PROGRAMMING METHODOLOGY

CHAPTER ONE

Overview of Computer Programming Methodology

Chapter Objectives

By the end of this chapter the learner should be able to;

- Describe the steps involved in solving a problem using a computer.
- Be able to represent the Algorithm to solve a problem in flow-charts and Pseudo-code.
- Be able to describe and differentiate the types of programming methods.

1.1 What is Computer Programming Methodology

A Methodology is a system of methods with its orderly and integrated collection of various methods, tools and notations. A computer program is a series of instructions written in the language of the computer which specifies processing operations that the computer is to carry out on data. It is a coded list of instructions that tell" a computer how to perform a set of calculations or operations.

Programming is the process of producing a computer program. Programming involves the following activities; writing a program, compiling the program, running the program, debugging the programs. The whole process is repeated until the program is finished.

1.2. Problem Solving with Computer;

There are a number of concepts of relevance to problem solving using computers. Two particular concepts includes computability and complexity. A problem is said to be computable if it can in principle be performed by a machine. Some mathematical functions are not computable. The complexity of a problem is measured in terms of resources required, time and storage

The steps involved in solving a problem using a computer program includes;

Step 1. Define the Problem: State in the clearest possible terms the problem you wish to solve. It is impossible to write a computer program to solve a problem that has been ambiguously or imprecisely stated.

Step 2. Devise an Algorithm: An algorithm is a step-by-step procedure for solving the problem. Each of the steps must be a simple operation which the computer is capable of doing. A universally-used representation of an algorithm is a flowchart or flow diagram, in which boxes representing procedural steps are connected by arrows indicating the proper sequence of the steps. In many problems you will need to define a mathematical procedure, expressed in strictly numerical terms since the use of computers to do higher level analytic processes such as solving algebraic equations or doing integrals in a non-numerical fashion is relatively limited. The Algorithm can also be represented using Pseudo-code

Step 3. Code the Program: The steps in an algorithm, translated into a series of instructions to the computer, comprise the computer program. There are many languages in which computer programs can be coded, each with its own syntax, vocabulary, and special features.

Step 4. Debug the Program: Most programs of any length don't work properly the first time they are run and must therefore be debugged." Often, during the debugging phase, errors and ambiguities in the original statement of the problem reveal themselves, calling for basic revisions in the solution algorithm.

Step 5. Run the Program: After the program has been fully debugged you run it, possibly using many sets of input data. This step may take anywhere from a few seconds to many hours depending on the complexity of the problem and the speed of the computer.

Step 6. Analyze the Results: Often the output from a computer program requires considerable further analysis. In some cases, even though the program worked perfectly, you may find that you solved the "wrong" problem. There is an acronym well known to computer users: GIGO, which stands for "garbage in, garbage out."

1.2.1. Problem Algorithm

An Algorithm is a logical sequence of discrete steps that describe a complete solution to a given problem in a finite amount of time independently of the software or hardware of the computer. It is the set of rules that define how a particular problem can be solved in finite number of steps. Algorithms are very essential as they instructs the computer what specific steps it needs to perform to carry out a particular task or solve a problem. Every algorithm should have the following five characteristics: Input, Output, Definiteness, Effectiveness and Termination. An Algorithm has the following properties;

- It must be precise and unambiguous
- It must give the correct solution in all cases
- It must eventually end.

Efficiency and Analysis of the Algorithm

The efficiency of an Algorithm means how fast it can produce the correct results for the given problem. The Algorithm efficiency depends upon its time complexity and space complexity. The complexity of an algorithm is a function that provides the running time and space for data, depending on the size provided by us. Two important factors for judging the complexity of an Algorithm are; **space complexity** which refers to the amount of memory required by the algorithm for it execution and generation of the final output and **time Complexity** which refers to the amount of computer time required by an algorithm for its execution, which includes both the compile time and run time. The compile time of an algorithm does not depend on the instance characteristics of the algorithm. The run time of an algorithm is estimated by determining the number of various operation, such as addition, subtraction, multiplication, division, load and store executed by it.

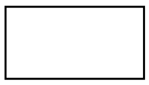
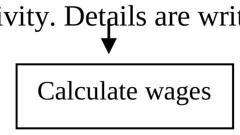
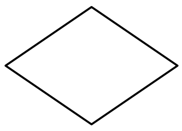
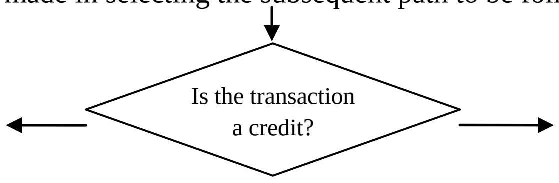
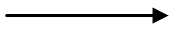
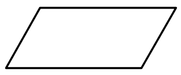
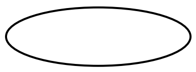
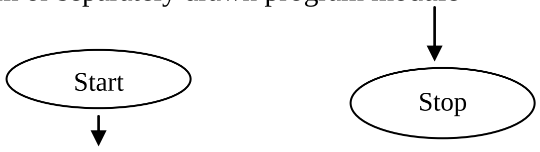
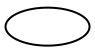
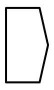
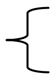
The analysis of an algorithm determines the amount of resources, such as time and space required by it for its execution. Generally, the algorithms are formulated to work with the inputs or arbitrary length. Algorithm analysis provides theoretical estimates required by an algorithm to solve a problem. The steps of an Algorithm, they can be presented using Flow charts and pseudo-codes.

1.3. Flow Charts

A flow chart is a traditional means of showing in diagrammatic form, the sequence of steps to be undertaken in solving a problem. Flowcharts or flow diagrams are important tools in writing a computer program. A flowchart allows you to plan the sequence of steps in a program before writing it. The flowchart serves as a visual representation which many programmers find indispensable in planning any program of at least moderate complexity.

1.3.1. Elements of a Flowchart.

A flowchart consists of a set of boxes, the shapes of which indicate specific operations. The separate boxes are connected with arrows to show the sequences in which the various operations are performed. We use these standard symbols:

Shape	Name	Description
	Rectangle	Process symbol: Used to represent any kind of processing activity. Details are written in the box 
	Diamond:	The decision Symbol: Used where a decision has to be made in selecting the subsequent path to be followed. 
		Used to show the flow/ path of ma sequence of symbols. • Vertical line without arrow head are assumes t flow top to bottom. • Horizontal lines without arrow heads are assumed to flow left to right. • Every operation box must have at least one incoming or outgoing arrow. Any arrow leaving a decision box must be labeled with the decision result which will cause that path to be followed.
	Parallelogram	Input/output symbol: Used where data input is to be performed
	Oval	The Terminal symbol: Used as the first or last symbol in a program or separately drawn program module 
 Or 	Small Circle	Connector symbol: Exit to or entry from another part of the chart
		Used to add explanatory notes or description,

Stages of Flow charting

Program flowcharts are generally produced in two stages representing different levels of details. The first step produces the o**utline program flow chart** which represents the first stage of turning the systems flow charts into the necessary detail to enable the programmer to write the programs. It represents the actual computer operations in an outline only. The second step produces the d**etailed program flow chart** which is prepared from the outline charts and contains the detailed computer steps necessary to perform a particular task. It is from thischarts that the programmer will prepare the program code

Limitations of program flowcharts

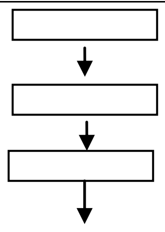
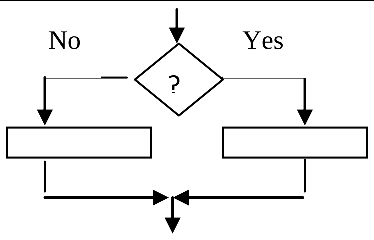
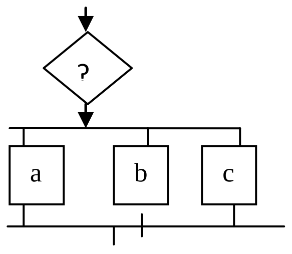
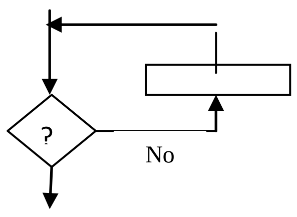
- Not easily translated into programming language.

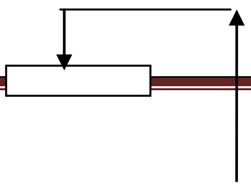
1.4. Pseudo code

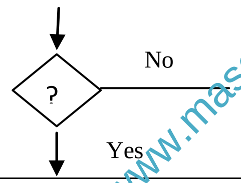
An alternative method of representing an Algorithm to the flowcharts. Pseudo code is halfway between English and programming language and is based upon a few simple grammatical construction which avoid the ambiguities of English but which can be easily converted into computer programming language. Pseudo code is an informal high-level description of a computer programming algorithm, which omits details that are not essential for human understanding of the algorithm, is easier for humans to understand than conventional programming language code, is compact and environment-independent description of the key principles of an algorithm and resembles skeleton programs including dummy code and can be compiled without errors.

Pseudo code assumes that programming procedures no matter how complex may be reduced to a combination of controlled sequences, selection, or repetition of basic operations. This gives rise to the control structures found in pseudo-code.

1.5. Program Control Structures

Control Structure	**Pseudo code**	**Flow Chart**
Sequence: In the absence of selection, or repetition, program statements are executed in the sequence in which the appear in the program	1st Instruction ↓ 2nd Instruction ↓ 3rd Instruction	
Selection Part of decision making and allows alternative actions to be taken according to the conditions that exist at particular stages in program execution	IF condition THEN actions ELSE actions ENDIF Or CASE a). Actions b). Actions c). Actions d). Actions ENDCASE	 
Repetition also called "looping" There are many programming problems in which the same sequence of statements needs to be performed again and again for a definite or indefinite number of times	WHILE condition DO Actions ENDWHILE	



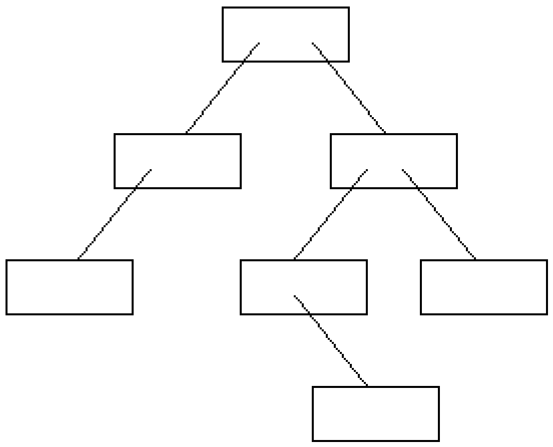
	REPEAT actions UNTIL condition	

1.6. Programming Methods

1.6.1 Top-down and Bottom-up methodology

A **top-down** approach (is also known as step-wise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

Top-*down-design* starts with a description of the overall system and usually consists of a hierarchical structure which contains more detailed descriptions of the system at each lower level. The lower level design details continue until further subdivision is no longer possible, i.e., until the system is described in terms of its "atomic" parts. This method involves a hierarchical or tree-like structure for a system as illustrated by the following diagram:



At the top level, we have that part of the system which deals with the overall system; a kind of system overview or main top-level module.

