SESSION 1-1: INTRODUCTION TO PROGRAMMING LANGUAGES

Programming languages are application programs that are used by Computer Programmers and/or Software Engineers in writing sets of instructions that can be read into a computer's memory and later executed on demand. Almost all software programs that you find on any computer are written using one of the Computer Programming languages.

1-1.1 Generation of Programming languages

In the 1930s, the Ananasoff-Berry Computer (ABC) was developed for solving large numbers of simultaneous equations. Shortly after, ENIAC (Electronic Numeric Integrator and Calculator), a military Computer, was invented for general computations. For both computers, only the data were input into the Computer's memory due to the electronic wiring set used. A mathematician, John von Neuman later introduced an alternative to the wiring for storing a computer's instructions in its own memory. Thus, with Neuman's computers both the data and instructions are stored in the Computer's memory. The stored instructions led to the development of multipurpose computers as only the instructions needed to be changed for different problems.

Computer programming languages have undergone dramatic evolution after the invention of the first electronic computers. The first Computer programmers used the machine language to write instructions for the Computer,that is programmers were communicating with their computers by using long chain of zeros and ones. Computers are designed to store information in bits. Each bit is capable of being one of two states. The two possible states are stated in a number of ways such as

True or False
Pass or Fail
On or Off
Yes or No
One or Zero
High or Low
Set or Clear
Charged or Discharged

In most cases, referring to the states above, the left ones (e.g. True) are considered as 1 bit and the right ones (e.g. False) as the 0 bit. As such, only the combination of the digits 1 and 0 are used to represent information in the Computer's memory for processing.

The use of zeros and ones had a number of disadvantages associated with it and this compelled Computer Scientists at that time to invent an easier way of writing computer instructions. Fortunately, the assembly language was invented a few years later. This level of programming requires the use of mnemonics (symbols) in writing Computer programs. Words such as ADD (to mean addition), SUB (to mean subtraction) and DIV (to mean division) were used. This made programming much easier compared to the machine language but since the instructions in assembly language were still not similar to the English language spoken by men, Computer programmers and scientists had to work much harder to come out with what is now known high level languages. These languages made use of instructions similar to the way instructions are given in English and in Mathematics. For example to assign the value 20 to a variable x, one could write 'Let x=20' or simply 'x=20'.

Today, programming languages fall under five major groups namely Visual Languages, Natural Languages, High level Languages, Assembly language and Machine language. Each of these languages has its own advantages and disadvantages with respect to both the Computer and the Computer programmer. The five major groups will be discussed briefly. Programming is the act of writing instructions that needs to be followed by the computer in a certain order as specified by the writer in order to solve a given problem. The writer is called a programmer. Please bear in mind that the Computer is a machine and therefore behaves like any ordinary machine that cannot make decisions on its own. Whatever it does is under the instructions of the Computer programmer or a program written by a programmer. This is the reason why in this course we are going to learn how to give the Computer precise instructions so that it can come out with the output that we expect or do exactly what we want.

All computer programs basically do the same thing. They instruct a computer to accept input data, to manipulate or process the data, and to produce the required output. As such, all programming languages provide the capabilities for carrying out these operations. You may be wondering why there are different programming languages if they provide the same features. The answer is there are differences in the types of data, the calculations needed and the nature of reports to be generated. Scientific and engineering applications require numerical results to a high degree of accuracy and to a reasonable number of decimal places while business applications usually require whole numbers in representing stock quantities, amounts in a particular currency usually accurate to two decimal places, etc. Some of the programming languages were invented purposely for teaching and therefore have their own set of requirements. For example, FORTRAN, an acronym for FORmular TRANslation was purposely designed for scientific and engineering problems since they handle a lot of formulae. Another programming language, COBOL, an acronym for COmmon Business Oriented Language , was designed for business or commercial problems. The languages BASIC, an acronym for Beginners All-purpose Symbolic Instruction Code and PASCAL, named after the inventor, were specifically invented for teaching purposes. We now consider the different levels of programming languages.

MACHINE LANGUAGE

Machine language also referred to as the first generation language (1GL) is the native language of the Computer, that is, the Machine language is the language that the Computer has been designed to understand. Although programmers usually write numbers in a decimal code using the digits 0 to 9 and words using the alphabets 'A' to 'Z' or 'a' to 'z', computers are built to store numbers and characters in binary form using the digits 0 and 1. Thus, the language consists of instructions coded in binary and is specific to the particular processor or computer for which it is used. The reason for use of the binary coding is because it is easier to build electronic computers that work with binary codes. A single binary digit is called a bit and eight bits is called a byte. A bit is the smallest addressable unit of the computer. Most computers are designed to store a single character as a byte using their American Standard Code Information Interchange (ASCII) collating sequence. For example, the ASCII value of the letter 'A' is 65 hence 'A' will be stored in a cell of 8 bits (1 byte) as 01000001.

In early days of computers, all programs were written in machine code, whereby each instruction was coded in binary (using only zeros and ones) in accordance with the particular manufacturer's computer. A sequence of numeric codes that instructs the computer to perform a particular task is called a machine language program. A machine language program is also known as the executable. This type of programming was very time consuming, cumbersome and prone to errors because of the binary coding. This also made it difficult to detect errors in the program.

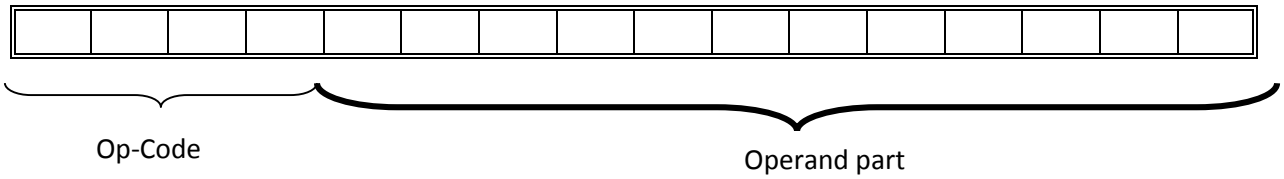
For example, if someone wanted to evaluate the physics expression for calculating speed, v = u + a*t where the v is the speed at time t, u the initial velocity and a is acceleration. Note that in evaluating the expression manually one will have to find the product of a and t first before adding the value of u to obtain the value of v. A programmer will usually write the sequence of evaluation as is follows:

```
…
Load        a
Multiply    t
Add         u
Store       v
…
```

The above instructions will then have to be written using the list of the bit patterns for each of the instructions according to the particular manufacturer's computer on which the instructions are to be executed. The table below defines some instruction bit patterns of a particular Computer manufacturer.

Instruction	Bit patterns	Instruction	Bit patterns
LOAD	0000	MULTIPLY	0011
ADD	0001	SUBTRACT	0100
STORE	0010	COMPARE	0101

Every instruction to be executed by the computer has two parts as follows:



The *Op-Code* (which stands for Operation Code) part is used to specify the instruction bits while the Operation part is used to specify the address of the memory location that contains the required operand or data. Let us assume that for the above instructions the values of u, a and t are stored in memory locations 100, 156 and 200 respectively, then the instructions in memory will be as follows if we also assume that the value of v will be stored at address 180. The italicised bits are the operations; add, multiply, etc.

0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
0	0	1	1	0	0	0	0	1	0	0	1	1	1	0	0
0	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0
0	0	1	0	0	0	0	0	1	0	1	1	0	1	0	0

Imagine you giving the above table to even a university professor who does not know machine language to explain what the instructions mean to you, all he may probably tell you is their decimal equivalents or just say I can see some zeros and ones. Assume you are also told that one of the bits in the third row is wrong, you may take some time to find which bit is incorrect. These were the major problems associated with machine language programming. Note that the list of instruction bit patterns given above is not the same for all computer manufacturers as such this line cannot be executed on all computers since their instruction sets are different. These problems are the reasons why a lot of programmers today prefer not to use the machine language in writing their computer programs.

