

BABCOCK e-LEARNING

Introduction to Programming in C++

Page | 1

- COURSE CODE: COSC102
- COURSE TITLE: INTRODUCTION TO PROGRAMMING IN C++
- AUTHOR: OMOTUNDE, AYOKUNLE A.
- SCHOOL: COMPUTING AND ENGINEERING SCIENCE
- DEPARTMENT: COMPUTER SCIENCE
- MODULE (OVERVIEW – OPTIONAL)
- At the end of this course, students should be able to write simple C++ codes to execute some basic programming task.

- **COURSE DESCRIPTION**

This material was prepared to serve as a first introduction to programming in C++. It is not assumed that the student has prior knowledge of programming hence the course does not cover programming in C++ in its entirety. To be precise, it doesn't cover any of the object oriented feature of C++. Because this is a first programming course emphasis is placed on the design of programs in a language-independent fashion. A brief introduction to computers is also given.

- **TIME (TO COMPLETE THE MODULE)**

It runs for fifteen weeks based on the assumption that three hours will be put into it per week.

- **COURSE PRE-REQUISITE (IF ANY)**

- **COURSE TEXT(S)**

1. Deitel, P., & Deitel, H. (2012). *C++ How to Program* (8th ed.). Boston, Massachusetts, USA: Pearson Education.
2. Liberty, J., & Jones, B. (2005). *Sams Teach Yourself C++ in 21 days* (5th ed.). Indianapolis, Indiana, USA: Sams Publishing.
3. Soulie, J. (2008). *C++ Language*. cplusplus.com. Retrieved from <http://www.cplusplus.com/doc/tutorial>

- **CONTENT MODULES AND UNITS**

MODULE 1: COMPUTER AND COMPUTER SYSTEMS

Unit 1.1 The Computer

- [1.1.1 What is a Computer](#)
- [1.1.2 Input/output](#)
- [1.1.3 Central Processing Unit \(CPU\)](#)
- [1.1.4 Memory](#)
- [1.1.5 System Bus](#)
- [1.1.6 Hardware and Software](#)
- [1.1.7 Data Processing](#)
- [1.1.8 Tutor Marked Assessment](#)
- [Unit 1.2 – Programming/Programming Languages.](#)
 - [1.2.1 Programs](#)
 - [1.2.2 Programming Languages](#)
 - [1.2.3. From Problem to Program](#)
 - [1.2.4 Tutor Marked Assessment](#)
- [Unit 1.3 - Operating Systems](#)
 - [1.3.1 Tutor Marked Assessment](#)
- [Unit 1.4 - Preparing a Computer Program](#)
 - [1.4.1 Tutor Marked Assessment](#)

Module 1 End of Module Assessment

MODULE – 2: DESIGN OF PROGRAMS

- [Unit 2.1 A “Disciplined” Approach](#)
 - [2.1.1 What is a Model?](#)
- [Unit 2.2 Algorithms](#)
 - [2.2.1 Flowchart](#)
 - [2.2.1.1 Symbols used in a flowchart](#)
 - [2.2.1.2 Guidelines for Flow Charting:](#)
 - [2.2.1.3 Advantages of Flow Charts](#)
 - [2.2.1.4 Disadvantages of Flow Charts](#)
 - [2.2.2 Pseudo Code](#)
 - [2.2.2.1 Rules for writing Pseudo Code:](#)
 - [2.2.2.2 Advantages of Pseudo Code](#)
 - [2.2.2.3 Disadvantages of Pseudo Code](#)

MODULE – 3: PROGRAMMING IN C++

- [Unit 3.1 A simple program in C++](#)
- [Unit 3.2 The Programming Environment](#)
 - [Text Editor:](#)
 - [C++ Compiler:](#)
- [Unit 3.3 C++ Program Structure](#)
 - [3.3.1 Comments](#)
 - [3.3.2 Pre-processor directives](#)
 - [3.3.3 using namespace std;](#)
 - [3.3.4 int main\(\)](#)
 - [3.3.5 cout](#)
 - [3.3.6 return 0;](#)
- [Unit 3.4 Variables and Data Types](#)
 - [3.4.1 Variables and Memory Concepts](#)
 - [3.4.2 Variable Scope](#)
 - [3.4.3 Keywords](#)
 - [3.4.4 Declaration of variables](#)
 - [3.4.4.1 int](#)
 - [3.4.4.2 float](#)
 - [3.4.4.3 bool](#)
 - [3.4.4.4 char](#)
 - [3.4.5 Modifiers](#)
- [Unit 3.5 Constants and the declaration of constants](#)
 - [3.5.1 The *const* keyword](#)
 - [3.5.2 The *#define* Pre-processor](#)
- [Unit 3.6 Introduction to String](#)
 - [3.6.1 The escape Character](#)

MODULE – 4: OPERATORS AND ASSIGNMENT STATEMENTS

- [Unit 4.1 Assignment Operator](#)
- [Unit 4.2 Arithmetic Operator](#)
 - [4.2.1 Increment ++/Decrement - - Operators](#)
 - [4.2.2 Priority of Arithmetic Operators](#)
- [Unit 4.3 Relational Operator](#)
- [Unit 4.4 Logical Operator](#)

MODULE – 5: CONDITIONAL STRUCTURE/DECISION MAKING STRUCTURE IN C++

- [Unit 5.1 If Statement](#)
- [Unit 5.2 If...else statement](#)

- [Unit 5.3 If...else if statement](#)
- [Unit 5.4 SWITCH STATEMENT](#)
 - [5.4.1 Switch Statement](#)
 - [5.4.2 Nested switch statement](#)

MODULE 6: LOOPS

- [Unit 6.1 while loop](#)
- [Unit 6.2 For loop](#)
- [Unit 6.3 Do...while loop](#)
- [Unit 6.4 Nested Loops](#)
 - [6.4.1 Nested for loop](#)
 - [6.4.2 Nested while loop](#)
 - [6.4.3 Nested do...while loop](#)

MODULE – 7: C++ FUNCTIONS AND NUMBERS

- [UNIT 7.1 C++ Functions](#)
 - [Library/Intrinsic Functions](#)
 - [User-defined function](#)
 - [7.1.1 Defining a Function](#)
 - [7.1.2 Functions with no parameters and no return value](#)
 - [7.1.3 Functions with no parameters but return value](#)
 - [7.1.4 Functions with parameters but no return value](#)
 - [7.1.5 Functions with parameters that return values](#)
- [UNIT 7.2 Numbers in C++](#)
- [Unit 7.3 Math Operators in C++](#)
- [Unit 7.4 Random Numbers](#)

MODULE – 8: ARRAYS

- [Unit 8.1 Arrays](#)
- [Unit 8.2 Declaring Arrays](#)
- [Unit 8.3 Initializing Arrays](#)
- [Unit 8.4 Accessing Array Elements](#)
- [Unit 8.5 More on Arrays](#)
 - [8.5.1 Multi-dimensional Array](#)
 - [Two-dimensional array](#)
 - [Initializing Two-Dimensional Arrays](#)

- [Accessing Two-Dimensional Arrays](#)
- [8.5.2 Pointer to an Array](#)
- [8.5.3 Passing Arrays as Function Arguments in C++](#)

MODULE – 9: MORE ON STRINGS

Page | 5

- [Unit 9.1 Character String in C](#)
 - [C-style character string](#)
- [Unit 9.2 Functions that manipulate null-terminated strings](#)
 - [9.2.1 strcpy function](#)
 - [9.2.2 strcat function](#)
 - [9.2.3 strlen function](#)
- [Unit 9.3 The String Class in C++](#)
- [9.4 Differences between C-Style character string <cstring> and C++ string class type <string>](#)

MODULE – 10: C++ BASIC INPUT/OUTPUT

- [Unit 10.1 I/O Library Header Files](#)
- [Unit 10.2 The standard output stream \(cout\)](#)
- [Unit 10.3 The standard input stream \(cin\)](#)
- [Unit 10.4. Inputting white spaces](#)
- [Unit 10.5 C++ Standard error stream](#)
- [Unit 10.6 C++ Standard log stream \(clog\)](#)

MODULE 11: INPUT/OUTPUT WITH FILES

- [Unit 11.1 Opening/Creating a file](#)
 - [11.1.1 Flag ios::out](#)
 - [11.1.2 Flag ios::binary](#)
 - [11.1.3 ofstream, ifstream, & fstream opening mode](#)
- [Unit 11.2 Closing a file](#)
- [Unit 11.3 Text files](#)
- [Unit 11.4 State Flags](#)

Appendix A: C++ Standard Library File Header

Appendix B: Math Library Functions

END OF MODULE ASSESMENT (EMA)

- **GRADING**

Attendance	5%
Quiz	10%
Assignment	10%
Mid-Semester	15%
Project	20%
Exam	40%
TOTAL	100%



Omotunde, Ayokunle A. is an Assistant Lecturer in Babcock University. He bagged a Bachelor's Degree in Computer Information System in 2006, after which he worked as a System Programmer in Tai Solarin University of Education, Ijagun, Ogun State. He later went for his Master's Degree in Computer Information Systems and graduated in 2012.

Ayokunle resumed as a teaching staff/faculty in Babcock University in 2012 and he is currently a PhD student of Computer Science in Babcock University. Together he has 10 years of experience, out of which 6 has been dedicated to teaching and research.

His research areas are Cloud Computing, Business Intelligence, Mobile systems, Content Delivery Networks, Artificial Intelligence, Web-Development and eLearning

MODULE 1: COMPUTER AND COMPUTER SYSTEMS

GENERAL OBJECTIVES

- To better understand the concept of computers and computer systems

SPECIFIC OBJECTIVES

At the end of this module, you will be able to

- Understand what is meant by a computer
- Identify the basic components of a computer system
- Distinguish between hardware and software of a computer system
- Understand what is meant by a program and programming language.

1.1 The Computer

1.1.1 What is a Computer?

The term computer is derived from the word *compute*. A computer is an electronic device that inputs (takes in) facts (known as data), and then processes (does something to it in accordance to a set of instructions called programs) it. Afterwards it outputs, or displays, the results for you to see. All these are done according to a set of instructions called programs. The act of developing these programs forms the basis of the course.

From this definition, it can be deduced that the computer is divided into two broad parts: the computer hardware and the computer software. Hardware consists of all the parts of a computer that can be seen or touched, while the software consists of instructions that a computer uses to perform tasks that it has been designated to carry out.



Figure 1.1. A Complete Computer System

The basic functions of a computer can better be expressed using figure 1.2

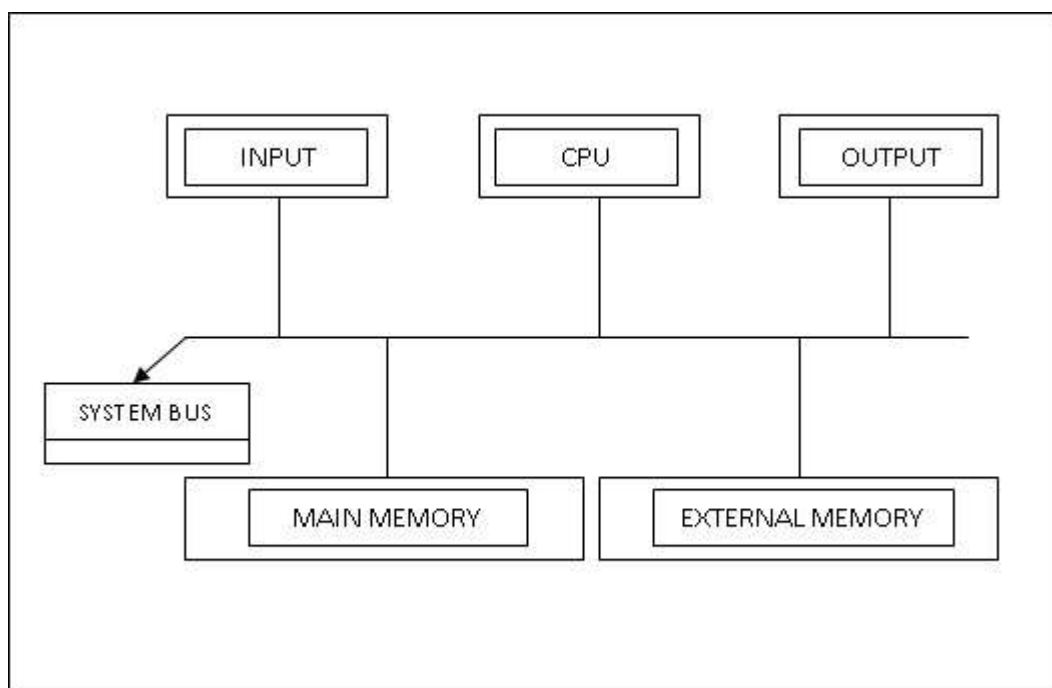


Figure 1.2 General Structure of a Computer

1.1.2 Input/output

When using a computer, it is necessary to capture text of programs, commands and data for processing into the computer system. After processing takes place, the results must also be made available to the users of the system. This requires the use of *input* and *output* devices (Input allows you to enter data into the system and output allows you to view the result of the processed data).