Top down programming method process

1. Define exactly what data the program will get and what it has to do with them.
2. If the task is simple enough, write the program code.
3. Otherwise, split the task into smaller parts and define exactly the duty of each part and interface to the rest of the program.
4. Repeat the steps 1–4 separately for each subtask.

Advantages of the Top-Down Design Method

- It is easier to comprehend the solution of a smaller and less complicated problem than to grasp the solution of a large and complex problem. Separating the low level work from the higher level abstractions leads to a modular design. Modular design means development can be self contained. Much less time consuming (each programmer is only involved in a part of the big project).

- It is easier to test segments of solutions, rather than the entire solution at once. This method allows one to test the solution of each sub-problem separately until the entire solution has been tested.
- It is often possible to simplify the logical steps of each sub-problem, so that when taken as a whole, the entire solution has less complex logic and hence easier to develop. A simplified solution takes less time to develop and will be more readable.
- The program will be easier to maintain. If errors occur in the output it is easy to identify the errors generated from each of the modules / sub-programs of the entire program.

A **bottom-up** approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

Top-down is a programming style, the mainstay of traditional procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces. The technique for writing a program using top-down methods is to write a main procedure that names all the major functions it will need. Later, the programming team looks at the requirements of each of those functions and the process is repeated. These compartmentalized sub-routines eventually will perform actions so simple they can be easily and concisely coded. When all the various sub-routines have been coded the program is ready for testing. By defining how the application comes together at a high level, lower level work can be self-contained. By defining how the lower level abstractions are expected to integrate into higher level ones, interfaces become clearly defined.

Top-down approaches emphasize planning and a complete understanding of the system. It is inherent that no coding can begin until a sufficient level of detail has been reached in the design of at least some part of the system. The Top-Down Approach is done by attaching the stubs in place of the module. This, however, delays testing of the ultimate functional units of a system until significant design is complete. Bottom-up emphasizes coding and early testing, which can begin as soon as the first module has been specified. This approach, however, runs the risk that modules may be coded without having a clear idea of how they link to other parts of the system, and that such linking may not be as easy as first thought. Re-usability of code is one of the main benefits of the bottom-up approach

1.6.2. Structured programming Method

Structured programming is a special type of procedural programming. It provides additional tools to manage the problems that larger programs were creating. Structured programming requires that programmers break program structure into small pieces of code that are easily understood. It also frowns upon the use of global variables and instead uses variables local to each subroutine.

Structured programming was developed during the 1950s after Edgar Dijkstra's insightful comments into the harmful nature of the GO TO statement. Dijkstra and others subsequently created a set of acceptable structures in programming that would enable development without GO TO statements. These structures produced programs that were easier to read by humans, easier to debug and easier to test. These structures have become some of the founding principles of modern programming methods.

Although the principles of structured programming have had a profound effect on the programming world, it was not until the 1970s that an actual language was created for teaching structured programming. Pascal was developed especially for this purpose, though it is much derided as a toy language, and appears to have never been used in commercial development. It appears that existing languages such as COBOL and FORTRAN were changed to accommodate Dijkstra's structures, or that programming included these structures through more indirect methods.

Later generation languages such as C are fully-fledged structured programming languages; these are from the third generation and procedural, in that they are both written and executed step-by-step. C, in its turn, has formed the foundation for the object-oriented language C++.

The three structures allowed in structured programming are *sequence*, *selection*, and *iteration*. Structures are also thought of in terms of substitution and combination, i.e. structures can be substituted or combined with other structures as long as the result equals a sequential structure. Structured programming also pays attention to design and testing with emphasis on a *top-down approach*. The top-down approach uses modularity as a means to ensure that the program is both legible and manageable, and also that these modules can be tested as they are developed. This is beneficial as it ensures that all modules should be tested and that bugs can be found in the modules that have most recently been added or altered.

Structured programming also places emphasis on program documentation, which can be in the form of a chart or the structured coding/listing. This documentation allows for subsequent updating of modules, making these modules easier to locate in the program. Modularity also ensures greater opportunity for re-use of modules during development.

1.7. Programming Aims

Good programming principles and practice aim at producing a program with the following characteristics;

- Reliability: the program can be depended upon always to do what it is supposed to do
- Maintainability: the program will be easy to change or modify when the need arises
- Portability: the program will be transferable to a different computer with a minimum modification.
- Readability: the program will be easy for a programmer to read and understand.
- Performance: the program causes the tasks to be done quickly and efficiently.
- Storage saving: the program is not allowed to be unnecessarily long

1.8. Programming Paradigms

A programming paradigm is a pattern of problem solving thought that underlies a particular genre of programs and languages. Four distinct and fundamental programming paradigms have evolved over the last three decades;

- Imperative programming;
- Object-oriented programming
- Functional programming
- Logic Programming;

1.8.1. Imperative programming;

The oldest and the most well-developed, it emerged with the first computers in the 1940s and its elements directly mirror the architectural characteristics of modern computers as well. The program and its variables are stored together and the program contains a series of commands that perform calculations, assign values to variables, retrieve input, produce out, or redirect control elsewhere in the series.

Procedural abstraction is an essential building block for imperative programming as are assignments, loops, sequences, conditional statements and exception handling. Imperative languages also support variable declaration and expressions. The predominant imperative programming languages include Cobol, Fortran, C Ada and Perl.

Commands are normally executed in the order they appear in the memory, while conditional and unconditional branching statements can interrupt this normal flow of execution. Originally the commands included assignment statements, conditional statements and branching statements. The assignments statements provided the ability to dynamically update the value stored in the memory location, while conditional and branching statements could be combined to allow a set of statements to be either skipped or repeatedly executed. The main features of Imperative programming includes;

- Control structures;
- Input/output
- Error and exception handling
- Procedural abstraction
- Expressions and assignments
- Library support for data structures

1.8.2. Object-oriented (OO) programming:

Provides a model in which the program is a collection of objects that interact with each other by passing messages that transform their state. The message passing allows the data objects to become active rather than passive. Object classification, inheritance and message passing are fundamental building blocks for OO programming. Major languages includes, C++, Java and C#.

1.8.3 Functional Programming:

Emerged in the early 1960s and its creation was motivated by the needs of researchers in artificial intelligence and its sub-fields- symbolic computation, theorem proving, rule-based systems and natural language processing. Models a computational problem as a collection of mathematical functions, each with an input (domain) and a result (range) spaces.

1.8.4. Logic Programming

Logic (declarative) programming allows a program to model a problem by declaring what outcome the program should accomplish, rather than how it should be accomplished. Sometimes called rule-based languages, since the program's declarations look more like a set of rules, or constraints on the problem, rather than a sequence of commands to be carried out.

Chapter Review Questions

1. Describe the processing of solving a problem using computers
2. Define a problem Algorithm
3. What are the advantages and disadvantages of using flow-charts to represent problem algorithm
4. What are the advantages and disadvantages of representing a problem algorithm using pseudo-codes.
5. A School is interested in computerizing their students grading system which is as follows;

Marks	Grade
80 - 100	A
60 - 79	B
50 - 59	C
40 - 49	D
0 - 39	E

a) Draw a flow chart to represent the solution.

b) Represent the solution in pseudo-code.

CHAPTER TWO

Programming Languages

Chapter Objectives

By the end of this chapter the learner should be able to

- Define a programming language
- Describe the types of programming languages
- Differentiate and explain advantages and disadvantages of the various types of programming languages
- Describe the High level languages translation processes
- Describe the criteria for Programming language evaluation

2.1. What is a Programming Language

There are many definitions of what constitutes a programming language, and none of these is the 'correct' answer. What might have defined a programming language in the 19th century would not necessarily be detailed enough for a modern definition. Programming languages are needed to allow human beings and computers to talk to each other. Computers, as yet, are unable to understand our everyday language or, in fact, the way we talk about the world. Computers understand logic expressed mathematically through what is known as machine code. Computer language consists of 1s and 0s or the *binary system*, which the majority of human beings would find very difficult to communicate in. Computer languages enable humans to write in a form that is more compatible with a human system of communication. This is then translated into a form that the computer can understand. Here are some different ideas on what constitutes a programming language.

- A programming language has been defined as a tool to help the programmer.
- A way of writing that can be read by both a human being and a machine.
- A sequence of instructions for the machine to carry out.
- A way for a human being to communicate with a machine that is unable to understand natural language.
- A computer language offers a means of writing algorithms that can be understood by both human being and machine. Machines are unable to understand natural language, so a human being uses algorithms that are translated into machine code by the programming language. Machine code is difficult for humans to use, so a language 'translates' human readable language into machine readable form.
- A computer program offers humans a standard way of expressing algorithms to solve particular problems. As languages offer a convention it allows other humans to read the program, and change it if they need to.

2.2. Types of Programming Languages

There are three levels of programming languages;

- Machine language (low level language)
- Assembly (or symbolic) language
- Procedure-oriented language (high level language)

2.2.1. Machine language

The lowest-level programming language (except for computers that utilize programmable microcode) Machine languages are the only languages understood by computers. While easily understood by computers, machine languages are almost impossible for humans to use because they consist entirely of numbers. It is a programming language in which the instructions are in a form that allows the computer to perform them immediately, without any further translation being required. Instructions are in the form of a Binary code also called machine code and are called machine instructions. Commonly referred to as the First Generation language

2.2.2. Assembly Language

Introduced in 1950s, reduced programming complexity and provided some standardization to build and applications. Also referred to second generation language. The 1 and 0 in machine language are replaced by with abbreviations or mnemonic code. It consists of a series of instructions and mnemonics that correspond to a stream of executable instructions. It is converted into machine code with the help of an assembler. Common features includes;

- Mnemonic code; used in place of the operation code part of the instruction eg SUB for substract, which are fairly easy to remember
- Symbolic Addresses which are used in place of actual machine addresses. A programmer can choose a symbol and use it consistently to refer to one particular item of data. Example FNO to represent First No.
- The symbolically written program has to be translated into machine language before being used operationally. A 1 to 1 translation to machine language, ie one symbolic instruction produces one machine instruction/code.

