P.O. Box 342-01000 Thika
Email: info@mku.ac.ke
Web: www.mku.ac.ke

DEPARTMENT OF INFORMATION TECHNOLOGY

BACHELOR OF BUSINESS INFORMATION TECHNOLOGY (BBIT)

COURSE CODE: BIT 2201

COURSE TITLE: COMPUTER PROGRAMMING METHODOLOGY

(INTRO. TO C PROGRAMMING)

INSTRUCTIONAL MANUAL FOR BBIT – DISTANCE LEARNING

PREPARED BY MERCY MAINA

BIT 2201: COMPUTER PROGRAMMING METHODOLOGY
Pre-requisite: BIT: 1203 BASIC DISCRETE MATHEMATICS

Course Purpose:

- To introduce structured computer programming methods of advanced algorithms for writing programs to solve general problems.

Course objectives:

By the end of the course unit, the student should be able to;

- Describe the structured computer programming paradigm.
- Use data control structures and advanced computer algorithms.
- Apply a high-level programming language like C/C++ and Pascal
- Write computer programs or commands to solve general problems.

Course Content

- Define; algorithm; a problem, including traitable and intractable
- Problem solving techniques; like -Documentation or solution, definition, down coding etc -top-down problem solving -drawing flowchart to represent solution
- Structured programming -algorithms, pseudo code, syntax, semantics, data types, expressions, statements, variable, input-output, control structures
- Sub-programming or modularization (to solve complex problems using functions and procedures (sub routines)
- Introducing functional and procedural observations and data abstractions
- File management and programming standards
- Writing correct and efficient programmes
- Designing and analyzing algorithms
- Mathematical foundations or algorithm design, summations, recurrences converting and probability
- Dynamic programming

Teaching/Learning Methodology

- Lectures
- Tutorials
- Computer laboratory exercises

Instructional Materials/Equipment: Audio visual aids in lecture rooms

- Computer hardware and software
- High programming language like C

Course Assessments: Continuous assessment 30% Examination 70% Total 100%

Required Textbooks:

1. Wu T Norman Theodore 1997, An introduction to programming, MC-Graw Hill
2. Balagurusamy E., Programming in ANSI C. TATA, MC-Graw Hill
3. Brian K.W. & Ritche D.M, The C Programming language, Prentice-Hall

Textbooks for further reading

1. Baday K. ANSI C, Problem solving and programming, Prentice-Hall
2. Brian K.W. et al, C programming Language 2nd edition, Prentice -Hall
3. Cormen et al, An introduction to Algorithms, MIT press

Other Support materials: C program compiler, various applicable manuals, computer packages, and journals Variety of electronic information resources as may be prescribed by the lecturer.

TABLE OF CONTENT

Page

CHAPTER ONE 10

Overview of Computer Programming Methodology 10

1.1 What is Computer Programming Methodology 10

1.2. Problem Solving with Computer; 10

1.2.1. Problem Algorithm 11

1.3. Flow Charts 12

1.3.1. Elements of a Flowchart. 12

1.4. Pseudo code 13

1.5. Program Control Structures 14

1.6. Programming Methods 15

1.6.1 Top-down and Bottom-up methodology 15

1.6.2. Structured programming Method 17

1.7. Programming Aims 18

1.8. Programming Paradigms 18

1.8.1. Imperative programming; 18

1.8.2. Object-oriented (OO) programming: 19

1.8.3 Functional Programming: 19

1.8.4. Logic Programming 19

Chapter Review Questions 19

CHAPTER TWO 20

Programming Languages 20

2.1. What is a Programming Language 20

2.2. Types of Programming Languages 21

2.2.1. Machine language 21

2.2.2. Assembly Language 21

2.2.3. High level language 22

2.3. High Level language Translation 23

2.3.1. Compiler 23

2.3.2. Interpreter: 24

2.4. Computer Program Compilation Process 24

2.5. Evaluating Languages 24

2.6. The Programming Language Generations 25

Chapter Review Questions 26

CHAPTER THREE 27

Introduction to C Programming 27

3.2. Characteristics of C 28

3.3. Executing a C program 29

3.4 Compiling a C Program 29

3.5. C Program File naming convention 30

3.6. Errors 31

3.6.1. Syntax Error 31

3.6.2. Logical or Intention Error 31

3.7. C Libraries 32

3.8. C Program Structure 32

3.9 C Program format 33

1.9.1. Documentation Section 34

1.9.2. Link Section 34

1.9.3. Definition Section 34

1.9.4. Declaration Section 34

3.9.5 main() Function Section 34
3.9.6 Sub-program section 35
3.10. Breaking out early 37
Chapter Review Questions 38
CHAPTER FOUR 39
C Programming: - Constants, Variables and Data Types 39
4.1. C Program Character set 39
4.2. C Program Tokens 40
4.4.1. Integer Constants 42
4.4.2. Real Constants 42
4.4.3. Single Character Constants 42
4.4.4. String Constants 42
4.4.5. Backslash Character constants 43
4.5 Variables 43
4.6. Data types 43
4.6.1. Primary data types 43
4.6.2. Integer data types 44
4.6.3. Floating point types 44
4.6.4. Void types 44
4.6.5. Character type 44
4.7. Declaring variables 45
4.7.1. User defined Type Declaration 45
4.8. Declaration of Storage Class 46
4.9. Assigning Values to Variables 46
4.9.1. Using the '=' operator 46

4.9.2. Reading data from the keyboard (scanf()) 46

4.10. Defining Symbolic Constants 47

CHAPTER FIVE 50

C- Programming:- Operators and Expressions 50

5.1. Overview 50

5.2. Arithmetic Operators 50

5.3. Relational Operators and Logical operators 50

5.3.1. Relative precedence of the Relational Operators 51

5.4 Assignment operators 51

5.5. Increment and Decrement Operators 52

5.6. Conditional Operators 53

5.7. Special Operators 53

5.8. Arithmetic Expressions 54

5.9. Precedence of Arithmetic Operators 55

Chapter review questions 57

CHAPTER SIX 58

C Programming:- Managing Input and Output Operations 58

6.1. Introduction 58

6.2. Reading a Character. 58

6.2.1. Character test functions. 59

6.3. Writing a character 60

6.4. Formatted Input 61

6.4.1. Inputting Integer Numbers 61

6.4.2. Inputting Real (floating point numbers) 62

6.5. Inputting Character Strings 63

6.6. Reading mixed Data types 63

6.7. Detecting Errors in Input 63

6.8. Commonly used Scanf formats 64

6.9. Formatted Out put 65

6.9.1. Formating Integer and Real numbers output 65

6.10. Commonly used Printf Formats 66

6.11. Enhancing readability of Output. 67

Chapter Review Questions 67

CHAPTER SEVEN 68

C Programming:- Decision Making and Branching 68

7.1. Introduction 68

7.2. Decision making with if statement 68

7.2.1. Simple if statement 69

7.2.2. The If …… Else Statement 70

7.2.3. Nesting *if* .... *Else* Statements 72

7.2.4. The Else If Ladder 74

7.3. Rules for indentation in the *if*…. *else* statements 76

7.4. Switch Statement 77

7.4.1. Rules for Switch 79

Chapter Review questions 79

CHAPTER EIGHT 80

C Programming:- Decision Making and Looping 80

8.1 Introduction 80

8.2. Classification of Loop Control Structures; 80

8.3. The While Statement 81

8.4. The Do Statement 83

8.5. The For Statement 84

8.6. Nesting of for loops 86

8.7. Selecting a Loop 87

8.8. Jumping out of a Loop 88

8.9 Structured programming 88

Chapter Review Questions 89

CHAPTER NINE 90

C- Programming: Arrays 90

9.1 Introduction 90

9.2. Data structures in C 90

9.3. One-Dimensional Arrays 91

9.4. Declaration of One-Dimensional Arrays. 91

9.5. Initialization of One-Dimensional Arrays. 92

9.5.1. Compile time initialization. 92

9.5.2. Run Time Initialization 93

9.6. Searching and Sorting using Arrays 94

9.7. Two – Dimensional Arrays 95

9.7. Initializing Two-Dimension Arrays 98

9.8. Character Arrays and Strings 98

9.9. Declaring and Initializing String Valuables. 98

9.10. Terminating with Null Character. 99

9.11. Reading Strings from Terminals 99

9.13. Writing Strings to Screen. 101

9.14. String Handling Functions 102

9.14.1. Strupr() function 102

9.14.2. Strcat() function 102

9.14.3. Strcmp() Function 102

9.14.4. Strcpy() Function 103

Chapter Review Questions 104

CHAPTER TEN 105

C- Programming:- User Defined Functions 105

10.1. Introduction 105

10.2. Need for user-defined functions 105

10.3. Modular programming; 106

10.4. A Multi-Function Program. 106

10.5. Elements of User-Defined Functions 107

10.6. Definition of Functions 108

10.6.1 Function header 108

10.6.2. Function Body 109

10.7. Return values and their types 109

10.8. Function call 110

10.9. Function Declaration 111

10.10. Categories of Functions 112

10.11. Nesting of Functions 118

10.12. Recursion 119

Chapter Review Questions 123

CHAPTER ELEVEN 124

C- Programming: Structures and Pointers 124

11.2. Arrays vs Structures 125

11.3. Declaring Structure Variables.. 125

11.4. Accessing Structure Members 126

11.5 Structure initialization... 127

11.7. Pointers 130

11.7.1. Accessing the Address of a Variable 131

11.7.2. Declaring Pointer Variables.. 132

11.7.3. Pointer declaration styles 132

11.7.4. Initialization of Pointer variables.. 132

11.7.5. Accessing a Variable through its pointer 133

11.8. Pointers and Arrays.. 134

Chapter Review Questions.. 135

CHAPTER TWELVE.. 136

C- Programming: Managing Files in C.. 136

12.1. Introduction.. 136

12.2. Defining and Opening aFile.. 137

12.3. Closing a File 138

12.4. Input / Output Operations on Files 138

12.5. The getw and putw Functions.. 140

12.6. The fprintf and fscanf Functions... 141

12.7. Error Handling during I/O Operations.. 143

Chapter Review Questions.. 145

Sample Examination Papers 146

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 Calculate wages
	Diamond:	The decision Symbol: Used where a decision has to be made in selecting the subsequent path to be followed. Is the transaction a credit?
		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 Start Stop
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
		No Yes
		?
		
		?
		a b c

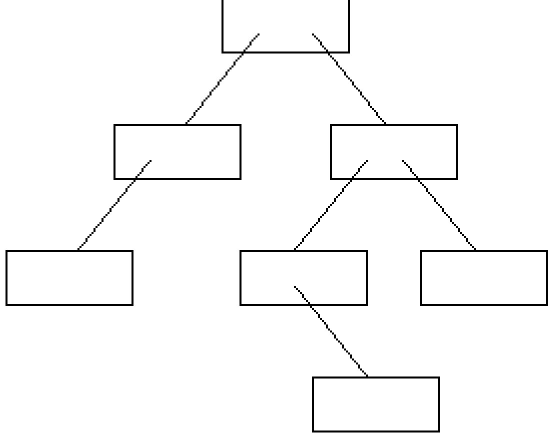
Repetition also called "looping"
There are many programming problems in which the same sequence of statements needs to be performed again and again for a definite or indefinite number of times
	WHILE
	condition
	DO
	Actions
	ENDWHILE
	
	REPEAT
	actions
	UNTIL
	condition
		?
		No
		
		No
		?
		Yes

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 messsages 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

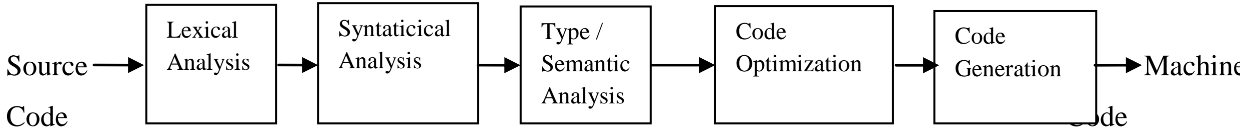
The commonly interpreted languages include BASIC and PERL. Though interpreters are easier to create as compared to compilers, the compiled languages can be executed more efficiently and are faster. Interpreters are appropriate in;

- 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.

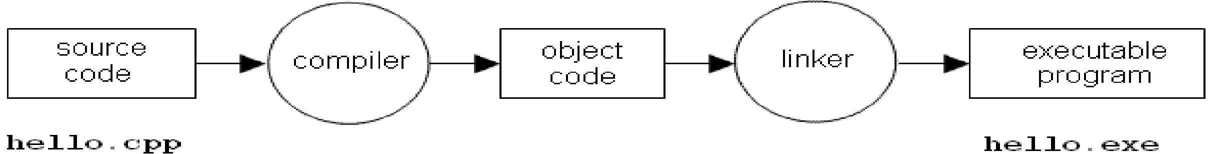
3.3. Executing a C program

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.

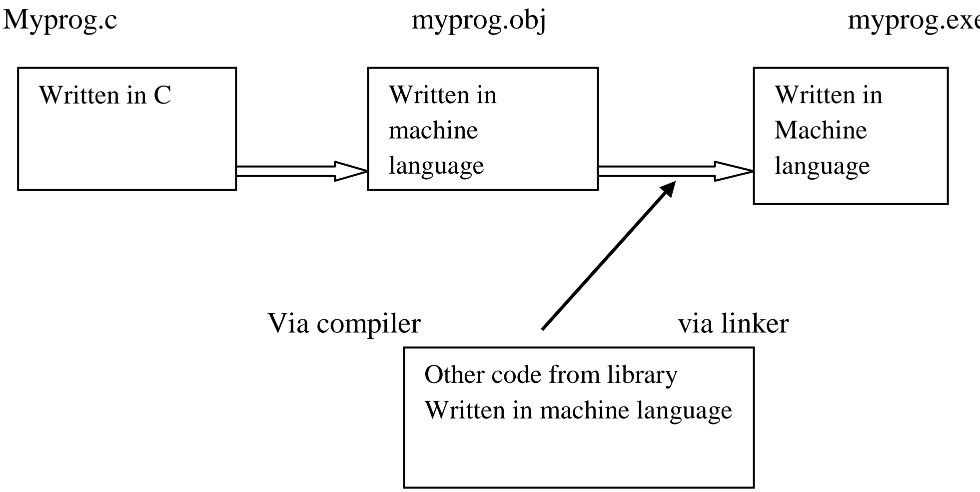
3.5. C Program File naming convention

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

The command filename <enter> will execute the program and give results.



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 ungratifying 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()	/* program for addition */
{	#include<stdio.h>
printf ("This is my \n");	#include<conio.h>

printf(“computer book”);	main()
getch();	{ int number;
}	float amount;
OR	number = 100;
main(()	amount = 30.75 + 75.35;
{	printf("%d\n", number);
print(“This is \n my computer book”);	printf("%5.2f", amount);
getch();	getch(); /* pause the results on the screen */
}	}

Program Example 3.3	**Program Example 3.4**
/* program to print Hello */	/* program to calculate the Interest rate */
#include <stdio.h>	#include <stdio.h>
#include<conio.h>	#include<conio.h>
	#define PERIOD 10
int main()	#define PRINCIPAL 5000.00
{	int main()
printf("hello, world\n");	{
getch();	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?
2. Name the most common library file in C.
3. Is it possible to define new functions with the same names as standard library functions?
4. What is another name for a library file?
5. Describe the structure of C Program
6. Distinguish between
   a) main() and main(void)
   b) int main() and void main()
7. 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

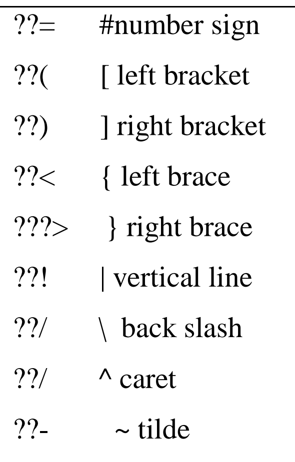
, comma	& ampersand
. period	^ caret
; semicolon	* asterisk
; colon	- minus sign
? Question mark	+ plus sign
‘ apostrophe	< opening angle bracket (less than sign
“ quotation mark	
! exclamation mark	> angle bracket or greater than sign
| vertical line	(left parenthesis
/ slash	) right parenthesis
\ back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

$ dollar sign	} right brace
% percent sign	# number 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.