The most common input devices used by the computer are the keyboard and the mouse. The keyboard allows the entry of textual information while the mouse allows the selection of a point on the screen by moving a screen cursor to the point and pressing a mouse button. Using the mouse in this way allows the selection from menus on the screen etc. and is the basic method of communicating with many current computing systems. Alternative devices to the mouse are tracker balls, light pens and touch sensitive screens.

The most common output device is a monitor which is usually a Cathode Ray Liquid Crystal Display (LCD) device which can display text and graphics. If hard-copy output is required, then some form of printer is used.

1.1.3 Central Processing Unit (CPU)

This is responsible for performing the actual processing of data. The data it processes is obtained, via the system bus, from the main memory. Results from the CPU are then sent back to main memory via the system bus. In addition to computation the CPU controls and co-ordinates the operation of the other major components. The CPU has two main components, namely:

- *The Control Unit* -- controls the fetching of instructions from the main memory and the subsequent execution of these instructions. Among other tasks carried out are the control of input and output devices and the passing of data to the Arithmetic/Logical Unit for computation.
- *The Arithmetic/Logical Unit (ALU)* -- carries out arithmetic operations on integer (whole number) and real (with a decimal point) operands. It can also perform simple logical tests for equality and greater than and less than between operands.

****It is worth noting here that the only operations that the CPU can carry out are simple arithmetic operations, comparisons between the result of a calculation and other values, and the selection of the next instruction for processing. All the rest of the apparently limitless things a computer can do are built on this very primitive base by programming!**

Modern CPUs are very fast. At the time of writing, the CPU of a typical PC is capable of executing many tens of millions of instructions per second.

1.1.4 Memory

The memory of a computer can hold program instructions, data values, and the intermediate results of calculations. All the information in memory is encoded in fixed size cells called bytes. A byte can hold a small amount of information, such as a single character or a numeric value between 0 and 255. The CPU will perform its operations on groups of one, two, four, or eight bytes, depending on the interpretation being placed on the data, and the operations required.

There are two main categories of memory, characterized by the time it takes to access the information stored there, the number of bytes which are accessed by a single operation, and the total number of bytes which can be stored.

- **Main Memory (Primary Memory)** is the working memory of the CPU, with fast access and limited numbers of bytes being transferred. It is directly accessed by the CPU to store and retrieve information. Most of the time, primary memory is also referred to as the RAM (Random Access Memory). It is a volatile memory, which loses its data when the power is turned off. Primary memory is directly accessible by the CPU through the address and memory bus and it is constantly accessed by the CPU to get data and instructions. Furthermore, computers contain a ROM (Read Only Memory), which holds instructions that are executed often such as the startup program (BIOS). This is a non-volatile memory that retains its data when the power is turned off. Since the main memory is accessed often, it needs to be faster. But they are smaller in size and also costly.
- **External memory (Secondary Memory)** is for the long term storage of information. Data from external memory will be transferred to the main memory before the CPU can operate on it. Access to the external memory is much slower, and usually involves groups of several hundred bytes. It is a storage device that is not accessible directly by the CPU and used as a permanent storage device that retains data even after power is turned off. CPU accesses these devices through an input/ output channel and data is first transferred in to the primary memory from the secondary memory before accessing. Usually, hard disk drives and optical storage devices (CDs, DVDs) are used as secondary storage devices in modern computers. In a secondary storage device, data are organized in to files and directories according to a file system. This also allows to associate additional information with data such as the access permissions, owner, last access time, etc. Furthermore, when the primary memory is filled up, secondary memory is used as a temporary storage for keeping least used data in the primary memory. Secondary memory devices are less costly and larger in size. But they have a large access time.

Difference between Primary and Secondary Memory

- Primary memory is the memory that is directly accessed by the CPU to store and retrieve information, whereas the secondary memory is not accessible directly by the CPU. Primary memory is accessed using address and data buses by the CPU, while secondary memory is accessed using input/ output channels. Primary memory does not retain data when the power is turned off (volatile) while secondary memory retains data when the power is turned off (non-volatile). Furthermore, primary memory is very fast compared to the secondary memory and has a lower access time. But, primary memory devices are costlier compared to secondary memory devices. Due to this reason, usually a computer comprises of a smaller primary memory and a much larger secondary memory.

1.1.5 System Bus

All communication between the individual major components is via the system bus. The bus is merely a cable which is capable of carrying signals representing data from one place to another. The bus within a particular individual computer may be specific to that computer or may (increasingly) be an industry-standard bus. If it is an industry standard bus, then there are advantages in that it may be easy to upgrade the computer by buying a component from an independent manufacturer which can plug directly into the system bus. For example, most modern Personal Computers use the PCI bus.

When data must be sent from memory to a printer then it will be sent via the system bus. The control signals that are necessary to access memory and to activate the printer are also sent by the CPU via the system bus.

1.1.6 Hardware and Software

The computer may be seen as a just a box but the reality is that it is not just a box as long as it has been empowered with the ability to process data under a set of instructions called programs. At this point we draw a line between hardware and software.

Hardware is the physical part of a computer system (which may consist of the keyboard, mouse, monitor screen, hard disk, memory, DVD Drive, processing unit, and so on) while software are the programs that run on the computer that enables it to “think”. *Software* uses the physical ability of the hardware, which can run programs, do something useful. It is called software because it has no physical existence and it is comparatively easy to change.

1.1.7 Data Processing

Computers are data processors. Information is fed into them, they do something with it, and then generate further information. A computer program tells the computer what to do with the information coming in. A computer works on data in the same way that a sausage machine works on meat, something is put in one end, some processing is performed, and something comes out of the other end.

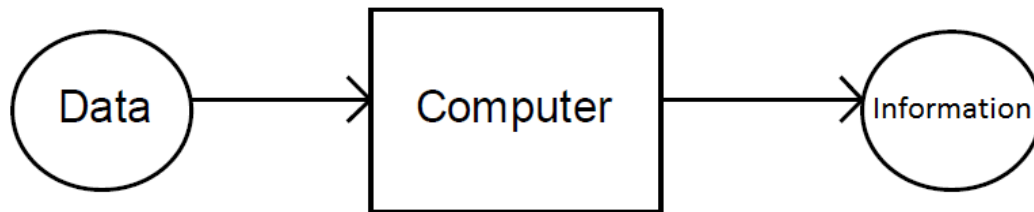


Figure 1.3 Transformation of Data into Information

A program is unaware of the data it is processing in the same way that a sausage machine is unaware of what meat is. Put a bicycle into a sausage machine and it will try to make sausages out of it. Put duff data into a computer and it will do equally useless things. It is only us people who actually ascribe meaning to data as far the computer is concerned it is just stuff coming in which has to be manipulated in some way.

A computer program is just a sequence of instructions which tell a computer what to do with the data coming in, and what form the data sent out will have.

Tutor Marked Assessment

1. Write short notes on the five major components of a computer system.
2. What are the main differences between main memory and external memory?

1.2 –Programming/Programming Languages

1.2.1 Programs

Programs are simply instructions in a programming language that tells a computer to perform a task.

Page | 14

Programmers write instructions in various programming languages, some of which are directly understandable by computers and others require intermediate *translation* steps. The act of programmers writing programs is referred to as programming.

1.2.2 Programming Languages

One might be tempted to ask the question “Why is it that we need programming languages”? The answer to this question is in two folds.

- Computers cannot understand English
- English would make a terrible programming language because of the ambiguity.

Programming languages get around both of these problems. They are simple enough to be made sense of by computer programs and they reduce ambiguity. There are very many different programming languages around, you will need to know more than one if you are to be a good programmer.

Although there are currently hundreds of computer languages in use today, the diverse offerings can be divided into three general:

- *Machine Language*: this is the lowest level of programming language. Machine language is the only language understood by the computer. While very easy for computers to understand this, it is next to impossible for humans to understand because it consists of only numbers.
- *Assembly Language*: this is used by programmers instead of machine language to make programming easier. This is so because it contains the same instructions but these instructions and variables have names (called mnemonics) instead of being just numbers.
- *High-Level Language*: this allows the specification of a problem solution in terms closer to those used by human beings. These languages were designed to make programming far easier, less error-prone and to remove the programmer from having to know the details of the internal structure of a particular computer. These high-level languages were much closer to human language. One of the first of these languages was Fortran II which was introduced in about 1958. In Fortran II our program above would be written as:

- $C = A + B$

This is obviously much more readable, quicker to write and less error-prone. As with assembly languages the computer does not understand these high-level languages

directly and hence they have to be processed by passing them through a program called a compiler which translates them into internal machine language before they can be executed.

1.2.3. From Problem to Program

It is necessary to note that programming is not necessarily about mathematics, it rather has a lot to do with organization. The art of taking a problem and breaking it down into a set of instructions that can be given to a computer to solve is the essence of programming.

There are many things you must consider when writing a program; not all of them are directly related to the problem in hand. I am going to start on the basis that you are writing your programs for a customer. He or she has problem and would like you to write a program to solve it. We shall assume that the customer knows even less about computers than we do! Initially we are not even going to talk about the programming language, type of computer or anything like that, we are simply going to make sure that we know what the customer wants. Coming up with a perfect solution to a problem the customer has not got is something which happens surprisingly often in the real world.

This is almost a kind of self-discipline. Programmers pride themselves on their ability to come up with solutions, so as soon as they are given a problem they immediately start thinking of ways to solve it, this almost a reflex action. What you should do is think "Do I really understand what the problem is?". Before you solve a problem you should make sure that you have a watertight definition of what the problem is, which both you and the customer agree on. In the real world this is often called a Functional Design Specification or FDS. This tells you exactly what the customer wants. Both you and the customer sign it, and the bottom line is that if you provide a system which behaves according to the design specification the customer must pay you. Once you have got your design specification, then you can think about ways of solving the problem. You might think that this is not necessary if you are writing a program for yourself; there is no customer to satisfy. This is not true. Writing an FDS forces you to think about your problem at a very detailed level.

Tutor Marked Assessment

1. What are the three main types of computer programming languages?
2. Enumerate the major advantages of using high-level languages rather than internal machine code

Unit 1.3 - Operating Systems

The Operating System of a computer is a large program which manages the overall operation of the computer system. On a simple one-user computer the Operating System will:

Page | 16

- Provide an interface to allow the user to communicate with the computer. This interface may be a text-oriented interface where the user types commands in response to a prompt from the computer or may be a mouse-driven Windows operating system.
- Control the various peripherals e.g. Keyboard, Video Display Unit (VDU), Printer etc. using special programs called Device Drivers.
- Manage the user's files, keeping track of their positions on disk, updating them after user makes changes to them etc. An important facility that the Operating System must supply in this respect is an Editor which allows users to edit their files.
- Provide system facilities, e.g. Compilers to translate from high-level programming languages used by the user to the internal machine language the computer uses.

Because of the disparity in speed between input/output devices (and the human entering data) and the CPU most modern operating systems will allow Multi-tasking to take place. Thus while the Computer is held up waiting for input from one program the operating system will transfer control to another program which can execute until it, in turn, is held up. Multi-tasking may take place in a stand-alone computer (for example using an operating system such as Windows 95 on a PC) and allow the user to simultaneously use several different programs simultaneously. For example a user may be running a large computational task in the background while using a word-processor package to write a report.

It is now common for computers to be linked together in networks. The network may consist of many dumb terminals, Personal Computers and workstations linked together with perhaps several larger, more powerful computers which provide a large amount of computer power and file storage facilities to the network. This allows many people access to computing facilities and access to common data-bases, electronic mail facilities etc. Networks may be local to a building or a small area (Local Area Network (LAN)) or connect individual networks across the country or world (Wide Area Network (WAN)).

A particular form of network operating system is a Timesharing operating system. Many large modern computers are set up to serve many simultaneous users by means of a time-sharing system. Each user has a direct connection to a powerful central computer, normally using a Visual Display Unit (VDU) which has a keyboard (and often a mouse) for user input and a screen for feedback from the computer to the user. There may be several hundred simultaneous users of a large computing system. Computing is Interactive in that the time from a user entering a

command until a response is obtained will typically be a few seconds or less. The Operating System will cycle in turn through each connected terminal and if the terminal is awaiting computation will give it a Time-slice of dedicated CPU time. This process is continuous thus each program receives as many time-slices as it requires until it terminates and is removed from the list of programs awaiting completion.

