used. In the same way that certain operations may be performed on numbers, there are a set of operations which are designed for use with characters. These will be described in detail later.

Strings

Strings are made up of an array of characters. For example a word or a sentence is made up of a string of characters, one following the other. Most programming languages provide a string type, which makes text handling much less tedious than having to process one character at a time. Operations such as string splitting and concatenation are common string operations, which will be described in more detail later.

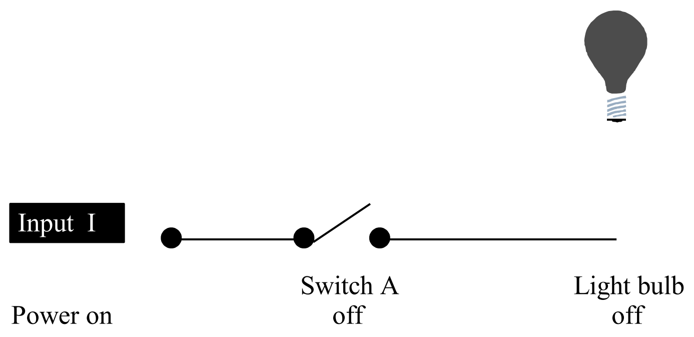
Boolean (Logical) Values

Boolean data can only take one of two values: TRUE or FALSE. Some computer languages represent TRUE and FALSE with the numbers 1 and 0. In either case, Boolean values are binary by nature and hence are closely related to the computer's own binary machine code. Boolean data is also known as logical data. There are certain operations which may be performed on Boolean values. These are described in the following section.

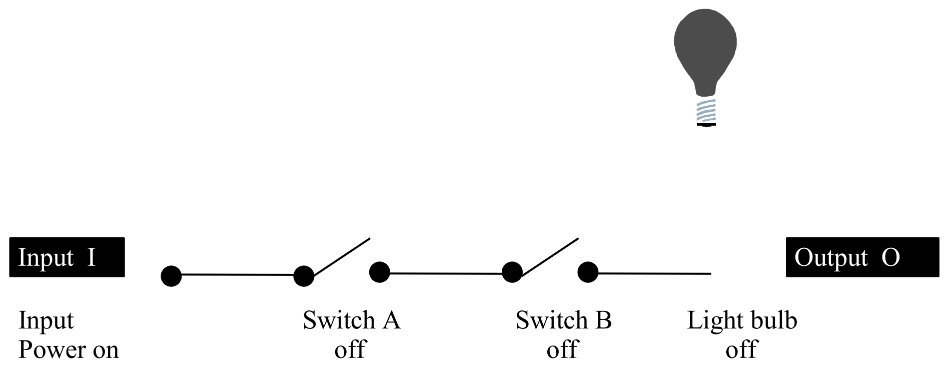
Binary logic

Essentially, computers are composed of a mass of minute electronic switches which control the flow of electric current. When current is flowing, this can be represented by the number one. When current is not flowing, this can be represented by a zero. The ones and zeros make up the bits and bytes of machine code, which ultimately result in useful work in the form of processed data. So, how can these ones and zeros be used to do something useful, like turn on the washing machine or start the VCR recording? How can these ones and zeros be used to make a decision? The answer is quite simple. Remember that a binary digit can only be in one of two states: either on or off - just like a light switch.

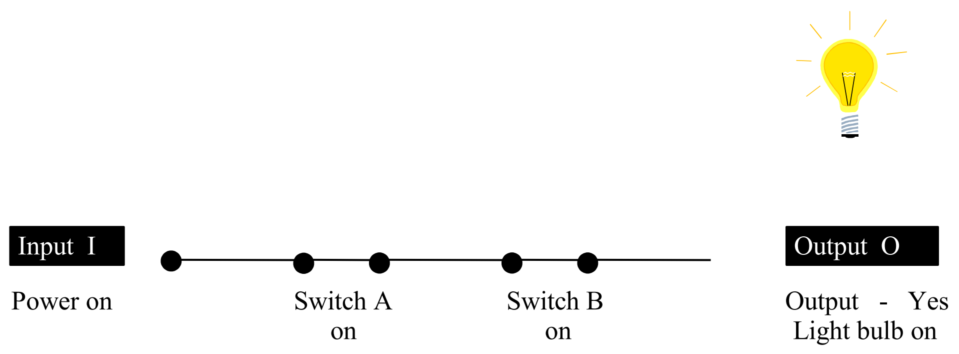
This diagram shows an electric circuit for a light switch:



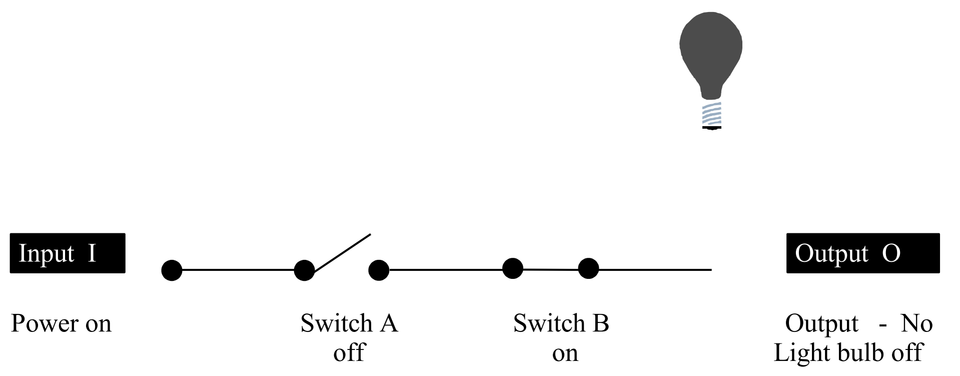
Take a look at the following diagram:

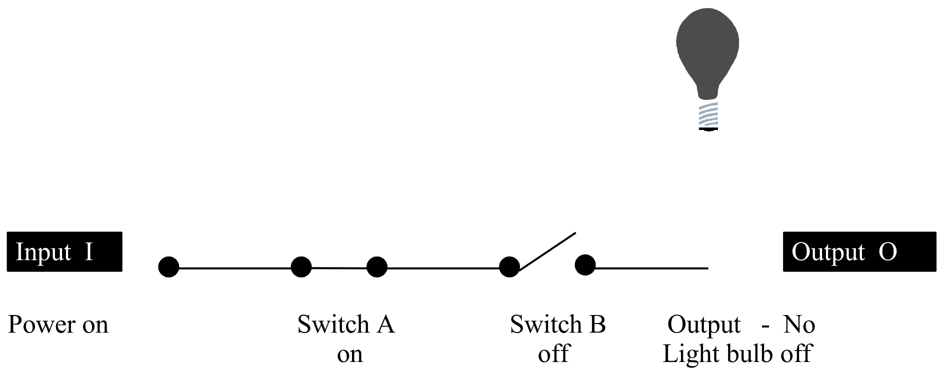


If there is an electrical input at **Input I**, there will be an electrical output at **Output O**, only if both switch A and switch B are closed (on) at the same time. This is illustrated in the following diagram:



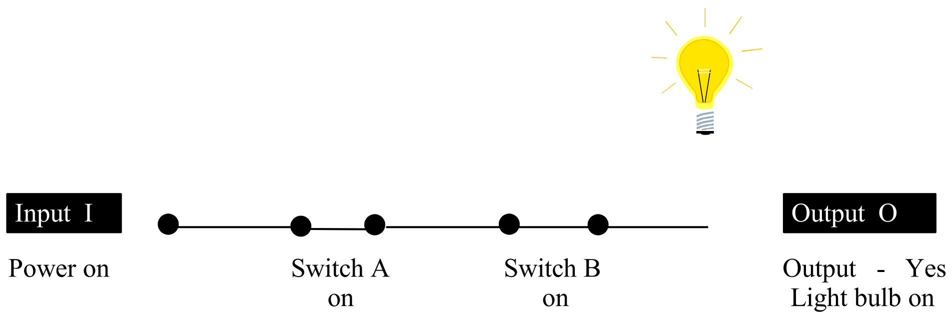
Notice that if either switch A or switch B is off, then there is no connection between the input and the output, hence there will be no output. This is illustrated in the following diagrams:




Input I
Output O
Power on
Switch A on
Switch B off
Output - No
Light bulb off

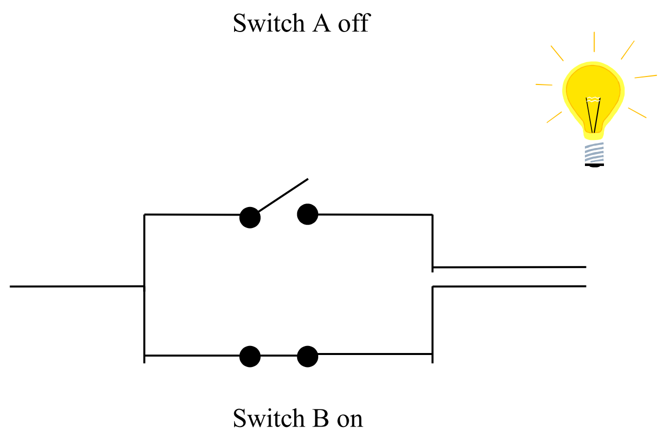
The AND gate

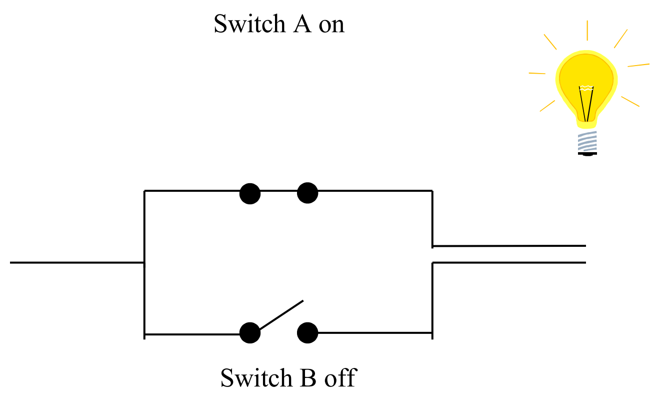
An electrical circuit of this type is known as a logic circuit. This particular logic circuit is known as an AND gate, because both switch A *and* switch B need to be on for the bulb to light up.

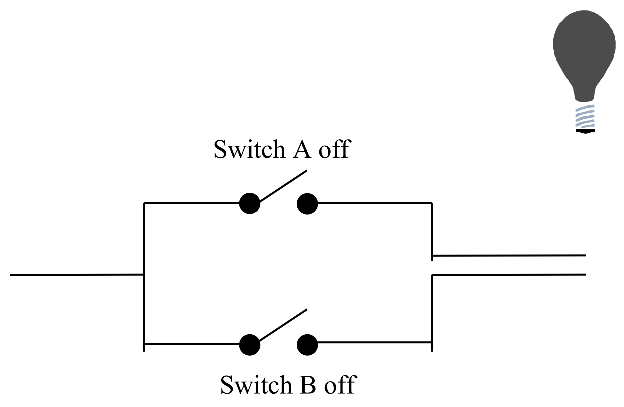


The OR gate

Compare the above circuit with the ones shown below:







The OR gate differs from the AND gate in that there will be an output if either switch A <u>**OR**</u> switch B is on.

Truth Tables

As we have seen, we can express a two state condition in a number of different ways:

	on	or	off
	1	or	0
we could also use	true	or	false

The four diagrams for the AND gate show the four possible combinations for the two switches. This can be represented in truth tables as follows:

<u>The AND gate</u>

Input		Output
A	B	
off	off	off
off	on	off
on	off	off
on	on	on

If on is represented by 1 and off is represented by 0, then these four combinations can be expressed as follows:

<u>The AND gate</u>

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

You can see that there is only an output when both switch A is turned on **<u>AND</u>** switch B is turned on.