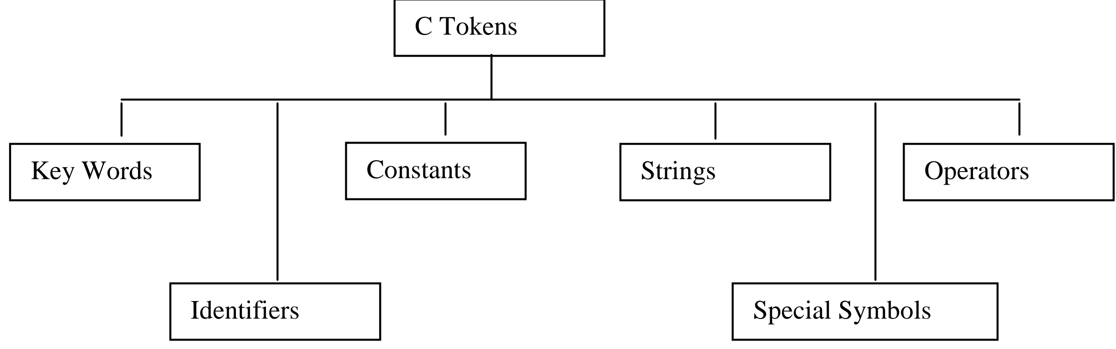


Figure 4.1 C Tokens

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

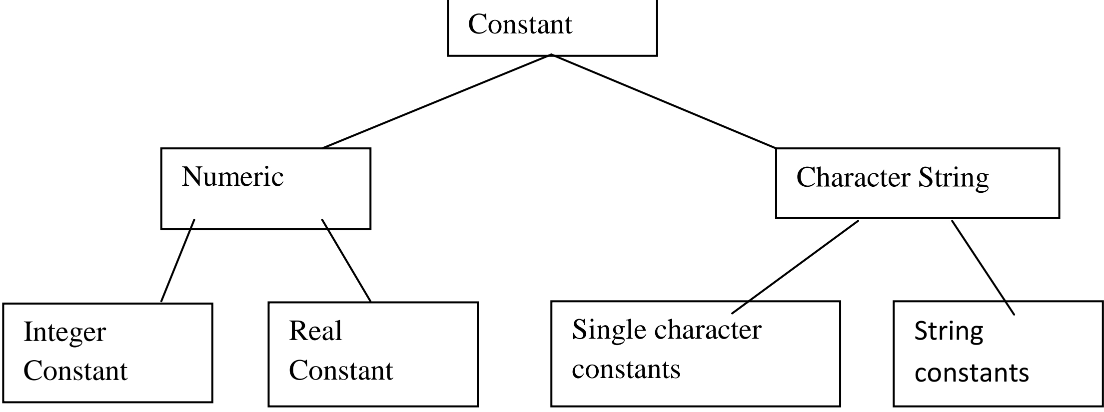


Figure 4.2 C program Constants

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

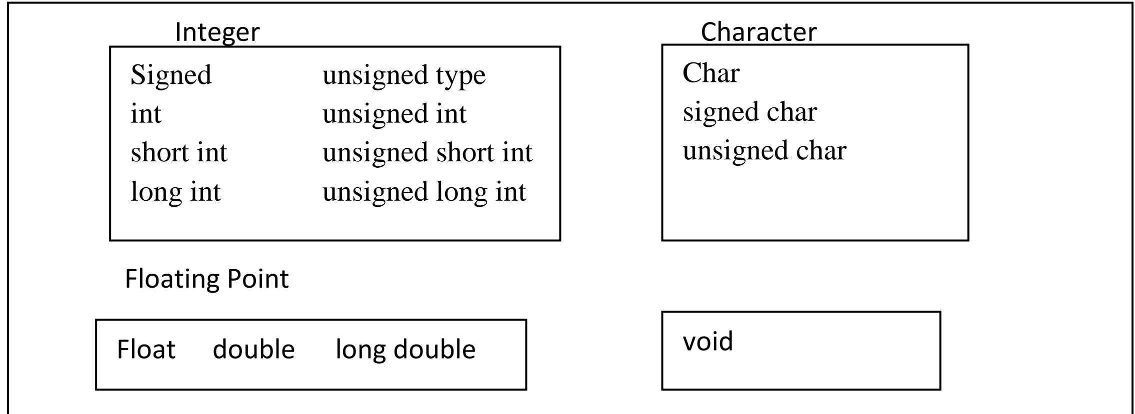


Figure 4.4. Primary data types in C

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

Rules for ++ and – Operators

- 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	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	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	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	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

6.1. Introduction

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();
}
```

<u>Out put</u>

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

CHAPTER SEVEN

C Programming:- Decision Making and Branching

Chapter Objectives

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

Describe and use appropriately the C decision making and branching control structures

- C program if statement
  - ✓ C program Simple if statements
  - ✓ C program If.. Else statements
  - ✓ C program Nesting of If.. Else statements
  - ✓ Else If ladder
- Switch statements
- Goto statement

7.1. Introduction

C language possesses such decision-making capabilities by supporting the following statements; if statements; switch statements; conditional operator statements; goto statement. This statements are commonly referred to as control statements because they 'control' the flow of program execution.

7.2. Decision making with if statement

The if statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is a two-way decision statement (Fig 7.1) and is used in conjunction with an expression, and takes the following form;

if (test expression)

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (non-zero) or 'false (zero), it transfers the control to a particular statement.

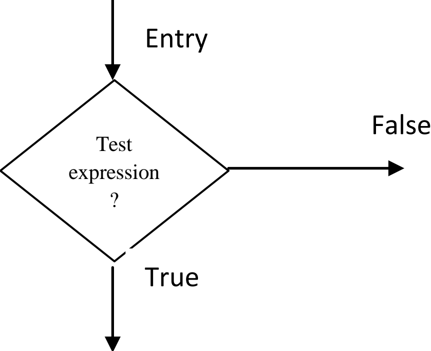


Fig 7.1 Two –way branching

Examples

If (room is dark)

Put on lights

If (code = 1)

Person if male

The if statement may be implemented in different forms depending on the complexity of conditions to be tested, example;

1. Simple if statement
2. If …. else statement
3. Nested if ….. else statement
4. Else if ladder

7.2.1. Simple if statement

The general form of a simple ***if*** statement is

```
  if  (test expression)
{
  statement block;
}
statement - x
```

The statement block may be a single statement or a group of statements. If the test expression is true the statement block is executed; otherwise it will be skipped and the program control jumps to statement –x. When the test expression is true both the statement block and statement-x are executed in sequence (Fig 7.2).

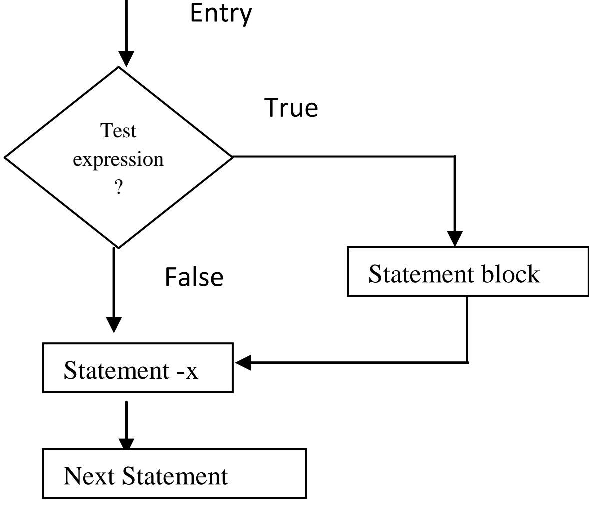


Fig 7.2 Flow diagram for a simple ***if statement***

Program example 7.1 Illustration of Simple if statement

```
#include<stdio.h>
#include<conio.h>
#include>ctype.h>
main()
{
int a, b, c, d;
float ratio;

printf("Enter four integers values\n");
scanf("%d, %d %d %d" &a, &b, &c, &d);
    if (c-d !=0)
{
    ratio = (float)(a+b) / (float)(c-d);
printf("Ratio = %f\n", ratio);
}
getch();
}
```

```
Out put
Enter four Integers values
12 23 34 45
Ratio = -3.181818

Enter for Integer values
12 23 34 34
```

The first test data will produce the answer ration = 3.181818, while the second test data will not give any answer as (c-d) equals zero, thus the statement block is skipped.

7.2.2. The If …… Else Statement

The ***if … else*** statement is an extension of the simple ***if*** statement. Its general form is;-

```
  if (test expression)
     {
      true-block statement(s)
     }
else
     {
       false-block statement(s)
     }
statement-x
```

If the test expression is true then the true-block statements(s), immediately following the if statement are executed; otherwise the false-block statements are executed. Either the true-block statement(s) or the false-block statement(s) will be executed and both. Fig 7.3 illustrates this

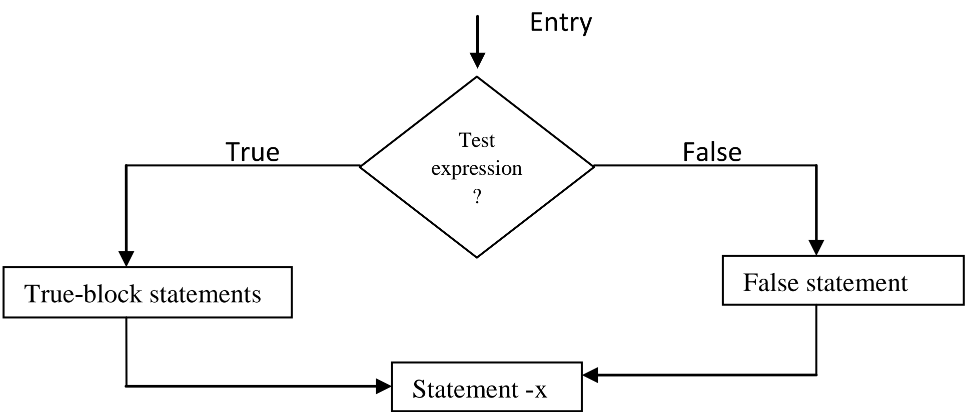


Fig. 7.3. Flowchart for ***if …. else*** control

Example the following code

```
if (code == 1)
 boy = boy + 1;
if (code == 2)
   girl = girl + 1;
  statement -x
```

can be written as

```
if (code ==1);
  boy = boy + 1
else
  girl = girl + 1;
statement-x
```

Program example 7.2. illustration of *if … else* control

The previous simple if statement program example can be re-written as follows;

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

 main()
 {
 int a, b, c, d;
```

```
float ratio;

printf("Enter four integers values\n");
scanf("%d %d  %d  %d", &a, &b, &c, &d);
if (c-d !=0)
    {
    ratio = (float)(a+b) / (float)(c-d);
     printf("Ratio = %f\n", ratio);
    }
else
printf("c-d is zero\n");

getch();
}
```

Use the same test data used in program example 7.1

7.2.3. Nesting *if …. Else* Statements

When series of decisions are involved, we may have to use more than one ***if .. else*** statements in nested form. The general form for nested ***if…. else*** statement is;

```
   if  (test-conditions-1)
   {
      if (test-conditions-2);
       {
              statement-1;
       }
  else
     {
        statement - 2;
         }
}
else
        {
     statement-3;
        }
statement-x
```

If the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If test condition-1 is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x. Fig 7.4. illustrates this.

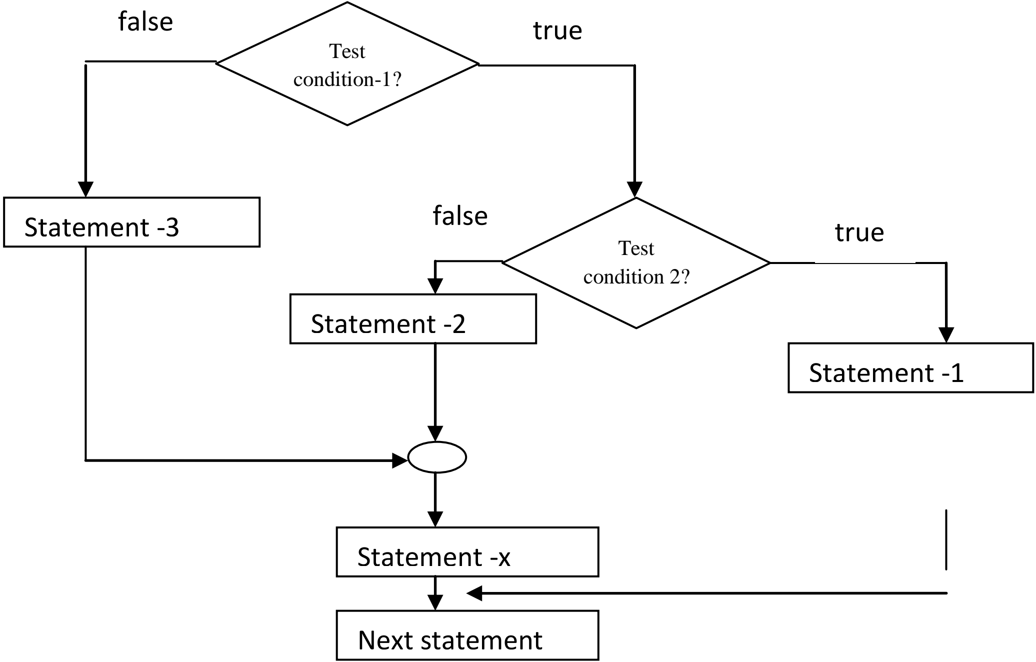


Fig 7.4. Flow chart for **Nested *if … else*** control

When nesting care should be exercised to match every ***if*** with ***else***.

A bank's policy is to a 2% bonus on the ending balance at the end of the year (31st December) irrespective of the balance, and a 5% bonus is also given to female account holders if the balance is more than 50,000. The logic can be coded as;

```
……
main()
if (sex is female)
{
        if (balance > 50000)
        {
                    bonus = 0.05 * balance;
else
                    bonus = 0.02 * balance;
}
else
{
 bonus = 0.02 * balance;
}
balance = balance + bonus;
………..
}
```

Program example 7.4.: Illustration of *if .. else* control to print the largest number of three numbers entered

```
#include<stdio.h>
#include<conio.h>
main()
```

```
{
float a, b, c;
printf("Enter three values\n");
scanf("%f   %f   %f", &a,  &b,  &c);
printf("\n Largest value is\n ");
  if (a>b)
          {
           if (a>c)
            {
               printf("%f\n", a);
            }
          else
            printf("%f\n ", c);
            }
else
       {
            if (c>b)
                printf("%f\n", c);
            else
                printf("%f\n", b);
        }
      getch();
}
```

Out Put

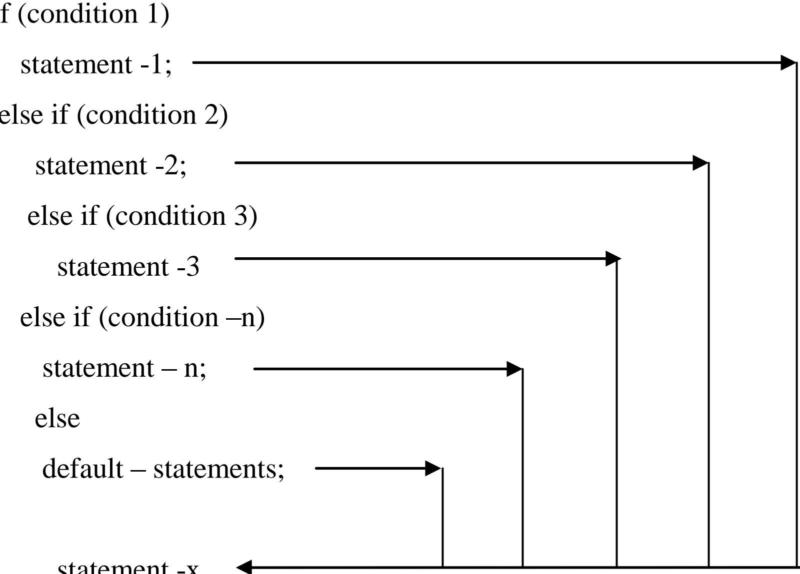
Enter three values

23455 67379 88845

Largest value is ***88845.0000***

7.2.4. The Else If Ladder

Another way of putting the ***ifs*** together when multipath decisions are involve is the ***else if*** ladder. A multipath decision is a chain of ifs in which the statement associated with each ***else*** is an ***if***. It take the following general form;



The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with it is executed and the normal control is transferred to the statement-x (skipping the rest of ladder). When all the n conditions become false then the final else containing ***default – statement*** will be executed.