In a system with multiple users the operating system must also carry out other tasks such as:

- Validating the user's rights to use the system
- Allocating memory and processor time to individual programs
- Maintaining the security of each user's files and program execution

In time-sharing system the processing power is contained in a central machine. Users access this central machine from a non-intelligent terminal that can do no processing itself. The advent of cheap powerful workstations has led to the distribution of computer power around the network. A distributed computer system consists of a central processor with a large amount of disk storage and powerful input/output facilities connected to a network of machines, each with its own main memory and processor.

The central processor (or Server) provides storage for all system files and user files. Each computing node in the network downloads any files and system facilities it requires from the server and then carries out all computation internally. Any changes to files or new files generated have to be sent by the network to the server. To make it easier to find a particular file it is usual to collect all related files into a separate directory. Each user will be allocated a certain amount of space on the external memory, this space will be set up as a single directory called the user's home directory. The user can further split this space into various other directories. For example a lecturer writing a course may well set up a directory to contain all the files relevant to the course. Within this directory it is best to organize the files into groups by setting up various sub-directories, a sub-directory to hold course notes, another to hold tutorials, another to hold laboratory sheets etc. Within one of these directories, say the tutorials directory, will be held the relevant files -- tutorial1, tutorial2 etc. This hierarchical file storage structure is analogous to the storage of related files in a filing system. A filing cabinet could hold everything relevant to the course, each drawer could hold a different sub-division, such as notes, and each folder within the drawer would be a particular lecture.

Space will also be allocated on the server for system files. These also will be allocated to directories to facilitate access by the operating system.

Tutor Marked Assessment

1. Highlight the responsibilities of the Operating System.
2. What are the various types of Operating Systems?

Unit 1.4 - Preparing a Computer Program

There are various steps involved in producing a computer program for a particular application. These steps are independent of which computer or programming language that is used and require the existence of certain facilities upon the computer. The steps are:

1. Study the requirement specification for the application. It is important that the requirements of the application should be well specified. Before starting to design a program for the application it is necessary that the requirement specification is complete and consistent. For example a requirement specification that says 'write a program to solve equations' is obviously incomplete and you would have to ask for more information on 'what type of equations?', 'how many equations?', 'to what accuracy?' etc.
2. Analyze the problem and decide how to solve it. At this stage one has to decide on a method whereby the problem can be solved, such a method of solution is often called an Algorithm.
3. Translate the algorithm produced at the previous step into a suitable high-level language. This written form of the program is often called the **source program** or **source code**. At this stage the program should be read to check that it is reasonable and a desk-check carried out to verify its correctness. A programmer carries out a desk-check by entering a simple set of input values and checking that the correct result is produced by going through the program and executing each instruction themselves. Once satisfied that the program is reasonable it is entered into the computer by using an Editor.
4. Preprocess the program, i.e. the compiler is ordered to compile the program. However, in a C++ program, a preprocessor program executes automatically before the compiler's translation phase begins. The C++ preprocessor obeys commands called preprocessor directives, which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually include other text files to be compiled, and perform various text replacements. ***A list of common Standard Library File Headers are found in Appendix A.***

5. Compile the program into machine-language. The machine language program produced is called the object code. At this stage the compiler may find Syntax errors in the program. A syntax error is a mistake in the grammar of a language, for example C++ requires that each statement should be terminated by a semi-colon. If you miss this semi-colon out then the compiler will signal a syntax error. Before proceeding, any syntax errors are corrected and compilation is repeated until the compiler produces an executable program free from syntax errors.
6. The object code produced by the compiler will then be linked with various function libraries that are provided by the system. This takes place in a program called a **linker** and the linked object code is then loaded into memory by a program called a **loader**.
7. Run the compiled, linked and loaded program with test data. This may show up the existence of Logical errors in the program. Logical errors are errors that are caused by errors in the method of solution, thus while the incorrect statement is syntactically correct it is asking the computer to do something which is incorrect in the context of the application. It may be something as simple as subtracting two numbers instead of adding them. A particular form of logical error that may occur is a run-time error. A run-time error will cause the program to halt during execution because it cannot carry out an instruction. Typical situations which lead to run-time errors are attempting to divide by a quantity which has the value zero or attempting to access data from a non-existent file.

The program must now be re-checked and when the error is found it is corrected using the Editor as in (3) and steps (4) and (5) are repeated until the results are satisfactory.

The program can now be put into general use - though unless the testing was very comprehensive it is possible that at some future date more logical errors may become apparent. It is at this stage that good documentation produced while designing the program and writing the program will be most valuable, especially if there has been a considerable time lapse since the program was written.

End of Module Assessment

1. Describe the steps involved in preparing a Computer program
2. What is the only language that a computer understands directly?

A. English, as spoken in London, UK	B. Machine Language
C. BASIC, the Beginners' All-purpose Symbolic Instruction Code	D. None of these
3. From the point of view of the programmer what are the major advantages of using a high-level language rather than internal machine code or assembler language?

A. Program portability	B. Easy development
C. Efficiency	D. None of these

4. How do the main components of the computer communicate with each other?

A. System Bus

B. Memory

C. Keyboard

D. Monitor

5. _____ is a logical unit of the computer that coordinates the activities of all the other logical units.

6. _____ is a logical unit of the computer that performs calculations.

7. _____ is a logical unit of the computer that makes logical decisions.

8. _____ Languages are most convenient to the programmer for writing programs quickly and easily.

MODULE – 2: DESIGN OF PROGRAMS

GENERAL OBJECTIVES

- To develop programs in a disciplined approach using various logic modelling tools

SPECIFIC OBJECTIVES

- At the end of the module, you should be to:
 - Explain why it is important to use logic modelling tools when designing a program.
 - Describe the advantages and disadvantages of using flow charting and pseudo-code to design a program.
 - Use basic logic modelling tools to represent solutions to a problem

Many new students of programming tend to be in a rush to write programs. This is natural and the availability of Integrated Development Environments (IDEs) like Codeblocks, Bloodshed DevC++, Microsoft Visual C++ amongst many other exacerbates this rush. However, it is pertinent that one has an understanding of the foundation of programming (program design) in order to be an effective and efficient programmer.

As earlier discussed in Module 1 (Unit 1.2.3), one must have a thorough understanding of the problem that is to be solved and a carefully planned approach to solving it. This module discusses logic modelling tools that are needed to appropriately design programs. This leads us to a “Disciplined” Approach to problem solving and programming

Unit 2.1 A “Disciplined” Approach

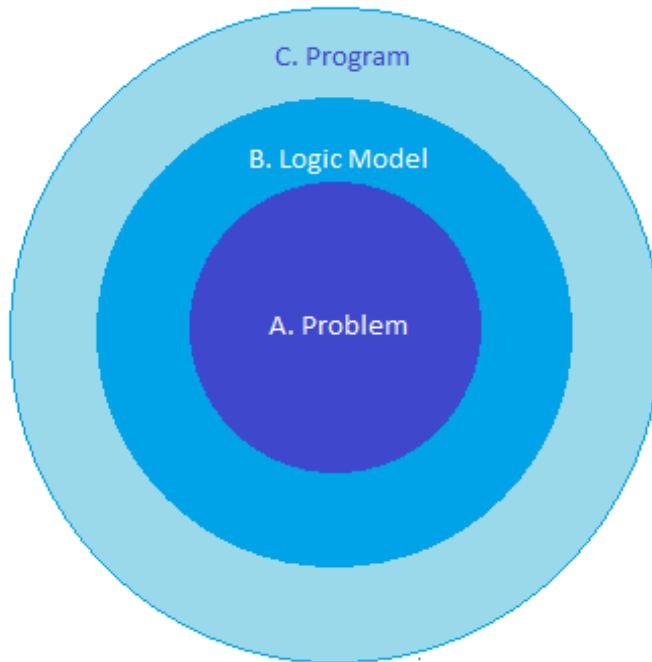


Figure 2.1. A “Disciplined” Approach to programming

This approach starts with a well-defined problem, followed by a proper documentation of the logic model that defines/represents the solution to the problem and lastly, the program which is an implementation of the logic model to solve the problem.

2.2.1 So what is a model?

A model is simply an abstraction of reality. When we model a design we use the requirements to put together a solution. The model allows us to experiment and fine tune our vision of the solution until it's correct. You need a step before programming starts to ensure that no requirement is left unnoticed. Think of your logic model as your blueprint as you design the “perfect” program.

A logic path is simply a sequence of instructions or statements. Sometimes a path of logic statements is also known as an algorithm.

Unit 2.2 Algorithms

An algorithm is a finite set of instructions that must be followed in a specified order so as to solve a problem. It is important that an algorithm must have the following attributes in order to be suitable:

- *Limitedness*: a suitable algorithm must terminate after a finite number of steps. If this attribute is not included, the program will continue in an infinite loop. For example:
 - produce first digit of $1/7$.
 - while there are more digits of $1/7$ do
 - produce next digit.

This algorithm never terminates because $1/7$ cannot be expressed in a finite number of decimal places.

- *Non-ambiguity*: each step of the algorithm must be explicitly defined. This is done because different programming languages may interpret statements differently.
- *Effectiveness*: this means that the operations performed in the algorithm can actually be carried out in a finite time.

Algorithms are essential to the way computers process data. Many computer programs contain algorithms that detail the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paycheques or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system.

Expressing Algorithms

Algorithms can be expressed in many kinds of notation, including natural languages, pseudo code, flowcharts, programming languages or control tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudo code, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms. We will lay more emphasis on the flowchart and pseudo codes.

2.2.1 Flowchart

A **flowchart** is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution to a given problem. Process operations are represented in these boxes, and arrows; rather, they are implied by the sequencing of operations. Flowcharts are used in analysing, designing, documenting or managing a process or program in various fields.

Flowcharts are used in designing and documenting complex processes or programs. Like other types of diagrams, they help visualize what is going on and thereby help the viewer to understand

a process, and perhaps also find flaws, bottlenecks, and other less-obvious features within it. There are many different types of flowcharts, and each type has its own repertoire of boxes and notational conventions. The two most common types of boxes in a flowchart are:

- a processing step, usually called *activity*, and denoted as a rectangular box
- a decision, usually denoted as a diamond.

A flowchart is described as "cross-functional" when the page is divided into different swim lanes describing the control of different organizational units. A symbol appearing in a particular "lane" is within the control of that organizational unit. This technique allows the author to locate the responsibility for performing an action or making a decision correctly, showing the responsibility of each organizational unit for different parts of a single process.

2.2.1.1 Symbols used in a flowchart

Flowcharts are usually drawn using some standard symbols

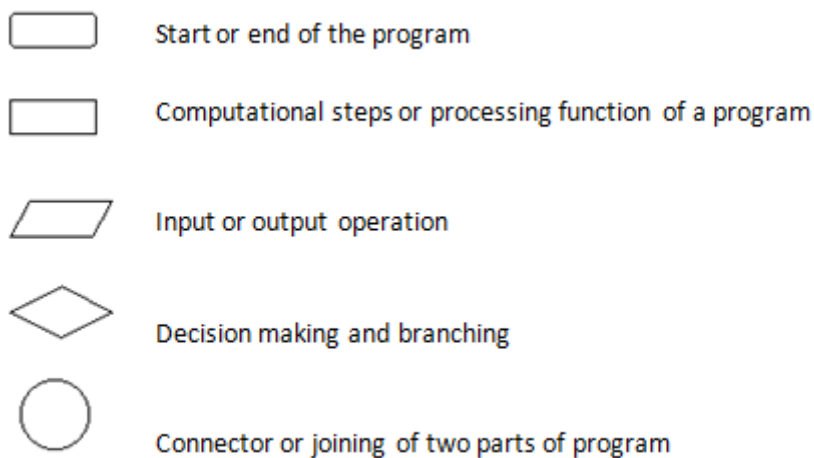
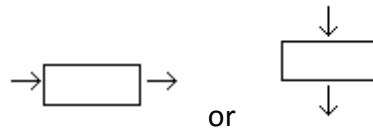


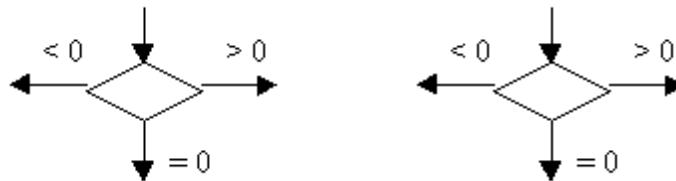
Figure 2.2 Symbols used in a flow chart

2.2.1.2 Guidelines for Flow Charting:

- In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
- The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
- The usual direction of the flow of a procedure or system is from left to right or top to bottom.
- Only one flow line should come out from a process symbol.