Advantages of Assembly language over machine language

- It is easy to locate and identify syntax errors, thus it is easy to debug it.
- It is easier to develop a computer application using assembly language in comparison with machine language
- Assembly language operates very efficiently.

2.2.3. High level language

A Machine independent and a Problem oriented (POL) programming language. High level language is portable across different machine types (architectures); The machine independence of the high level languages means that in principle it should be possible to make the same high-level language run on different machines. It reflects the type of problem solved rather than the features of the machine.

High level languages are more abstract, easier to use and more portable across platforms as compared to low-level programming languages. A programmer uses variables, arrays or Boolean expressions to develop the logic to solve a problem. Source programs are written in statements akin to English. A high level language code is executed by translating it into the corresponding machine language code with the help of a compiler or interpreter. High level languages can be classified into the following categories;

- Procedure-oriented languages (third generation)
- Problem-oriented languages (fourth generation)
- Natural languages (fifth generation).

Procedure languages.

High-level languages designed to solve general-purpose problems, example BASIC, COBOL, FORTRAN, C, C++ and JAVA. They are designed to express the logic and procedure of a problem. Though the syntax of the languages may be different, they use English-like commands that are easy to follow. They are portable.

Problem-oriented languages

Problem-oriented languages also known as Fourth Generation Languages (4GL) are used to solve specific problems and includes query languages, report generators and Application generators which have simple English like syntax rules. The 4GLs have reduced programming efforts and overall cost of software development. They use either visual environment or a text environment for program development similar to that of third-generation languages. A single statement of the 4GL can perform the same task as multiple line of a third-generation language. It allows a program to just drag and drop from the toolbar, to create various items like buttons, text boxes, label etc. A program can quickly create a prototype of the software applications

Natural Languages

Natural languages widely known as fifth generation languages, are designed to make a computer to behave like an expert and solve problems. The programmer just needs to specify the problem and the constraints for problem solving. Natural languages such as LISP and PROLOG are mainly used to develop artificial intelligence and expert systems.

Features of high level language

- Extensive vocabulary of words, symbols and sentences
- Whole sentences are translated into many machine codes instructions
- Portable across different machine types (architectures)
- Libraries of macros and sub-routines can be incorporated
- As they are problem oriented, the programmer is able to work at least to some extent independently of the machine.
- Have a set of rules that must be obeyed.
  - ➢ Syntax: the structure of the statements and the grammatical rules governing them. Grammatical rules that govern the way in which words, symbols, expressions and statements may be formed and combined.
  - ➢ Semantics: the meaning of the statements written in the language. The rules that governs its meaning. – what happens when the program is executed/run most are standardized by ISO/ANSI to provide an official description of the language

2.3. High Level language Translation

High level languages need to be translated into machine language which is the computer language. The translation is done by a Compiler or Interpreter

2.3.1. Compiler

A compiler is a manufacturer specifically written computer program which translates (or compiles) a source code computer program that translates the source code written in a high level language into the corresponding object code of the low level language. The translation process is called compilation. The entire high level source code / program is converted into the executable machine code file prior to the object program being loaded into main memory and executed. Translation done only ones and the object program can be loaded into the main storage and executed. A program that translates a low-level language into a high level language is called a Decompiler. Compiled languages includes C, C++, COBOL, FORTRAN etc.

Compilers are classified into single-Pass compilers and Multi-pass compilers. Single-pass compilers are generally faster than multi-pas compilers, but multi-pass compilers are required to generate high quality code

A Compiler:

- Translates the source program code into machine code
- Includes linkages for closed sub-routine
- Allocates areas of main storage
- Produces the object program.
- Produces a printed copy (listing) of the source code and object code
- Produces a list of errors found during compilation.

2.3.2. Interpreter:

The interpreter is a translation program that converts each high-level language statement into the corresponding machine code. The translation process is carried out just before the program statement is executed. Instead of the entire program, one program statement at a time is translated and executed immediately. When using an interpreter, the source code translated every time the program is executed

The commonly interpreted languages include BASIC and PERL. Though interpreters are easier to create as compared to compilers, the compiled languages can be executed more efficiently and are faster. Interpreters are appropriate in;

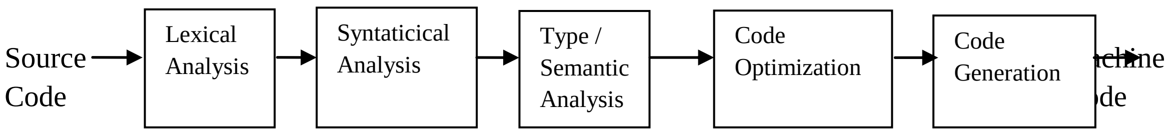
- Handling user commands in an interactive system
- Debugging programs as they run (removing program faults).
- Handling software produced for or by a different computer.

2.4. Computer Program Compilation Process

A computer program compilation process involves the following five stages;

1. Lexical analysis: the source program is transformed into tokens. During the transformation all whitespaces and comments are discarded. All character sequences that do not form valid tokens are discarded and an error message is generated.
2. Syntactical analysis: analysis to ensure the program syntax is appropriate to the language. Failure results in an error message been generated.
3. Type / Semantic checking: responsible for ensuring that the compile-time semantic rules of the language are enforced. An error message is generated if the semantic rules are violated
4. Code optimization: improves the intermediate code based on the target machine architecture
5. Code generation: target machine code is generated.

The first three stages are concerned with finding and reporting errors to the programmer, while the last two are concerned with generating efficient machine code to run on the targeted computer.



2.5. Evaluating Languages

Programming languages can be evaluated from a number of viewpoints, depending on either the programmer, the environment in which the programmer works or the standards of the organisation. When developing software, a programmer should consider which language is most suitable to the task, rather than relying on a language with which they are familiar. You wouldn't want to use a spreadsheet to develop a database. All the features of the language need to be considered rather than just one particular feature. A wrong choice can mean that the software has to be re-written, which can be very frustrating and time consuming.

There are a number of different ways that the programmer can think about the design of the system, from the top-down of structured programming to object oriented design issues. Some languages are geared towards one particular style of design, whilst others incorporate many types. Each of these language paradigms enables the programmer to consider the problem from a different viewpoint. There are a few basic questions that can be asked to help when making these decisions:

1. How readable is the language, to humans? If parts of the program are going to be read or altered separately from the entire program is might be worth considering how legible they are going to be. It is also useful to consider the length of names allowed in the language, for instance an early form of Fortran allowed for only 6 characters. This can lead to clumsy abbreviations that are difficult to read. Statements such as GO TO, FOR, WHILE and LOOP have increased the readability of programs, and lead to neater programs. These statements also affect the syntax or grammar.
2. When it comes to writing the program, how easy is it to write the program in this particular language? A programming language that is easy to write in can make the process easier and faster. It may help to reduce mistakes. FOR loops and other types of statement allow the programmer to write much simpler code. This will save time and money, and also make the program smaller.
3. How reliable is the language? Not all languages create robust programs, and some help the programmer to avoid making errors. A program that is not robust can cause errors, and code can ‘decay’. Any language that helps the programmer to avoid mistakes will make it easier to use.
4. How much would it cost to develop using a given language? Is the language expensive to use and to maintain? Programs may need to be updated or redeveloped, and an expensive language may make this prohibitive.
5. How complicated is the syntax going to be? Syntax is an important consideration. Clarity and ease of understanding are important, as is a syntax that seems logical and sensible. Errors are very likely to occur where one area of syntax too closely resembles another, and the program may prove difficult to debug. Some theorists reason that if it is difficult to write a program to parse the language, then it follows that it will be problematical for the programmer to get it right.
6. Does the language have standards? Languages that have standards for writing programs have greater readability; for instance Java has standards for naming, commenting and capitalization.

2.6. The Programming Language Generations

The language generations span many decades, and begin with the development of machine code. Each generation adds new features and capabilities for the programmer to use. Languages are designed to create programs of a particular type, or to deal with particular problems. Modern languages have led to the development of completely different styles of programming involving the use of more human-like or natural language and re-usable pieces of code.

- The first generation of languages was machine language. Instructions and addresses were numerical. These programs were linked to the machine they were developed on.
- The second generation allowed symbolic instructions and addresses. The program was translated by an assembler. Languages of this generation include IBM, BAL, and VAX Macro. These languages were still dependent on the machine they were developed on.
- Third generation languages allowed the programmer to concentrate on the problem rather than the machine they were writing for. Other innovations included structured programming and database management systems. 3GL languages include FORTRAN, COBOL, Pascal, Ada, C, and BASIC. All 3GL languages are much easier for the human being to understand.
- 4GL languages (fourth generation). These are known as non-procedural, they concentrate on what you want to do rather than how you are going to do it. 4GL languages include SQL, Postscript, and relational database orientated languages.

- 5GL (fifth generation). These languages did not appear until the 1990s, and have primarily been concerned with Artificial Intelligence and Fuzzy Logic. The programs that have been developed in these languages have explored Natural Language (making the computer seem to communicate like a human being).

Chapter Review Questions

1. Describe the C program compilation process
2. What criteria should a programmer use to evaluate a programming language
3. What are advantages of High-level languages over the Assembly language.
4. Describe the categories of the high-level languages

CHAPTER THREE

Introduction to C Programming

Chapter Objectives

By the end of this chapter the learner should be able to;

- Describe the characteristics of C programming language.
- Describe the process of developing and Executing a C program
- Describe the compilation process of a C program and C program file naming conventions.
- Differentiate between Syntax and Logical Errors
- Describe the structure / format of a C Program

3.1. What is C program

C is an imperative (procedural) systems implementation language. C is called a high level, compiler language. The aim of any high level computer language is to provide an easy and natural way of giving a programme of instructions to a computer (a computer program). The language of the raw computer is a stream of numbers called machine code. As you might expect, the action which results from a single machine code instruction is very primitive and many thousands of them are required to make a program which does anything substantial.

C is one of a large number of high level languages which can be used for general purpose programming, that is, anything from writing small programs for personal amusement to writing complex applications. **C** is a general-purpose computer programming language developed between 1969 and 1973 by Dennis Ritchie at Bell telephone Laboratories. It is unusual in several ways. Before C, high level languages were criticized by machine code programmers because they shielded the user from the working details of the computer, with their black box approach, to such an extent that the languages become inflexible: in other words, they did not allow programmers to use all the facilities which the machine has to offer. C, on the other hand, was designed to give access to any level of the machine down to raw machine code and because of this it is perhaps the most flexible of all high level languages.