The else .. if ladder can be used in the grading system for an academic institution. Suppose the grading is done following the following rules;

Average marks	**Grade**
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail

The logic can be implemented as follows;

```
main()
{
--------
  if (marks > 79)
       grade = " Honours";
  else if (marks > 59)
        grade = "First Division";
   else if  (marks > 49;
        grade = "Second Division";
      else if (marks > 39)
```

```
        grade = "Third Division";
    else
            grade = "Fail";
  printf ("the grade is \n", grade);
}
```

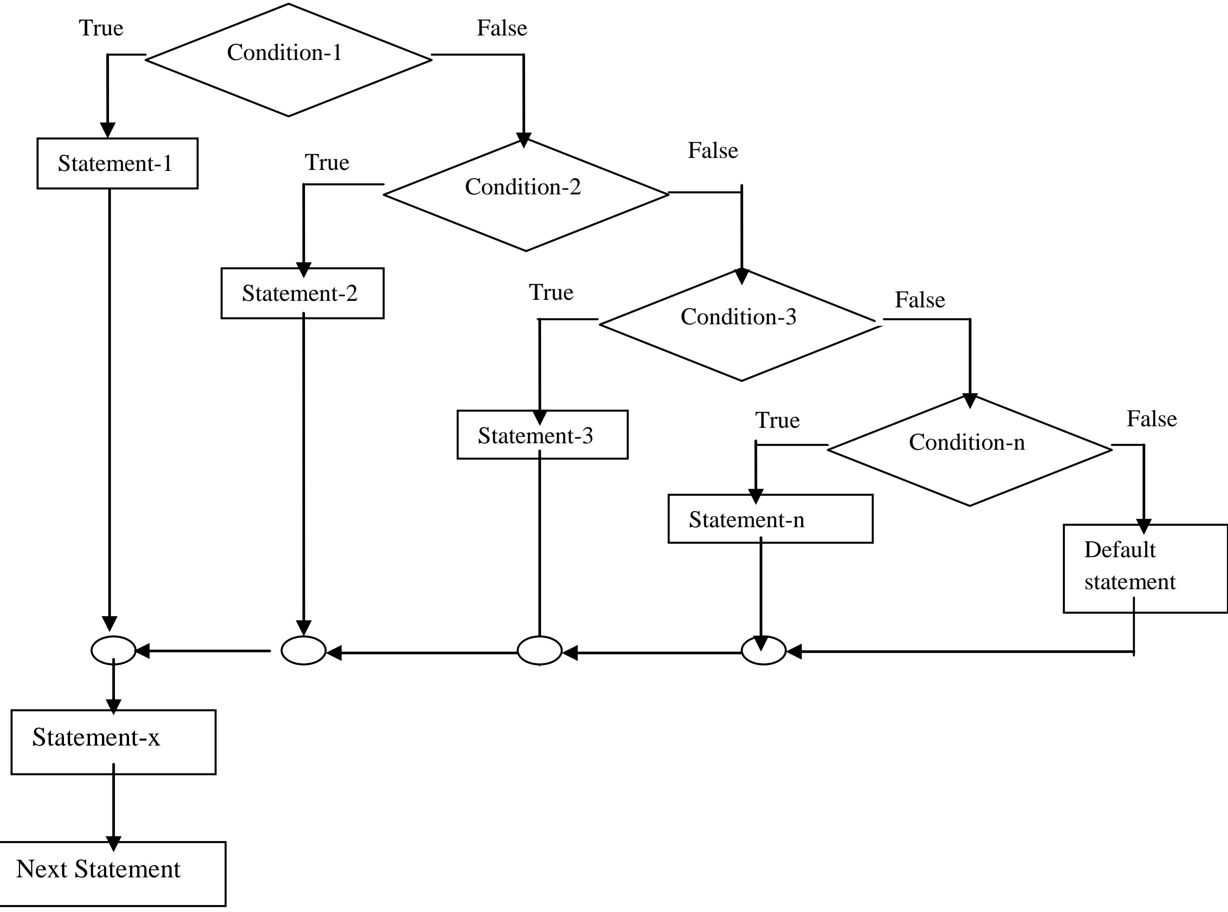


Fig 7.5 Flow chart for the else .. if ladder

7.3. Rules for indentation in the *if…. else* statements

- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation
- Align vertically else clause with their matching if clause
- Use braces on separate lines to identify a block of statements
- Indent the statements in the block by at least three spaces to the right of the braces
- Align the opening and closing braces
- Use appropriate comments to signify the beginning and end of block
- Indent the nested statements as per the above rules
- Code only one clause or statement on each line.

7.4. Switch Statement

If the number of the alternatives is to be chosen are few, then the if statement can be used, but when the number of alternatives increases, it becomes difficulty to us the if statement. C language has a built-in multi-way decision statement called **switch.** The switch statement test the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The Switch statement takes the following form;

```
switch (expression)
{
      case    value -1;
          block -1
          break;
      case value -2;
          block -2
         break;
         …….
        ……..
      default;
        default – block
       break;
}
statement = x
```

Switch Statement ***expression*** is an integer expression or character. ***Value-1, value -2*** are constants or constants expressions (evaluable to an integral constant) and are known as ***case labels***. Each of these values should be unique within a switch statement and the case labels end with semi-colon (;). ***Block-1, block-2 …*** are statement lists and may contain zero or more statements. There is no need to put braces around these blocks.

When the switch is executed, the value of the expression is successfully compared against the values value 1, value -2. If a case is found whose value matches with the value of the expression, then the block of statements that follow the case are executed. The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-4 following the switch. Default is an optional case, when present it is executed if the value of the expression does not match with any of the case values. If default is missing, no action takes place if all matches fail and the control goes to the statement- x

Program Example 7.5. Using the Switch statement in the grading for an Academic institution;-

```
      #include<stdio.h>
      #include<conio.h>
main()
{
index = marks/10;
switch (index)
{
            case 10:
            case 9:
            case8:
                        grade = 'Honours';
                        break;
            case 7:
            case 6:
                        grade =  'First Division';
                        break;
            case 5
                         grade =  'Second Division';
                        break;
            case 4
                        grade = 'Third Division';
                        break;
            default:
                         grade =  'Fail';
                        break;
}
printf ("%s\n", grade);
}
```

Index = marks / 10

Marks	index
100	10
90 - 99	9
80 - 89	8
70 - 79	7
60 – 69	6
50 - 59	5
40 - 49	4
30 - 39	3

20 - 29	2
10 - 19	1
0 - 9	0

Index 10, 9 and 8 executes the grade = "Honours"

Index 6 & 7 executes the grade = " First Division".

7.4.1. Rules for Switch

- The switch expression must be an integral type
- Case labels must be constants or constants expressions
- Case labels must be unique. No two label can have the same value
- Case labels must end with a semicolon
- The break statement transfers the control out of the switch statement
- The break statement is optional. That is two or more case labels may belong to the same statements.
- The default label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one default label.
- The default may be places anywhere but usually places at the end.
- It is permitted to nest switch statements

Chapter Review questions

1. Write a program to determine whether a given number is 'odd' or 'even' and print the message
   NUMBER IS EVEN
   Or
   NUMBER IS ODD
   (a) without using a else option (b) with an else option

2. What will be the output of the following segment when executed

```
int a = 10, b = 5;
if (a>b)
{  If (b>5)
     printf("%d", b);
}
else
   printf("%d", a);
}
```

CHAPTER EIGHT

C Programming:- Decision Making and Looping

Chapter Objectives

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

Describe and use appropriately the C decision making and Looping control structures

- ✓ The **while** statement
- ✓ The **do** statement
- ✓ The **for** statement
- ✓ Nesting for statements

Describe and use appropriately syntax to jumping in and jumping out of a Loop.

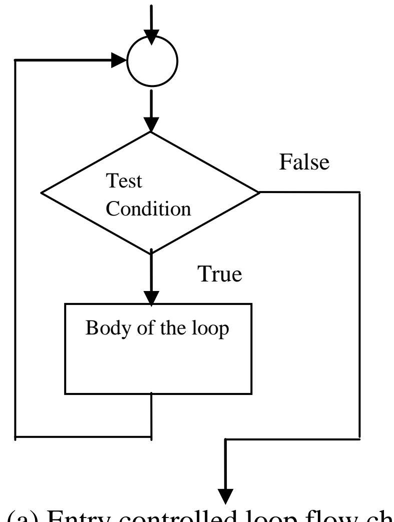
- Break statement

8.1 Introduction

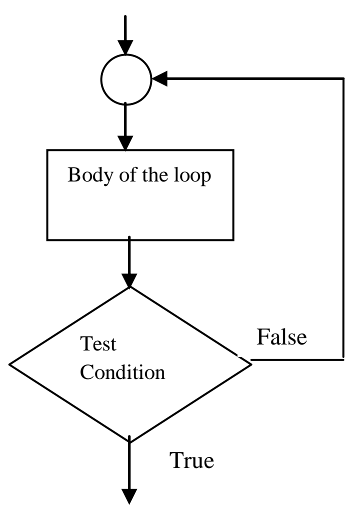
The looping capabilities in C enables us to develop concise programs containing repetitive processes without the use of GOTO statements. In looping the sequence of statements are executed until some conditions for termination of the loop is satisfied. A program loop consists of two segments, one known as the body of the loop and the other as the control statement. The control statement test certain conditions and then directs the repeated execution of the statement contained in the body of loop.

8.2. Classification of Loop Control Structures;

Depending on the position of the control statement in the loop a control structure may be classified as either a ***Entry-controlled loop(Pre-test)*** or an ***Exit-controlled loop (Post-test),*** Fig 8.1. In the entry controlled loop, the control conditions are tested before the start of the loop execution, while in the Exit-controlled loop the control conditions are tested before the end of the loop execution, thus the body of loop is executed unconditionally for the first time



(a) Entry controlled loop flow chart



(b) Exit controlled loop flow chart

Fig 8.1 Loop control structures

The test-condition should be carefully stated in order to perform the desired number of loop executions.
A looping process in general would include the following four steps;

- Setting an initialization of a condition variable.
- Execution of the statements in the loop.
- Test for a specified value of the condition variable for execution of the loop.
- Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.
The C language provides for three constructs for performing loop operations;

- The **while** statement
- The **do** statement
- The **for** statement

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loop may be classified into two general categories;

- **Counter-controlled loops**: when in advance the number of time the loop is to be executed is known. Control variable called Counter is used, which is initialized, tested and updated properly for the desired loop operations. The number of times the loop is to be executed may be a constant or a variable that is assigned a value. It is also called ***definite repetition loop***.
- **Sentinel**-controlled loops: a special value is used to change the loop control expression from true to false. The number of repetitions are not known before the loop begins executing and is thus called ***indefinite repetition loop***.

8.3. The While Statement

The simplest of all the looping structures in C. The while statement take the following form;

```
while (test condition)
{
        body of loop
}
```

The while is an entry-controlled loop statement. The test-condition is evaluated and if true the body of the loop is executed. After the execution the test-condition is evaluated again and if true the body of the loop is executed again. This process is repeated until the test-condition finally becomes false and the control is

transferred outside the loop where the program continues with the statements immediately after the body of the loop.

Program Example 8.1 Illustration of the while statement;

```
 #include<stdio.h>
 #include<conio.h>
main()
{       sum= 0;
        n = 1;
         while (n <= 10)
      {
       sum = sum + n*n;
         n = n+ 1;
      }
      printf ("Sum = %d\n", sum);
      getch();
}
```

Program example 8.2. Program to compute x to the power of n using while loop

```
 #include<stdio.h>
 #include<conio.h>
main()
{
int count, n;
float  x, y;
printf("Enter the values for x and b");
scanf("%f  %d", &x, &n);
y =1 ;
count = 1
while (count <= n)
{
y =  y*x;
count++;
}
printf("\nx = %f; n = %d; x to power n = %f\n", x,n,y);
getch();
}
```

8.4. The Do Statement

Use when the body of the loop needs to be executed first before evaluating the test-condition.

Format

```
   do
  {
     body of the loop
  }
   while (test-condition);
```

On reaching the do statement, the program proceeds to evaluate the body of the loop first, and at the end of the loop the test-condition in the while statement is evaluated. If the test-condition is true the programs evaluates the body of loop and these process is repeated until the test-condition is false. If the test-condition is false the loop is terminated and the control goes to the statement that appears immediately after the while statement.

Since the test-condition is evaluated at the bottom of the loop, the **do…..while** construct provides an exit–controlled loop and thus the body of the loop is always executed at least once.

Program example 8.3. Program to print the multiplication table 1x1 to 12x12 using Nested do .. while loops

```
#include<stdio.h>
#include<conio.h>
#define COLMAX 10
#define ROWMAX  12
main()
{
   int row, column, y;
   row = 1;
   printf("    MULTIPLICATION TABLE    \n");
   printf(" ---------------------------------------- \n");
do  /* Outer loop begins  */
 {
          column = 1;
          do              /* inner loop begins */
          {
               y = row * column;
               printf("%4d", y);
               column = column + 1;
```

```
        }
    while(column <= COLMAX);    /*Inner loop Ends*/
            printf("\n");
                row = row + 1;
    }
while (row <= ROWMAX);  /* Outer loop Ends*/
printf (" -------------------------------------------\n");
getch();
}
```

Out put

```
Multiplication table:

    1   2   3   4   5   6   7   8   9  10  11  12
---------------------
 1!   1   2   3   4   5   6   7   8   9  10  11  12
 2!   2   4   6   8  10  12  14  16  18  20  22  24
 3!   3   6   9  12  15  18  21  24  27  30  33  36
 4!   4   8  12  16  20  24  28  32  36  40  44  48
 5!   5  10  15  20  25  30  35  40  45  50  55  60
 6!   6  12  18  24  30  36  42  48  54  60  66  72
 7!   7  14  21  28  35  42  49  56  63  70  77  84
 8!   8  16  24  32  40  48  56  64  72  80  88  96
 9!   9  18  27  36  45  54  63  72  81  90  99 108
10!  10  20  30  40  50  60  70  80  90 100 110 120
```

The printf in the inner loop does not contain any new (\n) , thus allowing the printing of all row values in one line. The empty printf in the outer loop initiates a new line to print the next row.