Usually, different coding systems are used for writing machine language instructions. The common ones are:

ASCII CODE

ASCII, an acronym for American Standard Code for Information Interchange, in which 128 characters are given unique values. These values are then expressed in binary to represent characters usually using a byte. Note that the ASCII value of a character gives the collating sequence or order of the number in the set of the 128 characters. The ASCII value for the digit 7 is 55 and that of the character A is 65. This means that the digit 7 is the 55th character while the alphabet A is the 65th character. The 7 and A are encoded as follows:

55 0110111
65 1000001

EBCDIC CODE

This coding system is used by IBM and there are 256 characters used in this systems with each character having a unique value.

UNICODE

This coding system has a total of 65536 characters with unique numbers. Each character is written using two bytes (that is 16 digits long consisting of only 0s and 1s).

Programs written using other programming languages require either an interpreter, a compiler or an assembler to translate the program instructions to the machine language. The main advantages of the machine language are as follows:

- Machine language programs are the fastest in terms of program execution.
- Requires no translator to translate program instructions.

The disadvantages are as follows:

- Machine languages are machine dependent. As mentioned above, the instructions are written in a way specific to a particular computer for which it is to be used. The instructions cannot be run on a computer produced by a different manufacturer.
- Machine language programs are the most difficult to read and understand because of the long chain of zeros and ones.

- Machine language programs are the most difficult to debug.
- Machine language requires an intensive training by a Software Engineer or Computer Scientist to be able to use it.

ASSEMBLY LANGUAGE: THE SECOND GENERATION

As a result of the inherent problems and difficulties of writing programs in machine code, each computer manufacturer developed an assembly language for their particular models. This language is therefore considered as one level above the machine language. This language enabled programs to be written more easily and speedily. An assembly language enables a programmer to write instructions using mnemonic or symbolic codes instead of the numeric codes. Mnemonics are easy-to-remember names. Thus, assembly language programs are much easier for humans to understand and use. For example, one could write the mnemonic ADD instead of its machine language equivalent of 0101. Although one can say the assembly language is 'better' than the machine language, assembly programs can only be executed on the particular type of computer for which they were written. Programs written in assembly code have to be converted into machine code by a software known as an assembler. The assembly language is also referred to as second generation language (2GL). The following instruction:

v = u + a * t

can be written in a pseudo assembly language as follows:

```
LDA   a
MUL   t
ADD   u
STA   v
```

The above instructions may be taken to mean read into memory (LDA) the value of a, MULtiply what has been read into memory by t and then ADD the value of u and keep (store, STA) the result obtained in v. Compare the above instructions with their equivalents in machine language as given above and note for yourself which of these two languages are easy to use in writing computer programs. Please note that the Machine language and the assembly languages are also called low-level programming languages.

Let us consider a second example of a typicalassembly language program. We will consider a program to find $\sum_{n=1}^{n=3} n$, that is a simple program to find the sum of the numbers 1, 2 and 3. The needed instructions are as follows:

Memory Location	Memory Content	
0	LOAD	N
1	COMPARE	3

2		JUMP_IF_GREATER TO	9
3		ADD	SUM
4		STORE	SUM
5		LOAD	N
6		ADD	1
7		STORE	N
8		GOTO	0
9		STOP	
10	N	1	
11	SUM	0	

The above program uses the memory locations 0 to 9 inclusive for program statements and locations 10 and 11 for data (variables). Please bear in mind that the imaginary CPU has only one register as such only one value can be read into memory at any given time. Note that program execution starts with initialising data locations in memory. First, the N and SUM are initialised to 1 and 0 respectively. The value of N (1) is first LOADed into memory, it is then COMPAREd to 3. If the value of N is found to be greater than 3, program control is transferred to the instruction in location 9 (that is to stop the program as we are adding only 1, 2 and 3). Since the value of N is currently not greater than 3, the value of N is ADDed to SUM and the resultant value STOREd in SUM. The value of N (1) is again LOADed into memory, 1 is then ADDed to it and the result STOREd in N. The location 8 instruction causes control to be transferred back to instruction at location 0. This causes the instructions to be executed again until N attains a value greater than 3.

Assembly language has the following advantages:

Programs written in assembly language execute faster than those written in high level languages discussed below.

- Assembly programs are easier to write, read and understand than those written in Machine language.
- Can be used to solve all practical problems
- Assembly programs are easier to debug than those written in Machine language.

The following are the **disadvantages** associated with Assembly language:

- Assembly programs require an assembler to translate programs to machine language.
- Program execution is slow as compared to programs written in Machine language since instructions have to be translated.

- Assembly programs are difficult to read and understand as compared to High level language programs.
- Assembly programs are also machine dependent. That is, a program written on one kind of Computer would not ran on another kind of Computer. This is because different Computer Manufacturers have different assembly language instruction set for the Computers. For example, the instruction JUMP_IF_GREATER TO (as used in the example above) may cause syntax error if the program is to run on a different manufacturer's computer.

HIGH LEVEL LANGUAGES: THE THIRD GENERATION

High level languages (HLLs) are those languages designed to focus on the problem to be solved rather than on the details of how the computer works or on the internal design of the machine on which the program is to be run. In simple terms, high level languages do not require a Computer programmer to have a detailed understanding of how instructions are executed internally. Also a programmer need not necessary have to know which part of his computer's memory is a particular instruction stored. The Computer, more or less does everything when all the necessary instructions for solving a particular problem are given. In general, the high level languages are meant to further simplify the instructions needed to be coded or written by Computer programmers. Please note that even though it is easier to write programs using high level programming languages, most people still require some reasonable amount of training in order to use higher level programming languages.

A high level programming language is problem oriented and allows instructions to be written in a way familiar to the programmer. Each statement in a high level language generates to a number of machine code instructions. Note that the Computer understands instructions written using only zeros and ones. This will be explained further in due course. The source program (the program written by the programmer) must be written in accordance with a set of rules, referred to as syntax, which indicates the grammar of the language. What it means is that to be able to instruct a computer to solve a given problem, you must provide it with the needed and precise instructions written in a way that a particular program translator can accurately translate the program into machine language for the Computer to carry out the task given. Any deviation from the general rule will force the program translator to display an error or warning message for the programmer to correct a particular instruction. Thus, to be able to instruct the Computer to do something, you must first learn the Computer's language. This is the case with even human beings. For example, if an English man wants to communicate with a Chinese man, then he must first learn the Chinese language so that when he speaks the Chinese man will be able to understand him. Many variants of HLLs exist which prevent portability from one manufacturer’s computer to another. Since there are different high level languages, the rules for one high level language is different from the other. A summary of typical HLLs are as follows:

Procedural languages such as BASIC, COBOL, FORTRAN, C++, Pascal and ADA; Procedural languages specify the exact sequence of all operations for solving a problem. The program logic determines the next instruction to be executed in response to certain conditions and user requests. C++ is the most widely used procedural programming language.

Non-procedural (Declarative) languages e.g. Prolog;

High level languages are also referred to as third generation or implementation languages. A typical high level language instruction is as follows:

TakeHomePay = GrossPay + Allowances - TotalTax

The above instruction can easily be understood as one's take home pay is calculated by adding gross pay and allowances and then deducting total tax.

High level languages have also gone through evolution. What is known to be the first high level programming language to be invented was the FORTRAN. Even though FORTRAN had a number of advantages over the Machine and Assembly languages, some of its instructions were difficult to understand. Consider for example the following statement:

If (DISC) 100, 200, 300

This statement means program control is to be transferred to statement number 100, 200 and 300 if the value of the DISC is less than 0, equal to 0 and greater than 0 respectively. The use of the GO TO statements and other related ones made FORTRAN a bit difficult to understand. FORTRAN compilers also had to translated a source program into assembly language and the assembler then translate the program into binary code (machine executable form). These lead to the invention of another programming language that was much easier to use, that is the BASIC programming language in the 1960s. BASIC was therefore invented as a simplified version of the FORTRAN. It simplified the FORTRAN language in a number of ways. For example BASIC did not distinguish between real and integer numbers, did not restrict the number of variables, the number of letters in a variable name, etc. The BASIC was an interpreted language and BASIC code was not translated into assembly language but rather the interpreter analyses each instruction and if found to be syntactically correct, the instruction is executed.