- e. Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.



- f. Only one flow line is used in conjunction with terminal symbol.



- h. If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.
- i. Ensure that the flowchart has a logical *start* and *finish*.
- j. It is useful to test the validity of the flowchart by passing through it with a simple test data.

2.2.1.3 Advantages of Flow Charts

1. The greatest advantage that the flowchart has is its simplicity in the sense that it is simple to understand and create.
2. It is less intimidating than pseudo code for the fact that most people (programmers inclusive) tend to prefer concepts that are visually expressed.
3. With pseudo code you are still creating a program and programming syntax should not come into play when designing the model.

NB: These are advantages that flowchart has over pseudo codes.

2.2.1.4 Disadvantages of Flow Charts

1. Drawing a complex program cannot be done without spanning several sheets of paper.

2. The artistic nature of the drawing may be difficult for some programmers.
3. There may be the difficulty in translating flowcharting logic into programming language statements. Pseudo code looks much more like a programming language and can usually be implemented much faster and more efficiently than a flowchart.

Example of Flow Chart

The following is the logic model for calculating the area of a triangle.

The first thing we need to do is to gather all the variables needed to solve the problem. These variables are:

- Base of the triangle – b.
- Height of the triangle – h.

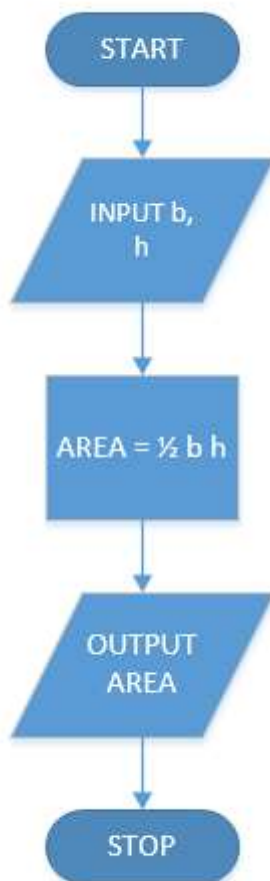


Figure 2.2 Flow chart for solving for area of a triangle

Tutor Marked Assessment

1. Draw a flowchart for finding the average of three numbers.
2. Mathematics department needs a tool that will calculate the roots of a quadratic equation. You are required to draw the logic model/Flow Chart for developing this tool.

2.2.2 Pseudo Code

Pseudo code is a non-standard English-like sentences and phrases that are quite different from any programming language syntax. This is because unlike programming languages which have very strict rules regarding keywords and syntax, there are no standards when it comes to pseudo code. As long as the statement is readable and defines what that logic step needs to accomplish, it is acceptable as pseudo code.

2.2.2.1 Rules for writing Pseudo Code:

- The pseudo code program should begin with the Start instruction and complete with the End instruction.
- Each pseudo code statement should contain at least one instruction.
- Each pseudo code statement should contain a verb that represents the action performed along with any identifier and operators that hold program values or perform calculations.

2.2.2.2 Advantages of Pseudo Code

1. The major advantage that pseudo code has is its closeness to programming language. It's a very simple and unstructured programming language and it is very easy to transition pseudo code to a programming language.

2.2.2.3 Disadvantages of Pseudo Code

1. The greatest disadvantage of pseudo code is its lack of standard. One person's logic instructions may not seem as logical as the next. Given the unstructured nature of pseudo code, it is few rules and is hard to standardize. One programmer might not see the logic written by someone else.
2. The other disadvantage arises as a result of comparing pseudo code to another form of expressing an algorithm (the flow chart to be precise). Pseudo Code doesn't have the ability to show logic flow. Whereas flowcharts provide an overview of logic and can be understood at a higher level, pseudo code is far more detail oriented and requires more concentration and practice to see the bigger picture.

End of Module Assessment

1. Describe the “Disciplined Approach” and explain why it is useful in programming.
2. Juxtapose the use of flow charts and pseudo codes paying attention to that advantages/disadvantages they have over each other.
3. It is said that pseudo code is more “English” like. What does this mean?
4. Name some flow charting rules?
5. Name some pseudo code rules?
6. A procedure for solving a problem in terms of the actions to execute and the order in which these actions execute is called a(an) _____

MODULE – 3: PROGRAMMING IN C++

SPECIFIC OBJECTIVES

- To write a simple, fully commented computer program in C++.

GENERAL OBJECTIVES

- At the end of the module, you will be able to:
 - Understand the basic structure of a C++ program.
 - Understand and use fundamental data types in C++.
 - Understand basic computer memory concepts.
 - Declare and initialize variables.

3.1 A simple program in C++

```
1 //This is my very first programm in C++
2 //I hope you like it
3 #include <iostream>
4 #include <cstdlib>
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "Hello World.\n";
11     cout << "This is first C++ Program.\n";
12     cout << "I'm loving it.";
13     return 0;
14 }
15
```

Figure 3.1 A Simple C++ Program written using CodeBlocks Compiler

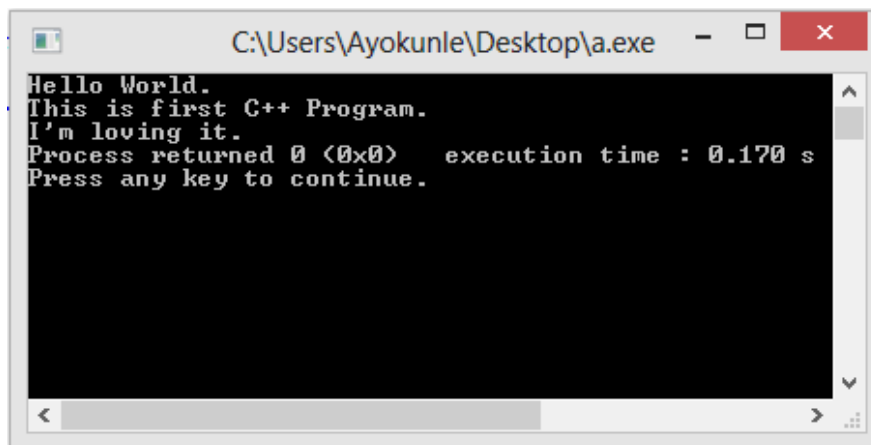


Figure 3.2 The output of the program in figure 3.1 after compilation and execution.

3.2 The Programming Environment

The way to edit and compile a program depends on the compiler you are using depending on whether it has a Development Interface or not and on its version. (CodeBlocks, used for the example above- and all other examples in the study pack- has a development interface and therefore doesn't need a text editor)

Text Editor:

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or Unix.

The files you create with your editor are called source files, and for C++ they typically are named with the extension .cpp, .cp, or .C. NOTE: C is in CAPITAL LETTER.

Before starting your programming, make sure you have one text editor in place and you have enough experience to type your C++ program.

C++ Compiler:

This is actual C++ compiler which will be used to compile your source code into final executable program.

Most C++ compilers don't care what extension you give your source code, but if you don't specify otherwise, many will use .c by default therefore it is Important to specify so that your C++ program do end up being saved as a c program.

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have respective Operating Systems.

NB: ALL EXAMPLES IN THIS STUDY PACK ARE WRITTEN, COMPILED AND EXECUTED USING CODEBLOCKS 10.05 COMPILER.

3.3 C++ Program Structure

The example in figure 3.1 will be used to describe the basic structure of a C++ program.

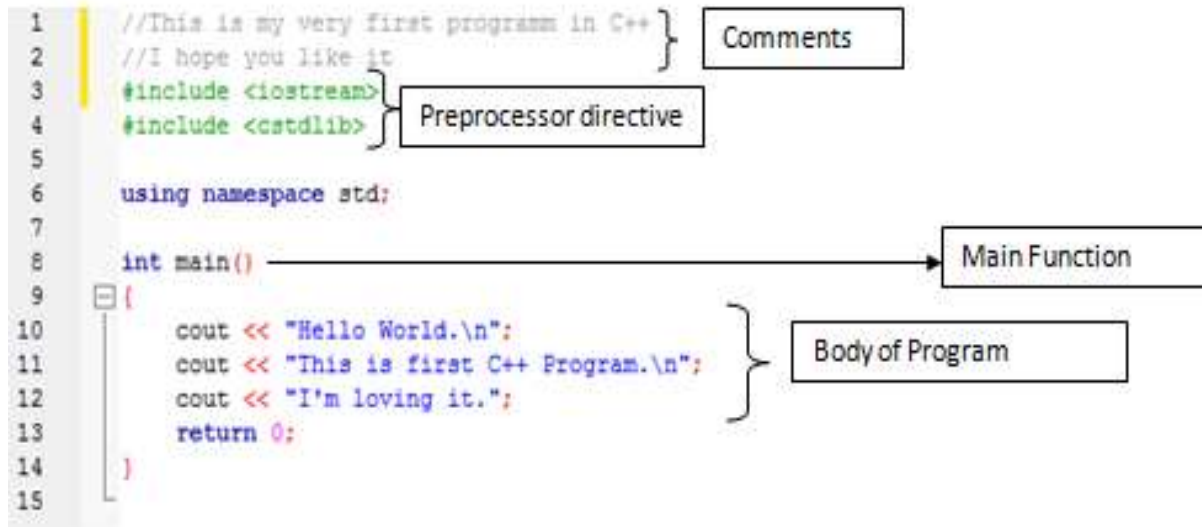


Figure 3.3 The basic structure of a C++ Program

3.3.1 Comments

Comments are messages that are inserted into your programs that make them readable and explain what is going on at that point in your program. The comments in the program (figure 3.3) indicate that this is the programmer's first C++ program and he hopes that you like it. It is not ideal to put C++ codes in your comment because a comment is but a message for people and not the compiler. Anything that goes into the comment will always be ignored by the compiler when compiling your source codes.

There are two different methods of writing a comment in C++:

- Single line comments: this commenting style starts each comment line with two backslashes "//". Refer to lines 1 and 2 of figure 3.3
- Multi-line comments/block comments: this commenting style starts a comment with backslash, asterix and ends the comment with asterix, backslash (/* multi-line comment */). Consider the comments on line 1 to 3 in figure 3.4.

```

1  /*This is my very first programm in C++
2  written on the 28 of June, 2013
3  I hope you like it */
4
5  #include <iostream>
6  #include <cstdlib>
7
8  using namespace std;
9
10 int main()
11 {
12     cout << "Hello World.\n";
13     cout << "This is first C++ Program.\n";
14     cout << "I'm loving it.";
15     return 0;
16 }
17

```

Figure 3.4 A simple program with multi-line comment

3.3.2 Pre-processor directives

The pre-processor directive instructs you C++ compiler to load a file from disk into the middle of the current program. Preprocessor directives are commands that you supply to the preprocessor. All preprocessor directives begin with a pound sign (#). Never put a semicolon at the end of preprocessor directives, because they are preprocessor commands and not C++ commands. Preprocessor directives typically begin in the first column of your source program.

The #include preprocessor directive merges a disk file into your source program. Remember that a preprocessor directive does nothing more than a word processing command does to your program; word processors also are capable of file merging. The format of the #include preprocessor directive follows:

```
#include <filename>
```

In the #include directive, the filename must be an ASCII text file (as your source file must be) located somewhere on a disk.

In this example above, the directive #include <iostream> tells the preprocessor to include the iostream standard header file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

A more comprehensive list of C++ Standard header files and their functions can be seen in Appendix A.

3.3.3 `using namespace std;`

All the elements of the standard C++ library are declared within what is called a **namespace**, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library.

3.3.4 `int main()`

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independent 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 word main is followed in the code by a pair of parentheses (). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces {}. What is contained within these braces is what the function does when it is executed.

3.3.5 `cout`

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect.

`cout` is the only statement that performs an action that generates visible effect in the program in figure 3.3.

`cout` represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

`cout` is declared in the *iostream* standard file within the `std` namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

3.3.6 `return 0;`

The return statement causes the main function to finish. `return` may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is

generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

Tutor Marked Assessment

1. Describe the role of the following in a simple C++ program
 - a. Pre-processor directives
 - b. std
 - c. main ()
 - d. comments

3.4 Variables and Data Types

The program we wrote in figure 3.3 is but a tiny fraction of what can be done with the programming language. We will take a look at a more complex program in order to explain variables and data types.