8.5. The For Statement

The ***for*** loop is an entry-controlled loop that provides a more concise loop control structure. The general form for the ***for*** loop is

for (initialization ; test-condition; increment)

{

body of the loop

}

The execution of the for statement is as follows;

1. Initialization of the control variables is done first using assignment statements such as **i =1** and **count = 0;.** The i and count are called the loop control variables.
2. The value of the control variable is tested using the test-condition. The test-condition is a relational expression such as i<10 that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the ***for statement*** after evaluating the last statement in the loop. The control is incremented using an assignment statement such as i = i + 1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. Example

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

   main()
   {
       int x;
               for (x =0; x<=9; x= x+1)
   {
              printf("%d", x);
   }
     printf("\n");
   getch();
   }
```

In the above example, the for loop is executed 10 times and prints the digits 0 to 9.

Note the three sections in the **for ( )** statement must be separated by semicolons(;). There is no (;) after the increment section (x = x+1)

The for statement allows for negative increments, example

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

main()
{
     int x;
    for (x =9; x>=0; x= x-1)
{
  printf("%d", x);
}
 printf("\n");

getch();
}
```

Program example 8.4 Use of for loop

```
#include<stdio.h>
#include<conio.h>
#define ROW 10
#define COLOUMN 12
main()
{
    int row,coloumn,product[ROW][COLOUMN],i,j;
    printf("Multiplication table:\n\n");
    printf(" ");
    for(j=1;j<=COLOUMN;j++)
  {
            printf("%4d",j);
```

```
}
        printf("\n");
        printf("-----------------------\n");
        for(i=0;i<ROW; i++)
        {
            row=i+1;
          printf("%2d|",row);
          for(j=1;j<=COLOUMN;j++)
          {
            coloumn=j;
            product[i][j]=row*coloumn;
            printf("%4d",product[i][j]);
            }
    printf("\n");
    }
    getch();
    }
```

To initialize more than one variable in the for statement

p = 1;

for (n=0; n<17; ++n)

becomes

for (p=1, n+0; n<17; ++n)

8.6. Nesting of for loops

Nesting of the for loop ie one for statement within another for statement is allowed in C. Two loops can be nested as follows;

```
…
….

for (i =1; i<10; ++i)  ──────────────┐
{                                    │
 **********                          │
************                         │
  for (j =1; j !=5; ++j)  ────┐      │
   {                          │      │
       **************         │ Inner   │   Outer
      ***********             │ loop    │   Loop
     } ◄──────────────────────┘      │
      ***************                │
     ***************                 │
} ◄──────────────────────────────────┘
```

****
****

Program example 8.5. Program to print the Multiplication table using Nested for loop

```
#include<stdio.h>
#include<conio.h>
#define COLMAX 10
#define ROWMAX  12
main()
{
    int row, column, y;
                 row = 1;
                printf("     MULTIPLICATION TABLE    \n");
                 printf(" ----------------------------------------- \n");
  for (row=1; row<=ROWMAX; ++row)
{
   for (column = 1; column <= COLMAX; ++column)
   {
                 y = row * column;
                printf("%4d", y);
  }
printf("\n");
}
printf (" ---------------------------------------------\n");
getch();
}
```

8.7. Selecting a Loop

- Analyse the problem and see whether it requires a pre-test or post-test loop.
- If it requires a post-test loop, then we can use the **do …. While**
- If it requires a pre-test loop, then use either **for** loop or **while** loop.
- Decide whether the loop termination requires counter-based control or sentinel-based control.
- Use for loop if the counter-based control is necessary
- Use while loop if the sentinel-based control is required.
- Both the counter-controlled and sentinel-controlled loops can be implemented by all the three control structures.

8.8. Jumping out of a Loop

An early exit from a loop can be accomplished by using the **break statement** or the **goto statement**. When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When loops are nested, the break would only exit from the loop containing it. Break only exits one loop.

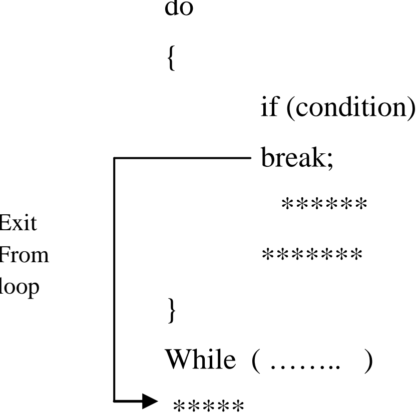


Fig 8.5. Exiting a loop with **break** statement

The Goto statement can transfer the control to any place on a program, and thus it is useful in proving branching in a loop. Use goto statement to exit from a deeply nested loops when an error occurs.

Example of using goto statement to jump out of a loop

```
while ( )
{
        if (error)
        goto stop;  ----------------+
        *********                   |
        if (condition)              |    Exit loop
        goto abc; ---+              |
        ************ | jump within  |
        **********   |   loop       |
         abc <-------+              |
        *********                   |
}                                   |
 stop; <----------------------------+
  *******
  *********
```

8.9 Structured programming

An approach to the design and development of programs. It is a discipline of making a program's logic easy to understand by using only the basic three control structures;

- Sequence (straight line) structures
- Selection (branching) structures
- Repetition (looping) structures.

The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured. Structured programming discourages the implementation of unconditional branching using jump statements such as goto, break and continue.

Chapter Review Questions

1. Write a for statement to print each of the following sequences of integer
   a) 1,2,3,4,8,16,32
   b) 1,3,9,27,81,243
   c) -4, -2, 0, 2, 4
2. Change the following for loops to while loops
   a. for (m=1; m<10; m= m+1)
      printf(m);
   b. for (; scanf("%d", &m) != -1)
      printf(m);

3. Write a program to compute the sum of the digits of a given integer.

CHAPTER NINE

C- Programming: Arrays

Chapter Objectives

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

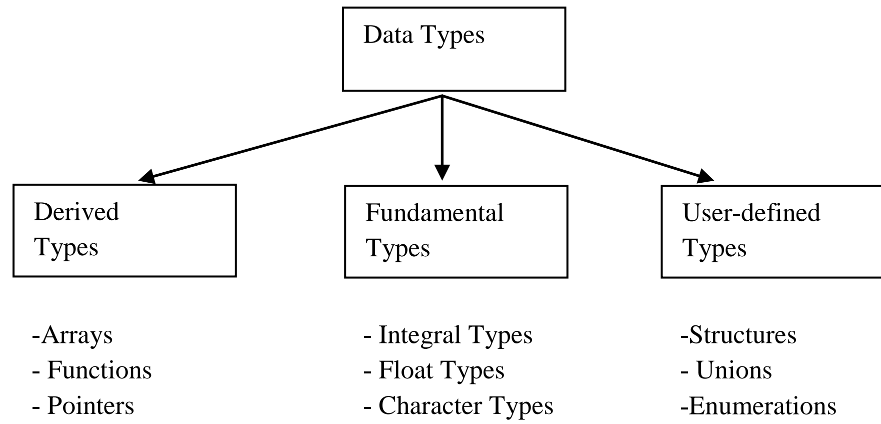
- Describe the C Program Data structures
- Declare and Initialize the C program Numeric Arrays
  - ✓ One-Dimensional
  - ✓ Two-Dimensional
  - ✓ Three Dimensional
- Declare and Initialize the C Program String Arrays
- Use String Arrays to manipulate Strings in C Program

9.1 Introduction

The data types so far encountered in C language, char, int, float, double are constrained by the fact that the variables of these types can store only one value at any given time. Thus they can only be used to handle limited amounts of data. C language supports a derived data type called **Array** that can be used in applications that need to handle large amounts of data.

An array is a fixed-size sequenced collection of elements of the same data type ie, a grouping of like-type data that share a common name. It can be used to represent a list of number or names. Since an array provides a convenient structure for representing data it is classified as one of the ***data structures*** in C.

9.2. Data structures in C



Arrays and structures are referred to as ***structured data types*** because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that

shows the relationships among he individual elements and facilitate efficient data manipulations. These structured data types are called ***data structures.*** Other data structures in C includes;

- Stacks
- Linked-lists
- Queues
- Trees

Since an array can represent a list of items, the individual values of the items are referred to as elements.

Example

Salary[10] represent the tenth element (10th employee salary in a list of organization employees salaries).

The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. We can use the arrays to represent not only a simple list of values but also tables of data in two, three or more dimensions, this giving rise to;

- One-dimensional arrays
- Two-dimensional arrays
- Three-dimensional arrays

9.3. One-Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array. Example

x[1]. x[2], x[3] ……. x[n]. x[0] is allowed

To represent 35, 40, 20, 57, 19) by a array variable number, the array variable number is declared as

int number[5];

the computer will reserve five storage spaces as;

Number[0]	35
Number[1]	40
Number[2]	20
Number[3]	57
Number[5]	19

These elements may be used in programs just like any other C variable. Example

a = number[0] + 10;

number[4] = number[2] + number[3];

9.4. Declaration of One-Dimensional Arrays.

Arrays must be declared like other variable before they are used so that the computer can allocate space for them in memory. The declaration takes the following form;

type ***variable-name[size]***

Type specifies the type of elements that will be contained in the array such as int, float, char etc. Size indicates the maximum number of elements that can be stored inside the array.

float height[50]; **d**eclares an array height to contain 50 real (floating point) numbers

Note: the size should either be a numeric or a symbolic constant.

C language treats character strings simply as arrays of characters. The size in the character string represents the maximum number of characters that the string can hold. Example the string "WELL DONE" is stored in a character 'name array' as follows;

Char name[10];

'W'
'E'
'L'
'L'
' '
'D'
'O'
'N'
'E'
'\0'

A compiler always terminates a character string with a ***null character '\0'***. Thus when declaring character arrays, we must allow one extra element space for the ***null terminator***.

9.5. Initialization of One-Dimensional Arrays.

After an array has been declared, its elements must be initialized, otherwise they will contain garbage. Arrays can be initialized at the following stages: at compile time and at run time

9.5.1. Compile time initialization.

Format: **type** ***array-name[size] = { list of value };***

The values in the list are separated by commas

Example int number[3] = {0,0,0};

- Float total[5] = {0.0, 15.75, -10} will initialize the first three elements to 0.0, 15.75, -10.0 and the remaining two elements to zero

If the size is omitted example

int counter[] = { 1, 1,1,1}; the compiler allocates enough space for all initialized elements.

int number[3] = {20, 10, 30, 40}; is not allowed in C

9.5.2. Run Time Initialization

This approach is usually applied for initializing large arrays.

```
-----------------
----------------
for ( i =1; i < 100; i = i+1)
{
  if  i < 50
    sum[i] = 0.0;
  else
   sum[i] = 1.0;
}
-----------
--------
```

Program Example 9.1. Program for frequency counting

```
#include<stdio.h>
#include<conio.h>
#define  MAXVAL  50
#define COUNTER 11

main()
{
float  value[MAXVAL];
int  i, low, high;
int group[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
printf("Enter a set of numbers Maximum 50 then press Enter\n");
                    for (i = 0; i < MAXVAL; i++)
{
        scanf("%f", &value[i]);
   ++ group[ (int ) (value[i]) / 10];
}
printf("\n");
printf(" GROUP    RANGE      FREQUENCY\n\n");
for ( i = 0; i < COUNTER; i++)
```

```
{
        low = i *10;
                if (i == 10)
                   high = 100;
                else
                  high = low + 9;
  printf(" %2d    %3d to %3d      %d\n", i+1,   low,   high,   group[i]);
}
getch();
}
```

Note int group[COUNTER] = {0,0,0,0,0,0,0,0,0,0}; can be replaced with int group[COUNTER] = {0};

9.6. Searching and Sorting using Arrays

Searching and sorting are the two most frequent operations performed on arrays. Computer scientists have devised several data structures and searching and sorting techniques that facilitate rapid access to data stored in lists. Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an ordered list. Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available. The three simple and most important among them are;

- Bubble sort
- Selection sort
- Insertion sort.

Other sorting techniques includes Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified elements in a list. The specified element is often called ***search key***. If the process of searching find a match of the search key with a list element value, the search is said to be successful otherwise it is unsuccessful. The two most commonly used search techniques are:

- Sequential search
- Binary search

Program Example 9.2. Program to sort Elements in an Array in Ascending order

```
#include<stdio.h>
#include<conio.h>
main()
{
  int arr[10],temp,i,j,n;
```

```
   printf("\nEnter any Value less Than 10 ");
    scanf("%d",&n);
   printf("\n\tEnter The Values into ARRAY ");
        for(i=0;i<n;i++)
        {
                printf("\n\n Enter Element no %d: ",i+1);
                scanf("%d",&arr[i]);
        }
                for(i=0;i<n-1;i++)
                {
                for(j=0;j<n-i;j++)
                {
                        if(arr[j] >arr[j+1])
                        {
                        temp=arr[j];
                        arr[j]=arr[j+1];
                        arr[j+1]=temp;
                        }
                }
                }
printf("\n\n-- Sorted Series --");
        for(i=0;i<n;i++)
        {
        printf("\n \n \t %d",arr[i]);
        }
        getch();
}
```

Test data 32, 43, 23, 57, 47, 34

9.7. Two – Dimensional Arrays

Two dimensional arrays can be used to store a table of values example

	Item1	Item2	Item3
Sales girl 1	310	275	365
Sales Girl 2	210	190	325
Sales Girl 3	405	235	240
Sales Girl 4	260	300	380

The table can be thought of as a Matrix containing 4 rows and 3columns. In mathematics we represents a matrix value by using two subscripts such as V_{ij} V denotes the entire matrix and V_{ij} refers to the value in the row (i) and column (j). example $V_{31} = 405$

Two dimensional arrays are declared as follows;

- **Type array_name[row_ size][column_size];**

Each dimension of the array is indexed from zero to its maximum size minus 1; the first index selects the row and the second index selects the column within that row.

Program example 9.3. Program to show; Total sales by each girl; Total value of each item sold and Grand total of sales of all items by all girls

```
#include<stdio.h>
#include<conio.h>
#define MAXGIRLS 4
#define MAXITEMS 3
main()
{
    int value[MAXGIRLS][MAXITEMS];
    int girl_total[MAXGIRLS], item_total[MAXITEMS];
    int i, j, grand_total;
    printf ("Input data\n");
    printf("Enter values, one at a time row-wise\n\n");
     for (i = 0 ;  i < MAXGIRLS ; i++)
        {
         girl_total[i] = 0;
           for (j = 0 ; j < MAXITEMS ; j++)
              {
                scanf("%d", &value[i][j]);
                  girl_total[i] = girl_total[i] + value[i][j];
               }
         }
         /* Computing the Items totals */
                for (j = 0 ; j < MAXITEMS ; j++)
                {
                  item_total[j] = 0;
                  for (i =0 ; i < MAXGIRLS ; i++)
                   item_total[j] = item_total[j] + value[i][j];
```

```
        }
    /* computing the Grand totals */
        grand_total = 0;
         for (i = 0 ; i < MAXGIRLS ; i++)
         grand_total = grand_total + girl_total[i];
    /* printing results */
     printf("\n GIRLS  TOTALS \n\n");
            for (i = 0 ; i < MAXGIRLS ; i++);
             printf("Salesgirl[%d] = %d\n", i+1, girl_total[i]);
             printf("\n ITEM TOTALs\n\n");
                  for (j = 0 ; j < MAXITEMS ; j++)
                   printf("Item[%d] = %d\n", j+1, item_total[j]);
               printf("\nGrand Total = %d\n", grand_total);
    getch();
    }
```

Program Example 9.4. /*Multiplication table using two-dimensional array: */

```
#include<stdio.h>
#include<conio.h>
#define ROW 5
#define COLOUMN 5
main()
{
        int row,coloumn,product[ROW][COLOUMN],i,j;
        printf("Multiplication table:\n\n");
        printf(" ");
        for(j=1;j<=COLOUMN;j++)
        {
                printf("%4d",j);
        }
        printf("\n");
        printf("-----------------------\n");
        for(i=0;i<ROW; i++)
        {
                row=i+1;
                printf("%2d|",row);
                for(j=1;j<=COLOUMN;j++)
        {
```

```
            coloumn=j;
            product[i][j]=row*coloumn;
            printf("%4d",product[i][j]);
            }
      printf("\n");
}
getch();
}
```

9.7. Initializing Two-Dimension Arrays

Like the one-dimension arrays, two-dimension arrays may be initialized by following their declaration with a list of initial values enclosed in braces. Example

int table[2][3] = { 0,0,0,1,1,1);

initializes the elements of the first row with zeros and the second row elements with one.