The C language has been equipped with features that allow programs to be organized in an easy and logical way. This is vitally important for writing lengthy programs because complex problems are only manageable with a clear organization and program structure. C allows meaningful variable names and meaningful function names to be used in programs without any loss of efficiency and it gives a complete freedom of style; it has a set of very flexible loop constructions (for, while, do) and neat ways of making decisions. These provide an excellent basis for controlling the flow of programs.

Another unusual feature of C is the way it can express ideas concisely. The richness of a language shapes what it can talk about. C gives us the apparatus to build neat and compact programs. This sounds, first of all, either like a great bonus or something a bit suspect. Its conciseness can be a mixed blessing: the aim is to try to seek a balance between the often conflicting interests of readability of programs and their conciseness. Because this side of programming is so often presumed to be understood, we shall try to develop a style which finds the right balance.

C allows things which are disallowed in other languages: this is no defect, but a very powerful freedom which, when used with caution, opens up possibilities enormously. It does mean however that there are aspects of C which can run away with themselves unless some care is taken. The programmer carries an extra responsibility to write a careful and thoughtful program. The reward for this care is that fast, efficient programs can be produced.

C tries to make the best of a computer by linking as closely as possible to the local environment. It is no longer necessary to have to put up with hopelessly inadequate input/output facilities anymore (a legacy of the timesharing/mainframe computer era): one can use everything that a computer has to offer. Above all it is flexible. Clearly no language can guarantee intrinsically good programs: there is always a responsibility on the programmer, personally, to ensure that a program is neat, logical and well organized, but it can give a framework in which it is easy to do so.

The C compiler combines the capabilities of an assembly language with features of a high-level language thus making it suited for writing both system software and business packages. C program uses a variety of data types and operators thus making programs written in C to be efficient and fast. C is highly portable and is well suited for structured programming. C is basically a collection of functions that are supported by the C library and because new functions can be added to the C library, C has the ability to extend itself.

3.2. Characteristics of C

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

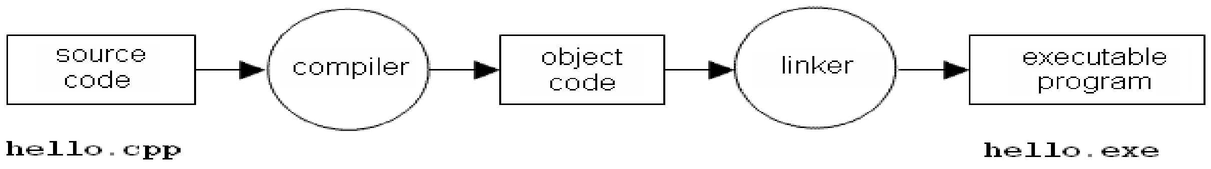
3.3. Executing a C program

The steps involved in executing a C program includes;

- Creating the program
- Compiling the program
- Linking the program with functions that are needed from the C library
- Executing the program

3.4 Compiling a C Program

A C program is first written in the form of a number of text files using a screen editor. This form of the program is called the source program. It is not possible to execute this file directly. The completed source file is passed to a compiler a program which generates a new file containing a machine code translation of the source text. The compiler translates the source code into machine code, and the compiled code is called the *object code*. The object code may require an additional stage where it is linked with other object code that readies the program for execution. The machine code created by the *linker* is called the *executable code* or *executable program*. Instructions in the program are finally executed when the executable program is executed (run). During the stages of compilation, linking, and running, error messages may occur that require the programmer to make corrections to the program source (debugging). The cycle of modifying the source code, compiling, linking, and running continues until the program is complete and free of errors.



A compiler usually operates in two or more phases (and each phase may have stages within it).
A two-phase compiler works in the following way:

Phase 1 scans a source program, perhaps generating an intermediate code (quadruples or pcode) which helps to simplify the grammar of the language for subsequent processing. It then converts the intermediate code into a file of object code (though this is usually not executable yet). A separate object file is built for each separate source file. In the GNU C compiler, these two stages are run with the command gcc -c; the output is one or more .o files.

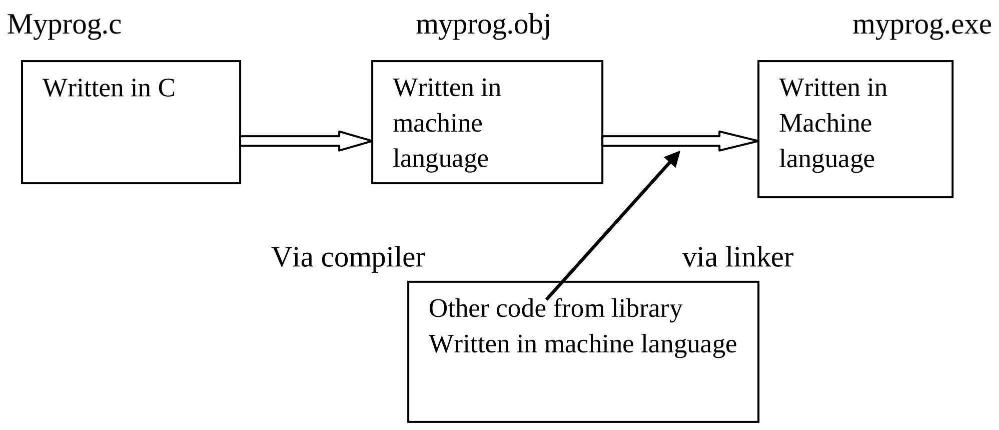
Phase 2 is a Linker. This program appends standard library code to the object file so that the code is complete and can "stand alone". A C compiler linker suffers the slightly arduous task of linking together all the functions in the C program. Even at this stage, the compiler can fail, if it finds that it has a reference to a function which does not exist. With the GNU C compiler this stage is activated by the command gcc -o or ld.

3.5. C Program File naming convention

The compiler uses a special convention for the file names, so that we do not confuse their contents. The name of a source program (the code which you write) is *filename*.c. The compiler generates a file of object code from this called *filename*.obj. The final program, when linked to libraries is called *filename.exe*

The endings `dot something' (called file extensions) identify the contents of files for the compiler. The dotted endings mean that the compiler can generate an executable file with the same name as the original source - just a different ending. The object file is only working files and should be deleted by the compiler at the end of compilation. The .c suffix is to tell the compiler that the file contains a C source program and similarly the other letters indicate non-source files in a convenient way.

Source code:	Filename.c
Object code:	Filename.obj
Executable code:	Filename.exe


Myprog.c
Written in C
myprog.obj
Written in machine language
myprog.exe
Written in Machine language
Via compiler
via linker
Other code from library Written in machine language

3.6. Errors

Errors are mistakes which we the programmers make. There are different kinds of error:

3.6.1. Syntax Error

Every language has got set of rules. If you make a mistake while using the language, then it is called syntax error.

Errors in the syntax, or word structure of a program are caught before you run it, at compilation time by the compiler program. They are listed all in one go, with the line number, in the text file, at which the error occurred and a message to say what was wrong. A program with syntax errors will cause a compiler program to stop trying to generate machine code and will not create an executable. However, a compiler will usually not stop at the first error it encounters but will attempt to continue checking the syntax of a program right to the last line before aborting, and it is common to submit a program for compilation only to receive a long and un-gratifying list of errors from the compiler.

As a rule, look for the *first* error, fix that, and then recompile. Of course, after you have become experienced, you will recognize when subsequent error messages are due to independent problems and when they are due to a cascade. But at the beginning, just look for and fix the first error.

Use of Upper and Lower Case

One of the reasons why the compiler can fail to produce the executable file for a program is you have mistyped something, even through the careless use of upper and lower case characters. The C language is *case dependent*. Unlike languages such as Pascal and some versions of BASIC, the C compiler distinguishes between small letters and capital letters. This is a potential source of quite trivial errors which can be difficult to spot. If a letter is typed in the wrong case, the compiler will complain and it will not produce an executable program.

3.6.2. Logical or Intention Error

Errors in goal or purpose (logical errors) occur when you write a program that works, but does not do what you intend it to do. You intend to send a letter to all drivers whose licenses will expire soon; instead, you send a letter to all drivers whose licenses will expire sometime. If the compilation of a program is successful, then a new file is created. This file will contain machine code which can be executed according to the rules of the computer's local operating system.

When a programmer wants to make alterations and corrections to a C program, these have to be made in the source text file itself using an editor; the program, or the salient parts, must then be recompiled.

3.7. C Libraries

In C, a library is a set of functions contained within a single "archive" file. The core of the C language is small and simple. Special functionality is provided in the form of libraries of ready-made functions. This is what makes C so portable. Libraries are files of ready-compiled code which we can merge with a C program at compilation time. Libraries provide ***frequently used functionality*** and, in practice, at least one library must be included in every program: the so-called C library, of standard functions.

Each library comes with a number of associated ***header files*** which make the functions easier to use. Header files contains the prototypes of the functions contained within the library that may be used by a program, and declarations of special data types and macro symbols used with these functions. It is up to every programmer to make sure that libraries are added at compilation time by typing an optional string to the compiler.

Including Library files in C Program

The most commonly used header file is the standard input/output library which is called stdio.h. This belongs to a subset of the standard C library which deals with file handling and provides standard facilities for input to and output from a program. Examples of Libraries header files

- Stdio.h, (printf() function)
- maths.h (for mathematical functions) etc.
- conio.h (for handling screen out puts such as pausing program execution getch() function)

The format for including the header file is **#include header.h**

3.8. C Program Structure

C program is can be divided into modules and functions.

Modules: A module is a set of functions that perform related operations. A simple program consists of one file; i.e., one module. More complex programs are built of several modules. Modules have two parts: the public interface, which gives a user all the information necessary to use the module; and the private section, which actually does the work.

Functions; the basic building block in a C program is the function. In general, functions are blocks of code that perform a number of pre-defined commands to accomplish something productive. It must have a name and it is reusable ie it can be executed from as many different parts in a C Program as required. Information passed to the function is called arguments and is specified when the function is called. And the function either returns some value to the point it was called from or returns nothing.

Function: a sub-program that may include one or more statements designed to perform a specific task

Every C Program will have one or more functions and there is one mandatory function which is called ***main()*** function. This function is prefixed with keyword *int* which means this function returns an integer value when it exits. This integer value is returned using *return* statement.

Structure of a Function

There are two main parts of the function. The function header and the function body.

```
int sum(int x, int y)
{
        int ans = 0;        //holds the answer that will be returned
        ans = x + y;        //calculate the sum
        return ans                  //return the answer
}
```

Function Header

It is the first line of a function, example; int sum(int x, int y). It has three main parts

- The name of the function i.e. ***sum***
- The parameters of the function enclosed in paranthesis
- Return value type i.e. ***int***

Function Body

What ever is written with in { } in the above example is the body of the function.