Let us assume that you are asked to retain the length and width of a rectangle in your head. You have just been asked to memorize two numbers. Now if you are asked to calculate the perimeter and the area of the rectangle using the numbers you have stored in your head, it means then that you have been asked to store four different numbers in your head (length, width, perimeter and area).

```

1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main()
7  {
8      int length, width;
9      int perimeter, area;
10     cout << "Length = ";
11     cin >> length;
12     cout << "Width = ";
13     cin >> width;
14     perimeter = 2*(length+width);
15     area = length*width;
16     cout << "Perimeter is " << perimeter << "\n";
17     cout << "Area is " << area;
18 }
19

```

Figure 3.5 A program to calculate the perimeter and area of a rectangle

A **variable** is the name used for values which are manipulated by a computer program. Line 8 and 9 in the program in figure 3.5 contain variables: length, width, perimeter and area. In order to distinguish between different variables, they must be given identifiers, names which distinguish them from all other variables. This is similar to elementary algebra, when one is taught to write "Let stand α for the acceleration of the body ...". Here is an identifier for the value of the acceleration. The rules of C++ for valid identifiers state that:

- start with a letter
- consist only of letters, the digits 0-9, or the underscore symbol _
- not be a reserved word otherwise known as key words

C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named length is different from Length, which is different from LENGTH.

3.4.1 Variables and Memory Concepts

In the program in figure 3.5, when the statement

`cin >> length;` (in line 11)

is executed, the integer typed in by the user is stored in the memory location tagged *length*. Let us assume that the user has entered 7, 7 is stored in the memory location tagged length.

Let us assume that 5 has been entered by the user in line 13,

`cin >> width;`

the statement also store that integer 5 in the memory location called width.

Going to line 14, the expression $\text{perimeter} = 2 * (\text{length} + \text{width})$ is executed. That gives us $2 * (7 + 5)$ which is 24. 24 is then stored in the memory location called *perimeter*. In line 16, the statement `cout<<"Perimeter is " << perimeter <<"\n";` displays "Perimeter is" and calls the value in the memory location *perimeter*. So from our example,

Perimeter is 24
is displayed on the screen.



Figure 3.6. Memory location showing the name and the value of variable `length` after the execution of the statement in line 11.

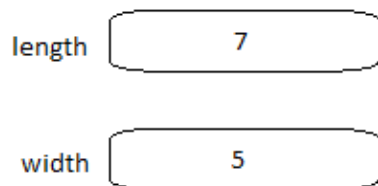


Figure 3.7. Memory location showing the names and the values of the variables `length` and `width` after the execution of the statement in line 13.

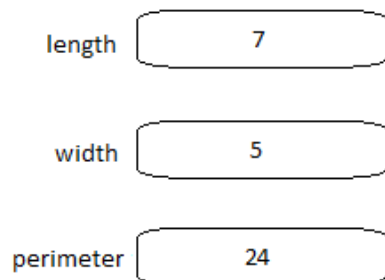


Figure 3.8. Memory location after calculating and storing the perimeter from `length` and `width`.

3.4.2 Variable Scope

In programming, a scope is a region of the program. Variable scope therefore, refers to regions in a program where variables can be declared. There are mainly three regions where variables can be declared in a program.

Page | 37

1. Inside a function – Local variable.

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

2. Outside a function – Global variable.

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the lifetime of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

3. Inside a function parameter.

```
1  #include<iostream>
2  using namespace std;
3
4  char oruko[20];
5
6  int main()
7  {
8      int age;
9      cout <<"Name: ";
10     cin >> oruko;
11     cout << "Age: ";
12     cin >> age;
13     cout << oruko << " is " << age << " years old";
14 }
```

Figure 3.9 Local variable vs. Global variable

```
C:\Users\Ayokunle\Desktop\variablescope.exe
Name: Omotunde
Age: 34
Omotunde is 34 years old
Process returned 0 (0x0)   execution time : 3.767 s
Press any key to continue.
```

Figure 3.10 Output of the program in figure 3.9

The following declares variable with the parameters of a function

```
1  int max(int num1, int num2) ← Declaration of variables
2  {                               within the function parameter
3      // local variable declaration
4      int result;
5
6      if (num1 > num2)
7          result = num1;
8      else
9          result = num2;
10
11     return result;
12 }
13
```

Figure 3.11 Declaration of variables within the function parameter

3.4.3 Keywords

The standard reserved words are:

Table 3.1 A table showing the reserved words/keywords in C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

The following are valid identifiers:

```
length days_in_year DataSet1 Profit95
Int Pressure first_one first_1
```

The following are invalid:

```
days-in-year 1data int first.val throw
```

Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using meaningful identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

At this stage it is worth noting that C++ is case-sensitive. That is lower-case letters are treated as distinct from upper-case letters. Thus the word `main` in a program is quite different from the word `Main` or the word `MAIN`.

3.4.4 Declaration of variables

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Table 3.2 A table showing the various data types in C++

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

In C++ (as in many other programming languages) all the variables that a program is going to use must be declared prior to use. Declaration of a variable serves two purposes:

- It associates a type and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.
- It allows the compiler to decide how much storage space to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

3.4.4.1 int

Variables can represent negative and positive integer values (whole numbers). There is a limit on the size of value that can be represented, which depends on the number of bytes of storage allocated to an int variable by the computer system and compiler being used. On a PC most compilers allocate two bytes for each int which gives a range of -32768 to +32767. On workstations, four bytes are usually allocated, giving a range of -2147483648 to 2147483647. It is important to note that integers are represented exactly in computer memory.

3.4.4.2 float

Variables can represent any real numeric value, which are both whole numbers and numbers that require digits after the decimal point. The accuracy and the range of numbers represented is dependent on the computer system. Usually four bytes are allocated for float variables; this gives an accuracy of about six significant figures and a range of about 10^{-38} to 10^{38} . It is important to note that float values are only represented approximately.

3.4.4.3 bool

Variables can only hold the values true or false. These variables are known as boolean variables in honour of George Boole, an Irish mathematician who invented boolean algebra.

3.4.4.4 char

Variables represent a single character -- a letter, a digit or a punctuation character. They usually occupy one byte, giving 256 different possible characters. The bit patterns for characters usually conform to the American Standard Code for Information Interchange (ASCII).

In the program in figure 3.5, variables length and width are declared as integers in line 8 and perimeter and area are also declared as integer in line 9. Other examples include:

```
int i, j, count;
float sum, product;
char ch;
bool passed_exam;
```

Variables may be initialised at the time of declaration by assigning a value to them as in the following example:

```
int i, j, count = 0;      //this assigns the value 0 to integer variable count
float sum = 0.0, product; // 0.0 is assigned to the real variable sum
char ch = 'A';           //character variable ch is initialized with character A
bool passed_exam = false; //Boolean variable passed_exam is initialized to
false or 0
```

For the variables that are not initialized, the program makes no assumption about their contents.

3.4.5 Modifiers

Situations may arise when the basic data types are not just enough to capture the appropriate values as required by a programmer. In such situations, a modifier is used. A modifier is used to alter the meaning of the base data type so that it more precisely fits that need of various situations. C++ allows the **char**, **int**, and **double** data types to have modifiers preceding them. The following are the data type modifiers available in C++:

- signed
- unsigned
- long
- short

The modifiers **signed**, **unsigned**, **long**, and **short** can be applied to integer base types. In addition, **signed** and **unsigned** can be applied to **char**, and **long** can be applied to **double**.

The modifiers **signed** and **unsigned** can also be used as prefix to **long** or **short** modifiers. For example **unsigned long int**.

C++ allows a shorthand notation for declaring **unsigned**, **short**, or **long** integers. You can simply use the word **unsigned**, **short**, or **long**, without the **int**. The **int** is implied. For example, the following two statements both declare unsigned integer variables.

```
unsigned x;  
unsigned int y;
```

*Refer to the table in table 3.2 to the variation in size for modified data types

Tutor Marked Assessment

1. What data type would properly represent the following variables
 - a. the number of students in a class
 - b. the grade (a letter) attained by a student in the class
 - c. the average mark in a class
 - d. the distance between two points
 - e. the population of a city
 - f. the weight of a postage stamp
 - g. the registration letter of a car
2. Write suitable declaration for variables in question 1. Be sure to choose meaningful identifiers.

Unit 3.5 Constants and the declaration of constants

Often in programming numerical constants are used, e.g. the value of. It is well worthwhile to associate meaningful names with constants. These names can be associated with the appropriate numerical value in a constant declaration. The names given to constants must conform to the rules for the formation of identifiers.

Page | 44

There are two ways in C++ to define constants

1. Using the `const` keyword.
2. Using `#define` pre-processor.

3.5.1 The *const* keyword

The general form of a constant declaration is:

```
const type constant-identifier = value;
```

`type` is the type of the constant, `constant-identifier` is the identifier chosen for the constant, which must be distinct from all identifiers for variables, and `value` is an expression involving only constant quantities that gives the constant its value. It is not possible to declare a constant without giving it an initial value.

Another advantage of using constant declarations is illustrated by the following declaration:

```
const float VatRate = 17.5;
```

This defines a constant `VatRate` to have the value 17.5, however if the Government later changes this rate then instead of having to search through the program for every occurrence of the VAT rate all that needs to be done is to change the value of the constant identifier `VatRate` at the one place in the program. This of course only works if the constant identifier `VatRate` has been used throughout the program and its numeric equivalent has never been used.

Constant definitions are, by convention, usually placed before variable declarations. There is no limit on how many constant declarations can be used in a program. Several constant identifiers of the same type can be declared in the same constant declaration by separating each declaration by a comma. Thus

```
const int days_in_year = 365,  
      days_in_leap_year = 366;
```

```

1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      const float agt = 9.8; //acceleration due to gravity ← Constant declaration
7      const char Newline = '\n';
8
9      float u, v, t; // initial velocity, final velocity, and time
10     cout << "Initial velocity? ";
11     cout << Newline;
12     cin >> u;
13     cout << "Time (in seconds): ";
14     cout << Newline;
15     cin >> t;
16
17     v = u + (agt * t)/2; // final velocity calculation
18
19     cout << "The final velocity of an object with initial velocity " << u << "m/s over a period of " << t << " seconds is " << v << "m/s";
20     return 0;
21 }
22

```

Figure 3.12 Constant using the const keyword

```

C:\Users\Ayokunle\Desktop\constantdeclaration.exe
Initial velocity?
5
Time (in seconds):
23
The final velocity of an object with initial velocity 5m/s over a period of 23 seconds is 117.7m/s
Process returned 0 (0x0) execution time : 999.985 s
Press any key to continue.

```

Figure 3.13 Output of the program in figure 3.9 (using const keyword)

3.5.2 The #define Pre-processor

The general form for using the #define pre-processor is:

```
#define identifier value
```

```

1  #include<iostream>
2  using namespace std;
3
4  #define agt 9.8 //acceleration due to gravity ← Constant definition
5  #define NewLine '\n'
6  int main()
7  {
8      float u, v, t; // initial velocity, final velocity, and time
9      cout << "Initial velocity? ";
10     cout << NewLine;
11     cin >> u;
12     cout << "Time (in seconds): ";
13     cout << NewLine;
14     cin >> t;
15
16     v = u + (agt * t)/2; // final velocity calculation
17
18     cout << "The final velocity of an object with initial velocity " << u << "m/s over a period of " << t << " seconds is " << v << "m/s";
19     return 0;
20 }
21

```

figure 3.14 Constant using #define pre-processor

```

C:\Users\Ayokunle\Desktop\constantdefinition.exe
Initial velocity?
4
Time (in seconds):
34
The final velocity of an object with initial velocity 4m/s over a period of 34 seconds is 170.6m/s
Process returned 0 (0x0)   execution time : 220.131 s
Press any key to continue.

```

Figure 3.15 Output of the program in figure 3.11 (using #define pre-processor)

Tutor Marked Assessment

1. Write a program that reads in the radius of a circle as an integer and prints the circle's diameter, circumference and area. Use the **constant** value 3.14159 for π .
2. Repeat the program in 1 using the **#define** preprocessor.

Unit 3.6 Introduction to String

Variables that can store non-numerical values that are longer than one single character are known as strings.