Or

Int table[2][3] = {{0,0,0}, {1,1,1}};

Or **Int table[2][3] = {**

```
                        {0,0,0},
                        {1,1,1}
                  }
```

9.8. Character Arrays and Strings

A string is a sequence of characters that is treated as a single data item.. Character strings are often used to build meaningful and readable programs. The common operations performed on character string includes;

- Reading and writing strings;
- Combining strings together
- Copying one string to another
- Comparing strings for equality.
- Extracting a portion of a string.

9.9. Declaring and Initializing String Valuables.

C does not support strings as a data type. C allows representing strings a character arrays. Format

char string_name [*size]*;

The size determines the number of characters in the string_name. example

char city[10] or char name[30]

The C compiler automatically supplies a null character (‘\0’) at the end of the string when it assigns a character string to a character array. Thus the array size should be equal to the required name size plus 1.

Initializing a string array;

char city[9] = “NEW YORK”; OR **char city[9] = {‘N’, ‘E’, ‘W’, ‘ ‘, ‘Y’, ‘O’, ‘R’, ‘K’, ‘\O’};**

C also permits initialization of a character string array without specifying the number of elements. In such a case, the size of the array is determined automatically based on the number of elements initialized. Example

char string[] = {‘G’, ‘O’, ‘O’, ‘D’, ‘\0’}; defines the array string as a five element string.

One can also declare a size much larger than the string size during initialization stage , example

char[10] = “GOOD”;

The compiler initializes an array of 10 elements and places the word “GOOD” into it terminating with null character.

G	O	O	D	\0	\0	\0	\0	\0	\0

Illegal formats;

Declaration such as;

char string2 [3] = “GOOD”.

char s1[4] = “abc”;

char s2[4];

s2 = s1; will give an error as an array name cannot be used as the left operand of an assignment operator.

9.10. Terminating with Null Character.

- A string is not a data type in C but is considered a data structure stored in an array.
- The string is a variable-length structure and is stored in a fixed-length array.
- The array size is not always the size of the string and most often it is much larger than the string stored in it. Thus the last element of the array need not be represent the end of the string.
- The Null character serves as the “end-of-string” marker.

9.11. Reading Strings from Terminals

Using Scanf() function.

Format

char address[10];

scanf(“%s”, address);

Scanf function terminates its input on the first white space (blanks, tabs, carriage returns, form feed and new line) it finds.

Example entering the name “NEW YORK” will only result with the “NEW” being read into the array, since the blank space between the NEW and YORK words will cause the scanf to terminate.

The & symbol is not required before the variable name.

To read the word "NEW YORK" then we need to use two character arrays example

```
Char adr1[5], adr2[5];
scanf ("%s  %s", adr1, adr2);
```

" NEW" will assigned to adr1 while "YORK" will be assigned to adr2

One can also specify the field width in the scanf() function,

```
scanf("%ws", variable_name);
```

9.12. Reading a Line of Test

C supports a format specification known as the edit set conversion code %[ ], that can be used to read a line containing a variety of characters, including white spaces

```
char  line[80;
scanf("%[^\n]", line);
printf("%s", line);
```

will read a line of text from the keyboard and print it out as entered.

9.12.1. Using *getchar* and *gets* Function instead of Scanf() function

The getchar and gets functions methods used in C to read a string of text containing whitespaces. Gets function is available from <stdio.h> header file

The getchar function takes the following form;

```
char ch;
ch = getchar( );
```

The gets function takes the following form;

gets(str); where ***str*** is a string variable declared properly. Example

```
char line[80];
gets(line);
printf("%s", line);
```

Program Example 9.5 A program to read a line of text from the Keyboard.

```
#include<stdio.h>
#include<conio.h>
main()
{
char line[81], character;
int c;
```

```
c = 0;
        printf("Enter text. Press <Return> at end\n");
        do
        {
                character = getchar();
                line[c] = character;
                c++;
        }
        while (character != '\n');
        c = c-1;
        line[c] = '\0';
        printf("\n%s\n", line);
        getch();
}
```

9.13. Writing Strings to Screen.

The function in C use to write string to the screen is the ***printf*** function. The printf function takes the following form

printf("%s", variable_name);

Program Example 9.6: Writing strings using %s format

```
/* documentation  section*/
#include<string.h>
#include<stdio.h>
#include<conio.h>
main()
{
char country[15] = "United Kingdom";
printf("\n\n");
printf("-------------------------\n");
printf("%15s\n", country);
printf("%5s\n", country);
printf("%15.6\ns", country);
printf("%-15s\n", country);
printf("%15.0\ns", country);
printf("%.3s\n", country);
printf("%s\n", country);
getch();
}
```

9.14. String Handling Functions

The string functions are in the <string.h> header file

Function	Action
Strcat()	Concatenates two string
Strcmp()	Compares two strings
strcpy	Copies one string over another
Strlen()	Finds the length of a string
Strupr()	Converts to upper case
Strlwr()	Converts to lower case

9.14.1. Strupr() function

Used to convert string into upper case.

Program Example 9.7. Program to Convert String to Upper case

```
/* documentation  section*/
#include<string.h>
#include<stdio.h>
#include<conio.h>
main()
{
char mystring[10];
printf("Enter your string ");
scanf("%s", &mystring);
printf("string to upper case is %s", strupr(mystring));
getch();
}
```

9.14.2. Strcat() function

Function used to join two strings together. It takes the form

strcat(string1, string2);

String1 and string2 are character arrays. When the function ***strcat()*** is executed, string2 is appended to string1. It does so by removing null characters at the end of string1and placing string2 from there.

The size of the string1 should be large enough to accommodate the combined string (string1 + string2);

9.14.3. Strcmp() Function

The function compares two strings identified by the arguments and has a value of 0 if they are equal. If they are not, it has the numeric difference between the first non matching characters in the string. The function takes the following form;

strcmp(string1, string2);

String1 and string2 may be string variable or string constants example

strcmp(name1, name2);   strcmp(name1, "John");   strcmp("Rom", "Ram");

strcmp("there", "their");   returns the value -9 which is the numeric difference between ASCII "i" and ASCII "r" (i – r) = -9.

9.14.4. Strcpy() Function

The function Copies / assigns the contents of string2 to string1. String2 may be a character array variable or a string constant. It takes the following g form

strcpy(string1, string2);

Example

Strcpy(city, "NAIROBI"); will assign the string "NAIROBI" to string variable city.

9.14.5 Strlen() Function

The function counts and returns the number of characters in a string. It takes the following form

n = strlen(string);

n = integer variable which receives the value of the length of the string.

Program Example 9.8: Program example using the string handling functions

```
/* documentation  section*/
#include<string.h>
#include<stdio.h>
#include<conio.h>
main()
{
 char s1[20], s2[20], s3[20];
  int x, k1, k2, k3;
printf("\n\n Enter two string constants \n");
printf("?");
scanf("%s  %s", s1, s2);

x = strcmp(s1, s2);                          /* compares s1 and  s2 */
            if (x !=0)
             {
               printf ("\n\n Strings not Equal  \n");
              strcat(s1, s2);                          /* joins s1 and  s2 strings */
            }
```

```
        else
         printf("\n\n Strings are equal \n");
        strcpy(s3, s1);                    /* copies s1 into s3 */

        k1 = strlen(s1);                   /* gets the length of s1 */
        k2 = strlen(s2);                   /* gets the length of s2 */
        k3 = strlen(s3);                   /* gets the length of s3 */
printf("\n s1 = %s\t length = %d characters \n", s1, k1);
printf("\n s2 = %s\t length = %d characters \n", s2, k2);
printf("\n s3 = %s\t length = %d characters \n", s3, k3);

getch();
}
```

Test data
London / London
Nairobi / City

Chapter Review Questions

1. Discuss how initial values can be assigned to a multidimensional array.
2. What is a data structure? and why is an array considered a data structure?
3. Write a program to read two matrices A and B and print the following
   a) A + B
   b) A – B
4. What is the error in the following program

```
main()
{
 int x;
float y[ ]
…….
}
```

5. Marks for a Programming Methodology Exam for a class of 50 students are as follows;

43	65	51	27	79	11	56	61	82	09	25	36	07	49	55
63	74	81	49	37	40	49	16	75	87	91	33	24	58	78
65	56	76	67	45	44	36	63	12	21	73	49	51	19	39
49	68	93	85	59										

a). Write a program to count the number of students belonging to each of the following groups of marks

0 -9, 10 – 19, 20 – 29, 30 – 39 …… 90-99, 100

CHAPTER TEN

C- Programming:- User Defined Functions

Chapter Objectives

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

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

- Describe and design a function
- Describe function integration and integrate a function into a program
- Describe how functions communicate

10.1. Introduction

C functions can be classified into two categories; ***library functions*** and ***user-defined functions***. Library functions are not required to be written by the programmers, while user-defined functions have to be developed by the user at the time of writing the C program. ***main function*** is an example of user-defined functions while ***scanf*** and ***printf functions*** are examples of library functions. Some user-defined functions can eventually become library functions.

10.2. Need for user-defined functions

Large programs are divided in functional parts (figure 10.1), and each of the functional part is code separately and independently but later combined into a single unit. The independently coded programs are called sub-programs, and in C they are referred to as **"functions"**. The functions can be called and used whenever needed. The division of large programs into sub-programs have the following advantages;

1. It facilitates top-down modular programming. The high level logic of the overall problem is solved first while the details of each lower-level functions are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on whatever others have already done, instead of starting all over again from scratch.

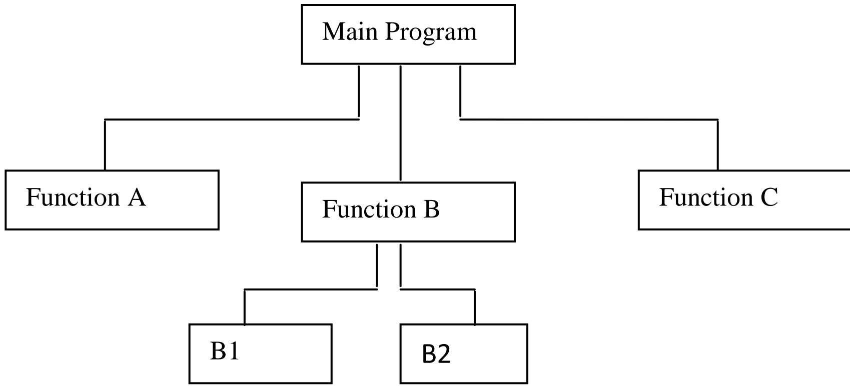


Figure 10.1 Top-down modular programming using functions.

10.3. Modular programming;

A strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called modules that are separately named and individually callable program units. These modules are carefully integrated to become a software system that satisfies the system requirements. It is basically a "divide – and –conquer" approach to problem solving. Modules are identified and designed such that they can be organized into a to-down hierarchical structure (similar to an organization chart).

Characteristics of modular programming;

1. Each module should do only one thing
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module
4. No communication can take place directly between modules that do not have calling – called relationship.
5. All modules are designed as single-entry-exit systems using control structures.

10.4. A Multi-Function Program.

A function is a self-contained block of code that performs a particular task. A C program is designed using a collection of functions.

Program Example 10.1: Program to illustrate the use of Functions

```
/* documentation  section*/
#include<string.h>
#include<stdio.h>
#include<conio.h>
Void printline(void);  /* declares a function*/
main()
{
printline();        /* first call to the printline function */
```

```
    printf("This illustrated the use of C functions");
printline();   /* second call to the printline function */
}
void printline(void)    /* printline function  */
{
int i;
for(i =1; i<40, i++)
printf("-");
printf("\n");
}
```

Out put

This Illustrates the use of C functions

The program contain two user-defined functions ***main()*** function and ***printline()*** function. In the above program the main function call the user-defined function printline twice and the library function printf() once. A called function can also call another function example printline function in the above example calls printf() function 39 times to draw a line.

10.5. Elements of User-Defined Functions

Functions in C are classified as one of the derived data types in C. Functions can be defined and used like any other variable in C programs.

Similarities between functions and variables

1. Both function name and variable names are considered identifiers and therefore they must adhere to the rules of identifiers.
2. Like variables, functions have types (such as int) associated with them.
3. Like variables, function names and their types must be declared and defined before they are used in a program.

A user-defined functions consists of thee elements;

1. **Function Definition:** an independent program module that is specially written to implement the requirements of the function.
2. **Function Call:** invoking a function at an appropriate place in the program. The function/ module that call a function is referred to as the ***calling function*** or ***calling program***
3. **Function declaration:** declaration of the function within the calling function that will be used later. Also referred to as ***function prototype.***

10.6. Definition of Functions

A function definition also known as function implementation has the following elements;

1. Function name
2. Function type
3. List of parameters.
4. Local variable declarations
5. Function statements; and
6. A return statement.

All the six elements are grouped together into two parts, namely; function header and function body. The general format for a function definition is as follows;

```
function_type  function_name(parameter list)        /* function header */
{                                                   /*  function body */
        local variable declaration;
        executable statement 1'
        executable statement 2;
        ------
        -------
         return statement;
}
```

10.6.1 Function header

The functional header consists of the first three (3) elements; function type (also called return type); function name and the formal parameter list. No semi colon is used at the end if the function header section.

Name and Type

Function type specifies the type of value (int, float, void) that the function is expected to return to the calling function. If not stated C assumes int (integer) data type. If the function is not returning anything we should specify return type as void.

Example   printline(void);   or printline();

Function Name is any valid C identifier and thus must follow the same rules of formulation as other variable names in C.

Formal Parameter List

Parameter list declares the variables that will receive data sent by the calling program. They serve as input data to the function to carry out the specified task. They are called **Formal parameter** list since they

represent actual input values. The parameters can also be used to send data to the calling program where they are referred to as **Arguments.**

10.6.2. Function Body

Contain the declaration and statements necessary for performing the required task. It is enclosed on braces { } and contain three parts;

1. Local declarations that specify the variables needed by the function
2. Function statements that perform the task of the function
3. Return statement that returns the value evaluated by the function.

Examples of functions definition;

a). float mul (float x, float y)

```
{
        float results;          /* declare local variable*/
        results = x * y;        /*computes  the products*/
        return (results);       /*returns the results*/
}
```

b). void sum(int a, intb)

```
{
  printf("Sum = %s", a+b);      /* no local variables */
  return;                       /* Optional */
}
```

<u>**Note;**</u>

1. If a function is not returning any value one can omit the return statement or use the return(void) statement.
2. When a program reaches the return statement, the control is transferred back to the calling program. In the absence of the return statement the closing braces } acts as avoid return.
3. A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.

10.7. Return values and their types

While it is possible to pass to the called function any number of values, the called function can only return ***one value*** per call at the most. The return statement can take either of the following forms;

return; or return(expression);

- return; does not return any value and only acts as the closing brackets for the function,
- return(expression); returns the value and once reached the program control is immediately returns to the calling function

A function may have more than one return statements, especially when the value returned is based on certain conditions. Example

```
 if (x <= 0)
         return(0);
else
         return(1);
```

All functions by default returns **int** type of data. We can force a function to return another type of data type by using the type specifier in the function header. When a value is returned, it is automatically cast to the function's type

10.8. Function call

A function may be called by simply using the name followed by a list of actual parameters (or arguments) if any enclosed in parenthesis. Example

```
main()
{
int y;
y = mul(10,5)     /* Function Call */
printf("%d\n", y);
}
         int mul (int x, int y)   /* Mul function */
         {
                  int p;
                  p = x*y;
                  return (p);
            }
```

Function Call

A function call is a postfix expression. The operator (..) is at a very high level of precedence. Therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.