3.9 C Program format

A C program includes the following sections

1. Documentation Section
2. Linker Section
3. Definition Section
4. Global Declaration Section
5. Main() Function Section

```
  {
       Declaration section
       Executable Section
  }
```

6. Sub-program Section

```
     Function 1
     Function 2
     Function3   etc   is User defined functions
```

1.9.1. Documentation Section

This section consists of a set of comments lines giving the name of the program, the author and other details which the programmer would like to use later. Comments starts with /* and ends with */ and enhances readability and understandability. Comment lines are not executable statements ie. executed and are ignored by the compiler. Comments can be inserted wherever there is a blank a space but cannot be nested (having a comment within a comment)

example /* …………………./* ……………..*/ ……………….*/

1.9.2. Link Section

This section provides instructions to the compiler to link functions from the system library. C program have predefined functions stored in the C library. Library functions are grouped category-wise and stored in different file known as header files. To be able to access the library files it is necessary to tell the compiler about the files to be accessed.

Instruction Format: **#include<file_name>**

Example #include<stdio.h> a standard I/O header file containing standard input and output functions

1.9.3. Definition Section

This section allows the definition of all symbolic constants. Statements begin with # sign and do not end with a ; because the statements are compiler directive statements

Example #define PRINCIPLE 10000

Symbolic constants are usually written in upper case to distinguish them from lower case variables. Values defined here remain constant throughout the program.

1.9.4. Declaration Section

Section used to declare global variables. The section is also used to declare user defined functions.

3.9.5 main() Function Section

The main() function is a special function used by C system to tell the computer where the program starts. Every program must have exactly **one main function**. The main function is the point by where all C programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C program. For that same reason, it is essential that all C programs have a main function. The main function section has the following sections

- Declaration part: where all variables used in the executable section are declared.
- Executable part: consist of the statements to be executed.

```
{
  Declaration part
 Executable part
}
```

There must be a least one statement in the executable part. The two part must be included between the opening { and the closing }. Program execution begins at the opening { and closes at the closing }, which signifies the logical end of the program. All the statements between the { and } forms the function body and are the instructions to perform the given task. All statements in the Declaration and Executable parts must end with a semicolon **(;)**

Formats of main()

- main()
- int main()
- void main()
- main(void)
- int main(void)

( ) or word **void** means the function has no arguments or parameter and thus does not return any information to the operating system. **int** means the function return an integer value to the operating system

3.9.6 Sub-program section

Contains all user defined functions that are called in the **main** function

```
main()                    /* Main program */
{
do_nothing();              /* Function call */
}
/******************************************************/
do_nothing()              /* Function called */
{
}
```

Example

```
  // Program using function          --  comment
  #include <stdio.h>
  #include<conio.h>
  int mul (int a, int b);

  int main()              /* function body*/
  {
        int a, b, c;
         a = 5;
         b = 10;
         c = mul (a,b);                  /* function call*/

      printf("Multiplication of %d and %d is %d", a,b,c);     /* request to print the answer on the Screen*/
     getch();                                   /*command used to pause the results on the screen*/
  }
     /* mul()                                   /* ……….. Sub-program mul */
       int mul(int x,int y)
       {
         int p;
         p = x*y;
         return (p);
     }
```

NB. the values of a & b are passed to x & y respectively when the sub-program is called.

Note the followings

- C is a case sensitive programming language. It means in C *printf* and *Printf* will have different meanings.
- C has a free-form line structure. End of each C statement must be marked with a semicolon.
- Multiple statements can be done on the same line.
- White Spaces (ie tab space and space bar) are ignored.
- Statements can continue over multiple lines.
- Printf is a predefined C function for printing out put
- Everything between the starting and ending quotation marks “ ” to be printed.
- To print on separate line; Use the command \n
- int = integer data type
- float = floating point number data type
- %d = formatting command that prints the output as a decimal integer
- %5.2f’ = formatting command that prints the output as a floating point integer with five places in all and two places to the right of the decimal point.

Program Example 3.1	**Program Example 3.2**

main()
{
printf (“This is my \n”);
printf(“computer book”);
getch();
}
OR
main(()
{
print(“This is \n my computer book”);
getch();
	/* program for addition */
	#include<stdio.h>
	#include<conio.h>
	main()
	{ int number;
	float amount;
	number = 100;
	amount = 30.75 + 75.35;
	printf("%d\n", number);
	printf("%5.2f", amount);
	getch(); /* pause the results on the screen */

}	}
Program Example 3.3	**Program Example 3.4**

```
/* program to print Hello */
#include <stdio.h>
#include<conio.h>

int main()
{
   printf("hello, world\n");
   getch();
}
```

```
/* program to calculate the Interest rate */
#include <stdio.h>
#include<conio.h>
#define PERIOD  10
#define PRINCIPAL    5000.00
int main()
 {
 int year;
 float amount, value, inrate;
 amount = PRINCIPAL;
 inrate = 0.11;
 year = 0;
 while(year <= PERIOD)
 {printf("%2d %8.2f\n", year, amount);
    value = amount + inrate * amount;
    year = year + 1;
    amount = value;
    }
    getch();
    }
```

3.10. Breaking out early

Return statement

The program can simply call return(value) anywhere in the function and control will jump out of any number of loops or whatever and pass the value back to the calling statement without having to finish the function up to the closing brace }.

The exit() function

The function called exit() can be used to terminate a program at any point, no matter how many levels of function calls have been made. This is called with a return code, like this:

```
#define CODE  0
exit (CODE);
```

This function also calls a number of other functions which perform tidy-up duties such as closing open files etc.

Chapter Review Questions

1. How is a library file incorporated into a C program? Name the most common library file in C.
2. What is another name for a library file?
3. Describe the structure of C Program
4. Distinguish between
   a) main() and main(void)
   b) int main() and void main()
5. Find errors if any in the following

```
#include <stdio.h>
Void main()
{ Print (“Hello C);
}
```

CHAPTER FOUR

C Programming: - Constants, Variables and Data Types

Chapter Objectives

By the end of the chapter the learner should be able to;

- Describe the C program character set and Trigraph characters
- Describe the C Program Tokens;

  Constants, Keywords, Strings, Identifiers Special symbols and Operators
- **Declare and assign values to C variables.**
- Differentiate between Constants and Symbolic Constants

4.1. C Program Character set

A computer program consists of instructions formed using certain symbols and words according to a rigid rules called syntax rules (Grammar) of the programming language used. Each program instruction must confirm precisely to the syntax rules of the language. Like all programming languages, C has its own set of vocabulary and grammar

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. The characters in C are grouped into the following categories;

1. Letters : Upper case A ……. Z and Lower casa a ……..z.
2. Digits: 0……..9
3. Special characters

, comma	& ampersand
. period	^ caret
; semicolon	* asterisk
; colon	- minus sign
? Question mark	+ plus sign
‘ apostrophe	< opening angle bracket (less than sign
“ quotation mark	> angle bracket or greater than sign
! exclamation mark	(left parenthesis
| vertical line	) right parenthesis
/ slash	] right bracket
\ back slash	[left bracket
~ tilde	{ left brace
_ under score	} right brace
$ dollar sign	# number sign
% percent sign	

C compiler ignores white spaces (Blank, Horizontal tab, Carriage return, New line Form feed) unless they are a part of a string constant. White spaces may be used to separate words but are prohibited between the characters of keywords and identifiers.

Trigraph characters

C has the concept of "trigraph" sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character

??=	#number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
???>	} right brace
??!	| vertical line
??/	\ back slash
??/	^ caret
??-	~ tilde

4.2. C Program Tokens

In a passage text, individual words and punctuations marks are called tokens. Similarly in a C program the smallest individual units are known as C Tokens. C program has six types of tokens shown in figure 4.1 below and C is written using this tokens and the syntax of the language.

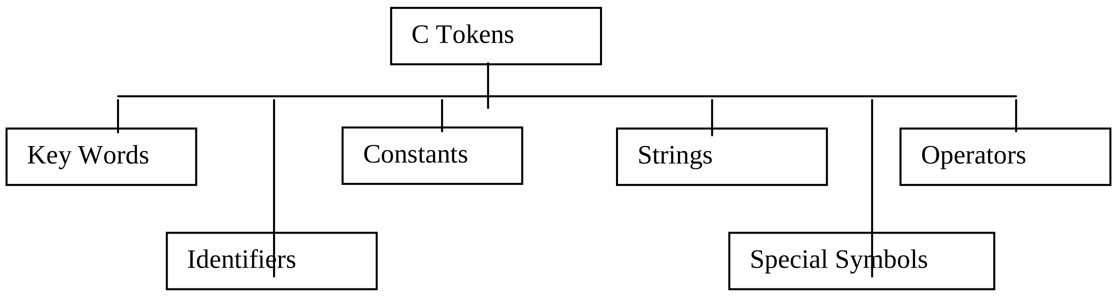


Figure 4.1 C Tokens

4.3. Reserved words / Key words and Identifiers

Reserved words (occasionally called keywords) are one type of grammatical construct in programming languages. These words have special meaning within the language and are predefined in the language's formal specifications.

Every C programs word is classified as either a ***keyword*** or ***identifier***. All keywords have the fixed meaning and these meaning cannot be changed and acts as the building blocks for program statements. List of ANSI C keywords are listed in table 3.2 below.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 4.1. ANSI C Keywords

Identifiers refer to the names of variables, functions ad arrays. These are user defined names and consists of a sequence of letters and digits, with a letter as a first character.