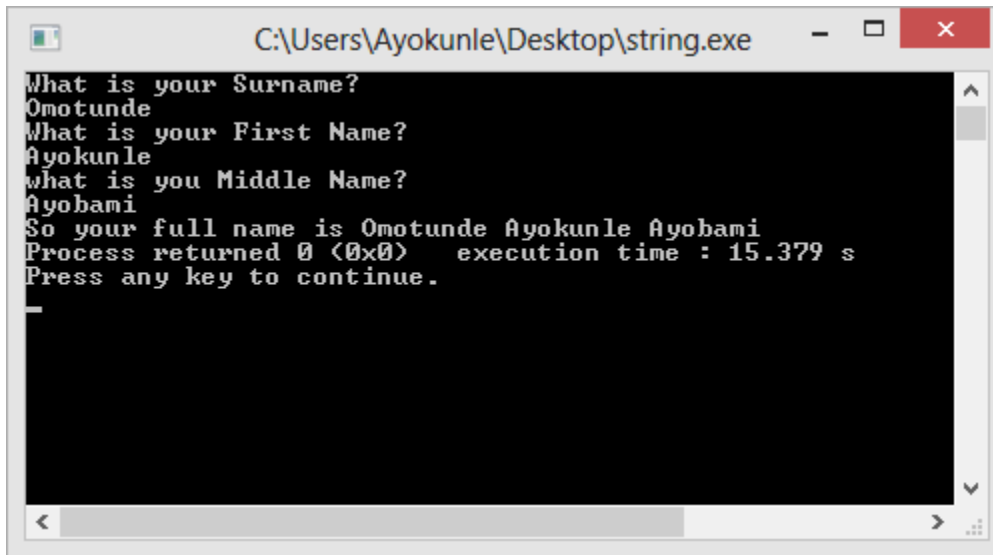
We will take a look at an example of how the string is used in a program.

```
1  /* The following program shows
2  how the string is used*/
3  #include <iostream>
4  #include <cstdlib>
5  #include <string>
6
7  using namespace std;
8
9  int main()
10 {
11     string lname, fname, mname;
12     cout << "What is your Surname? \n";
13     cin >> lname;
14     cout << "What is your First Name? \n";
15     cin >> fname;
16     cout << "What is your Middle Name? \n";
17     cin >> mname;
18
19     cout << "So your full name is " << lname << " " << fname << " " << mname;
20     return 0;
21 }
22
```

Preprocessor directive to import the string file

Declaration of variables lname, fname and mname as string

Figure 3.16 A program showing how the string is used



```
C:\Users\Ayokunle\Desktop\string.exe
What is your Surname?
Omotunde
What is your First Name?
Ayokunle
What is your Middle Name?
Ayobami
So your full name is Omotunde Ayokunle Ayobami
Process returned 0 (0x0)   execution time : 15.379 s
Press any key to continue.
_
```

Figure 3.17 Output of the program in figure 3.6

The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: `<string>` and have access to the `std` namespace (which we already had in all our previous programs thanks to the `using namespace` statement).

3.6.1 The escape Character

The backslash character (`\`) is called the *escape character*. It is used to signal that a special character follows. It is not uncommon to see the character `\` being followed by `n` in the form: `(\n)` which implies the new-line character. It causes the output device to go to the beginning of the next line, similar to a return key on a typewriter. Table 3.2 summarizes these characters

Table 3.3 A table of special characters

Character	Name	Meaning
\b	Backspace	Move the cursor to the left one character
\f	Form feed	Go to top of a new page
\n	New line	Go to the next line
\r	Return	Go to the beginning of the current line
\t	Horizontal Tab	Advance to the next tab stop (eight-column boundary)
\v	Vertical Tab	
\a	Alert	Bell
\'	Apostrophe	The character '
\"	Double quote	The character "
\nnn	The character nnn	The character number nnn (octal)
\NN	The character NN	The character number NN (hexadecimal)

End of Module Assessment

1. What is the output of the following program?

```

1  /* The following program shows
2  how the string is used*/
3  #include <iostream>
4  #include <cstdlib>
5  #include <string>
6
7  using namespace std;
8
9  int main()
10 {
11     cout << "Hello My name is \n";
12     cout << "Omotunde \t Ayokunle \t Ayobami\n";
13     cout << "\"Welcome to C++ \"" << "I hope you like it";
14     return 0;
15 }
16

```

- Every C++ program begins execution at the function _____.
- A(n) _____ begins the body of every function and a(n) _____ ends the body.
- Every C++ statement ends with a(n) _____.
- The escape sequence \n represents the character _____, which causes the cursor to position to the beginning of the next line on the screen.
- The _____ statement is used to make decisions.
- State whether each of the following is true or false. If false, explain why.
 - Comments cause the computer to print the text after the // on the screen when the program is executed.

- b. The escape sequence `\n`, when output with `cout` and the stream insertion operator, causes the cursor to position to the beginning of the next line on the screen.
- c. All variables must be declared before they're used.
- d. All variables must be given a type when they're declared.
- e. C++ considers the variables `number` and `Number` to be identical.
- f. Declarations can appear almost anywhere in the body of a C++ function.
- g. The modulus operator (`%`) can be used only with integer operands.

MODULE – 4: OPERATORS AND ASSIGNMENT STATEMENTS

GENERAL OBJECTIVES

- To write expression and perform operations on variables in C++

Page | 51

SPECIFIC OBJECTIVES

- At the end of the module, you will be able to:
 - Perform basic assignment operation in C++
 - Use relational operators to compare values

Now that we are familiar with variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.'

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides following type of operators:

- Assignment Operator
- Arithmetic Operators
- Relational Operators
- Logical Operators

4.1 Assignment Operator

The basic features in a C++ program are:

1. Statements
2. Expressions
3. Values

A **statement** is a building block of a program, it is the smallest unit that performs an action. For example,

```
int myAge;
```

is a statement that tells the compiler that `myAge` is a variable that should be stored in the memory of the computer as an integer.

An **expression** is a statement that has value. The statement `myAge` may be extended to form an expression by adding a value to it.

```
int myAge = 40;
```

The value 40 has been assigned to the variable `myAge` which is stored in memory as an integer. The assignment operator (`=`) assigns a value to a variable.

This statement assigns the integer value 40 to the variable `myAge`. The part at the left of the assignment operator (`=`) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The *lvalue* has to be a variable whereas the *rvalue* can be either a constant, a variable, the result of an operation or any combination of these.

The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way. This means that the value of whatever it is you have on the right side of the operator is assigned to the variable on the left side of the operator.

Right `myAge = 40; area = pi * radius * radius;`

Wrong `40 = myAge; pi * radius * radius = pi;`

```

1  /* The following program shows
2  how the assignment operator is used*/
3  #include <iostream>
4  #include <cstdlib>
5  #include <string>
6
7  using namespace std;
8
9  int main()
10 {
11     int length = 4;           // 4 is assigned to the integer variable length
12     int width = 5;           // 5 is assigned to the integer variable width
13     int perimeter = 2 * (length + width);
14     int area = length * width;
15     cout << "Perimeter = " << perimeter << endl;
16     cout << "Area = " << area << endl;
17     return 0;
18 }
19

```

Figure 4.1 A program showing how the assignment operator is used.

At line 11, the statement assigns 4 to the integer variable `length` and at line 12, the statement assigns 5 to the integer variable `width`.

There is a special property of C++ that allows assignment operation to be used as the *rvalue* or part of an *rvalue*. An example is the following line of code:

```
a = 2 + (b = 5);
```

this is the same as:

```
b = 5;  
a = 2 + b;
```

Another example is this: `a = b = c = 5;`

This implies that 5 is being assigned to all three variables: a, b, and c.

There are many more assignment operators supported by C++ language. The following is a table of those operators.

Table 4.1 A table showing the assignment operators accepted by C++ language

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	<code>C = A + B</code> will assign value of <code>A + B</code> into <code>C</code>
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	<code>C += A</code> is equivalent to <code>C = C + A</code>
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	<code>C -= A</code> is equivalent to <code>C = C - A</code>
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	<code>C *= A</code> is equivalent to <code>C = C * A</code>
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	<code>C /= A</code> is equivalent to <code>C = C / A</code>
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	<code>C %= A</code> is equivalent to <code>C = C % A</code>
<<=	Left shift AND assignment operator	<code>C <<= 2</code> is same as <code>C = C << 2</code>
>>=	Right shift AND assignment operator	<code>C >>= 2</code> is same as <code>C = C >> 2</code>
&=	Bitwise AND assignment operator	<code>C &= 2</code> is same as <code>C = C & 2</code>
^=	bitwise exclusive OR and assignment operator	<code>C ^= 2</code> is same as <code>C = C ^ 2</code>
=	bitwise inclusive OR and assignment operator	<code>C = 2</code> is same as <code>C = C 2</code>

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  main()
6  {
7      int a = 21;
8      int b = 10;
9      int c ;
10     c = a;
11     cout << "Line 1:  = Operator, Value of c = : " <<c<< endl ;
12
13     c += a;
14     cout << "Line 2:  += Operator, Value of c = : " <<c<< endl ;
15
16     c -= a;
17     cout << "Line 3:  -= Operator, Value of c = : " <<c<< endl ;
18
19     c *= a;
20     cout << "Line 4:  *= Operator, Value of c = : " <<c<< endl ;
21
22     c /= a;
23     cout << "Line 5:  /= Operator, Value of c = : " <<c<< endl ;
24
25     c = 200;
26     c %= a;
27     cout << "Line 6:  %= Operator, Value of c = : " <<c<< endl ;
28
29     c <<= 2;
30     cout << "Line 7:  <<= Operator, Value of c = : " <<c<< endl ;
31
32     c >>= 2;
33     cout << "Line 8:  >>= Operator, Value of c = : " <<c<< endl ;
34
35     c &= 2;
36     cout << "Line 9:  &= Operator, Value of c = : " <<c<< endl ;
37
38     c ^= 2;
39     cout << "Line 10: ^= Operator, Value of c = : " <<c<< endl ;
40
41     c |= 2;
42     cout << "Line 11: |= Operator, Value of c = : " <<c<< endl ;
43
44     return 0;
45 }

```

Figure4.2 A Program showing all assignment operators

```

C:\Users\Ayokunle\Desktop\assignment.exe
Line 1: = Operator, Value of c = : 21
Line 2: += Operator, Value of c = : 42
Line 3: -= Operator, Value of c = : 21
Line 4: *= Operator, Value of c = : 441
Line 5: /= Operator, Value of c = : 21
Line 6: %= Operator, Value of c = : 11
Line 7: <<= Operator, Value of c = : 44
Line 8: >>= Operator, Value of c = : 11
Line 9: &= Operator, Value of c = : 2
Line 10: ^= Operator, Value of c = : 0
Line 11: != Operator, Value of c = : 2

Process returned 0 (0x0) execution time : 0.055 s
Press any key to continue.

```

Figure 4.3 The result of the program in figure 3.8

Tutor Marked Assessment

1. What is the output of the following program?

```

#include <iostream>
using namespace std;
int main ()
{
    int a, b;
    a = 10;
    b = 4;
    a = b;
    b = 7;
    cout << "a:";
    cout << a;
    cout << " b:";
    cout << b;
    return 0;
}

```

4.2 Arithmetic Operators

There are seven arithmetic operators supported by C++ language.

Table 4.2 A table of acceptable arithmetic operators in C++

Operator	Description
+	Add two operands
-	Subtracts second operand from the first

*	Multiply both operands
/	Divide numerator by de-numerator
%	Modulus Operator and remainder of after an integer division
++	Increment operator, increases integer value by one
--	Decrement operator, decreases integer value by one

Let us assume that there are two variable num1 and num2. num1 holds 21 and num2 holds 10. The following programs perform all the supported arithmetic operators on the variables.

4.2.1 Increment ++/Decrement - - Operators

The increment operator adds 1 to its operand while the decrement operator subtracts 1 from its operand.

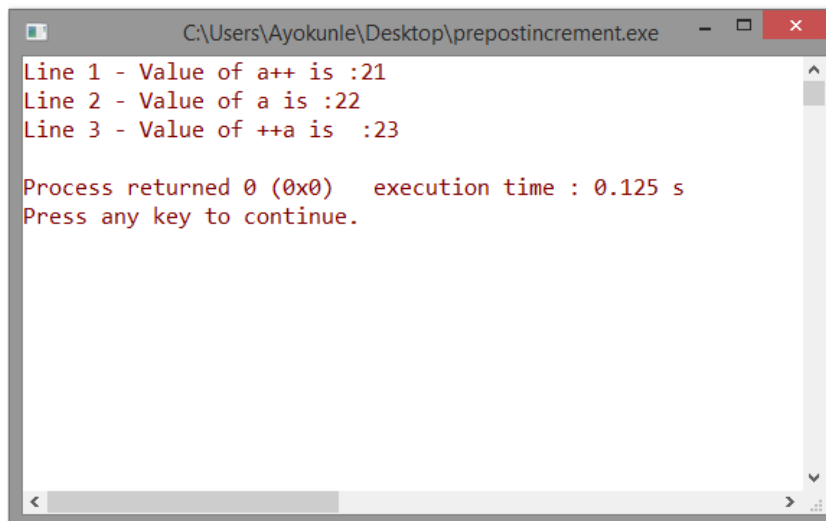
count = count +1; is the same as count++;	Post Increment
count = count +1; is also the same as ++count	Pre Increment
count = count -1; is the same as count - -;	Post Decrement
count = count -1; is also the same as - - count;	Pre Decrement

When an increment or decrement is used as part of an expression, there is an important difference in prefix and postfix forms. If you are using prefix form then increment or decrement will be done before rest of the expression, and if you are using postfix form, then increment or decrement will be done after the complete expression is evaluated.

The example in figure 4.3 explains the difference.