The OR gate can be similarly represented in a truth table:

<u>The OR gate</u>

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

You can see that there is an output when either switch A is turned on **<u>OR</u>** switch B is turned on (or both).

Propositions and Logical Expressions

A statement which can be either true or false is known as a *proposition*.

The following statement is an example of a proposition: 'there is water in the kettle'. This statement is either true or false. There is either water in the kettle or there isn't. Another example of a proposition is: 'the water is boiling'. You can build a logical expression from joining propositions together. For example, consider the following:

'there is water in the kettle' AND 'the water is boiling'

Programming Languages

To start programming, you need to select the most appropriate language for the job. Languages differ in many ways - some are cryptic in appearance, some are more English-like and readable, some are more suitable for text processing than numerical analysis. Whatever their appearance or design, all programming languages are based on the binary numbering system.

In this section, the various types of programming language are described, with examples of programming code.

Low Level Languages

Low level languages are computer programming languages which are closely related to the hardware of the computer itself. In other words, low level languages work directly on the bits and bytes of data. Programs written in a low level language perform tasks 10 to 100 times quicker than their high level language equivalent, because little translation of the programming instructions is required.

Machine Language

Machine language is written entirely using bits. The following is an example of a program written in machine code:

```
10000101
11011010
11110110
10101010
01010101
11100001
```

Not very informative is it?

Early computer programmers had to write programs using machine code. It was an extremely tedious process. The ones were often represented by an indicator light which was switched on, the zeros were then represented by the light being off. As each instruction was carried out by the computer, the lights went on and off accordingly.

Imagine how easy it would be to make an error in writing a program using machine language. One bit in the wrong place would cause the computer to perform a completely different task to the one intended, and the result could be disastrous. Imagine for example, the consequences of an error in a program which was controlling the navigation of a space ship!

For these reasons, early computer programmers decided to develop a simpler system of programming which would be less error prone and quicker to enter. The resulting language is known as *assembly language* which uses short sequences of English letters to represent tasks, which can then be translated by the computer into the machine code which the computer can work with.

Aside

You may well ask, 'Why don't computers use decimal, it's a lot easier!'. Good question. Let's turn that question around.

Why don't people use binary to count?

Well, for people it is convenient to count using decimal because we carry the tools to do that with us at all times. Together, our two hands give us a counting frame from 1 to ten with our 8 fingers and 2 thumbs.

The computer is an electronic device. Computers cannot operate without electricity. Imagine the insides of the computer being composed of numerous electric switches. A switch may be in only one of two states, either **on** or **off**. If we represent on as the binary digit 1, and off as the binary digit 0, then a series of switches can be used together to represent any decimal number, each of which can be linked to a task like 'add', 'divide', 'multiply' etc.

Assembly Language

Assembly language is one step up from machine language. Instead of using zeros and ones, assembly language uses short English statements which represent chunks of machine language. A program known as an *assembler* converts the assembly language back to the machine language which can be directly worked on by the computer to carry out the required operations.

Assembly language is a bit more palatable than machine language. An assembly language program is made up of a sequence of two-to-four-letter assembly language commands, often accompanied by an additional statement that represents locations in the computer's memory.

The following is an example of an assembly language statement:

```
Add  R3  R4
```

Which means, add the contents of location R3 to the contents of location R4, and store the result in R4.

A bit more informative than machine language, but it could be better!

High Level Languages

One step up from assembly language are the *high level languages.* Examples of high level languages are BASIC, Pascal, C and C++. These languages take the process of deciphering program code one step further. High level languages provide programming statements which are considerable easier for us to understand. They use English words like 'if', 'then', 'else', 'begin' and 'end'. High level languages were designed in response to the different types of problem to be solved. For example, BASIC (which stands for Beginner's All-purpose Symbolic Instruction Code) is a simple language which was designed to teach basic computer concepts. You could write most programs using any programming language, but it makes sense to select the appropriate language for the job.

Compilers and Interpreters

Of course, before the computer can carry out any command issued in a high level language, they must first be converted back down to the low level language - the zeros and ones of machine code. The programs which perform this conversion task are known as *compilers*. Another type of program which performs a similar conversion is known as an *interpreter.* Compilers convert all the high level code to machine code before the machine code instructions are carried out.

Interpreters work slightly differently in that they take a chunk of high level code at a time. An interpreter converts the first chunk of high level code to machine code, then *runs* the machine code (performs the machine code tasks), then takes the next chunk of high level code, converts that to machine code, then runs that set of machine code. This process continues until all instructions are carried out.

Programs usually run more quickly if they have been compiled in one go, rather than interpreted a chunk at a time.

Source code and Object code

Machine language instructions can be processed by the computer without any further modification. A set of machine language instructions (or machine code) is called the *object program*. A computer program is simply a collection of computer instructions to perform a specified task.

Instructions written in a high level language (BASIC, Pascal, FORTRAN) need converting to machine code before they can processed by the computer. This set of instructions is known as the *source code*. The process of compilation converts source code into object code.

Beginner's Programming Languages

BASIC and Pascal are the two main programming languages designed for use by beginners.

BASIC

Although it's easy to learn, BASIC is limited in what it can do. A major disadvantage of BASIC is that it allows poor programming practices which can create bad habits that may be hard to correct later. BASIC is also an interpreted (rather than compiled) language, which means that compared to compiled code, programs run much slower.

There are many different versions of BASIC available on the market today. Many have improvements which counter some of the original criticisms. Whatever its faults, for many people BASIC was the first introduction to programming, thus sparking an interest which led to more serious study of the subject. For this reason alone, BASIC will always have an important place in the history of computer programming.

The following is an example of coding in BASIC:

```
LET A = 2
LET B = 4
LET C = A + B
PRINT C
```

Pascal

Pascal is named after the 17th century mathematician Blaise Pascal. Since it is not an acronym (where each letter represents a word) only the first letter is capitalized. Pascal was created with two main objectives: 1) To provide a teaching language that would bring out concepts common to all programming languages, and 2) To define a standard language which would be cheap and easy to implement on any computer.