The main **advantages** of these languages are as follows:

- Programs written in these languages are easy to read and understand.
- Programs are easy to debug as compared to the other programming languages
- Can be used to solve all problems

- These languages are machine independent, that is a program written in a high level language on one Computer can also ran on another Computer of different kind

The **disadvantages** are as follows:

- Programs written in these languages require either a compiler or an interpreter to translate the program instructions into the native language of the Computer.
- Program execution is slow

PROBLEM-ORIENTED LANGUAGES: THE FOURTH GENERATION

As mentioned earlier, third generation programming languages require most programmers to have some training before they can program using the third generation languages. The fourth generation programming languages (4GLs) was developed to reduce the amount of training that one requires in developing computer programs, as such, 4GLs are widely used by nonprogrammers. The 4GLs are problem-oriented and very high programming languages designed to solve specific problems. Unlike 3GLs that focus on procedures and how a problem solution (program) will accomplish a task, the 4GLs are non procedural and focus on specifying what a program is to accomplish.

Most of the fourth Generation Languages are part of a database management systems. Examples are ORACLE, SQL (Structured Query Language).

The 4GLs can be classified into two main groups namely the Query Languages and Application generators. The query languages are those that are part of a database management systems, that allow easily understood query commands to be used in searching databases and to generate reports from databases. A typical example of an SQL command is as follows:
SELECT name FROM employee WHERE yrsWorked>24
This SQL statement is to pick or select the names of all employees (that have their data stored in the employee file and) who have worked for more than 24 years.

The application generators, also known as program coder, are programs that provide modules of already written or prewritten codes that help programmers to quickly create a program by just referencing the appropriate module(s). A typical application generator is Microsoft Access which has a report generation application and Report wizard for creating different types of reports from databases.

Generally, 4GLs are very high level languages which translates user's requests into procedural steps.

NATURAL PROGRAMMING LANGUAGES: THE FIFTH GENERATION

The fifth generation languages (5GLs), also known as Visual programming languages, are languages that were initially taught to be computer languages that incorporates the concepts of artificial intelligence to allow direct human and computer communication but this definition has been expanded to include visual programming languages that provide natural visual interface for program development by providing icons, menus and drawing tools for different controls in generating program codes. The 5GLs generally allow computers to learn and to apply new information just as we humans do. Typical examples of 5GLs are Visual C++, Visual C#, VISUAL BASIC, etc. For these languages, programs are not procedural, that is they do not follow a sequential logic. Here, programmers do not control and determine the order of program execution but rather program users either press keys or click different buttons for specific actions to be taken. In fact, each of the action cause an event to occur.

1-1.2 Program Translators

Once a program is written it must be entered into the Computer using a text editor and then stored on file. Since the program is not in the language that the Computer has been designed to understand, there is the need to have a program translator or compiler. A program translator may be defined as any computer package capable of translating a source program written in either a high level language or an assembly language into a machine language. As mentioned earlier, to date, the Computer only understands instructions written using zeros and ones. Apart from the machine language, the other programming languages do not use only zeros and ones. This means that whenever one writes his or her own Computer programs, there is the need to have the instructions translated into zeros and ones so that the Computer can understand. A translator capable of translating symbols and numbers into zeros and ones are therefore needed. Fortunately, early programmers have designed a number of translators to translate the programs that we write into machine language. Presently, we have Computer programmers who continue to write better translators or improve on the existing translators. Since these translators are computer programs, they are usually referred to as program translators. There are three types of program translators namely (1) Compilers, (2) Interpreters and (3) Assemblers.

INTERPRETER

Most of the early microcomputers were programmed in BASIC language and equipped with an interpreter (BASIC interpreter) stored in a Read Only Memory (ROM) chip. Every program instruction is translated or interpreted on each occasion the program is run (not just once initially as with assemblers and compilers). If any statement does not conform to or violates the rules or grammar (syntax) of the language an error message is displayed on the screen. Until this error is corrected, program execution cannot proceed and this has the disadvantage of slowing down the

running of the program. An interpreter, however, allows errors to be corrected more quickly than a compiler which prints a (long) list of errors which have to be corrected as a whole rather than individually before program execution can continue.

ASSEMBLER

This type of software translates and assembles a program written in assembly code into machine (object) code. The assembler translates the symbolic function codes into the equivalent machine operation code; symbolic addresses are allocated actual internal memory locations. After assembling, the object program is retained on a storage device in machine code. Errors are diagnosed during the assembly process. This is a once only process and does not have to be performed each time a program is run as with an interpreter. Assembly programs tend to run faster than interpreted ones, because assembly language is low level and nearer to machine code and fewer instructions are necessary to accomplish a desired result.

COMPILER

This is another translation program for compiling high level language instructions into machine code. A compiler has more capabilities than an assembler because each source program instruction generates a number of machine code instructions. A compiled program may not be so efficient in the time taken to process a task because of the nature of the high level language used, COBOL for instance. This results in more detailed and less direct instructions than those generated by an assembled program written in assembly code. A compiler is usually used together with a program called the linker, capable of linking together the different translated modules. The use of a linker makes it possible for standard functions (provided by the compiler developers) to be added to one's program to make it a complete working program. The output of the linker is the executable version of the source program.

1-1.3 Introduction to Software Engineering

Software Engineering is a term generally used to describe the various processes involved in a software development. The objective of software engineering is to produce software products to be delivered to a customer with the documentation that describes how the software is to be installed and used. Generally, software products fall into two main groups. These are (1) Generic products, stand-alone software that are produced and sold on the open market to any customer who wants to buy and use them. A typical example of such a product is Microsoft word, and (2) Customized products that are commissioned by a particular customer. That is, the software is developed specifically for a particular customer. A typical example may be software designed purposely for the military, immigration or police.

To design a software product, it has to go through a software process. A software process is the set of activities and the corresponding results that will produce a software. These activities are usually carried out by software engineers or Computer programmers. Most of the time, the software engineers use CASE (Computer-Aided Software Engineering, example VISIO) tools in carrying out most of the activities. There are four main processes involved in software process. These are as follows:

Step 1 The Software or program specification:- Software specification is also known as program definition or program analysis. This is the stage where the functionality of the software and the constraints on its operations are defined. Here, the programmer defines the layout of the program and then gets inputs from all stakeholders who are usually the expected users or owners of the software to be developed. Note that this stage is not simply identifying inputs and outputs of a problem as you will normally do with a class exercise or assignment, as the owners of the software usually do not fully understand the problem themselves. For example, you may get someone coming to ask you to computerise his payroll system or Stock Control, pharmacy shop, hospital, etc. In problems of this nature, a number of questions would have to be asked in order to get a full specification of the software. Some of the questions may be the kind of information to be kept on all employees, what salary scales are there, what are the various allowances and deductions are in place, how will the program be used, should the program be user friendly or it will be used by experience computer scientists, in what format are results to be represented, if a number of people are to use the same software for different operations or tasks within an organisation, then what rights should be given to each user, etc. Answers to all and such questions should be obtained before one begins with the development of the software. Generally, this stage can be considered to be made up of the following and should be considered in the order given.

(i) **Program objectives**: This involves defining the problem at hand so as to meet the users requirements. Please bear in mind that programmers do not normally write programs for their own use but rather for others. Thus, it is important to first know from the user his/her objectives for wanting a software to be developed for him/her.