Rules for Identifiers

1. First letter must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore
3. Only first 31 characters are significant
4. Cannot be a Keyword.
5. Must not contain white spaces

4.4 Constants

Constants in C Program refers to fixed values that do not change during the execution of the program. Figure 4.2 below shows the types of constants supported by C program

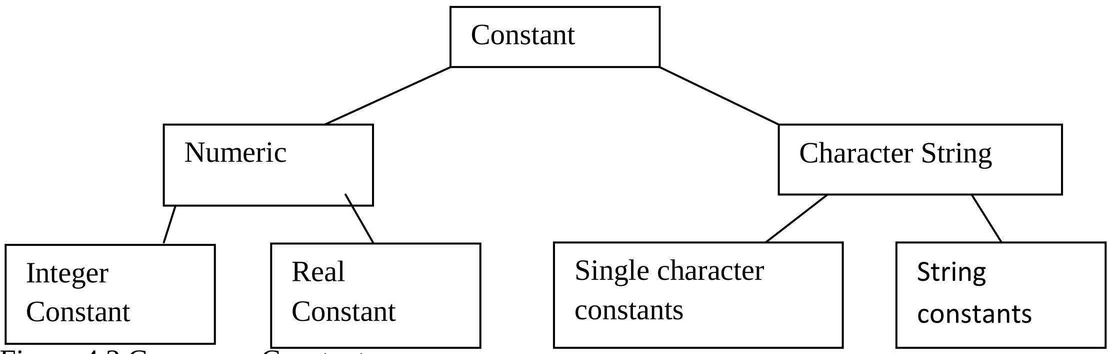


Figure 4.2 C program Constants

4.4.1. Integer Constants

An integer constant refers to a sequence of digits. There are three types of integers namely;-

1. Decimal integer: Digits 0 – 9 example +78, -321 Embedded spaces commas and non digit characters are not permitted between digits
2. Octal integer: consists of any combination of digits from the set 0 trough 7, with a leading 0. Example 037, 0, 0435, 0551
3. Hexadecimal integer: a sequence of digits preceded by 0x or 0X, they may also include alphabets A through F or through f. A through F represents number 10 to 15. Examples 0X2, 0x9F, 0Xbcd

4.4.2. Real Constants

Real Constants are used to represent quantities that vary continuously, such as distance, height, temperatures, prices etc. which integer constants are inadequate to represent. These quantities are represented by numbers containing fractional parts example 12.58. Such numbers are called real (***floating point***) constants. A real number may also be expressed in exponential (or scientific) notation example the value 215.66 may be written as 2.1566e2 in exponential notation. e2 means multiply by 10^2

4.4.3. Single Character Constants

A single character constant contains a single character enclosed within a pair of ***single quote*** marks. Example '5', 'x', ' ' etc. The character constants have integer values known as ASCII values. To find out the ASCII value for any character eg "a" use the following program

```
include<stdio.h>
include<conio.h>

main()
{
printf("%d", 'a');
getch();
}
```

The program will output 97as the output. Since each character constant represents an integer value, it is possible to perform arithmetic operations on character constants.

4.4.4. String Constants

A string constant is a sequence of characters enclosed in ***double quotes***. They may be letters, numbers, special characters and blank spaces. Example "hello', "1987", "5+5" etc. Note 'X' is not the same as "X". A string constant does not have a ASCII value.

4.4.5. Backslash Character constants

C supports some special backslash constants are used in output functions Example '\n' stands for new line. Though having two characters they represent one character, these combinations are known as ***escape sequences.*** Table 4.2 below gives a list of the C backslash constants

'\a' audible alert (bell)	'\t' carriage return	'\?' question mark
'\b' back space	'\v' vertical tab	'\\' backslash
'\f' form feed	'\'' single quote	'\0' null
'\n' new line	'\"' double quote	

Table 4.2. C program backslash constants

4.5 Variables

A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. A variable name should be chosen in a meaningful way so as to reflect its function or nature in the program. Example price, rate, total, amount etc. Naming follows the same rules as stated in the identifiers section. Valid variable names include;

- John, Value distance, Sum1 etc

Invalid variable names includes;

- 123, (area) % etc

4.6. Data types

C program is rich in its data types. Storage representations and machine instructions to handle constants differ from machine to machine. ANSI C supports three classes of data types;

1. Primary (or fundamental) data type
2. Derived data types
3. User-defined data types

The derived data type and the user-defined data types will be discussed in later chapters

4.6.1. Primary data types

All C compilers support five (5) fundamental data types namely Integer (int), Character (Char), Floating point (float), Double-precision floating point (double) and Void. Figure 4.4. below show C program primary data types. The size and range for the primary data types in a 16-bit machine are shown in table 4.5

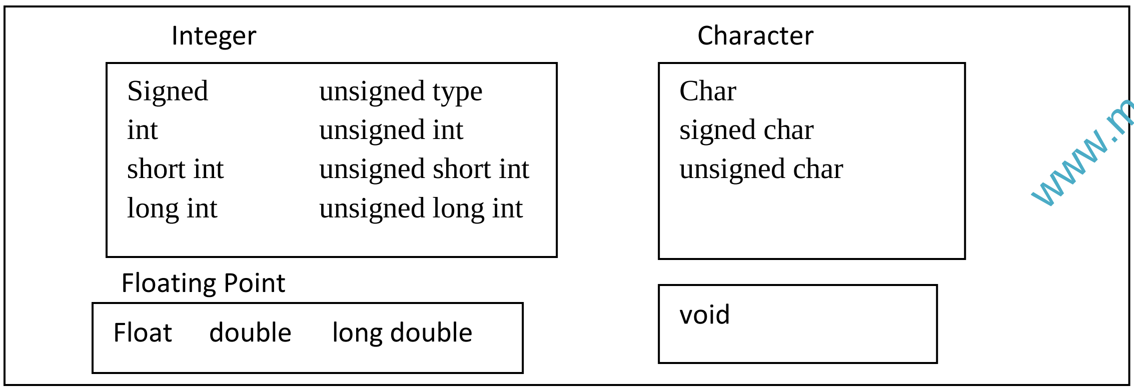


Figure 4.4. Primary data types in C

Table 4.5. Size and Range of basic data types on a 16-bit machine

Data type	Range of values
char	-128 to 127
int	-32,768 to 32,768
float	3.4e-38 to 3.4e +38
double	1.7e-308 to 1.7e+308

4.6.2. Integer data types

Integers are whole numbers with a range of values supported by a particular machine. Integer data types are defined in C as **int**. C supports three classes of integer storage, **short int, int** and **long int** in both signed and unsigned forms.

4.6.3. Floating point types

Floating point (or real) numbers are stored in 32 bits (in all 16-bit and 32-bit machines) with 6 digits of precision. Floating point data type is defined in C as **float.** When the accuracy provided by float is not sufficient, the type double can be used to define the number.

4.6.4. Void types

Void data type has no values, and usually used to specify the **void type** of function which does not return any value to the calling function example **main(void)**

4.6.5. Character type

A single character can be defined as character (char) type of data. Characters are usually stored in 8-bit (one byte) of internal storage.

4.7. Declaring variables

Variable to be used in a C program need to be declared to the C compiler. Declaration of variables does the following two things; it tells the compiler what the variable name is and it specifies what type of data the variable will hold. A variable must be declared before it is used in a C program. A variable can be used to store a value of any data type. Syntax for declaring a variable

data_type v1, v2, vn;

v1, v2 and vn are names for variables and the variables are separated with commas. A declaration statement must end with a semi-colon.

examples

```
int count;
int count, price
double ratio;
```

Declaration of variables is done immediately after the opening brace ({) in the **main()** function body. Variables can also be declared outside the **main()** function either before or after. When declared before the main() function they are called ***Global variables*** and can be used in all the functions in the program. Global variables do not need to be declared again in other functions and are called ***external*** variables. Variables declared within a function are called ***local variables*** as they are only visible and meaningful inside the function they are declared. Example of local variable declaration;

```
main()
{
int  code;
float x, y;
char  c;

statements;
}
```

4.7.1. User defined Type Declaration

C supports type definition that allows a programmer to define an identifier that would represent an existing data type. These data types can later be used to declare variable. Format;

typedef *type identifier*;

where ***type*** refers to existing data type and ***identifier*** refers to the new name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. This method only changes the name of the identifier and not the data type. Eexamples

typedef int units;

typedef float marks;

Once defined the new identifiers names can be to declare variable example

units code;

marks x, y;

4.8. Declaration of Storage Class

Variables in C have not only the data type but also storage class that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized. C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of a variable. This concept is important only in multifunction and multiple file programs.

4.9. Assigning Values to Variables

Variables are created for use in the program statements. Values can be assigned to variables using the operator '=' in a C program or through reading the values from the keyboard using the scanf() function.

4.9.1. Using the '=' operator

The format is as follows; **variable_name = value;** OR **variable_name = constant;**

The assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity on the right. Example

gross_salary = basic_salary + commission;

Variables can be assigned an initial value when declaring it in a process called ***initialization***, using the format;

data type variable_name = constant

4.9.2. Reading data from the keyboard (scanf())

Another way of assign value to variables is to input data through the keyboard using the ***scanf function***. The general format for scanf() is as follows

scanf("control string", &variable1, variable2, …..);

The control string contains the format of data being received, the ampersand **&** before the variable name is an operator that specifies the variable name's address. Example

scanf("%d", &marks);

When the computer encounters this statement, the programs stops and waits for the value ***marks*** to be keyed in through the keyboard and the <enter key> pressed. ***"%d"*** signifies that an integer data type is expected. The use of scanf() provides an interactive feature and makes the program more user friendly.

Program Example 4.1. Program to show use of Scan for interactive programming

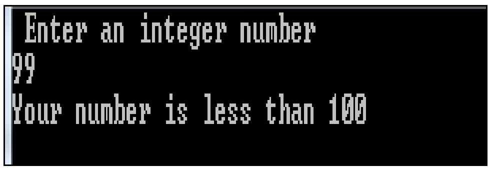
```
#include<stdio.h>
#include<conio.h>

main()
{
    int number;
   printf(" Enter an integer number\n");
    scanf("%d", &number);

    if (number > 100)     /* Statement does not end with a semi-colon */
    printf("Your number is smaller than 100\n\n");
    else
     printf("Your number is less than 100\n\n");

   getch();         /* Pause the out put on the Screen */
  }
```

Out Put



```
 Enter an integer number
99
Your number is less than 100
```

4.10. Defining Symbolic Constants

When the use of a numeric value in a program is not very clear, especially when the same value means different things in different places C allows the use of symbolic name to differentiate the different values and enhance the understandability of the program. The format for defining a symbolic constant is as follows

- ***#define symbolic_name value of the constant***

Example

- **#define PASS_MARK 50**
- **#define MAX 50**

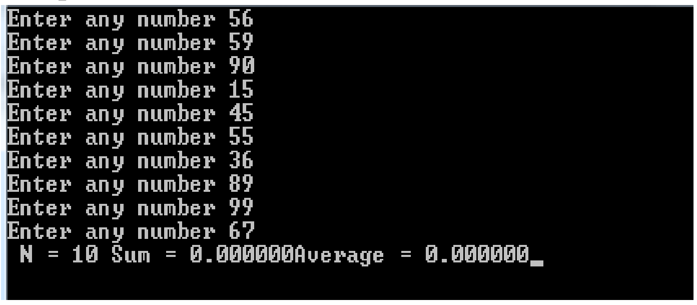
Symbolic names are constants and not variables and thus do not appear in the declaration section. The rules that apply to the #define statement which defines a symbolic constants are;

- Symbolic names have the same form as variable names. Symbolic names are usually written in CAPITAL letters to visually distinguish them from the normal variable names.
- No black space between the # and word **define** is permitted
- '#' must be the first character in the line
- A black space is required between **#define** and **symbolic name** and between the **symbolic name** and the **constant.**
- **#define** statement must not end with a semicolon.
- After definition, the symbolic name should not be assigned any other value within the program using an assignment statement. Example MAX = 200; this is illegal
- Symbolic names are NOT declared for data type. Its data type depends on the type of constant.
- **#define** statements may appear anywhere in the program but before it is referenced in the program, Usual practice is to place the #define statements at the beginning of the program.