Pascal is a compiled language, and hence a program written in Pascal will run faster than say, the same program written in BASIC. Pascal is easy to learn, promotes good programming practices and hence provides an excellent base for learning other computer programming languages

The following is an example of a Pascal program:

```
program example1 (input, output);
   {reads and prints the value of a variable}
var   yearborn:  integer;
   {this declares a variable called yearborn}
begin
   writeln  (‘Enter the year of your birth’);
   readln (yearborn);
   write (‘You were born in ‘);
   writeln (yearborn)
end.
```

Scientific Programming Languages

FORTRAN

FORTRAN is one of the earliest programming languages. It was designed, as its name suggests (FORmula TRANslator) for scientific and mathematical applications. A source program in FORTRAN is written using a combination of algebraic formulae and English statements of a standard but readable form. The following line is an example of a program statement in the FORTRAN language:

```
RESULT = (B + SQRT(C))  /  (2*A)
```

System Programming Languages

C and C++

C and C++ are programming languages which were designed for writing computer systems such as operating systems, compilers and assemblers. C and C++ are Pascal-like in design, and like Pascal, are compiled languages. They are very economical languages in that commands are terse to the point of being obscure, but take up the least amount of computer storage space and memory. Because C was designed as a systems programming language, it is able to cope with complex problems with minimal code.

This is an example of a program written in C:

```
/*  Copyright  A H DAWSON 1996 */

#include <stdio.h>

main()
{
   float feet, metres, centimetres;

   printf("Enter the number of feet to be converted to metres: ");
   scanf("%f",&feet);

   while(feet > 0) {
        centimetres = feet * 12 * 2.54;
        metres = centimetres/100;
        printf("%8.2f feet equals \n", feet);
        printf("%8.2f metres equals \n", metres);
        printf("%8.2f centimetres equals \n", centimetres);
        printf("\nEnter another value to be converted (0 ends program): ");
        scanf ("%f",&feet);
   }
   printf(" End of program");

   return(0);
}
```

C++ is a superset of the C language. C++ retains all of C's strengths of efficiency and economy, and also provides object-oriented programming. Object-oriented programming makes use of abstract data objects which can be manipulated in a manner defined by the programmer. For example, a database which might be represented by multiple arrays of data in a scientific language like Fortran, could be declared as a simple object, (*student_database*) in C++. The *student_database* object can receive messages to *add_student*, *delete_student* or *display_student* information contained within it. Inserting a new record into the student_database object becomes as simple as this:

```
student_database.add_student (new_recruit)
```

Business Programming Languages

COBOL

COBOL (Common Business Oriented Language) was developed as a standard for business computing. Many COBOL commands are designed specifically for payroll and accounting tasks. The following line is an example of a program statement in the COBOL language:

```
ADD OVERTIME TO NORMAL-HOURS
READ INPUT-X AT END GO TO FINAL
WRITE LINE-5 AFTER ADVANCING 2 LINES
```

Compared to other programming languages, COBOL is quite verbose. COBOL programs which perform even the most trivial tasks can be quite lengthy.

Data Storage

Information which is used by a program to perform a task can be stored in one of two ways depending on how it is to be used.   Storing data as a *variable* allows for the value to be changed. Data stored as a *constant* is intended to remain at the same value throughout the duration of the program run. These two types of data storage are explained in more detail in the following sections.

Variables

Variables are used to represent storage locations in the computer.   Each variable is given a name to distinguish it from other variables. When a variable is named within a program, it is also given a data type. The data type associated with a variable name is one of the types allowed by the programming language being used.   The four main data types are: integer, real, character and boolean.

Imagine a variable as a mailbox. When it is given a name and type, data of that type can be stored within it.

The following is an illustration of a variable named 'Salary', of real number data type, with a value of 2500.00.

Salary   (type: real)

$\boxed{2500.00}$

The following is an illustration of a variable named 'Grade', of character data type with a value 'A'.

Grade (type: character)

$\boxed{}$

A

It is important to realize that computer memory can only hold *bits*. The bits *represent* the data to be stored. For example, the character 'A' corresponds to the binary number 01000000 which in decimal is the number 64 - the ASCII code for the character 'A'.

The amount of storage space required to hold the data depends on the data type:

- Boolean data can be represented by one bit: either a 1 for TRUE or a 0 for FALSE.
- Character data ('a', 'b' or 'C' etc.) may be stored in one byte (8 bits) of memory
- Real and integer numbers may take multiple bytes of memory

Constants

A constant is a value in a program that cannot change during the program's execution. A constant value is stored in a memory location in the same way as a variable value. Again, the size of the memory location will depend on the type of data to be stored.

The values which would be stored as constants rather than variables are values such as π (3.142), hours in a day (24), seconds in a minute (60) etc. These values are fixed, and although technically, they could be stored as variables, it is safer to store these values as constants, since the programming language will not allow any unintentional changes to be made to values stored as constants during the execution of the program.

Data Processing

Essentially there are three stages in the processing of data by a computer program. Any useful program will have *input*, *processing* and *output*.

All programs are made up of a series of statements - where a *statement* is a single command. Statements are used for assigning values to variables, controlling input and output as well as processing information in a program. Examples of input, output and processing statements are shown in the next sections.

Input

Input is the process of getting data into your program. The most common means of inputting data are via the keyboard or from the disk drives. There are numerous other methods of obtaining input data such as use of text scanners, touch screens and other peripherals. Data may also be obtained via another computer on a local or global network.

Inputted data is stored in memory in the computer. Generally, data is stored in units of one byte as binary numbers. Each memory location has an address so that the data contained within it can be referred to whenever required. However, unless permanently stored on the computer's floppy or hard disk, data stored in memory will be lost when the computer is switched off. This does not mean that the memory locations are empty of data when the machine is first switched on. It is up to the programmer to ensure that, when the contents of a memory location are used, the contents are meaningful in the current context.