(ii) **Output**: Once a user requirements are known, the next thing is to determine the kind of output or results to be generated by the program. The output should be based on the inputs that the user will be provided to the program.

(iii) **Input Data**: Once the output of a program is known, the next thing to do is to determine the inputs required in order to generate the expected output. Here, it is important to know the source of the input. Some inputs are to be either entered via the keyboard, to be read from a file or are to be generated by the program itself.

(iv) **Processing** : Once the inputs and outputs of a program have been determine the next thing to do is to determine how the inputs can be processed or manipulated in order to generate the required output. This is the stage where a lot of thinking has to be done. In fact, processing requirement is normally one of the most difficult part as most people find it difficult to get the necessary steps required to produce the required result correctly.

(v) **Specification documents**: Once the objectives, inputs, outputs and the processing requirements have been identified, it is important to have these documented.

Step 2 The Software development:- This involves using the software specifications to develop the software such that it does exactly what is expected. The solution can therefore be a complex one. But if a good work is down with respect to the software specification, then this stage can be a bit simple. One method of simplifying the work to be down here is to draw a structure chart. A structure chart shows the various tasks and their relationship to one another. Let us consider the example of students' cumulative weighted average calculations. The structure chart may be given by figure 1.1.

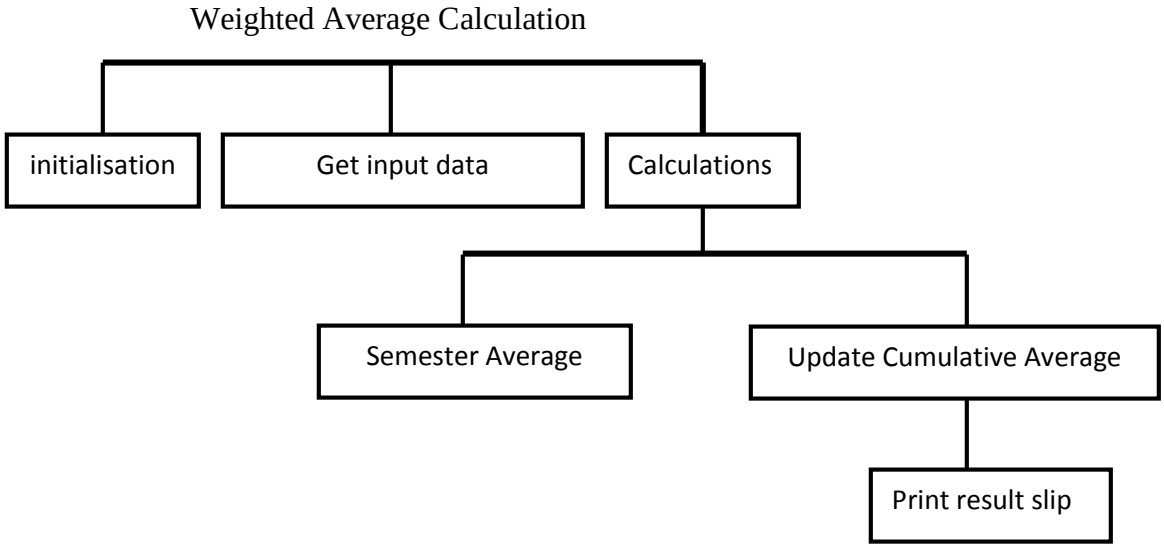


Figure 1.1: A typical structure chart for weighted average calculations

Figure 1 [illegible] relationships. At least it shows that the problem can be broken down into roughly five major tasks, namely variable declarations and initialisations, getting the input data, computing semester average, computing cumulative average and printing of the results. Each task can then be written as a procedure. Initially, one procedure will be called to initialise arrays and variables to be used in a program to zeros as they will be used in summing examination scores, etc. Another procedure will be used to get the current cumulative details, semester examination marks of students and other information as input. Once the inputs are entered calculations can then be performed. The

calculations start with the semester average computations. The semester details are then used to update the cumulative averages and result slips are then printed for the students.

Please note that after a structure chart is drawn, some of the tasks will require further refinement. In the figure 1.1, even though some of the tasks such as 'calculations' have been refined, some may still require further refinement. Successive refinement of tasks should continue until each task is sufficiently simple so that the algorithm can be designed easily.

Note that for very large programs, teams (made up of system analysts and programmers) are usually used for the different tasks as each task may take a couple of days, weeks, months or years to complete. Each of the tasks would have to be coded and tested independently to see if it does exactly what it is supposed to do. Once all the different program modules are doing exactly what is required, they are then combined into one big solution. The combined programs needs to be tested before it is given to the owner or the user. The software development can be seen to be made up of two main stages and these are program design and program coding.

Program design: In program design, normally one designs the solution using a structured programming technique. These techniques may be either of the following:

(i) Top-down design where the main or major steps are broken down into smaller units called program modules.

(ii) Algorithms or Pseudocode where outline of the various instructions needed to solve the problem are written down in a logical order using simple English and mathematical expressions.

(iii) Flowcharts where the outline of the various instructions needed to solve the problem are written in a pictorial or graphical form in a logical manner. In fact, a flowchart is just a pictorial representation of an algorithm or a pseudocode.

(iv) Logic structures where the three control structures are used to specify the sequence of the instructions, the decision making statements and the loop or repetitive structures.

Program code: This is the stage where the program instructions are written in a form that the computer can understand by using one of the Computer programming languages.

Step 3 The Software validation:- This is the stage that the software is validated to see if it does exactly what the customer wants. That is, the software is assembled and then tested for functionality and where necessary modified. Please note that no matter how good a programmer or programmers involved in developing a software may be, there is always the possibility that the software may not do virtually all that is required correctly and is therefore necessary that a validation is carried out to ensure that the program is correct. Getting a well structured program layout does not necessarily mean that the program will always give the correct outputs. Programs may even run without errors and in most cases generate the correct outputs but its only rigor testing with all sorts of inputs may give out some of the hidden problems associated with the software.

This stage is normally carried out by the software developers and later by the software users. Where the software users feel that certain things are not being done correctly, it will be the responsibility of the programmers to sort that out.

Step 4 The Software evolution:- The software needs to be updated periodically to meet the changing needs of the customer. This is basically a software maintenance as programs used in real world are normally used for many years. Generally, when a software is developed, because of the size and the operations of the company for example, only few functionalities may be required at the onset of a company but as the company expands and more operations are needed, the programmers may be required to maintain the program so that all new operations are incorporated. For example, when Microsoft Word (WinWord) was first developed and released onto the market, it did not have most of the functionalities that it offers today. Functionalities such as spell check, grammar check, mail merge, reference, table of contents, building indexes, etc were all missing. These were added as a result of the changing world that requires things to be done with very little input or effort within the shortest possible time. Generally, this steps involves two main stages and these are program documentation and program maintenance.

Program documentation is usually about the description of the program and the procedures used. The documentation is normally used by (i) users as they need to know how the program is used before they can be able to use it. (ii) programmers or the software developer who may have to update the software program from time to time when the needs be. Please note that documentation is not to be done at the end of the software development but rather at every stage of the software development there is the need for this to be done. Some of the documentation are to be done in the program itself whereas the others are in printed documents.

Program maintenance: The program maintenance is done to ensure that the program works correctly, efficiently and effectively. To achieve this (i) it is necessary to locate and correct errors and improve program usability. (ii) the changing need of the organisation as organisations do change over time and as such their operations.

Please note that software maintenance is not done for free as charges made initially are based on users specifications at the time and hence when specifications changes, charges must also change to reflect the efforts required by the changes.

Different software models such as the Waterfall model and the Boehm's spiral model are used in the software process. For this course we shall not go into such details. You are required to read a little more of software process models.