Program Example 4.2. Program to calculate the Average of 10 number entered through the keyboard

```
#include<stdio.h>
#include<conio.h>
#define  N  10
        main()
        {
             int count;
             float sum, average, number;
             sum = 0;
             count = 0;
           while(count < N)
             {
                          printf("Enter any number ");
                          scanf("%f", &number);
               count = count + 1;
              }
        average = sum/N;
        printf(" N = %d Sum = %f", N, sum);
        printf("Average = %f", average);
        getch();
           }
```

Out put



```
Enter any number 56
Enter any number 59
Enter any number 90
Enter any number 15
Enter any number 45
Enter any number 55
Enter any number 36
Enter any number 89
Enter any number 99
Enter any number 67
 N = 10 Sum = 0.000000Average = 0.000000_
```

Chapter Review Questions

1. What is a variable and what is meant by the "value" of a variable?.
2. What is variable initialization?.
3. Find error in the following declaration statements
   int x;

```
float letter, DIGIT;
double p, q;
n, m, z INTEGER;
```

4. Write a program to read two floating point numbers using **scanf** statement assign their sum to an integer variable and then output the values of all the three variables,

CHAPTER FIVE

C- Programming:- Operators and Expressions

Chapter Objectives

By the ends of this chapter the learner should be able to;

- Describe the C program operators and their classifications
- Use C program operators appropriately and correctly in a C Program
- Describe the C program expressions
- Use C program Expression appropriately in a C program

5.1. Overview

C supports a rich set of built-in operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations on data or variables. Categories of C operators includes; Arithmetic operators; Relational operators; Logical operators; Assignment operators; Increment and decrement operators; Conditional Operators; Bitwise operators; Special operators

5.2. Arithmetic Operators

C provides all the basic arithmetic operators listed in table 5.1. below

+	Addition of unary plus
-	Subtraction of unary minus
*	Multiplication
/	Division
%	Modulo division

Table 5.1: Basic arithmetic operators

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Example

a-b a+b a*b a/b a%b -a*b

Here a and b are variables and are known as ***operands***.

5.3. Relational Operators and Logical operators

We often compare two quantities and depending on their relation make certain decision. The comparison is done using ***relational operator***. C supports six relational operators shown in table 5.1;

<	Is less than
<=`	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
==	Is equal to
!=	Is not equal to

Table 5.1 C relational operators

C supports the following three logical operators

1. && meaning logical AND
2. || meaning logical OR
3. ! meaning logical NOT

The logical operators && and || used when testing more than one condition and make a decision.

if mark >= 80 && mark < 90

An expression combining two or more relational expressions is called logical expression or compound relational expression and yields the value of either 1 or 0 / true or false.

5.3.1. Relative precedence of the Relational Operators

The relative precedence of the relational and logical operators are as follows

Highest	!
	> >= < >=
	== !=
	&&
Lowest	||

5.4 Assignment operators

Assignment operators are used to assign the results of an expression to a variable. The usual assignment operator is "=". C has a set of 'shorthand' assignment operators of the form

vop = exp;

Where v is a variable, exp is an expression and op is a C binary arithmetic operator. The operator op= is known as the shorthand assignment operator. Shorthand operators in C are shown in table 5.2. below

Statement with simple assignment operator	Statement with shorthand operator
a = a+1	a + = 1
a = a-1	a -+1
a = a*(n+1)	a *=n+1
a = a/(n+1)	a /= n+1
a = a%b	a %=b

Program Example 5.1: Program to print a sequence of squares of numbers.

```
/* documentation  section*/
#include<string.h>
#include<stdio.h>
#include<conio.h>
#define N 100
#define A 2
main()
{
    int a;
    a=A;
    while (a < N)
    {
        printf("%d\n", a);
         a *= a;
     }
         getch();
 }
```

<u>Out Put</u>

```
2
4
16
```

5.5. Increment and Decrement Operators

C allows two very useful operators not generally found in other languages called increment and decrement operators **++** and **--** operators

++ adds 1 to the operand

-- Subtracts 1 from the operand

Format **++x** or **--x**

Examples

++m is the same as m = m+1 (or m +=1;)

--m is equivalent to m = m-1 (or m -+1);

Increment and decrement operators and extensively used in the ***for*** and ***while*** loops

<u>Rules for ++ and – Operators</u>

- They are urinary operators and requires a variable as their operand
- When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by 1
- When prefix ++ (or --) is used with a variable in an expression, the expression is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of ++ and – operators are the same as those of unary + and unary -.

5.6. Conditional Operators

A ternary operator pair "?:" is used in C to construct conditional expressions. The format for using the conditional operator is;

exp1 ? exp2 : exp3

where exp1, exp2 and exp3 are expressions. If exp1 is evaluated and found to be nonzero(true) then expression exp2 is evaluated and becomes the value of the expression. If Exp1 is found to be nonzero(false) the Exp3 is evaluated.

```
 If exp=true
   then exp2
    else exp3
```

example

```
a= 10
b = 15;
x = (a>b)? a:b;
```

x will be assigned value of b

```
if (a>b)
  x = a
else
x = b
```

5.7. Special Operators

C supports some special operators namely, comma operator, size-of operator, pointer operator (& and *) and member selection operators (. &->) . the pointer and member selection will be discussed in later chapters.

5.7.1. Comma Operator

Used to link the related expressions together. A comma linked list of expressions is evaluated left to right and the value of right-most expression is the value of the combined expression

value = (x =10, y =5, x+y);

first assigns 10 to x then assigns 5 to y and finally assigns 15 (10+5) to value

5.7.2. Sizeof Operators

The sizeof operator is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable constant or a data type qualifier. Example

```
m = sizeof(sum);
n = sizeof(long int);
```

The sizeof operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer

5.8. Arithmetic Expressions

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. Example

a x b-c	= a*b –c
(m+n)(x+y)	= (m+n)*(x+y)

Expressions are evaluated using an assignment statement of the form

variable = expression;

example

```
x = a * b – c;
y = b/c * a;
```

Program example 5.2. Program to illustrate the use of variables in expressions and their evaluation

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
#define N 100
#define A 2
main()
{
     int a, b, c, d, x, y, z;
     a=9;
b = 12;
c = 3;
x = a-b/3 +c*2 -1;
y = a-b/ (3+c) * (2-1);
z = a- (b /(3+c) * 2) -1);

 printf("x =  %f\n", x);
printf("y =  %f\n", y);
printf("z =  %f\n", z);
```

```
    getch();
        }
```

Out Put

```
x = 10.000000
y = 7.000000
z = 4.000000
```

5.9. Precedence of Arithmetic Operators

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority: * / %
Low priority: + -

Rules for evaluation of Expression

- First parenthesized sub-expression from left to right is evaluated
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expression within parentheses assume highest priority.

5.9.1. Operator Precedence and Associativity

Each operator in C has a precedence associated to it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to any of these levels. Operators on the higher precedence level are evaluated first, operators at the same level are evaluated either from left-to-right or from right-to-left depending on the level. This is called ***associativity*** of the operator.

C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied. Table 5.3. below C operators precedence and associativity

Operator	**Description**	**Associativity**
()	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (change *type*)	
*	Dereference	
&	Address	

sizeof	Determine size in bytes	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x * z++;** the current value of **z** is used to evaluate the expression (*i.e.*, **z++** evaluates to **z**) and **z** only incremented after all else is done. See **postinc.c** for another example.

Example
If (x== 10 +15 && y<10)
+ has a higher precedence than the && and the relational operators && and <
Then
 if (x ==15&&y<10)
Next determine if x == to 15 and y<10. If x= 20 and y = 5 then
x==25 is false
y<10 is true

Chapter review questions

1. Which of the following expressions are true
   a). !(5+5>=10)
   b). 5 + 5 = =10 || 1+3 ==5.
   c). 10! = 15&& !(10<20) || 15>20.
2. Identify unnecessary parentheses in the following arithmetic expressions.
   a). ((x-(y/5)+z)%8) +25

b). (m*n) + (-x/y)
c). x/(3*y)

4. Find the output of the following program

```
main()
{
int x = 100;
printf("%d/n", 10  +  x++);
printf("%d/n", 10 +  ++x
```

CHAPTER SIX

C Programming:- Managing Input and Output Operations

Chapter Objectives

By the end of this chapter the learner should be able to;

- Use the C program input and Output functions appropriately
  - ✓ Link the appropriate header file
  - ✓ Use the scanf function to format inputs
  - ✓ Use the printf function to format output
  - ✓ Be able to detect errors in in puts
  - ✓ Enhance the readability of the Out put

6.

Reading, processing and writing of data are the three essential functions of a computer program. Most programs take data as input and display the processed data, often known as information or output on a suitable medium. The methods of providing data to a programs variable are;

- Assignment statements example x = 60;
- Input function scanf () which reads data from a keyboard. (Scanf stands for scan formatted)

Unlike other high-level languages, C does not have any built-in input/output statements as part of the syntax. All input/output operations are carried out through function calls such as **printf** and **scanf.** These standard input/out put functions are contained in the ***<stdio.h>*** library file. To able to call these standard input/output functions the library/header file ***<stdio.h*** must be included into the program through the command;
#include<stdio.h> (stdio.h = standard input-output header file).

6.2. Reading a Character.

The simplest of all input/output operations is reading a character from the 'standard input' unit (Usually keyboard) and writing it to the 'standard output' unit (usually the screen). In C program, reading a character can be done by using the getchar() function. The getchar function takes the following format;

variable_name = getchar();

variable_name is a valid C name that has been defined as ***char type.*** When the statement is reads, the program waits until a key is pressed and then assign the character as a value to the getchar() function. Example;

```
char  name;
name = getchar();
```

Example 6.1. Program Example: use of getchar() function to read a character