Processing

Once data has been inputted into the computer's storage locations (memory), it can then be used by the program in a manner specified by the programmer in the program code itself. For example, the following piece of BASIC code assigns integer values to variables named A and B, the calculates their sum in a variable named C.

```
LET A = 10
LET B = 20
LET C = A + B
```

The integer value 30 will be stored in the variable C. If this program was typed in to a computer which supports the BASIC language, and was then executed by issuing the RUN command, each line of the program would be executed in sequence. The first line of the program results in the allocation of computer memory to allow for the storage of an integer value. The computer memory allocated is represented by the variable name A. Once the memory location has been reserved, the integer value 10 is stored at this memory location as the 8 bit binary number: 00001010 (decimal 10).

```
LET A = 10
```

In a similar manner, the second line of the BASIC program would result in the allocation of computer memory to allow for the storage of the integer value 20 (binary 00010100).

```
LET B = 20
```

The third line of the program instructs the computer to take the value of variable A, add it to the value of variable B, then store the result in a memory location to be named C.

```
LET C = A + B
```

After the 3 line BASIC program has run.....

the memory location known as variable A will contain the binary equivalent of the decimal number 10.

the memory location known as variable B will contain the binary equivalent of the decimal number 20.

the memory location known as variable C will contain the binary equivalent of the decimal number 30.

and that is all!

When the program runs, each statement is carried out exactly as specified. However, because we cannot see the internal memory locations and their contents, we cannot be sure that anything has happened at all.
In order to confirm the contents of a variable, it is necessary to send a copy of the contents to the screen or printer. This process is known as outputting, and is described in the next section.

Output

After input and processing of data, the next logical step is to output the results of the processing to the screen or printer. Every programming language has commands for input, processing and output. In the BASIC language, the command which results in a print to the screen is the word PRINT. Our 3 line BASIC program is expanded here to make use of the PRINT command.

```
LET A = 10
PRINT  A
LET B = 20
PRINT  B
LET C = A + B
PRINT  C
```

When this program is executed by issuing the RUN command, the following output will appear on the computer screen:

```
10
20
30
```

Notice that the command: PRINT A

is the output command to print out to the screen the contents of the memory location which corresponds to the variable named A. This command would not print out the character 'A'.

Program Flow Control

In the previous section, the following BASIC program was used to illustrate the processing and output aspects of computer programming:

```
LET A = 10
PRINT  A
LET B = 20
PRINT  B
LET C = A + B
PRINT  C
```

In this program, each statement is carried out in sequence, one after the other. This is known as *sequential* program control.

There are two main programming structures which control the sequence of execution of program statements. One control structure allows a group of program statements to be executed multiple times - a process known as *looping* - and the other control structure which uses the IF statement - which causes the program to branch off to execute different parts of the program if certain conditions are met. This latter program structure allows *decisions* to be made within a program. The decision and repetition program control structures are discussed in more detail in the following sections.

Decision - the If Statement

The **if** statement performs a branch in a program depending on the outcome of a conditional test. If the test expression is true, the body of the **if** statement executes. If it is false, the statement body is skipped. The following is an example of a simple **if** statement written in the C language:

```
if  (score  <  70)
   grade = ‘F’   /*  Fail */
else
   grade = ‘P’;   /*  Pass */
```

This piece of code can be verbalized as:

If the score is less than 70, then the grade is F (for fail), otherwise the grade is P (for pass).

Let’s look at this line of code in detail. The single line of code shown above could equally well have been written as follows:

```
if  (score  <  70) grade = ‘F’   /*  Fail */ else grade = ‘P’;   /*  Pass */
```

or even as:

```
if(score<70) grade=‘F’/*Fail*/ else grade=‘P’;/*Pass*/
```

The computer would have interpreted each version of the statement in exactly the same way. The first version is clearly easier for the programmer to read. So let’s get back to that version.

```
if  (score  >  70)
   grade = ‘P’;   /*  Pass */
else
   grade = ‘F’;   /*  Fail */
```

This section of code is actually a single if statement. Indentation is used to clarify the meaning of the code. Programming style will be covered in more detail later.

The **if** statement takes the general form:

If (expression) statement

or

If (expression) statement **else** statement

The If statement is also known as a *selection statement* because a different route through the program is selected depending on the outcome of the test expression.

In this case, the test expression is whether the score is greater than (>) 70. The greater than symbol (>) is known as a *relational operator*.

Relational operators relate one *operand* to another. Relational operators are described in detail in the following section. In our example line of code, the operands are the variable 'score' and the constant value 70. This line of code has two variables: score and grade. Score is an integer variable. Grade is a character variable which takes on the character value 'P' or 'F', depending on whether the value of score is above 70 or not.

```
if  (score  >  70)
   grade = 'P';   /*  Pass */
else
   grade = 'F';   /*  Fail */
```

This piece of code also shows the use of *comments*. A comment is added to program code to clarify parts of the code for the reader. Comments are completely ignored when the program is executed by the computer. In the C language, the characters /* begins a comment and the characters */ ends it.

Note the use of the semi-colon in C to terminate a statement.

Relational Operators

In the C language, there are six relational operators. These are listed below with their meaning.

Relational Operators

Operator	**Meaning**
<	**less than**
>	**greater than**
<=	**less than or equal to**
>=	**greater than our equal to**
==	**equal to**
!=	**not equal to**

Repetition - the For and While statements

A **for** loop is used to execute a statement or block of statements a specified number of times. If it is necessary to repeat a piece of code for a known number of times, then the **for** statement would be used.
If the number of times that the code is to be repeated depends on a condition, then a **while** statement would be used.

The For statement

The **for** statement uses an index variable to keep track of the number of times that the loop has been run. In constructing a **for** loop, you first initialize the index variable, and then set a limit that will cause the program to exit the loop. The general format for the **for** statement in C is shown below. This format allows for a single statement to be executed each time through the loop.

```
for  ( init ;  test  ;  increment )
     statement;
```

An example of this type of statement in the C language is shown below

```
for ( i  = 0;   i  <  11;  i  =  i + 1 )
     printf ( "i = %d \n" i);
```

The index i is assigned an initial value of 0 with the assignment statement: i = 0
The initialization is done only once.