1-1.4 Types of Computer Programming Errors

Any good programmer will agree with us that it is always not possible to write a reasonable length of a Computer program or code without an error. An error here may be defined as anything that can prevent a program from executing smoothly to completion or can prevent a program from producing an expected result. There are basically three types of programming errors namely syntax error, logical error and runtime error.

SYNTAX ERRORS

This type of error is as a result of a violation in the programming rule of a particular programming language. As such, it would be very wrong to use a BASIC compiler to compile a FORTRAN program and vice versa. This is because BASIC and FORTRAN programming languages although they are high level languages have their own set of rules or grammar that a Computer programmer must observed in writing his programs. This type of error normally occurs during Computer coding. Such errors can be identified by either the Computer programmer or the compiler/translator. A Compiler/translator is able to detect such errors during program execution or program compilation. In C++ syntax errors are detected during compilation.

Example, in BASIC, it is valid to write **READ A, B, C** but READS **A,B,C** would result in a syntax error. This is because the Compiler or translator would not be able to identify the main command in the instruction. Similarly, in C++, the equivalent of READ A,B,C is **cin>>A >>B >> C**. Even though C++ uses the read statement, the syntax is completely different. In C++, the read statement has the format read(x,n) where *x* is the name of the variable to hold the data to be entered and n is the number of characters to be entered for x thus in C++ read A, B,C has no meaning and hence why it will result in a syntax error.

LOGICAL ERROR

This type of error is the commonest one. It occurs when there is a mistake in one's reasoning. For example, if in the process of writing a Computer program one is not able to identify or define a problem correctly a logical error will occur. Such errors can only be identified by the programmer and that can be done when a program is tested with a test data. A test data here refers to a data that has been used in solving a given problem and the result or output is known and accepted. If the known result when compared with the output of a program written to solve the same problem are found to be the same then one can say the program is functioning as expected otherwise it is likely to have a logical error.

RUNTIME ERROR

This type of error is also very common. It usually occurs during program execution or runtime. It can be caused by a number of factors. Some of the major causes are

1. an attempt to divide a number by zero, this would result in an overflow. Mathematically, any number divided by 0 is undefined.

2. an attempt to read from a disk or write to a device such as the printer when no printer is connected to one's PC or when the device is not available. This would result in a device time out error.

3. Specifying arguments outside the allowable range of an inbuilt function. For example, the inbuilt function SQR(x) returns the square root of a positive number x. An instruction such as cout <<sqrt(-64) would result in an illegal function call error. The C++ compiler may not indicate there is an error however it will display to the screen -1.#IND and anyone who has been programming in C++ will know that there is something wrong with the output.

4. if an array say X is declared as int X(5), then an expression such as cin >>X[5] would result in an runtime error. Consider the following program. Do not worry if you do not understand what the program is suppose to do at this stage. The program as it stands is correct and will therefore compile without any error messages.

```
#include<iostream>
using namespace std;
int main()
{
        int x[5];
        cin >>x[5];
        cout <<x[5];
        return 0;
}
```

If you key in this program and name it as test then the following screen will be displayed on the screen if you use the visual C++ or the Bloodshed C++ compiler to compile and run the program. The error message is apparently being displayed by the operating system. The error is as a result of line 5. You will understand why when we treat arrays.

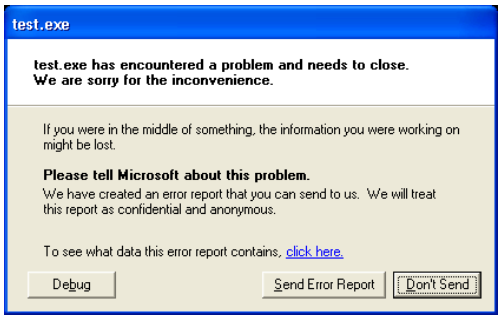


Figure 1.2: An Error Message Screen

Self Assessment 1-1

1. Write briefly on the different generations of Programming Languages

2. Explain why one will prefer a high level programming language to a Machine Language even though the Machine language is known to be the only language that the Computer understands

3. Program translators are Computer programs that convert a programme written in a high level language or the assembly language into a machine language. Briefly describe how each of these program translators work.

4. No matter how good one may be at programming, when a program is compiled or executed there is a high probability that the program may not run at all because there are errors in the program or the program may run but will not produce the expected results. In your own opinion, what are some of these possible errors that can occur when writes a computer program?

SESSION 2-1: INTRODUCTION TO PROGRAMMING IN C++

2-1.1 Creating Executable C++ File

INDENTATION AND COMMENTS

INDENTATION

Indentation is not required by the C++ compiler however indenting a program makes it more readable. Moreover, it makes a program easy to read for analysis and debugging. Indentation may be thought of as a means of structuring one's program such that statements within a particular

block are made to start at the same column. I will advise that you use indentation in all your programs.

COMMENTS

Comments are very useful in computer programs as they help to explain most of the program statements such that programmers not involved in writing a particular program can read, understand and to make corrections if the need be. A programmer may find it difficult to understand his own 'uncommented' program if he should have a look at the same program some years later. It is therefore advisable to add as much comment as possible to your program. Comments can appear anywhere in a program but it is not allowed to place a comment at the beginning or middle of an executable statement. It is alright if the comment follows some executable statements on the same line. To add comment to a program, the following formats are used.

/* comments*/ or // comments

Each of the formats can be used for any comment at any time. However, the pair /* and */ are best used for long comments that spread over a number of lines while the // is best for short or single line comments.

In using the pair /* and */ to spread a comment over several line, the /* is required at the start of the first line of the comments and */ at the end of the last line in the comment. The // can equally be used to spread a comment line over several lines. However, in this case, each comment line should be preceded with //. Wherever a // appears on a line, all characters following it are treated as part of the comment as such any executable statement that comes after the comment and on the same line as the comment will be treated as part of the comment. Since comments are non executable statements, the executable statement on the same line as the comment will not be executed. The same is true when executable statements are placed between the pair /* and */. Also, when a comment contains spelling mistakes or wrong spelling of a reserve word, the compiler will not be flag that as a syntax error nor will it give a warning since strictly speaking comments are skipped by the compiler during program compilation.

Please note that for some compilers nested comments are not allowed if the pair /* and */ are used for comments. For example */* my comment /* your comment */ */* will result in a syntax error. In the case of the use of //, each // is seen as the start of new comment hence with // there is nothing like nested comment.

SPACING: Whenever the enter key is presses a white-space character is created. In C++, whenever extra spaces are left between token they are invincible to the Compiler. This means that programmers are at liberty to leave as many spaces as they may wish between token to enhance

program readability. Note that keywords cannot be entered with spaces between the characters of the keyword. Thus, the keyword ***while*** will result in an error if it is written as *w hile*. With this example, the compiler will display a message indicating that the 'w' is undeclared. Note that a token is te smallest element that a C++ compiler does not brek it down into smaller parts. Thus a token can be a reserved word, a function name, etc.

SEMICOLON: In C++, executable statements appear in the body of functions and they must be terminated by a semicolon. The use of C++ semicolon is just like using the period or full stop at the end of English sentences. It is a terminator. Note that not all C++ statements must have a terminating semicolon. This will be clear to you as we move along.

CASE SENSITIVITY: Unlike programming languages like Pascal which are not case sensitive, the C++ programming language distinguishes between lower- and uppercase characters or letters, that is C++ is case sensitive. In C++ all reserved words must be written in lowercase letters. In naming variables one is at liberty to use either lower- or uppercase or a combination of the cases. For example, the words NetPay, netPay, Netpay and NEtPay are different words to the compiler.