```
1  #include <iostream>
2  using namespace std;
3
4  main()
5  {
6      int a = 21;
7      int c;
8
9      // Value of a will not be increased before assignment.
10     c = a++; //post increment
11     cout << "Line 1 - Value of a++ is :" << c << endl ;
12
13     // After expression value of a is increased
14     cout << "Line 2 - Value of a is :" << a << endl ;
15
16     // Value of a will be increased before assignment.
17     c = ++a; // pre increment
18     cout << "Line 3 - Value of ++a is  :" << c << endl ;
19     return 0;
20 }
21
```

Figure 4.3 Program showing the difference between pre and post increment



```
C:\Users\Ayokunle\Desktop\prepostincrement.exe
Line 1 - Value of a++ is :21
Line 2 - Value of a is :22
Line 3 - Value of ++a is :23

Process returned 0 (0x0)   execution time : 0.125 s
Press any key to continue.
```

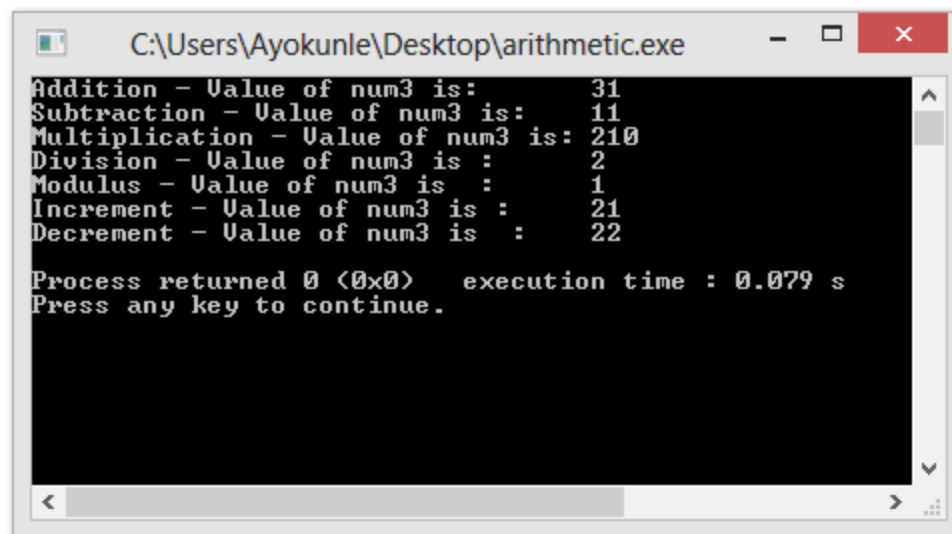
Figure 4.4 Output of the program in figure 4.3 (pre and post increment)

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  main()
6  {
7      int num1 = 21;
8      int num2 = 10;
9      int num3 ;
10
11     num3 = num1 + num2;
12     cout << "Addition - Value of num3 is:      " << num3 << endl ;
13     num3 = num1 - num2;
14     cout << "Subtraction - Value of num3 is:    " << num3 << endl ;
15     num3 = num1 * num2;
16     cout << "Multiplication - Value of num3 is: " << num3 << endl ;
17     num3 = num1 / num2;
18     cout << "Division - Value of num3 is :      " << num3 << endl ;
19     num3 = num1 % num2;
20     cout << "Modulus - Value of num3 is :       " << num3 << endl ;
21     num3 = num1++;
22     cout << "Increment - Value of num3 is :     " << num3 << endl ;
23     num3 = num1--;
24     cout << "Decrement - Value of num3 is :    " << num3 << endl ;
25     return 0;
26 }
27

```

Figure 4.5 A program showing how to use the arithmetic operators in C++



```

C:\Users\Ayokunle\Desktop\arithmetic.exe
Addition - Value of num3 is:      31
Subtraction - Value of num3 is:   11
Multiplication - Value of num3 is: 210
Division - Value of num3 is :     2
Modulus - Value of num3 is :      1
Increment - Value of num3 is :    21
Decrement - Value of num3 is :    22

Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.

```

Figure 4.6 Result of the arithmetic program written in figure 3.8

4.2.2 Priority of Arithmetic Operators

Questions may arise concerning the order of evaluation of arithmetic operators. For example,

$$a + b * c$$

Which should be evaluated first?

$$(a + b) * c \text{ or}$$

$$a + (b * c)?$$

Page | 59

C++ solves this problem by assigning priorities to operators, operators with high priority are then evaluated before operators with low priority. Operators with equal priority are evaluated in left to right order. The priorities of the operators seen so far are, in high to low priority order:

()

/ %

+ -

=

Thus $a + b * c$ is evaluated as if it had been written as: $a + (b * c)$.

Tutor Marked Assessment

1. $10/3 + 5 - 2 + 6\%3 =$ _____
2. Writing $c /= 7$ is shorthand for writing _____
3. The operator for and (as in $a \text{ and } b$) is _____
4. The operator for or (as in $a \text{ or } b$) is _____
5. The operator that is used to see if the contents of the memory location designated by the variable b is the same as the contents of the memory location designated by the variable a is _____

4.3 Relational Operators

These are used to check the relationship between variables. There are six relational operators supported by C++ language.

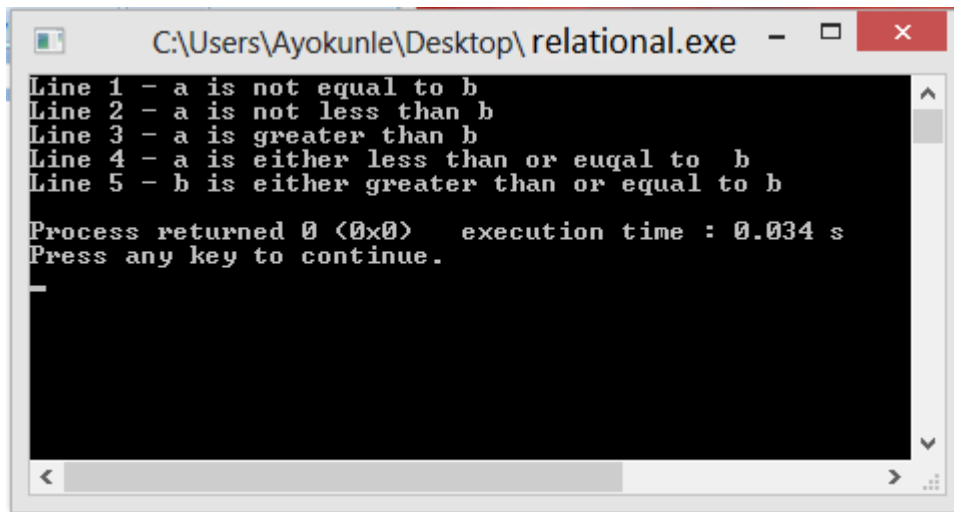
Table 4.3 A table showing the acceptable relational operators

Operator	Description
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

```
1  #include <iostream>
2  using namespace std;
3
4  main()
5  {
6      int a = 21;
7      int b = 10;
8      int c ;
9
10     if( a == b )
11     {
12         cout << "Line 1 - a is equal to b" << endl ;
13     }
14     else
15     {
16         cout << "Line 1 - a is not equal to b" << endl ;
17     }
18     if ( a < b )
19     {
20         cout << "Line 2 - a is less than b" << endl ;
21     }
22     else
23     {
24         cout << "Line 2 - a is not less than b" << endl ;
25     }
26     if ( a > b )
27     {
28         cout << "Line 3 - a is greater than b" << endl ;
29     }
30     else
31     {
32         cout << "Line 3 - a is not greater than b" << endl ;
33     }
34     /* Let's change the values of a and b */
35     a = 5;
36     b = 20;
37     if ( a <= b )
38     {
39         cout << "Line 4 - a is either less than or eugal to b" << endl ;
40     }
41     if ( b >= a )
42     {
43         cout << "Line 5 - b is either greater than or equal to b" << endl ;
44     }
45     return 0;
46 }
47
```

Figure 4.7 A program showing how to use the relational operators in C++



```

C:\Users\Ayokunle\Desktop\ relational.exe
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or euqal to b
Line 5 - b is either greater than or equal to b

Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.

```

Figure 4.8 The result of the relational operator program in figure 4.6

4.4 Logical Operators

The following are the logical operators acceptable by C++ language

Table 4.4 A table showing the acceptable logical operators

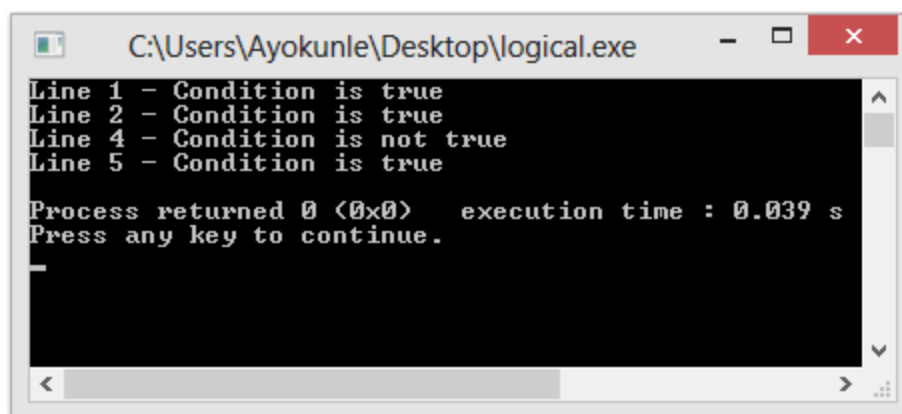
Operator	Description
&&	Called Logical AND operator. If both the operands are non-zero then condition becomes true.
	Called Logical OR Operator. If any of the two operands is non-zero then condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false

```

1  #include <iostream>
2  using namespace std;
3
4  main()
5  {
6      int a = 5;
7      int b = 20;
8      int c ;
9
10     if ( a && b )
11     {
12         cout << "Line 1 - Condition is true"<< endl ;
13     }
14     if ( a || b )
15     {
16         cout << "Line 2 - Condition is true"<< endl ;
17     }
18     /* Let's change the values of a and b */
19     a = 0;
20     b = 10;
21     if ( a && b )
22     {
23         cout << "Line 3 - Condition is true"<< endl ;
24     }
25     else
26     {
27         cout << "Line 4 - Condition is not true"<< endl ;
28     }
29     if ( !(a && b) )
30     {
31         cout << "Line 5 - Condition is true"<< endl ;
32     }
33     return 0;
34 }

```

Figure 4.9 A program showing how to use the logical operators in C++



```

C:\Users\Ayokunle\Desktop\logical.exe
Line 1 - Condition is true
Line 2 - Condition is true
Line 4 - Condition is not true
Line 5 - Condition is true

Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.