The test statement: i < 11
is executed before each iteration (repeat) of the loop. If this test statement evaluates to true, (i.e. the value of i is less than 11), then the loop continues to execute.

The increment statement: i = i + 1
is executed at the end of the statement (or block of statements) within the loop. (Note that in the C language, the statement: i = i + 1 can be replaced by the single increment operator i++).

The statement which is carried out for each iteration of the loop is:

```
printf ( "i  = %d  \n", i);
```

This needs a little bit of explanation. The printf statement in this form causes any output to be printed to the screen - this is called *standard* output. The data to be output to the screen is shown between the quotemarks. The %d \n is a format string which directs the computer to print out the value i as an integer (%d) followed by a newline (\n).

```
for ( i  = 0;   i  <  11;  i  =  i + 1 )
      printf ( "i = %d \n" i);
```

On running the section of code above, the following output will be obtained:

```
i  =  0
i  =  1
i  =  2
i  =  3
i  =  4
i  =  5
i  =  6
i  =  7
i  =  8
i  =  9
i  =  10
```

The While statement

There are several types of **while** statement in C. Consider the following variations:

```
do
   statement;
while (condition);
```

The above variation of the while loop executes the single statement at least once. After the first iteration, the condition is tested. If found to be true, the loop is repeated. The loop repeats until the condition equates to false, in which case the next statement to be performed will be the one following the last semi-colon which marks the end of the while statement.

```
do
{
   statement1;
   statement2;
}
while (condition);
```

The second version of the while loop performs in exactly the same way as the first version except that, instead of a single statement executed at each iteration, two statements are executed. The two statements are enclosed within braces {} and are considered as a single compound statement.

A third version of the while statement in C is shown below:

```
while (condition)
   statement;
```

For multiple target statements, the format looks like this:

```
while (condition)
{
   statement1;
   statement2;
}
```

In this form of the while statement, the condition is evaluated before the loop is executed. If the condition evaluates to false, then the statement or block of statements is never executed. This is an important difference between the two versions of the while command. If the condition evaluates to true, then the statement or block of statements is executed, and the program loops back to test the condition. If the condition evaluates to true, the statement block is executed once again, and the program returns to the top of the loop again to test the condition. If the condition evaluates to false, the statement block is then skipped, and control will continue from the next statement in the program.

Program Design Methodologies

The last, but not least aspect of programming to be covered in this course is that of program design. A good program design is crucial to the smooth and efficient running of a program. A good design allows for future enhancements to the program with the least amount of effort.

This chapter will outline some of the methods used in program design. Although several different methodologies exist, by following the advice given in this chapter, you can avoid many of the pitfalls experienced by new programmers.

The Stages of Program Development

The practice of software engineering is concerned with building large and complex software systems in a cost-effective way. The techniques and methodologies proposed by software engineers for good software design can also be used on small projects to good effect.

In his book on the subject of software engineering, Sommerville [Som 87] described the software life cycle as made up of the following phases:

1. Requirements analysis and definition
2. System and software design
3. Implementation and unit testing
4. System testing
5. Operation and maintenance

Sommerville points out that for management purposes it is useful to consider the phases of the software life cycle to be distinct but, in practice, the development stages overlap and feed information to each other. This is illustrated in Fig. 10.1.

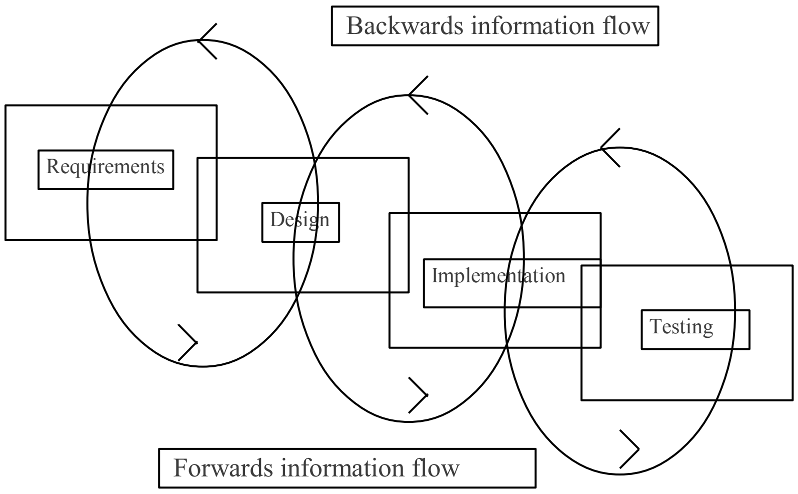


Figure 10.1 The software development cycle

For simplicity the final life cycle phase (operation and maintenance) has been left out of Fig. 10.1 as, during this phase, information is fed back to all previous life cycle phases. This information feedback of the final phase results from maintenance activity which may involve changes in the requirements, design or implementation, or may highlight the need for further system testing.

The system testing phase of the software life cycle represents the ultimate validation stage in the development cycle. At this stage, the system developer must convince the system user that his or her requirements have been met by the system.

In his book, Sommerville quotes the work of Boehm [Boe 81] who gave figures for the relative costs of each life cycle phase. These figures are reproduced in Table 10.1.

Table 10.1 Relative costs of software systems

System Type	Phase costs (%)		
	Requirements/Design	Implementation	Testing
Command/control system	46	20	34
Spaceborne system	34	20	46
Operating system	33	17	50
Scientific system	44	26	30
Business system	44	28	28

The inference which was drawn from the above figures was that the software development costs are greatest at the beginning and at the end of the development cycle. As Sommerville pointed out, this suggests that a reduction in overall software development costs is best accomplished by more effective software design and testing. What is not shown in the table is the cost of the final life cycle phase - operation and maintenance, where maintenance is the process of correcting errors in a system and modifying a system to reflect environmental changes. Sommerville explains that for most large, long-lived software systems, maintenance costs normally exceed development costs by factors which range from two to four. Indeed, he quotes Boehm's example of an avionics system where the development cost was $30 per instruction, but the maintenance cost was $4000 per instruction [Boe 75]. Sommerville goes on to explain that the majority of software maintenance costs result not from system errors, but from changing system requirements, and therefore, to reduce maintenance costs and hence reduce total software life cycle costs, a more accurate picture of the user's real requirements must be established. He explains that because of this emphasis on user requirements, some workers have suggested that the life cycle model should be scrapped and replaced by a more evolutionary model of software development [McC 82] & [Gla 82].