In a function call, the function name is the operand and the parenthesis set (..) which contains the actual parameters is the operator. The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

Note

1. If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
2. On the other hand, if the actual are less than the formals, the unmatched formal arguments will be initialized to some garbage.
3. Any mismatch in data types may also result in some garbage values.

Program Example 10.2. : Program Using Function

```
/* 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);
getch();
}
                              /* mul()      Sub-program  */
    int mul(int x,int y)
      {
        int p;
        p = x*y;
        return (p);
     }
```

10.9. Function Declaration

Like variables, all functions in a C Program must be declared, before they are invoked. A function declaration consists of four parts; Function type (return type), Function name, Parameter list and Terminating semicolon. The function declaration statement takes the following form

function_type function-name(parameter list);

Note

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order
4. Use of parameter names in the declaration is optional
5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns **int** type data.
7. The retype must be **void** if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

When a function does not take any parameters and does not return any value, its prototype is written as;

- ***void display(void);***

A prototype declaration may be placed in two places in a program.

1. Above all the functions (including main): referred to as ***Global prototype*** and is available to all functions in the program.
2. Inside a function definition: referred to as ***Local prototype*** and is available only to the local function.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the scope of the function. Declare the prototypes in the global variables declaration section of a C program (above the main () function) to add flexibility, provide an excellent quick reference to the functions use in the program, and enhance documentation.

Parameters

Parameters (also known as arguments) are used in three places.

1. In declaration (prototypes)
2. In function call
3. In function definition

The parameters use in prototypes and function definitions are called formal parameters and those used in calling statements may be simple constants, variables or expressions.
The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need match.

10.10. Categories of Functions

A function depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories

Category 1: Functions with no arguments and no return values

Category 2: Functions with arguments and no return values.

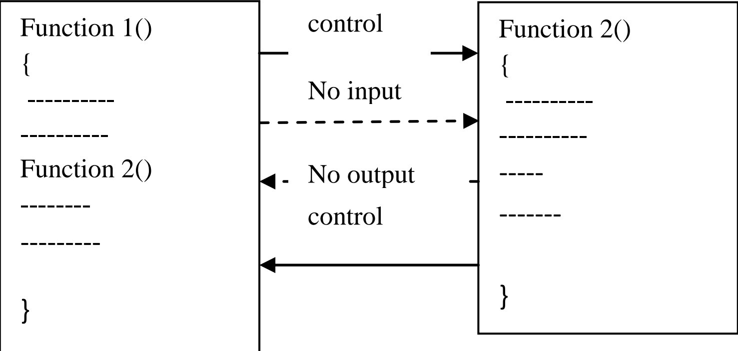
Category 3: Functions with arguments and one return value.

Category 4: Functions with no arguments but return a value.

Category 5: Functions that return multiple values.

Category 1: Functions with no arguments and no return values

When a function has no arguments, it does not receive any data from the calling function. When it does not return an value the calling function does not receive any data from the called function. Thus there is no data transfer between the calling and called functions, there is only transfer of control.



No data communication between functions

Program Example 10.3. Program to show Functions with no arguments and no return values.

```
/* Program to calculate the value of  principal in a bank after a certain period of time */
#include <stdio.h>
#include<conio.h>
                    /* functions declaration */
void printline (void);
void value(void);
main()
{
printline();
value();
printline();
}
                    /* function  print line */
void printline(void)     /* contains no arguments */
{
 int i;
for (i =1; i<=35; i++)
printf("%c", '-');
printf("\n");
}
                    /* function 2: value()    */
void value (void)   /* Contains no arguments */
{
int year, period;
float inrate, sum, principal;
printf("Principal Amount?");
```

```
scanf("%f", &principal);
printf("Interest rate?      ");
scanf("%f", &inrate);
printf("Period?     ");
scanf("%d", &period);

sum = principal;
year = 1;
while(year <= period)
{
sum = sum *(1+inrate);
year = year + 1;
}
printf ("\n%8.2f  %5.2f  %5d  %12.2f\n", principal, inrate, period, sum);
getch();

}
```

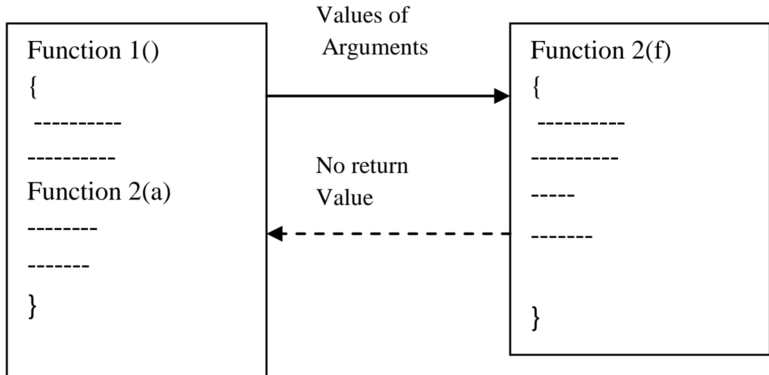
Test data

Principal = 50000
Interest = 0.12
Period = 5

Program Out put

```
----------------------------------------------------
Principal amount        5000
Interest rate?          0.12
Period                              5
5000.00                 0.12                5         8811.71
--------------------------------------------------------------------------------
```

Category 2: Arguments but no Return Values



One-way data communication

The calling function passes on a list or arguments to the called function. In the last example, the following changes will be done at the definitions of the functions;

void (printline(char ch)

void value (float p, float r, int n)

ch, p,r, &bn are called formal arguments. The calling function can thus send values to the arguments using the function call value (5000, 0.12, 5). The values 5000, 0.12, & 5 are called actual arguments

The formal and actual arguments should match in number, type and order

Program Example 10.4. Program to demonstrate functions with arguments and no return Value.

```
/* Program to calculate the value of  principal in a bank after a certain period of time */
#include <stdio.h>
#include<conio.h>

void printline (char c);
void value(float, float, int);
main()
{
float principal, inrate;
int period;
printf("Enter Principal amount:  ");
scanf("%f", &principal);
printf("Enter Interest rate?  ");
scanf("%f", &inrate);
printf("Enter Period?      ");
scanf("%d", &period);

printline('-');
value(principal, inrate, period);
printline('-');
}
void printline(char ch)
{
 int i;
for (i =1; i<=52; i++)
printf("%c", ch);
printf("\n");
}
```

```
void value (float p, float r, int n)     /* value function */
{
int year;
float sum;

sum = p;
year = 1;
while(year <= n)
{
sum = sum *(1+r);
year = year + 1;
}
printf ("\%8.2f\t  %5.2f\t  %d\t  %8.2f\n", p, r, n, sum);
getch();
}
```

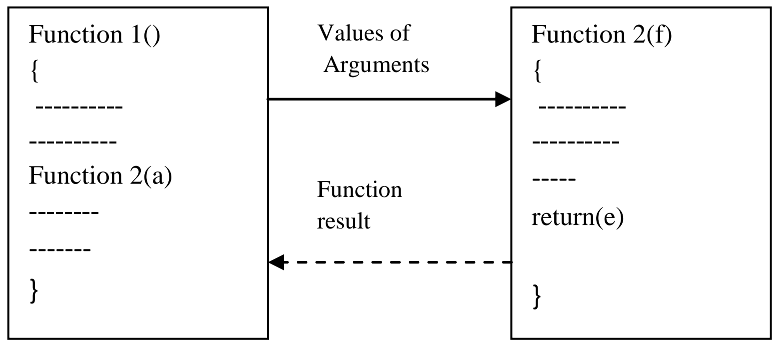
Test data
Principal = 50000
Interest = 0.12
Period = 5

Program Out put

```
----------------------------------------------------
Principal amount       5000
Interest rate?         0.12
Period                          5

5000.00                0.12             5          8811.71
---------------------------------------------------------------------------------
```

Category 3: Arguments with Return Values



Two- way data communication between functions

Program Example 10.5. Program to demonstrate functions with Arguments and Return Values.

```
/* Program to calculate the value of  principal in a bank after a certain period of time */
#include <stdio.h>
#include<conio.h>

int value(float, float, int);
void printline (char c, int len);

main()
{
 float principal, inrate;
 int period, amount;
 printf("Enter Principal amount:  ");
 scanf("%f", &principal);
 printf("Enter Interest rate?  ");
 scanf("%f", &inrate);
 printf("Enter Period?      ");
 scanf("%d", &period);

 printline('*', 52);
 amount = value(principal, inrate, period);
 printf ("\%8.2f\t  %5.2f\t  %d\t %d\n\n", principal, inrate, period,amount);

 printline('=', 52);
 getch();
}
void printline(char ch, int len)
{
 int i;
 for (i =1; i<=len; i++)
 printf("%c", ch);
 printf("\n");
}

int value (float p, float r, int n)     /* value function  */
{
int year;
float sum;
```

```
sum = p;
year = 1;
while(year <= n)
{
sum = sum *(1+r);
year = year + 1;
}
return(sum);
}
```

Test data
Principal = 50000
Interest = 0.12
Period = 5

Out put

```
----------------------------------------------------
Principal amount      5000
Interest rate?        0.12
Period                        5

5000.00               0.12          5        8811.71
--------------------------------------------------------------------------------
```

10.11. Nesting of Functions

C permits nesting of functions freely, ie. main() can call function1(), which calls function2() which calls function3()

Program Example 10.6: Program to demonstrate Nesting of Functions

```
/* Program to demonstrate Nesting of Functions  calculation of ratios*/
#include <stdio.h>
#include<conio.h>

float ratio (int x, int y, int z);
int difference (int x, int y);

main()
{
        int  a, b, c;
        printf("Enter values for a, b,& c\n");
        scanf("%d  %d %d", &a, &b, &c);
        printf("%f\n", ratio(a,b,c));
        getch();
```

```
}

        float ratio (int x, int y, int z)
        {
                if(difference(y,z))  /* difference function checks if    b-c = 0 */
                return (x/(y-z));
        else
                return(0.0);
        }
        int difference (int p, int q)
        {
          if (p != q)
                return (1);
        else
                return(0);
        }
```

- The program calculate the ratio **a/(b-c)**
- It has the following functions: **main(), ratio() and difference().**
- main() reads the values a, b, c and calls the function ratio to calculate the value a/(b-c). the ration function cannot be evaluated if (b-c) = 0, thus the ration() function calls another function difference to test whether the difference (b-c) is zero or not; difference returns 1 if b is not equal to c; otherwise returns zero to the function ratio.
- ratio function calculates the value a/(b-c) if it received 1 and returns the result in float, or sends zero if (b-c) = zero

10.12. Recursion

When a called function in turn calls another function a process of "chaining" occurs. Recursion is a special case of this process, where a function calls itself. The program runs endlessly until terminated by force. Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem.

Program Example 10.7: Program to demonstrate Recursion in functions

```
main()
{
printf("This is an example of recursion\n");
        main();
        getch();
```

}

10.13. Passing Arrays to functions

One-Dimensional Arrays.

It is possible to pass the values of an array to a function. To pass a one-dimensional array to a called function, it is sufficient to list the name of the array without any subscripts and the size of the array as arguments. Example

largest(a,n),

will pass an array a to the called function. The called function should be appropriately defined to receive the array,

Program Example 10.8. Program to find the largest value in an array of elements

```
#include <stdio.h>
#include<conio.h>
main()
{
float largest(float a[ ], int n);
float value[4] = {2.5, -4.75, 1.2, 4.67};
printf("%f\n", largest(value,4));
}
float largest (float a[], int n);
{
int i;
float max;
max = a[0];
for (i = 1; i< n; i++)
if(max < a[1])
max = a[1];
return (max);
getch();
}
```

Rules to Pass and Array to a Function

- The function must be called by passing only the name of the array.
- In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
- The function prototype must show that the argument is an array.

Program Example 10.9: Program that uses a function to sort an array of integers

```
#include <stdio.h>
#include<conio.h>
void sort(int m, int x[]);
main()
{
        int i;
       int marks[5] = {40, 90, 73, 81, 35};
        printf("Marks before sorting \n");
         for(i = 0; i <5; i++)
          {
                printf("%4d", marks[i]);
                  printf("\n");

               sort(5, marks);
               printf("Marks after sorting \n");
               for ( i= 0; i <5; i++)
                printf("%4d", marks[i]);
               printf("\n");
         }
getch();
}
void sort(int m, int x[])
{
  int i, j, t;
  for (i = 0; i <= m-1; i++)
        for (j =1; j <= m-1;j++)
        if(x[j-1] >= x[j])
       {
         t = x[j-1];
   x[j-1] = x[j];
   x[j] = t;
}
}
```

10.14. Scope, Visibility and Lifetime of Variables

In C all variables not only have a ***data type***, but they also have a ***storage class***. The variables can be broadly categorized depending on the place of their declaration, as internal (local) and External (Global). Internal variables are declared within a function, while external are declared outside of any function

The following variable storage classes are most relevant to functions;

1. Automatic Variables: declared inside a function in which they are utilized, they are created when the function is called and destroyed automatically when the function is exited. They are private/ local/ internal to the function in which they are declared in. A variable declared inside a function without storage class specification is by default an automatic variable. One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other functions in the program.

```
 main()
{
  auto int number;
  ----
   -----
}
```

2. External Variables .Variables that are **both alive and active** throughout the program. They can be accessed by any function in the program and are declared outside a function. Example

```
 int count;
main()
{
   count = 10;
   ………….
   ………….
}
function(1)
{
  int count = 0;
 …………
………
count = count + 1;
}

function2()
{
  int count;
………..
count = count + 2;
}
```

Note the variable count is available for use to all the functions (main(), function1() & function2())

3. Static Variables: Variables whose value persists until the end of the program. It can either be internal or external and is initialized only once when the program is compiled.
4. Register Variables: Variables kept in a machine register instead of in memory.

- The **scope** of a variable determines over what region of the program a variable is actually available for use ('Active')
- **Longevity** refers to the period during which a variable retains a given value during execution of a program ('alive'). Longevity has a direct effect on the utility of a given variable
- **Visibility** refers to the accessibility of a variable from the memory.

Scope Rules

1. The scope of global variable is the entire program life
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an auto variable declared in **main()** is the entire program execution time, although its scope is only the **main** function
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A static local variable, although its scope is limited to its functions, its lifetime extends till the end of program execution.
7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is re-declared within its scope again, it loses its visibility in the scope of the re-declared variable.

Chapter Review Questions

1. Describe any two ways of passing parameters to functions
2. Distinguish between the following;
   a) Scope and visibility
   b) Global and local
   c) Actual and formal arguments
3. Write a program that invokes a function called find() to perform the following tasks;
   a) Receive a character array and a single character
   b) Return 1 if the specified character is found in the, 0 otherwise.
4. Write a function that receives a floating point value x and returns it as a value rounded to the nearest decimal places. Example 123.4567 to be 123.46

CHAPTER ELEVEN

C- Programming: Structures and Pointers

Chapter Objectives

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

- Define a structure and declare structure variables
- Initialize a structure
- Use arrays of structures

11.1 Defining a structure

C supports a constructed data type called ***structures*** as a mechanism for packing data of different types. A structure is a convenient tool for handling a group of logically related data items.