2-1.2 C++ Compiler

As mentioned earlier on, there are a number of different versions of the C++ compilers. All programs in this book were typed and run using either visual C++ or the Bloodshed Dev-C++ compiler. Please note that the Dev-C++ compiler is free and can be downloaded from the following website.

www.bloodshed.net

Even though it is free, it will be a good idea to donate some few dollars to the developer. We shall at this time look at the Bloodshed Dev-C++ as to how we can run our programs. First, you will have to download the compiler from the above website address and install on your Computer's hard disk. In widows operating system environment, the compiler is started in the same way as you would have started any application program such as the Microsoft Word. When the Bloodshed starts you will obtain figure 1.3

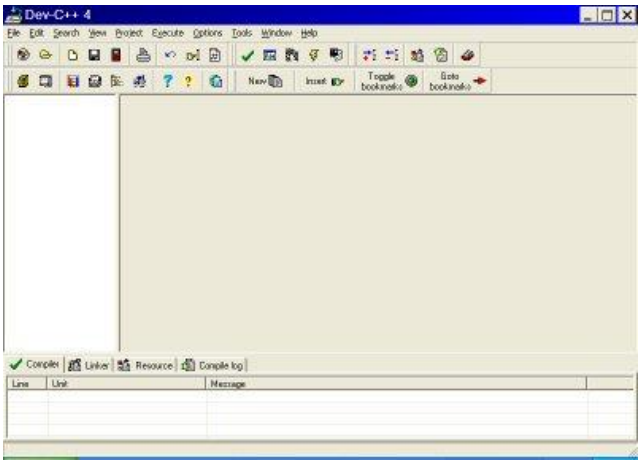


Figure 1.3: Initial Screen of Bloodshed C++ Compiler

and for visual C++ (version 6.0) is as shown in figure 1.4

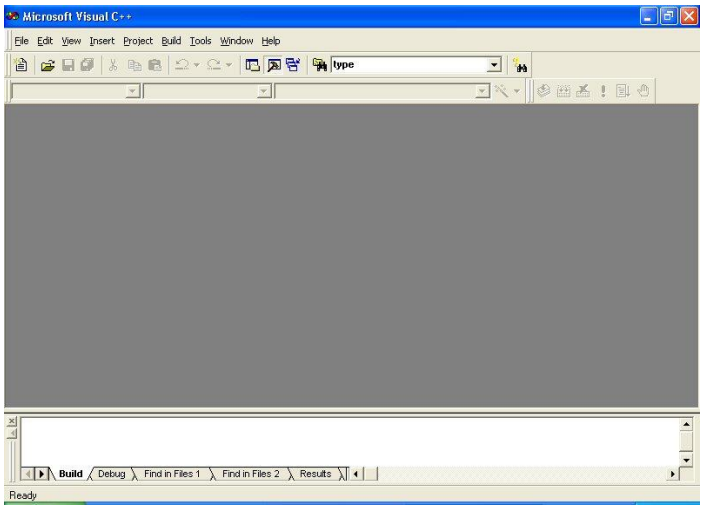


Figure 1.4: Initial Screen of Visual C++ Compiler

C++ Reserved or keyword words

Table 1.1 shows the list of C++ reserved words. These reserved words cannot be used as variable name. Each of these reserved words will be explained in detail where appropriate.

Table 1.1 List of C++ Reserved Words

Asm	Do	Long	sizeof	Union
Auto	double	Mutable	static	Unsigned

Bool	dynamic_cast	namespace	static_cast	Using
Break	else	New	struct	Virtual
Case	enum	Operator	switch	Void
Catch	explicit	Private	template	Volatile
Char	extern	protected	this	wchar_t
Class	false	Public	throw	While
Const	float	Register	true	
const_cast	for	reinterpret_cast	try	
Continue	friend	Return	typedef	
Default	goto	Short	typeid	
Delete	int	Signed	typename	

C++ SYNTAX: Every programming language has a set of rules for writing statements and these rules must be followed strictly by the programming in writing his/her programming with respect to the programming language being used. These rules are known as *syntax rules*. In writing programs in C++, we will use the syntax of the language.

2-1.3 Variables

Most programming languages such as Pascal and C++ require that all variables to be used in a program be declared before their first usage. A variable is a symbolic name used by a programmer in a program to store values or data. For example a variable name can be used for items such as number of students, height, weight and age. Variables used in programming are similar to those used in mathematical equations except that unlike mathematical equations where the variables more or less have a fixed value, variables in programs can have different values at different times if necessary. In mathematics, X and Y are frequently used as variables. You may think of variables as buckets or containers that one can put values in, take the values out or change the values that are in the container or the basket.

Variables are to be given names in a program so that they can be referred to. These names must be meaningful so that their purpose or function can be easy to identify without having to go through an entire program to figure out what they are being used for. For example, to declare a variable to store the hourly rate paid to workers, it will be better to use a name such as HourlyRate or Hourly_rate rather than using H, hhhh or HR. The name of a variable is called an **identifier**. There are some rules that one must follow when naming variables. Below are some of the rules in forming identifiers:

1. The first character must be an English alphabet or the underscore, but it is advisable to use alphabets rather than the underscore.
2. The remaining characters can be alphabets, digits or the underscore.

3. They can be of any length. Some C++ writers tend to give limits to the number of characters that can be used in a variable name but our experience have shown that programmers usually use names that are relatively short and hence such limits are not all that important.
4. They should not contain special characters such as . , ; @ # $ “ etc
5. They should not be reserved words such as iostream and else.

Note that C++ is case sensitive hence the names Rate, RATE and rate are all different so far as the compiler is concerned. It is not advisable to use the same identifiers with just differences in the cases as different variables as such naming system can be problematic. Note that all variables defined or used in a program must be defined with one of C++ data types. We will soon look at the different data types.

Question. Indicate whether or not the following are valid C++ identifiers. If not state why it is not.

(i) X	(ii) base 2	(iii) X_squared	(iv) _Rate
(v) 5Star	(vi) D.O.B	(vii) why?	(viii) interest-rate
(ix) else	(x) oh!	(xi) John	(xii) int

Answer:

(a) The valid ones are X, X_squared, _Rate and John
(b) The else and the int are invalid identifiers as they are reserved words.
(c) The base 2 (space), D.O.B (special character .), why? (special character ?), interest-rate (special character -) and oh! (special character !) contain special characters as shown in brackets after each.
(d) 5star does not start with an alphabet or the underscore.

VARIABLE DECLARATIONS

As mentioned earlier, all variables used in a C++ program must be declared before they are used. By declaring a variable, it tells the compiler the type of data that will be stored in the variable during program execution. Moreover, it enables the compiler to display or indicate an error or give a warning message when an attempt is made at storing a wrong data type into a variable. By declaring variables the compiler gets to know the variables you intend to work with and assign the variables memory locations accordingly. Variables can be declared anywhere within a program but good programmers usually declare variables at the start of the main function or the user defined function in which they are to be used. We will consider declaring variables at the start of the main function or a user function. The syntax for declaring a variable is as follows:

dataType variableList;

where *variableList* is the list of variables of the same data type separated by commas. The *dataType* is one of the valid C++ data types given in table 1.2. Later on, we shall add user defined data types that can also be used as a valid C++ data type.