The evolutionary model of software development is based on the idea that the user ought to be presented with a prototype system for experimentation as quickly as possible. The prototype should be built in such a way that it may readily be re-implemented as required. The process of improvement and user validation continues until the user is satisfied with the delivered system. However, Sommerville quotes Boehm who explains that, for systems which are based on existing manual systems the requirements of which are reasonably well understood, the evolutionary approach is less likely to be cost effective, and he suggests that a mix of approaches might be the best overall solution [Boe 84]. The derivation of new software development models is currently an area of active research, but in the light of present knowledge it seems unlikely that one development model will be appropriate for all types of software system.

It is clear from the above that software maintenance plays a very important role in software development. Indeed, the process of system software change which occurs during maintenance is a subject that has been considered important enough to be given its own title of 'Software Evolution' and is discussed in an important paper by Lehman [Leh 80]. Lehman suggests that the evolution of a software system is subject to a number of 'laws' which he derived from experimental observations of a number of large software systems [Leh 76]. The first, and perhaps the most important of Lehman's laws, 'the law of continuing change' is applicable to small as well as to large systems and states that:

"A program that is used in a real-world environment must change or become less useful in that environment"

In other words, as the environment changes or becomes more fully understood, the software system must either adapt to these changes or become progressively less useful until, ultimately, it must be discarded.   In order to adapt to environmental changes, a program must be maintainable. Hence, maintainability along with reliability are considered to be the most important attributes of a well engineered software system.

Software Design and Implementation

Once a problem has been analyzed and defined in terms of the user's requirements, the software solution must be designed and then implemented. Many methods for software design have been proposed [Pet 80] & [Bla 83], and most involve structured programming techniques [Dah 72]. Software design methods generally fall into one of the following categories:

Software design methods

1. top down functional decomposition design
2. object-oriented design
3. data-driven design

Arguably, top down functional decomposition was the first truly systematic method of program design. It is associated with the names of Dijkstra and Wirth, and evolved hand-in-hand with the ideas of structured programming in the 1960s. Top down functional decomposition has been widely used for both small-scale and large-scale projects in diverse application areas. Because of its general suitability for software design, top down functional decomposition is still the chosen design method for many current software systems. However, with the advent of object-oriented programming languages such as C++, object-oriented design is gaining popularity and may become the predominant design method of the future. Object-oriented design is covered in the course "Introduction to Programming in C++". In this course, the method of top down functional decomposition will be discussed in more detail.

Since top down functional decomposition is based on structured programming techniques, any discussion of the design method must also be accompanied by a description of these techniques. Since the subject of structured programming is well documented (see for example [Dah 72]) only a brief resume of the main topics of the subject is given here.

Aspects of Structured Programming

Structured programs exhibit:

- modularity
- readability
- abstraction
- simplicity
- appropriate use of control structures
- top down development

Each of these programming features is discussed as follows.

Modularity

A maintainable program is a program composed of modules (procedures and functions) which are highly *cohesive* and loosely *coupled*. A module is said to exhibit a high degree of cohesion if the elements in that module exhibit a high degree of functional relatedness. This means that each element in the program module should be essential for that module to achieve its function. Coupling is an indication of the strength of interconnections between program modules. Highly coupled systems have strong interconnections with modules dependent on each other, whereas loosely coupled systems are made up of modules which are independent or almost independent. The obvious advantages of highly cohesive, loosely coupled software systems is that any module can be replaced by an equivalent module without adversely affecting the rest of the system.

Module independence is also enhanced by 'information hiding', that is, the concealing of information that is not required by a module. This is mainly achieved by control of data objects through scoping, that is, restricting the realm of visibility of data objects by making local declarations wherever possible.

Readability

The objects in a program such as constants, variables, procedures, functions and types represent real-world entities. Moreover, the action of an object in a program reflects the action of an entity in the real world. Hence, the names of objects in a program should be closely related to or, if practical, be identical to the names of the real-world entities which are modeled. For example, the variable identifier 'numberofobservers' might represent the number of observers in a visual assessment experiment, or the procedure identifier 'calcSD' might represent the procedure of calculating the standard deviation of a number of values. The use of meaningful identifier names certainly aids the understanding of a program, but this can be augmented by the prudent use of comments in the code and of a meaningful program layout. Guidelines for program layout are given by Sommerville ([Som 87] pp 118 - 120), and include the liberal use of blank lines, paragraphing to highlight separate blocks of the program and indentation to highlight statements executed within a loop.

Abstraction

Abstraction is probably the most important feature of structured programming. Abstraction allows a computer programmer (or indeed program designer) to work in a familiar environment, that is, one in which the objects and rules are fully understood. For instance, rather than bothering about how the computer is going is going to do it, the programmer need only consider details of exactly what is required.

A program may be expressed at a high level of abstraction, for instance:

```
Input the values
Add the values
Find the average
Output the result
```

A structured programming language has facilities to express actions and data in an abstract way such as this, but it stands to reason that abstraction is not possible without the qualities of modularity and readability.

Simplicity

Simple programs are easy to understand and hence are easy to maintain and adapt. Bell et al in their book on the programming approach to software engineering [Bel 87] say:

"...a simple program is more likely to work quickly and then go on working after it is put into service."

The importance of simplicity in program code is often underestimated, and no doubt its absence contributes to the misunderstanding of program actions.