Structures must be defined first for their format that may be used later to declare structure variables. General format

```
struct    tag_name
{
data_type          member 1;
data_type                    member 2;
………..            ………
………..            ………
};
```

Example to create a books database record to hold **title, author, no of pages and price**

```
struct  book_bank
{
        char    title[20];
        char  author[15];
        int  pages;
        float price;
};
```

The keyword ***struct*** declares a structure to hold the details of four data fields;- title, author, pages and price. These fields are called ***structure elements*** or ***members*** and each may belong to different data type. **Struct** defines the following template to hold data

Title:	Array of 15 characters
Author:	Array of 20 characters
Pages:	integer
Price:	float

Note:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as book_bank can be used to declare structure variables of its type, later in the program

11.2. Arrays vs Structures

Both are classified as structured data types as they provide a mechanism that enables us to access and manipulate data in a relatively easy manner. They differ in a number of ways;

- An array is a collection of related data elements of same type. Structures can have elements of different data types.
- An array is derived data type whereas a structure is a programmer-defined one.
- Any array behaves like a build-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

11.3. Declaring Structure Variables

After defining a structure format we can declare variables of that type. A structure variable declaration includes the following elements

- The keyword struct
- The structure tag_name
- List of variable names separated by commas
- A terminating semicolon.

Example: **struct book_bank, book1, book2, book3;**

declares book1, book2, & book3 as variables of type ***struct book_bank***. Each of the variables has four members as specified by the template. The complete declaration thus has the format;

```
struct  book_bank
{
        char    title[20];
```

```
        char  author[15];
        int  pages;
        float price;
    };
    struct  book_bank, book1, book2, book3;
```

The members of a structure are not variables themselves and do nor occupy any memory until they are associated with structure variables such as **book1**

11.4. Accessing Structure Members

Members of a structure must be linked to structure variables in order to make them meaning full and this link is established using ***member operator (.)*** called ***dot operator*** or ***period operator***. Example book1.price

```
strcpy(book1.title ,"BASIC");
strcpy(book1.author, "Balaburusamy");
book1.pages = 250;
book1.price = 120.50;
```

or use scanf to get the value from the keyboard

```
scanf("%s\n", book1.title);
scanf("%d\n", &book1.pages);
```

Program Example 11.1 Program to demonstrate the use of structures in C

Define a structure type, struct personal that would contain person name, date of joining and salary. Using this structure to write a program to read this information for one person from the keyboard and print the same on the screen

Example Version 1	Version 2
/* documentation section*/	/* documentation section*/
#include<string.h>	#include<string.h>
#include<stdio.h>	#include<stdio.h>
#include<conio.h>	#include<conio.h>
struct personal	struct personal
{	{
char name[20];	char name[20];
int day;	int day;
char month[10];	char month[10];
int year;	int year;
float salary;	float salary;
};	};
main()	main()

{ struct personal person; printf("Enter Name:"); scanf("%s", person.name); printf("Enter day of joining eg 10:"); scanf("%d", &person.day); printf("Enter Month eg. May:"); scanf("%s", person.month); printf("Enter Year:"); scanf("%s", &person.year); printf("Enter Salary:"); scanf("%s", &person.salary); printf("Name\t Date\t Month\t Year\t Salary\n"); printf("%s\t %d\t %s\t %4d\t, %8.2f\n", person.name, person.day, person.month, person.year, person.salary); getch(); }	{ struct personal person; printf("Enter Name, Date, Month, Year & Salary:"); scanf("%s %d %s %d %f", person.name, &person.day,person.month,&person.year,&person.salary); printf("\nName\t Date\t Month\t Year\t Salary\n"); printf("%s\t %d\t %s\t %4d\t %8.2f\n", person.name, person.day, person.month, person.year, person.salary); getch(); }

11.5 Structure initialization

Structures can be initialized at compile time. Example

```
main()
{
struct
{
int weight;
float height
}
student = {60, 180.75};
…….
……….
}
```

This will assign 60 to student.weight and 180.75 to student.height.

Or

```
main()
{
struct str_record
{
int weight;
float height;
};
struct str_record student1 = {60, 180.75};
struct str_record student2 = {53, 170.60};
```

……..
………
)

Note

At compile time initialization of a structure variable must have the following elements

- The keyword struct
- The structure tag name
- The name of the variable to be declared
- The assignment operator =
- A set of values for the members of the structure variable, separated by commas an enclosed in braces.
- A terminating semicolon.

Rules for Initializing Structures

- We cannot initialize individual members inside the structure template
- The order of values enclosed in braces must match the order of members in the structure definition.
- It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The un-initialized members should be only at the end of the list.
- The un0initialized members will be assigned default values as follows;
    - Zero for integer and floating point numbers
    - '\0' for character and string

11.6. Arrays of Structures

We use structures to describe the format of a number of related variables. Example in analyzing students marks we use a template to describe student's name and marks obtained in various subjects and then declare all the students as structure variables.

struct class student[100]; defines an ***array called student*** that consists of 100 elements, and each element is defined to be of the type **struct class**

Consider the following

```
struct marks
{
int subject1;
int subject2;
int subject3;
};
main()
{
```

struct marks student[3] = {{45,68,81}, {75,53,69}, {57,36,71}};

This declares the student as an array of three elements **student[0], student[1]** and **student[2]** and initializes their members as follows;

student[0].subject1 = 45;

student[0].subject2 = 65;

……..

……

student[2].subject3 = 71;

Program Example 11.2: Program calculate the totals per students and subject

Three students results were as follows in three subjects

Student	Subject1 Marks	Subject2 Marks	Subject3 marks
Student1	45	67	81
Student2	75	53	69
Student3	57	36	71

```
/* documentation  section*/
#include<string.h>
#include<stdio.h>
#include<conio.h>
struct marks
    {
        int sub1;
int sub2;
int sub3;
int total;
    };
main()
{
int i;
struct marks student[3] = {{45,68,81,0},{75,53,69,0},{57,36,71,0}};
struct marks total;
for (i =0; i<= 2; i++)
{
  student[i].total = student[i].sub1 +
                     student[i].sub2 +
                     student[i].sub3;
```

```
  total.sub1 = total.sub1 + student[i].sub1;
  total.sub2 = total.sub2 + student[i].sub2;
  total.sub3 = total.sub3 + student[i].sub3;
  total.total = total.total + student[i].total;
}
printf("STUDENT         TOTAL\n\n");
for (i =0; i<=2; i++)
   printf("Student[%d]          %d\n", i+1, student[i].total);
   printf("\n SUBJECT          TOTAL\n\n");
   printf("%s                 %d\n%s        %d\n%s       %d\n",
"Subject 1  ", total.sub1,
            "Subject 2   ", total.sub2,
            "Subject 3  ", total.sub3);
printf("\n Grand Total  = %d\n", total.total);

getch();
}
```

11.7. Pointers

A pointer is a derived data type in C. a pointer is built from one of the fundamental data types available in C and contains memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are undoubtedly one of the most distinct and exciting features of C language. It has added power and flexibility to the language. Pointers are used frequently in C as they offer a number of benefits to the programmers. They include;

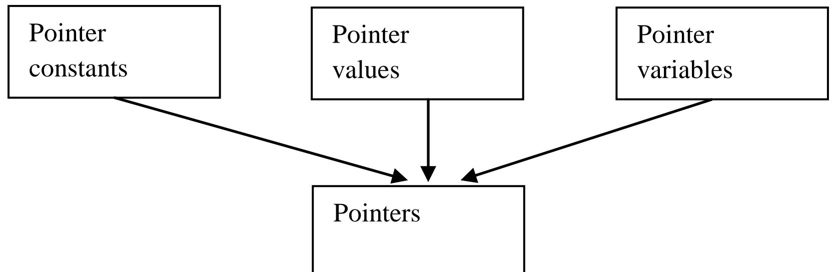
1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of a data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs
8. They increase the execution speed and thus reduce the program execution time.

A computer memory is a sequential collection of ***storage cells***, and each sell commonly known as ***byte*** has a number called ***address*** associated with it. Typically the addresses are numbered consecutively, starting from zero, and the last address depends on the memory size.

When a variable is declared, the computer allocates somewhere in the memory, an appropriate location to hold the value of the variable. Access to the variable value can either be through the variable name of the memory address allocated. Memory addresses can be assigned to variables and the variables that hold memory addresses are called pointer variables.

Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below;



- Memory addresses within a computer are referred to as ***pointer constants***. We cannot change them, we can only use them to store data values. They are like house numbers.
- We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as ***pointer value***. The pointer value (i.e. the address of a variable) may change from one run of the program to another.
- Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a ***pointer variable***.

11.7.1. Accessing the Address of a Variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately.

The operator & (available in C and referred to as **"address of"**) immediately preceding a variable returns the address of the variable associated with it. The & operator can only be used for simple variable or an array element.

Program Example 11.3: Program to print the Address of a variable along with its value

```
#include <stdio.h>
#include<conio.h>
main()
{
```

```
char a;
int x;
float p, q;

a = ‘A’;
x = 125;
p = 10.25, q = 18.76;
printf(“%c   is stored at addr %u.\n”, a, &a);
printf(“%d   is stored at addr %u.\n”, a, &x);
printf(“%f   is stored at addr %u.\n”, p, &p);
printf(“%f   is stored at addr %u.\n”, q, &q);
getch();
}
```

11.7.2. Declaring Pointer Variables.

Pointers must be declared as pointers before they can be used. Format

- **data_type *pt_name;**

Declaration of pointers tell the compiler three things about the pointer pt_name;

- The Asterisk (*) tells that the variable pt_name is a pointer variable;
- Pt_name needs a memory location;
- Pt_name points to a variable of type data_type.

Example ***int *p***;   /* integer pointer */

Declares the variable **p** as a pointer variable that points to an integer data type. The declaration causes the compiler to allocate memory location for the declared pointer variable.

11.7.3. Pointer declaration styles

Pointer variables are declared similarly as normal variables except for the addition of the Unary * operator. This symbol can appear anywhere between the type name and the pointer variable name. Programmers uses the following styles;

i). int* p;   ii). int *p;   iii). int * p;

11.7.4. Initialization of Pointer variables

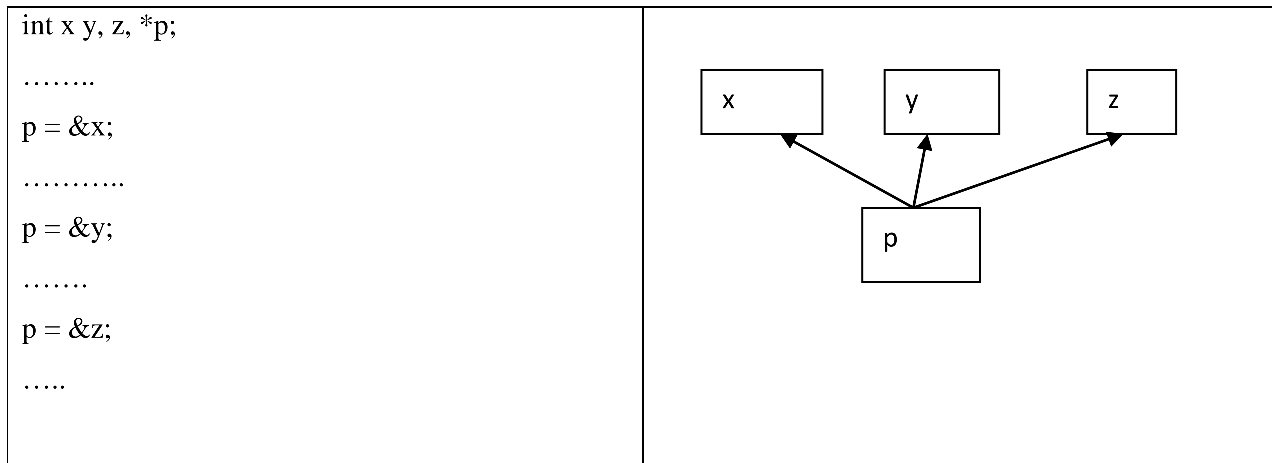
Initialization is the process of assigning the address of a variable to a pointer variable. Example

```
int quantity;
int *p;            /* declaration */
p = quantity;      /* initialization */
```

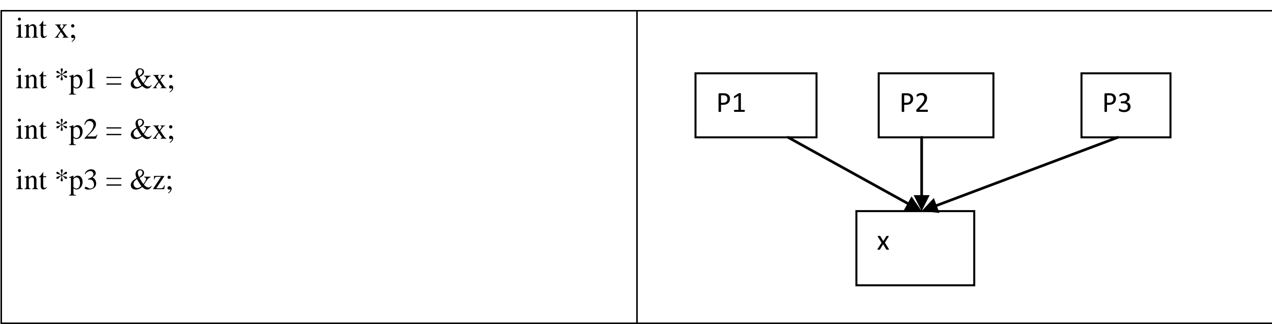
or int *p = quantity ;

Pointer Flexibility

Pointers are flexible. We can make the same pointer to point to different data variables in different statements.



We can also use different pointers to point to the same data variable. Example



11.7.5. Accessing a Variable through its pointer

Unary operator * usually known as ***Indirection operator*** is used to access the value of the variable using a pointer. Another name for the indirection operator is the *dereferencing* operator. Example

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
```

Program Example 11.4: Program to illustrate the use of indirection operator '*' to access the value pointed by a pointer.

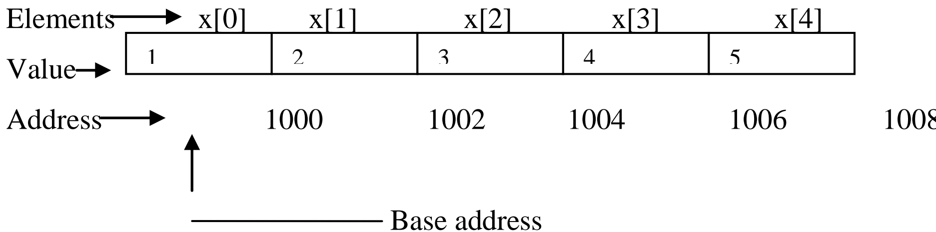
```
#include <stdio.h>
#include<conio.h>
main()
{
int x y;
int *ptr
 x = 10;
```

```
ptr = &x;
y = *ptr;
printf("Value of x is %d\n\n", x);
printf("%d   is stored at addr %u.\n", x, &x);
printf("%d   is stored at addr %u.\n", *&x, &x);
printf("%d   is stored at addr %u.\n", *ptr, ptr);
printf("%d  is stored at addr %u.\n", y, &y);
*ptr = 25;
printf("\nNow x = %d\n" x);
getch();
}
```

11.8. Pointers and Arrays

When an array is declared the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory allocation. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Example

int x[5] = {1,2,3,4,5};



X is defined as a constant pointer pointing to the first element x[0] and therefore the value of x is 1000, the location where x[0] is stored. Ie.

x= &x[0] = 1000

Program Example 11.5: Program using Pointers to compute the sum of all elements stored in an Array.