Table 1.2: The Different Data Types in C++

Data type	Memory (bytes)	Range of values	Precision
short	2	**signed** -32768 to 32767 **unsigned** 0 -65535	-
long int or int	length normally depends on the length of the system's word length. in a 32-bit system such as windows 98, 2000 and nt, the length is 4 bytes.	**signed** -21474836476 to 21474836476 **unsigned** 0 to 4294967295	-
float	4	10^{-38} to 10^{38}	7
double	8	10^{-308} to 10^{308}	15
long double	10	10^{-4932} to 10^{4932}	19
char	1		-
bool	1	true / false	-

The first five data types are used for numeric. However, the *int* and *long int* are for integers while the float, double and long double are for real numbers. Examples of variable declarations are as follows:

```
int No_of_Students, age, Student_Number;
double average_age, stdev, variance;
```

In the above table, precision refers to the number of meaningful digits, consisting of the integer and the decimal parts. The number of bytes is the amount of memory required to store an item. Note that the number of bytes may vary from one system to another. You can know the number of bytes required by a data type by using the sizeof() function. You may key in the following program and run to determine the number of bytes used by each of the data type on your system. Do not worry if you do not understand the programs in this chapter and also do not worry if your system uses different number of bytes to store the different data type from my system.

```
#include<iostream>
using namespace std;
int main()
```

```
{
        int a=1;
        long int b=1;
        float c=1.0;
        double d=1.0;
        long double e=1.0;
        bool z=true;
        char ch='A';
        cout <<"Bytes required by int is " <<sizeof(a) <<endl;
        cout <<"Bytes required by long int is " <<sizeof(b) <<endl;
        cout <<"Bytes required by float is " <<sizeof(c) <<endl;
        cout <<"Bytes required by double is " <<sizeof(d) <<endl;
        cout <<"Bytes required by char is " <<sizeof(ch) <<endl;
        cout <<"Bytes required by long double is " <<sizeof(e) <<endl;
        cout <<"Bytes required by bool is " <<sizeof(z) <<endl;
 return 0;
}
```

Program output: When I ran the above program on my PC, the output obtained is as follows:

```
Bytes required by int is 4
Bytes required by long int is 4
Bytes required by float is 4
Bytes required by double is 8
Bytes required by char is 1
Bytes required by long double is 8
Bytes required by bool is 1
Press any key to continue
```

The following program can also be used to determine the range of values of the various data types of your system:

```
#include<iostream>
#include<limits>
using namespace std;
```

```
int main()
{
        numeric_limits<int> inte;
        numeric_limits<long int> lint;
        numeric_limits<float> flt;
        numeric_limits<double> dbl;
        numeric_limits<long double> ldouble;
        numeric_limits<bool> bbool;
        cout <<"The following are the limits on my system\n";
        cout <<"\n\n\t\tMinimum\t\tMaximum\n";
        cout <<"int         :" <<inte.min() <<"\t" <<inte.max() <<endl;
        cout <<"long int    :" <<lint.min() <<"\t" <<lint.max() <<endl;
        cout <<"float       :" <<flt.min() <<"\t" <<flt.max() <<endl;
        cout <<"double      :" <<dbl.min() <<"\t" <<dbl.max() <<endl;
        cout <<"long double :" <<ldouble.min() <<"\t" <<ldouble.max() <<endl;
        cout <<"Bool        :" <<bbool.min() <<"\t" <<bbool.max() <<endl;
return 0;
}
```

Program output: Running the above program on my computer system gave the following result:
The following are the limits on my system

	Minimum	Maximum
int	:-2147483648	2147483647
long int	:-2147483648	2147483647
float	:1.17549e-038	3.40282e+038
double	:2.22507e-308	1.79769e+308
long double	:2.22507e-308	1.79769e+308
Bool	:0	1

Care must be taken when dealing with char variables. Consider the following program.

```
#include<iostream>
int main()
{
        char ch='A';
```

```
        cout <<ch;
        cout <<ch+1;
        return 0;
}
```

The variable *ch* is initialised to 'A'. The output statement, cout<<ch is to display the content of ch, thus the character A is displayed to the screen. For the next output statement, **cout <<ch+1**, which is a valid statement, one would have expected that the character 'B' will be displayed. Rather, it is the ASCII value of 'B' (66) that is displayed. You should therefore be very careful when using char variables in arithmetic expressions.

Note that most C++ compilers do not allow users to store values of data types different from what a variable has been declared to store as doing this will result in the compiler displaying an error or a warning message such as
"warning C4244: 'initializing' : conversion from 'const double' to 'int', possible loss of data" or "warning C4244: '=' : conversion from 'double' to 'int', possible loss of data"

during compilation. These messages are displayed when during variable initialisation a double data type value is assigned to an int variable or the receiving field is of int type but the right hand side of the assignment is of the type double respectively. For example, if age is of int type then

```
age = 30.50;
```

will result in a warning message being displayed on the screen during program compilation. If this warning is ignored then during program execution, the value stored in age will be 30 and not 30.50 as one would have expected. However, it is possible to change the data type of one variable to another by a process called **type casting**. Type casting should be used whenever real and integer variables or constant are to be used in the same arithmetic expression else final results may not be accurate especially if floats and doubles are being converted to integers. To change a real number to an integer the format is as follows:

```
        [integer_variable =] int(real_variable);
or      [integer_variable =] long int(real_variable);
```

where the *real_variable* is a real variable whose value is being converted to an integer type. Similarly, to change a variable from an integer value to real value by casting the format is as follows

```
[real_variable =] RealDataType(int_variable)
```

where the *RealDataType* is one of the valid real number data type, that is float, double and long double.

Examples of casting

1. double(7)/2 will give 3.5 instead of 3 if the 7 had not been converted to a real number.
2. int(7.5)/2.0 will also give 3.5 instead of 3.75.

EXCEEDING INTEGER RANGE

Generraly, in C++, when a variable is declared to be of the type int it can store values in the range of -32768 to +32767 due to the use of two bytes of memory used for storing integer values. Normally the Compiler treats the range of values as a clock with -32768 as the starting number through to +32767 such that after 32767 the next number is -32768. Now, if the value 32775 is stored in an *int* variable, the compiler will detect that the upper limit has been exceeded by 8. It will therefore count 8 after the number +32767 and that will give -32759. Similarly if an attempt is made to store -32770 then it will be stored as +32765. It is therefore important than when dealing with numbers one should be careful in choosing the appropriate data type for the variable.

The points discussed above holds for all numeric data types.

Question:

What data types would you use to store the following type of information in a program.

(i) The acceleration due to gravity (ii) The number of children one has

(iii) The first character of your surname (iv) The name of your best friend

(v) A question such as 'Do you own a car? (vi) Marital Status

Answers:

int for (ii)
float/double for (i)
char for (iii) and (v)
char (string) for (iv) and (vi)

INITIALISING VARIABLES DURING DECLARATIONS

Variables can be initialised (or assigned values) during the declaration stage. The syntax for this is as follows:

dataType variable1 = value1, variable2 = value2,…;

where *dataType* is one of the C++'s data types. *Variable1*, *variable2*, etc are identifiers and *value1*, *value2*, etc are also constants. An example is as follows:

int count = 0, sum = 0,number_of_students = 100;

It is also possible to declare and initialise variables using the following syntax:

dataType variable1(value1), variable2(value2),…;

where *dataType, variable1, variable2, value1 and value2* have the same meaning as above. The following example declares x and y to be of type int and then initialises them to 10 and 20 respectively.

```
int x (10), y (20);
```

Please note that the use of the open and closing brackets as you may get confused when arrays are introduced later. Arrays use the square brackets. I will advise that you stick to the format,

```
dataType variable1 = value1, variable2 = value2,…;
```

Note that if an uninitialised variable is used in an expression on the right hand side of the assignement operator no error message will be displayed as the C++ compiler cannot detect such type of errors at compile time or runtime error. Such type of errors are up to the programmer to recognise and that can only be done when the results from the program are incorrect. It is therefore very important to always initialise any variable you intend to use in an expression that appears at the right hand side of the replacement sign.