Appropriate use of control structures

Control structures should be used so that flow of control is strictly top down. For example, if you consider a loop as a single compound statement, execution should commence with the first program statement, each of the following statements should be executed in turn, and execution should end with the last statement. Program units, loops and decision statements would have a single point of entry and exit. Strictly, this precludes the use of conditional and unconditional goto statements except in exceptional circumstances [Knu 74].

Another control structure whose misuse can lead to programs which are difficult to understand is the if_then_else two armed conditional statement. If such statements are deeply nested, it can become very difficult to follow the flow of control and to determine which 'else' belongs to which 'if'. Nested if_then_else statements may be avoided by using multiple conditions instead. For example,

```
if C1 then                        if C1 then A1;
    A1           becomes          if C2 then A2;
else
    if C2 then
        A2;
```

the latter being less efficient, but shorter and more understandable.

Top down development

Top down development is related to top down functional decomposition which is described in section 10.5.

Programs which incorporate the features of top down development and structured programming have a number of advantages - these are outlined in the following section.

Advantages of Structured Programs

Structured Programs are:

- easy to understand
- easy to test
- easy to debug
- easy to maintain
- reliable

Top Down Functional Decomposition

As its name suggests, top down functional decomposition is a program design method that concentrates on the functions (or actions) that a program must carry out. The design method is based on the idea that the structure of a problem should determine the structure of its software solution. Top down functional decomposition makes use of abstraction by expressing the overall problem solution in terms of simpler actions. For instance, the earlier example of a program to calculate the average of a number of values may be broken down into the following steps:

```
Input the values
Add the values
Find the average
Output the result
```

Each of these steps may be further broken down (or refined) into simpler steps, for instance the step:

```
Input the values
```

may consist of the following:

```
While there are values to enter, do
    Enter a value
```

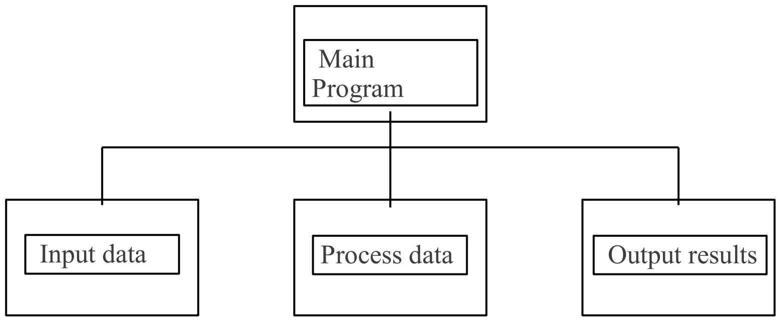
which may be further refined into actual code:

```
while not eof(datfile)
    read(value)
```

The method is a 'top-down' method since it starts with a statement of the overall task of the program, which is broken down in a stepwise manner into simpler functions. Stepwise refinement of functions continues until the solution is expressed in sufficient detail to enable coding to commence. A statement that is to be broken down may be regarded as a procedure call and may be represented on a program structure chart as a rectangular box containing the name of the procedure.

Structure charts are hierarchical diagrams which show the structural relationship of components in a software design. Fig. 10.5 gives an example of a simple structure chart to add a set of numbers.

Figure 10.5 A Simple Structure Chart



The top-level structure chart can be broken down into further structure charts for each module.

Testing and Performance

System testing is the penultimate software life cycle phase which occurs before system operation and maintenance. Testing involves running the system with test data, and later with real data, and comparing the obtained output with the expected output. Anomalies in output data infer errors in the program code.

Testing is the process of detecting program errors, whereas debugging is the process of finding and correcting these errors.

The most popular testing methods available at the present time are top-down and bottom-up testing, so-called because they are based on the corresponding top-down and bottom-up software development methods.

In top-down development, the software system starts off as the single top-most module of the software design which is tested using appropriate data. Lower level modules may be represented by stubs - objects which have the same interface as the module but are very much simpler.

Once the top-most module has been tested, a module from the next level is coded and further testing is carried out. The process continues until all modules have been integrated into the system. If at any stage in this process an error is detected, then the fault must be confined to the latest added module or its interface with the higher level modules. This greatly simplifies the process of locating the fault.

Another advantage of top-down testing is that a working system, albeit limited, is available at an early stage in the development of the software. Not only does this boost the morale of the software developer, it also demonstrates the feasibility of the system to the originator.

And finally. . .

This **Introduction to Programming** course was designed to provide a good general background for those wishing to embark upon a study of computer programming. The information included in this course is general enough to be relevant to a study of any computer language. A thorough understanding of the concepts introduced here will ease the path of the new computer programmer coming across language manuals for the first time.

This document is under constant review, and any suggestions for improvement would be gratefully received.

References

Bell,D., Morrey,I., Pugh,J. (1987) Software Engineering - A Programming Approach, Prentice-Hall, p29

Blank,J., Krijger,M.J. eds. (1983) Software Engineering: Methods and Techniques: Wiley-Interscience

Boehm,B.W. (1975) Practical Strategies for Developing Large Software Systems, ed. E.Horowitz, Addison-Wesley

Boehm,B.W. (1981) Software Engineering Economics, Prentice Hall

Boehm,B.W. (1984) Proc. 1st Process Workshop, Egham, UK

Dahl,O.J., Dijkstra,E.W., Hoare,C.A.R. (1972) Structured Programming, Academic Press

Gladden,G.R. (1982) ACM Software Engineering Notes, 7(2), 35-39

Knuth,D.E. (1974) Computing Surveys, 6(4), 261-301

Lehman,M.M. (1980) Proc.IEEE, 68(9), 1060-76

Lehman,M.M. ,Belady,L.A. (1976) IBM Systems J., 15(3), 225-52

McCracken,D.D., Jackson,M.A. (1982) ACM Software Engineering Notes, 7(2), 29-32

Peters,L.J. (1980) Proc.IEEE, 68(9), 1085-93

Sommerville,I. (1987) Software Engineering, 2nd ed., Addison-Wesley