```
#include <stdio.h>
#include<conio.h>
main()
{
int *p, sum,  i;
int x[5] = {5,9,6,3,7);
 i =0;
p = x;
```

```
printf("Element       Value      Address \n\n");
while (i<5)
{
  printf(" x[%d]    %d       %u\n", i *p, p);
sum = sum + *p;
i++, p++;
}
printf ("\n  Sum     =  %d\n",  sum);
printf ("\n  &x[0]  =  %u\n" , &x[0]);
printf ("\n  p         =  %u\n", p);
getch();
}
```

Out put

```
Element                Value             Address
x[0]          5
x[1]          9
x[2]          6
x[3]          3
x[4]          7

sum   =
&x[0]          =
p      =
```

Chapter Review Questions

1. State whether the following statements are true or false
   a) A ***struct t***ype in C is built-in data type
   b) Structures contains members of only one data type
   c) The keyword ***typedef*** is used to define a new data type
2. Given the following declaration

   Struct abc a, b, c;

Which of the following statements are legal?

   a) scanf("%d, &a);
   b) printf("%d", b);
   c) a = b;
   d) a = b+c;
3. Define a structure data type called time_struct containing three members integer hour, integer minute and integer second. Develop a program that would assign values to the individual members and display the time in the following form'

   16:30:*45*

CHAPTER TWELVE

C- Programming: Managing Files in C

Chapter Objectives

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

- Describe the concept of files in C
- Describe and use/implement the C program basic file operations
  - ✓ Define and open a file
  - ✓ Close a file
  - ✓ Read from a file
  - ✓ Write into a file
  - ✓ Close a file

12.1. Introduction

A file is a place on the disk where a group of related data is stored. C supports a number of functions that have the ability to perform basic file operations, which includes;

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file
- Closing a file

There are two distinct ways to perform file operations in C

1. Low-level I/O and used the Unix system calls
2. High-level I/O operation and uses functions in C's standard I/O library
3. High Level I/O functions in C

Function	**Operation**
fopen()	Creates a new file for use / opens an existing file for use
fclose()	Closes a file that had been opened for use
getc()	Reads a character from a file
putc()	Writes a character to file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads an integer from a file

putw()	Writes an integer in a file
fseek()	Set the position to a desired point in a file
ftell()	Fives the current position in the file(in terms of bytes from the start)
rewind()	Sets the position to the beginning of the file

There are many other functions and not all are supported by all the C compilers

12.2. Defining and Opening aFile.

If we want to store data in a file in the secondary memory, there is need to specify the following things about the file;

- Filename: File name is a string of characters that make up a valid filename for the operating system. It may contain two parts a primary name and an optional period with the extension. Examples
  Input.data, Student.c, Text.out

- Data structure: Data structure of a file is defined a FILE in the library of standard I/O function definitions. Therefore all files should be declared as type FILE before they are used. FILE is a defined data type

- Purpose. When we open a file we must specify what we want to do with the file, example read from or write to it. Format
  FILE *fp;
  fp = **fopen**("filename", "mode");

  FILE *fp declares the variable fp as a "pointer to the data type FILE. FILE is a structure that is defined in the I/O library.

  fp = fopen('filename, "mode"); open the file named and assigns an identifier to the FILE type pointer fp. The fp pointer which contains all the information about the file is subsequently used as a communication link between the system and the program.
  The statement also specifies the purpose of opening the file. Options includes;
- r = open a file for reading only
- w – opens a file for writing only.
- a – opens a file for appending (or adding) data to it.

  When opening a file;
- When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted if the file already exists.
- When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.

- If the purpose is ‘reading’ and if it exists, then the file is opened with the current contents safe otherwise abd error occurs.

Example

FILE *p1, *p2;

p1 = fopen(“data”, “r”);

p2 = fopen(“results”, “w”);

- p1 is opened for reading and if it does not exist an error occurs.
- p2 is opened for writing and if it exists, its contents are deleted and the file opened a new file.

Additions modes includes;

r+ - existing file is opened to the beginning for both read and write.

w+ - same as w except both for reading and writing.

a+ - same as a except both for read and writing.

12.3. Closing a File

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. Format

- **fclose(file_pointer);**

example

FILE *p1, *p2;

p1 = fopen(“INPUT”, “w”);

p2 = fopen(“OUTPUT”, “r”);

………

…….

fclose(p1);

fclose(p2);

Once a file is closed its file pointer can be used for another file.

12.4. Input / Output Operations on Files

Once a file is opened, reading out of or writing into it is accomplished using the standard I/O routines mentioned earlier.

Getc and putc functions

These are analogous to getchar and putchar functions that handles one character at a time

Example;

- getc(c, fp1); writes the character contained in the character variable c to the file associated with FILE pointer fp1, when the file is opened with write mode.

- c = getc(fp2); reads a character from the file whose file pointer is fp2.

Prgram Example 12.1: Program to read data keyed in through the keyboard and write it into a file, then read the data from the file and display it on the screen

- The End-Of-File (EOF) marker is achieved through pressing ***<control-z>***

Data to enter: ***This is a program to test the file handling features of C program.***

```
#include <stdio.h>
#include<conio.h>
main()
{
FILE *f1;
char c;
printf("Data Input\n\n");
                                        /* open the INPUT file for writing */
f1 = fopen("INPUT", "w");
                                        /* get character from the Keyboard */
while ((c=getchar()) != EOF)
putc(c, f1);
                                        /* close the INPUT file */
fclose(f1);
printf("\nData Output \n\n");
                                        /* Open the INPUT file for reading */
f1 = fopen("INPUT", "r");
                                        /* Read a character from the INPUT file */
 while ((c=getc(f1)) !=EOF)
                              /* Display the character on the screen */
 printf("%c", c);
                              /* Close the INPUT file */
fclose(f1);

getch();
}
```

Testing for End-Of-File (EOF) condition is important, as attempting to read past the EOF might either cause the program to terminate with an error or result in an infinite loop situation.

12.5. The getw and putw Functions

getw and **putw** are integer-oriented functions. Format

putw(interger, fp);

getw(fp);

Program Example 12.2: Write a program to read integers from a file called DATA and write all odd integers into a file called ODD and all even integers into a file called EVEN.

Test data

111	222	333	000	121
444	555	666	232	343
777	888	999	454	-1

-1 is entered to terminate the reading and file is closed

```
#include <stdio.h>
#include<conio.h>
main()
{
FILE *f1, *f2, *f3;
int number, i;

printf ("Content of DATA file \n\n");
f1 = fopen("DATA", "w");
for (i = 1; i<= 30; i++)
{
 scanf ("%d", &number);

if (number == -1) break;
putw(number, f1);
}
fclose(f1);

f1 = fopen("DATA", "r");
f2 = fopen("ODD", "w");
f3 = fopen("EVEN", "r");

while((number = getw(f1)) != EOF)
{
```

```
  if(number  %2 == 0)
 putw(number, f3);
else
 putw(number, f2);
}

fclose(f1);
fclose(f2);
fclose(f3);

f2 = fopen("ODD", "r");
f3 = fopen ("EVEN", "r");

printf ("\n\nContents of ODD file\n\n");
  while ((number = getw(f2)) != EOF)
  printf("%4d", number);

printf("\n\nContents of EVEN file\n\n");
    while ((number = getw(f3)) != EOF)
    printf("%4d", number);

fclose(f2);
fclose(f3);

getch();
}
```

12.6. The fprintf and fscanf Functions

fprintf and fscanf perform I/O operations on files.

Format for fprint()

fprintf(fp, “control string”, list);

- fp is the file pointer associated with a file that has been opened for writing
- control string contains output specifications for the items in the list
- list may include variables, constants and strings

Example

fprintf(f1, “%s %d %f”, name, age, 7.5);

fscanf Format

fscanf(fp, control string", list);

example

fscanf(f1, "%s %d", item, &quantity);

Program Example 12.3: Write a program to open a file named INVENTORY and store in it the following data;

Item Name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Use file name 'INVENTORY'

```
#include <stdio.h>
#include<conio.h>
main()
{
FILE *fp;
int  number, quantity,  i;
float  price, value;
char  item[10], filename[10];

printf("Input file name \n");
scanf ("%s", filename);
fp = fopen(filename, "w");
printf("Input Inventory data\n\n");
printf("Item name  Number  Price Quantity\n");

for (i =1; i <=3; i++)
{
fscanf(stdin, "%s    %d   %f   %d", item, number, &price, &quantity);
fprintf(fp, "%s    %d    %.2f  %d", item, number, price, quantity);
}

fclose(fp);
```

```
fprintf(stdout, "\n\n");

fp = fopen(filename, "r");
printf("Item name     Number    Price    Quantity    Value\n");
  for (i =1 ; i <=3; i++)
{
  fscanf(fp, "%s     %d    %f    %d", item, &number, &price, &quantity);
   value = price * quantity;
fprintf(stdout, "%-8s  %7d  %8.2f  %8d  %11.2f\n" ,
                 item, number, price, quantity, value);
}
fclose(fp);
getch();
}
```

12.7. Error Handling during I/O Operations

Typical I/O operations errors include;

1. Trying to read beyond the end-of-file mark.
2. Device overflow
3. Trying to use a file that has not been opened
4. Trying to perform an operation on a file, when the file is opened for another type of operation
5. Opening a file with an invalid filename
6. Attempting to write to a write-protected file.

C has two status-inquiry library functions; **feof** and **ferror** that helps to detect I/O errors in files.

feof is used to detect end-of-file condition. It take the file pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read and returns zero otherwise.. example if fp is a file pointer that has been opened then the statement ;

```
if(feof(fp))
printf("End ofdata");
```

would display "End of data" on reaching the end of file condition.

Ferror function reports the status of the file indicated. It also takes the file pointer as its only argument and returns a nonzero integer value if an error has been detected up to that point, during processing. It returns zero otherwise. Example

```
if (ferror(fp) != 0)
printf ("An error has occurred. \n");
```

or to check if file is opened

- when a file is opened using the **fopen** function, the file pointer is returned, thus if the file could not be opened the function returns zero.

  if (fp == NULL)

  printf("File could not be opened");

Program Example 12.4: Program to Illustrate error handling in file operations

Use file name 'TEST'

```
#include <stdio.h>
#include<conio.h>
main()
{
char   *filename;
FILE *fp1, *fp2;
int i, number;

fp1 = fopen("TEST", "w");
for (i = 10; i<= 100; i+=10)
     putw(i, fp1);

fclose(fp1);

printf("\nInput file name\n");

open_file:
scanf("%s", filename);
if((fp2 = fopen(filename, "r"))== NULL)
{
printf("Cannot open the file.\n");
printf("Type file name again.\n\n");
goto open_file;
}
else
  for (i = 1; i<= 20; i++)
{
  number = getw(fp2);
```

```
    if (feof(fp2))
    {
    printf("\nRan out of data.\n");
    break;
    }
    else
    printf("%d\n", number);
    }

    fclose(fp2);
    getch();
}
```

Chapter Review Questions

1. what do the following statements do?

   a). while ((c=getchar() != EOF)

   putc(c, fl);

   b). while ((m=getw(fl)) != EOF

   prinbtf ((“%5d”, m)

2. Write a program t copy the content from one file to another.
3. Write a program to append one file at the end of another
4. Write a program to compare two files and return 0 is they are equal and 1 if they are not.

Sample Examination Papers

Mt Kenya University

SCHOOL OF APPLIED AND SOCIAL SCIENCES
DEPARTMENT OF INFORMATION TECHNOLOGY

EXAMINATION FOR BACHELOR OF BUSINESS INFORMATION TECHNOLOGY

BIT 2203: COMPUTER PROGRAMMING METHODOLOGY TIME 2 HOURS

Instructions

*Answer question **ONE** and any other **TWO** questions*

QUESTION ONE (30 MARKS)

a). Write a C program to calculate the value of money at the end of each year over a period of 10 years assuming an interest rate of 12 percent and the initial amount was 10,000. The program should print the year and its corresponding amount in two columns.
Use the appropriate data types for the program variables.
[20 Marks]

b). Distinguish between high level programming language and machine level programming language [4 Marks]

c). Syntax errors occurs when a program is compiled.
i) Define Syntax [2 Marks]
ii). Differentiate between Syntax error and logical error. [4 Marks]

QUESTION TWO (20 MARKS)

a). Describe the following as applied in C Programming language and provide an example where appropriate
i). Type conversion
ii). Global variables
iii). Compiler
iv). Arithmetic operator
v). The DO…. WHILE loop [10 Marks]

b). Write a C program to covert a string into uppercase using in-built function from the string.h header. [6 Marks]

c). List four advantages of C program [4 Marks]

QUESTION THREE (20MARKS)

a). What do you understand by array? Give declaration of 1-dimensional, 2-dimensional and 3- dimensional array.
Write a C program to sort the list in ascending order. [12 Marks]

b). Write short notes on the following :

(i) Sequential input/output operators

(ii) Operator overloading **(Recursion)**

(iii) Structures

(iv) Control structures. [8 Marks]

QUESTION FOUR (20MARKS)

a). Describe the four basic data types in C programming language [4 Marks]

b). State the difference between the declaration of a variable and the definition of a symbolic name [4 Marks]

c). Write a C program that inputs the subject marks of a student and tell him/her the grade that he/she has obtained in the subject. The grading criteria is as follows;

Marks	Grade
90 and above	A
80 - 89	B
70 - 79	C
60 - 69	D
Below 60	Fail

If the student fails specify that the student will be required to sit for a Supplementary examination [10 Marks].

QUESTION FIVE (20MARKS)

a). Using the Switch statement, create a C program that prints the name of an input number if it's a digit. If it is not it informs the user that the number is not a digit.
Use numbers 0 to 3 in the program [8 Marks]

b). Describe the importance of program documentation [4 Marks]

c). Describe the advantages of structured programming. [4 Marks]

d). Differentiate between white box and black box testing [4 Marks]

Mt Kenya University

SCHOOL OF APPLIED AND SOCIAL SCIENCES
DEPARTMENT OF INFORMATION TECHNOLOGY

EXAMINATION FOR BACHELOR OF BUSINESS INFORMATION TECHNOLOGY

<u>BIT 2203: COMPUTER PROGRAMMING METHODOLOGY</u> **TIME 2 HOURS**

Instructions

*Answer question **ONE** and any other **TWO** questions*

<u>QUESTION ONE (30 MARKS)</u>

a). Write a C program to out-put the following multiplication table of a number, when it is keyed in from the keyboard.

5 x 1 = 5
5 x 2 = 10
5 x 3 = 15

----
5 x 10 = 50

15 Marks]

a). Define the following terms
i). Algorithm
ii). Pseudocode
iii). Portability
iv). Recursion
v). Repetition control structures [10 Marks]

c). List the five (5) relational operators in C [5 Marks]

<u>QUESTION TWO (20 MARKS)</u>

a). Explain the following as they relate to functions

i). User defined function
ii). Function prototype
iii). Local variable
iv). Structure

[8 Marks

b). Give two differences between While …. and Do ….. While. [6 Marks]

c). Identify syntax errors in the following program. After corrections, what output would you expect when you execute it

```
#define PI 3.14159
Main()
{
 Int R, C
Float perimeter
Float area;
C = PI
R = 5;
Perimeter = 2.0*C*R;
Area = C*R*R;
Printf("%f" "%d", &perimeter, &area)
}
```

[6 Marks]

QUESTION THREE 20 MARKS

a). What is meant by

i). Data structures

ii). Array of structures

iii) Sub-program

[6 Marks]

b). Write a C program to calculate the average of a set of N numbers. [10 Marks]

c). Describe the difference between printf and fprintf as used in C programming [4 Marks]

d). Describe the uses of compilers and text editors as software resources used in programming [2 Marks]

QUESTION FOUR (20 MARKS)

a). List four characteristics of a good Algorithm [4 Marks]

b). List for qualities of a good program [4 Marks]

c). A Sales person for a mobile handset selling company, earns a basic salary of 20,000, a commission of 2% for all total sales for the months and a bonus of 300 for every handset sold during the month.
Write a C program to calculate the gross salary for a worker at the end of the month when the number of hand sets sold is keyed in. [12 Marks]

QUESTION FIVE (20 MARKS)

a). Differentiate between a reserved word and Identifier [4 Marks]

b). Briefly describe basic structure of a C Program [8 Marks]

c). What is the output of the following C programs

i).

```
main()
{
```

```
        char x;
        int y;
        x = 100;
        y = 125;
        printf("%c\n", x);
        printf("%c\n, y);
        printf(%d\n", x);
    }
```

ii).

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