CONSTANT VARIABLES

The declaration above is used when a variable is to store different values of the same data type but at different times during program execution. However, at times, a variable can be declared such that its value cannot be changed. To declare a variable of such data type, the word const is placed in front of the declaration. A variable declared using the const modifier is called a declared constant. The syntax is as follows:

```
const dataType variable1 = constant1, variable2 = constant2…;
```

where *dataType, variable* and *constant* have their same usual meanings. If for example, we want to use the value of Pi, that is 3.14159, we can declare a variable Pi and assign it the value 3.14159 using the const so that wherever the variable Pi appears, the value 3.14159 is substituted. Since we are declaring the variable with a const, it means that its value 3.14159 cannot be changed within the program. Since the value of Pi contains a decimal, we will declare it to be of the data type double. If more or fewer digits of Pi is required, then one can modify the value assigned to Pi in the const declaration. The following declares Pi as a constant.

```
const double Pi = 3.14159;
```

DEFINED CONSTANTS

Defined constants are similar to constant variables. Constant variables are defined using the *const* while for a defined constant the #define directive is used. The #define is used to define names for constant. The general syntax is as follows:

```
#define definedConstant  value;
```

where *definedConstant* is the name to be given to the constant and value is the constant. Consider the following examples.

```
#define integer int;
```

```
#define PI  3.14159;
#define NextLine '\n';
#define tab4 "    ";
```

The first example even though it is syntactically correct, variables cannot be defined to be of the data type integer. The compiler will list variables declared to be of the type integer as undeclared. Since the int is not enclosed in quotes, it is not even a string hence using the cout function to display the content of integer will in error. Even if the int is placed in quotes one cannot still use it to define constants to be of the int type.

The last three lines are all correct. PI is assigned the value 3.14159, whenever *NextLine* is used in a cout statement (e.g. cout << NextLine <<…), the compiler will move to the next row before displaying any information while the variable *tab4* is assigned four white spaces. Note that the name of a defined constant cannot have the constant assigned changed later in the program. As such, the define constant variable cannot appear in an *l-value*. Note also that a defined contant variable name cannot be used to accept input data from the keyboard. In very simple term, this directive tells the compiler to replace all appearances of a defined constant variable in a program by the constant assigned to the variable during declaration.

2-1.4 Scope of Variables

In almost all structured programming there are two main types of variables with regards to their scope. The scope of a variable is the block for which the variable exist and that is the only place the variable can be used. In C++, we have global and local variables. Local variables are variables declared within the body of a function, that is either within the main function or within a user defined function. Local variables are local only to the function within which they are declared and as such the compiler will display an error message such as follows when a local variable is used outside its scope.

```
x c:\program_name.cpp  `localVariable' undeclared (first use this function)
```

where *program_name* is the name of the main program, *x* is the line number where the local variable name appears and *localVariable* is the local variable that is being referenced outside where it was declared.

Global variables are those variables that can be used by either the main program or other functions. They are declared before the int main() in a main program. Global variables are best used when more than one function is to use them.

Note that the same variable name can be declared before the start of the main function as a global variable and within a function as a local. When this happens, the two variables will be treated as being different by the compiler. In functions that do not have the same variable being declared within, references to such variables will assume their current global values. However, those

declared within a function will assume their local values rather than their global values hence care should be taken when a global variable is re-declared in a function.

It must be noted that the value of a global variable can be modified by any function within the main program. Let us consider the following diagram.

The figure shows a program that uses three user functions, namely main, SquareUs and SwapUs.. A function is a list of instructions describing how a particular task can be performed. Functions usually perform only a specific task, however, it can be called a number of times when needed. In the figure to the left, a box represent a function and also shows the scope or boundary of the variables declared within it. The following can be said about the program.

```
#include<iostream.h>
int x, y, z;

int main()
        {
           int b, x;
                :
           return 0;
        }

int SquareUs(int m,int n)
        { int l, d;
          return a*a +d*d;
        }

int SwapUs(int u, int v)
        {
          int y, t, d;
              :
          return y;
        }
```

(i) The variables x, y and z declared before the main function are global variables and can be used in any of the three functions. Since the variable x and y have been redeclared in main and SwapUs respectively, the use of x or y in these functions will imply their local variables. Thus all references to x in the main function are not references to the global variable x but rather the local x.

(ii) The variables b, l and t can be used only in main, SquareUs and SwapUs respectively as they are local variables.

(iii) The variable d in SquareUs and SwapUs are two different variables. They are local to the function in which they are declared. During program execution, they will store different values and will refer to different memory locations.

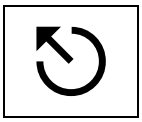
Self Assessment 2-1

1. Explain how a program you have just written can be made an executable file so that the executable file can be copied to another Computer and run without having to put a copy of the compiler on the new computer?

2. State some of the rules that one needs to take into consideration when naming a variable in C++.

3. With the aid of a diagram, briefly explain what is meant by the scope of a variable.

4. Variables can be used to hold all types of data, however, C++ requires that before a variable is used it must first be declared so that the compiler will know the nature of data that will be stored in a variable. Briefly explain the different types of data as well as giving reason(s) why a wrong data type of a variable can pose a a problem.



Unit Summary

1. There are different generations of programming languages but in this course we have considered the generations or type of programming languages as Machine language, Assembly Language, High level programming languages, problem oriented and visual or natural programming languages. In moving from one generation to the next, the main rational was to come out with a programming language that is as close as possible to common languages such as English language spoken by men and also to make programming as easy as possible.

2. Program translators are very important to a computer programmer unless one is using a machine language. Programmers normally use a programming language that is very easy for them to use to write, read and understand computer programs. However, since these languages that can easily be understood by men and not the computer there is the need for a program to convert a programmer's program to the language that the computer understands. As said earlier such programs that are capable of converting a program that is not in a machine language into a machine language are referred to as program translators.

3. Normally when you have finished writing a computer program you will expect that once you attempt to compile and run you will obtain the needed results. This is not always the case as your program is likely to fail the compilation process or to execute. The failure of a program to run is generally due to some errors in the program. We have learnt in the unit that there are basically three types of errors namely the syntax, logic and runtime errors. It is very easy to correct syntax and runtime error but not logic error. This is because the

Computer ca help us to locate syntax and runtime errors and correct them but does not help in any way when there is a logic error. It is therefore important that whenever you have a computer program to write you break the problem into smaller modules, write and test each module before integrating them. By so doing, even if there are logic errors they can easily be identified and corrected.

4. There is no program that you will write to do something very meaningful without using variables. As such variables do play very important roles in programming. Instead of knowing exactly the memory location where a given data is stored, variables simply allow us to refer to the name given to the stored data and not where it is stored in memory. There are different types of data and understanding how each type of variable is treated is very important so as not to loose any vital information. We learnt that when an integer value is divided by an integer value the result will always be an integer. For small numbers a loss of the decimal part may not create a big problem but then one still has to be careful. If you consider the expression ½*1000000, the computer will return a zero in C++ an for that matter in most programming languages, however, if we had made either added just point zero (.0) to the 1 or 2 in the ½ and the 1000000, the result of this expression would have been 500000. It is always advisable to decide very early what type of values you intend to store in variables so as to avoid possible truncation of some decimal parts of an expression.

Key terms/ New Words in Unit

1. Computer Program-a set of logical instructions coded in a particular programming language for solving a specific problem

2. Program Translator-A software program that takes a source program written in either assembly or high level language and translate it into the machine language for the computer to execute.

3. Compiler-This is a program translator capable of generating a machine code equivalent of a source program so that the machine code or executable file can be run without the compiler.

4. Variables-names that are used in a computer program to hold data in memory during program execution.

5. Reserved word-Words that have special meaning to the computer when used in a computer program. For example in C++ words such as if, while, for, etc are reserved or key words.

6. Syntax-this defines the structure of the statement and how the statement should be written for the computer to understand when using a particular programming language.

Unit Assignments 1

1. Explain each of the following:

a) Program b) Bit c) Programming d)Programming language
e) Byte f) Memory g) Variable h) identifier i) program logic

2. write briefly on each of the following

a) The different types of programming languages, their advantages and
disadvantages

b) The different types of translators, their advantages and disadvantages

c) The stages involved in writing a computer program

d) The main stages in Software Engineering

e) The different data types

f) How variables are named in general

g) The three different types of operators discussed in this unit

h) The three different types of expressions discussed in this unit

i) The three different types of programming errors, pointing out how each can be located, identified and corrected.