```
/* program example on use of getchar function */
#include<stdio.h>
#include<conio>
main()
{
char  answer;
printf('Would you like to know my name?\n);
printf("Type Y or YES and  N or NO:");
answer = getchar();
if (answer  == ‘Y’ || answer == ‘y’)
printf(“\n\n My name if BUSY BEE\n”);
else
printf(“\n\n You are Good for nothing\n”);

getchar();
}
```

The getchar function may be called successively to read the characters contained in n a line of text.

6.2.1. Character test functions.

Used to test whether a character is a digit, alphanumeric, lowercase, uppercase, etc. The functions are contained in the library header **<ctype.h>**

Function	**Test**
isalnum(c)	Is c alphanumeric?
isalpha(c)	Is c an alphabetical character?
isdigit(c)	Is c a digit?
islower ©	Is c lower case letter?
isprint(c)	Is c printable character?
ispunct(c)	Is c a punctuation mark?
Isspace(c)	Is c a white space character?
Isupper(c)	Is c upper case letter?

Example 6.2. Program to test the character type

```
#include<stdio.h>
#include<conio.h>
#include>ctype.h>
main()
{
char character;
printf(“Press any key\n”);
character = getchar();i
if (isalpha(character) > 0) /* Test for letter */
  printf(“The character  is a letter”);
 else
 if (isdigit(character) > 0)
 printf(“The character is a digit”);
else
 printf(“The character is not alphanumeric’):

getch();
}
```

6.3. Writing a character.

Like getchar there is an analogous function ***putchar*** for writing characters one at a time to the terminal. It takes the form;

putchar (variable_name);

the variable name is a type char variable containing a character.

Example 6.3 Program to show use of putchar function and to convert case of a character

```
 #include<stdio.h>
 #include<conio.h>
 #include>ctype.h>
 main()
 {
 char alphabet;
printf (“Enter any alphabet\n”);
putchar (“\n”);  /* move to the next line*/
alphabet = getchar();
if (islower(alphabet))
putchar (toupper (alphabet)); /* Reverse the case and display */
else
putchar(tolower(alphabet));

getch();
}
```

6.4. Formatted Input

Formatted input refers to an input data that has been arranged in a particular format. C program uses the scanf function to format in the inputs. The scanf function takes the following format;

scanf(“control string”, arg1, arg2, ….);

The control string specifies the field format in which the data is to be entered and the arguments arg1, arg2 specify the address of locations where the data is stored. Control string and the arguments are separated by commas. Control string (also called format string) contains field specifications which direct the interpretation of input data and includes;

- Field (or format) specification, consisting of the conversion character %, a data type character (or type specifier), and an optional number, specifying the field width
- Blank, tabs and newlines

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional.

6.4.1. Inputting Integer Numbers

The field specification for reading an integer number is

% w sd

% - specifies a conversion specification follows. ***w*** is an integer number that specified the field width of the number to be read and ***d*** is the data type character and indicates the number to be read is a integer. Consider the following example;

```
scanf(“%2d  %5d”, &num1, &num2);
```

If number entered are 50, 31426, 50 will be assigned to ***num1*** and 31426 will be assigned to ***num2***. The output will be 50, 31425. If number entered are 31426, 50, the ***num1*** variable will be assigned 31 (because of the %2d) and ***num2*** variable will be assigned the remaining part - 426 (unread part of 31426). The output will thus be 31 426

Input data must be separated by spaces, tabs or new lines, punctuation marks do not count as separators. The input data should not contain more digits than specified by the input format example, %2d requires 2 digits maximum, while %5d will require maximum of 5 digits.

Program example 6.4. Reading integers using scanf function

```
#include<stdio.h>
#include<conio.h>
#include>ctype.h>
main()
{
int a, b, c, x, y
printf("enter three integer numbers\n");
scanf("%d %d %d", &a, &b, &c);
printf("Enter two 4 digit integer numbers\n");
scanf("%2d %4d", &x, &y);
getch();
}
```

Output

```
Enter three integer number
1 2 3
1 3 -3577

Enter two 4 digit integer number
6789 4356
67 89
```

6.4.2. Inputting Real (floating point numbers)

Unlike the integers the width of the real numbers is not to be specified, scanf reads real numbers using the simple %f for both the decimal point notation and exponential notations. Example
scanf (%f %f %f", &x, &y, &z); with the input 475.90 43.25 689.0, 475.90 will be assigned to x, 43.25 will assigned to y and 689.0 will assigned to z.

Program example 6.5. Reading float numbers using scanf function

```
#include<stdio.h>
#include<conio.h>
#include>ctype.h>
main()
{
float a, b, c, x, y
printf("Enter Values for A and B\n");
scanf("%f %f", &a, &b);
printf("x = %f\n y = %f\n\n", x, y

printf("Enter values for X and Y \n");
scanf("%1f %1f", &x, &y);
printf("\n\n x = %.12f\n y = %12e". x y);
getch();
}
```

```
Output
Enter values or A and B
199.75    12.567
a = 199.95
b = 12.567
Enter values for X and Y
4.123458967  18.56768974363

x = 4.123458967
y = 18.56768974363e001
```

6.5. Inputting Character Strings

The scanf function format for inputting character strings are;
scanf(%ws) or scanf(%wc);
When we use ***%wc*** for reading a string the program waits until the wth character is keyed in.
%s terminates reading at the encounter of a blank space.

6.6. Reading mixed Data types

C allows inputting of strings of mixed data types as long as the data items match the control specifications in ***order and type***. When an attempt is made to read an item that does not match the type expected, the scanf function does not read any further and immediately returns the values read. Example;
scanf("%d %c %f %s", &count, &code, &ratio, name); will read the data
15 p 1.575 coffee correctly and assign the values to the variables in the order in which they appear.

6.7. Detecting Errors in Input

When a scanf function completes reading its list, it returns the value of number of items that are successfully read. When scanf() encounters an error example in data type, it returns the number of the last correctly read value. Example ***scanf("%d %f %s", &a, &b, name)***; will return 1 when 20, motor, 15.25 data is input.

Program Example 6.6. Program to test correctness of the data input

```
#include<stdio.h>
#include<conio.h>
#include>ctype.h>

main()
{
 int a
float f;
char c;
printf("Enter values of a, b, c\n");
if scanf("%d %f %c", &a, &b, &c) ==3)
  printf("a = %d, b = %f  c = %c\n", a, b, c);
else
printf("Error in input. \n");
getch();
}
```

```
Out put
Enter values for a, b, c
12 3.45 A
a = 12, b = 3.450000  c = A

Enter values for a, b, c
23  78  9
a = 23   b = 78.00000   c =9
```

6.8. Commonly used Scanf formats

%s	Read a string
%d	Read an integer
%f	Read a floating point value
%c	Read a single character
%i	Read a decimal, hexadecimal or octal integer

Note when using scanf

- All function arguments, except the control string must be pointers to variables
- Format specification contained in the control string should match the arguments in order
- Scanf terminates if it encounters a mismatch.
- Input data items must be separated by spaces and must match the variables receiving the input data in the same order.
- Any unread data items in a line is considered as part of the data input line in the next scanf call.
- When the field width specifier w is used it should be large enough to contain the input data size.

Rules of scanf()

- Each variable to be read must have a filed specification
- For each field specification, there must be a variable address of proper type
- Any non-whitespace character used in the format string must have a matching character on the user input.
- Never end the format string with whitespaces. It is a fatal error.
- The scanf reads until
    - A white space character is found in a numeric specification or
    - The maximum number of characters have been read or
    - The end of file is reached.

6.9. Formatted Out put

The printf function is used to format the C program out puts. The printf statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminal. The printf function format is

printf(" control string", arg1, arg2,arg3, …..);

The control string consists of;

- Characters that will be printed on the screen as they appear
- Format specification that defines the output format for display of ach item.
- Escape sequence characters such as \n (for new line), \t (for tab) and \b.

The control string indicates how many arguments follow and what their types are. The arguments ***arg1, arg2***, are the variables whose values are formatted and printed according to the specifications of the control string.

The argument should match in number, order and type with the format specifications. A simple format specification has the following form;

Format (%w.p type-specifier)

w = integer number that specifies the total number of columns for the output value, p is another integer number that specifies the number of digits to the right of the decimal point or the number of characters to be printed from a string. Both w and p are optional.

Printf function never supplies a newline automatically and therefore multiple printf statements may be used to build one line of output.

6.9.1. Formating Integer and Real numbers output

The format specification for printing an integer number is;-

% w d

w specifies the minimum field width for the output, however if the number is greater than the specified field width, it will print in full overriding the minimum specification.

The format specification for printing a real number is;

% w.p f

w specifies the minimum field width for the output, p an integer indicates the number of digits to be displayed after the decimal point (precision). The value when printed out is rounded to p decimal places and printed right-justified.

Examples of integer and real numbers formatting

Format	**Out put**
printf("%6d", 9867);	_ _ 9876
prinff("%-6d",9867);	9876__
printf("%o6d", 9876);	0098676
printf("%7.4f", 98.7654)	98.7654
printf("%7.2f", 98.7654)	98.77
printf("%-7.2f", 98.7654)	98.77
printf("%f", 98.7654)	98.7654

Program Example 6.7. Formatted output for Real numbers

```
#include<stdio.h>
#include<conio.h>
#include>ctype.h>
main()
{
 float  y = 98.7654;
printf("%7.4f\n" , y);
printf("%f \n", y);
printf("%-7.2f\n" , y);
printf("%-07f \n", y);

getch();
}
```

```
Out put
98.76754
98.765404
98.77
98.77
```

6.10. Commonly used Printf Formats

%d	Out put for integer
%f	Out put for floating number
%c	Out put for single character
%s	Out put for string

6.11. Enhancing readability of Output.

- Provide enough blank spaces between two numbers.
- Introduce appropriate headings and variable names in the output
- Print special messages whenever a peculiar condition occurs in the output.
- Introduce blank lines between the important sections of the output.

Chapter Review Questions

1. State whether the following statements are true or false
   a) The purpose of the header file <studion.h> is to store the programs created by the user
   b) The C standard function that receives a single character from the keyboard is getchar.
   c) Format specifiers for out put convert internal representations for data to readable characters
2. Write scanf statements to read the following data lists
   a) 78 B 45
   b) 123 1.23 45A
   c) 15 – 10 – 2011

3. The variables count, price and city have the following values

Variable	Value
count	1275
price	235.75
City	Nairobi

Show the exact output that the following output statements will produce;

a) printf(“%d %f \”, count, price);
b) printf(“%2d\n   %f”, count, price);
c) printf(“%d %f”, price, count);

4. Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form;
   35.7   50.21 -2373   -46.45