```

Figure 4.10 The result of the logical operator program in figure 4.8

Figure 4.11 shows a gives a snapshot of common operators that are acceptable in C++

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
<pre> a = b a += b a -= b a *= b a /= b a %= b a &= b a = b a ^= b a <<= b a >>= b </pre>	<pre> ++a --a a++ a-- </pre>	<pre> +a -a a + b a - b a * b a / b a % b ~a a & b a b a ^ b a << b a >> b </pre>	<pre> !a a && b a b </pre>	<pre> a == b a != b a < b a > b a <= b a >= b </pre>	<pre> a[b] *a &a a->b a.b a->*b a.*b </pre>	<pre> a(...) a, b (type) a ? : </pre>

Figure 4.11 Common operators in C++

End of Module Assessment

1. Determine the output of the following codes

```

#include<iostream>
using namespace std;
int main()
{
    int y = 4;
    int z = y + (y = 10);
    cout << z;
    return 0;
}

```

- | | |
|-------|-------------|
| A. 14 | B. 20 |
| C. 8 | D. 14 or 20 |

2. The syntax: **#define <identifier> <value>** is the for what type of constant

- | | |
|----------------------|---------------------|
| A. Declared Constant | B. Defined Constant |
| C. None of these | |

3. Identify and correct the errors in the following statements

```

a. if ( c < 7 );
    cout << "c is less than 7\n";

```



```
b. if ( c >= 7 )  
    cout << "c is equal to or greater than 7\n";
```

4. Given the algebraic equation $y = ax^3 + 7$, write a simple C++ expression for the equation.
5. Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers.
6. Assume that a video store employee works 50 hours. He is paid \$4.50 for the first 40 hours, time-and-a-half (1.5 times the regular pay rate) for the first five hours over 40, and double-time pay for all hours over 45. Assuming a 28 percent tax rate, write a program that prints his gross pay, taxes, and net pay to the screen. Label each amount with appropriate titles (using string literals) and add appropriate comments in the program.

MODULE – 5: CONDITIONAL STRUCTURE/DECISION MAKING STRUCTURE IN C++

GENERAL OBJECTIVES

Page | 66

- To better understand how decisions are made within a program

SPECIFIC OBJECTIVES

- At the end of this module, you will be able to:
 - Use if, if...else, and if...else if selection statements to choose among alternative choices.

Conditional Structures/Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

5.1 If Statement

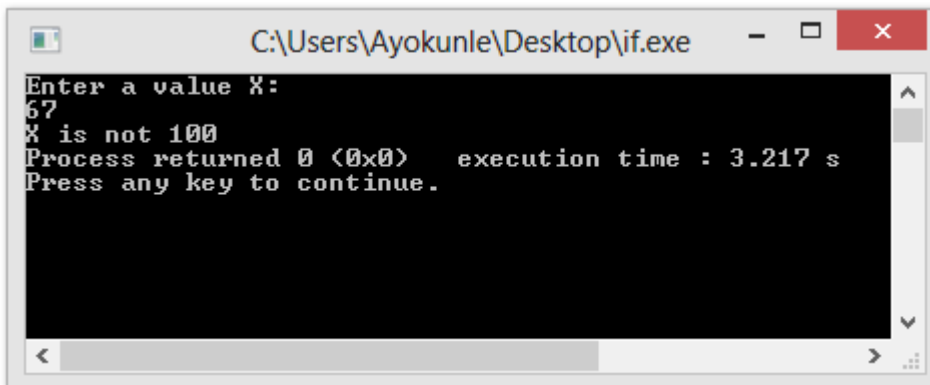
This is the simplest form of the conditional structure. The **if** keyword is used to execute a statement or block only if a condition is fulfilled. It is in the form:

```
if (condition)
{
    statement
}
```

Where `condition` is the expression that is being evaluated. If this condition is true, `statement` is executed. If it is false, `statement` is ignored (not executed) and the program continues right after this conditional structure.

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5  int main()
6  {
7      int x;
8      cout << "Enter a value X: " << endl;
9      cin >> x;
10     //displays X is 100 if the value entered is 100
11     if (x == 100)
12     {
13         cout << "X is: ";
14         cout << x;
15     }
16     //displays X is not 100 if the value entered is not 100
17     if (x != 100)
18     {
19         cout << "X is not 100 ";
20     }
21     return 0;
22 }
23
```

Figure 5.1 A program that implements the if statement



```
C:\Users\Ayokunle\Desktop\if.exe
Enter a value X:
67
X is not 100
Process returned 0 (0x0)   execution time : 3.217 s
Press any key to continue.
```

Figure 5.2 Output if the program in 5.1

Tutor Marked Assessment

1. Write a C++ program that accepts score from user and displays "You passed" if score is above 50.

5.2 If...else statement

This is a more complex form of if statement. An **if** statement is followed by else statement which is executed when the Boolean expression is false. The structure of the if-else statement is as follows:

```
If (condition)
{
    Action1
}
else
{
    Action2
}
```

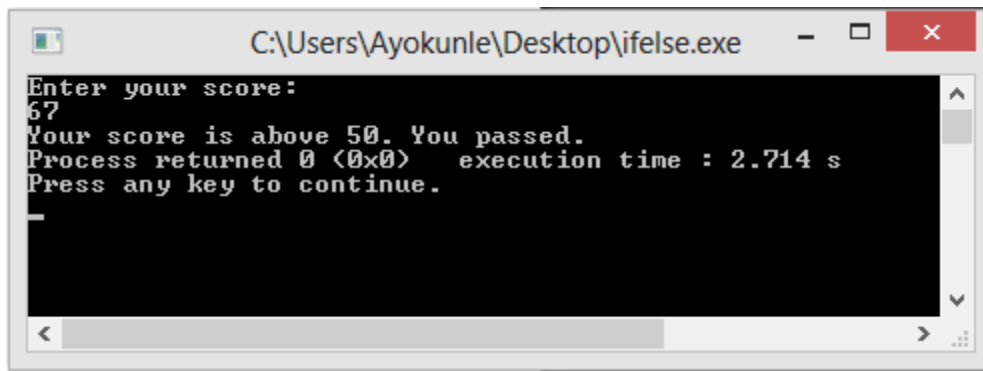
This implies that `action1` is executed when the condition is true while `action2` is executed when the condition is false.

The following is an implementation of the if-else statement

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5  int main()
6  {
7      int score;
8      cout << "Enter your score: " << endl;
9      cin >> score;
10
11     if (score > 50)
12     {
13         //if score is above 50 it outputs You passed
14         cout << "Your score is above 50. You passed.";
15     }
16     else
17     {
18         //if score is below 50 it outputs You failed
19         cout << "Your score is below 50. You failed.";
20     }
21     return 0;
22 }
23
```

Figure 5.3 A program that implements the if-else statement

The following is the output of the program



```
Enter your score:
67
Your score is above 50. You passed.
Process returned 0 (0x0) execution time : 2.714 s
Press any key to continue.
```

Figure 5.4 Output of the if-else program in figure 5.3

Tutor Marked Assessment

1. Write a C++ program that accepts an integer from a user, displays “This is a positive number” when the entered number is positive and displays “This is a negative number” when the entered number is negative.

5.3 If...else if statement

This is a more complex form of if...else statement. While if statement test a single condition and if...else statement test for two conditions, if...else if statement tests various conditions. The general format for the if-else if statement is as follows:

```
If (condition)
{
    Action statement
}
else if (condition)
{
    Action statement
}...
```

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5  int main()
6  {
7      int score;
8      cout << "Enter your score: " << endl;
9      cin >> score;
10
11     if (score >= 80 && score <= 100)
12     {
13         cout << "Grade is A";
14     }
15     else if (score >= 60 && score <= 79)
16     {
17         cout << "Grade is B";
18     }
19     else if (score >= 50 && score <= 59)
20     {
21         cout << "Grade is C";
22     }
23     else if (score >= 45 && score <= 49)
24     {
25         cout << "Grade is D";
26     }
27     else if (score >= 40 && score <= 44)
28     {
29         cout << "Grade is E";
30     }
31     else if (score < 40)
32     {
33         cout << "Grade is F";
34     }
35     return 0;
36 }
37
```

Figure 5.3 Implementation of the if-else statement

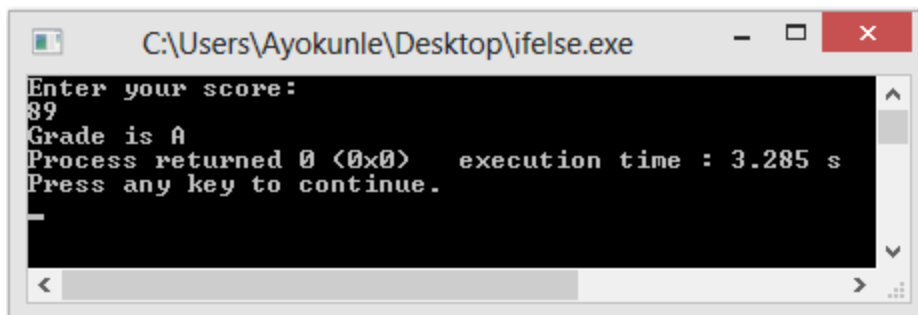


Figure 5.4 Output of the if-else program in figure 5.3

Tutor Marked Assessment

1. Write a program that displays a message depending on the temperature entered by user.

Temperature range (° F)	Message
0 – 30	It is cool!
31 – 60	It is warm!
61 – 90	It is hot!
Above 90	It is scorching!

5.4 SWITCH STATEMENT

5.4.1 Switch Statement

The switch statement is sometimes called the *multiple-choice statement*. The switch statement enables your program to choose from several alternatives. The format of the switch statement is a little longer than the format of other statements you have seen. Here is the switch statement:

```
switch (expression)
{
    case (expression1): { one or more C++ statements; }
    case (expression2): { one or more C++ statements; }
    case (expression3): { one or more C++ statements; }
    .
    .
    .
    default: { one or more C++ statements; }
}
```

The expression can be an integer expression, a character, a literal, or a variable. The *sub* expressions (expression1, expression2, and so on) can be any other integer expression, character, literal, or variable. The number of case expressions following the switch line is determined by your application. The one or more C++ statements is any block of C++ code. If the block is only one statement long, you do not need the braces, but they are recommended.

The default line is optional; most (but not all) switch statements include the default. The default line does not have to be the last line of the switch body.

If expression matches expression1, the statements to the right of expression1 execute. If expression matches expression2, the statements to the right of expression2 execute. If none of the expressions match the switch expression, the default case block executes. The case expression does not need parentheses, but the parentheses sometimes make the value easier to find. Using the switch statement is easier than its format might lead you to believe. Anywhere an if-else-if combination of statements can go, you can usually put a clearer switch statement. This

is much easier to follow than an if-in-an-if-in-an-if statement, as you have had to write previously. However, if and else-if combinations of statements are not difficult to follow. When the relational test that determines the choice is complex and contains many && and || operators, if statement might be a better candidate. The switch statement is preferred whenever multiple-choice possibilities are based on a single literal, variable, or expression.

The syntax for a switch statement in C++ is as follows:

```
switch(expression)
{
    case constant-expression :
        statement(s);
        break; //optional
    case constant-expression :
        statement(s);
        break; //optional

    // you can have any number of case statements.
    default : //Optional
        statement(s);
}
```

The following rules apply to a switch statement:

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram of the switch statement

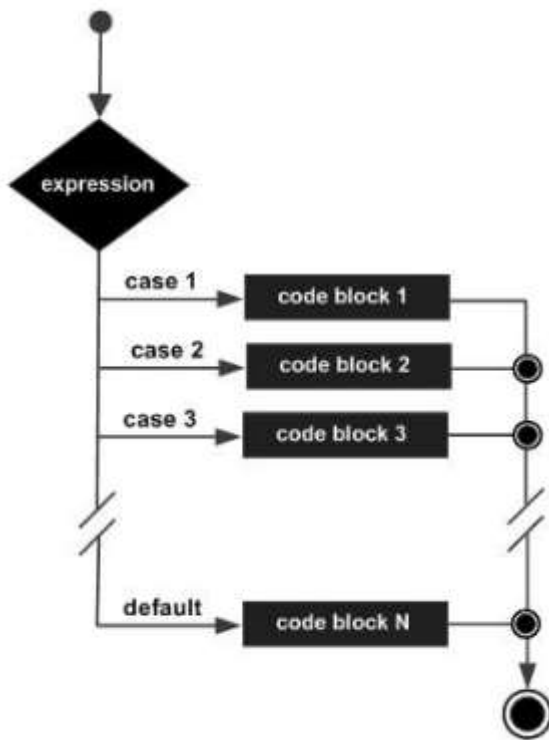


Figure 5.5 The flow diagram for the switch statement

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6      // local variable declaration:
7      char grade;
8      cout << "Enter Grade: \n";
9      cin >> grade;
10     switch(grade)
11     {
12     case 'A':
13         cout << "Excellent!" << endl;
14         break;
15     case 'B' :
16         cout << "Good" << endl;
17         break;
18     case 'C' :
19         cout << "Well done" << endl;
20         break;
21     case 'D' :
22         cout << "You passed" << endl;
23         break;
24     case 'F' :
25         cout << "Better try again" << endl;
26         break;
27     default :
28         cout << "Invalid grade" << endl;
29     }
30     cout << "Your grade is " << grade << endl;
31
32     return 0;
33 }
34
```

Figure 5.6 A program the implements the switch statement

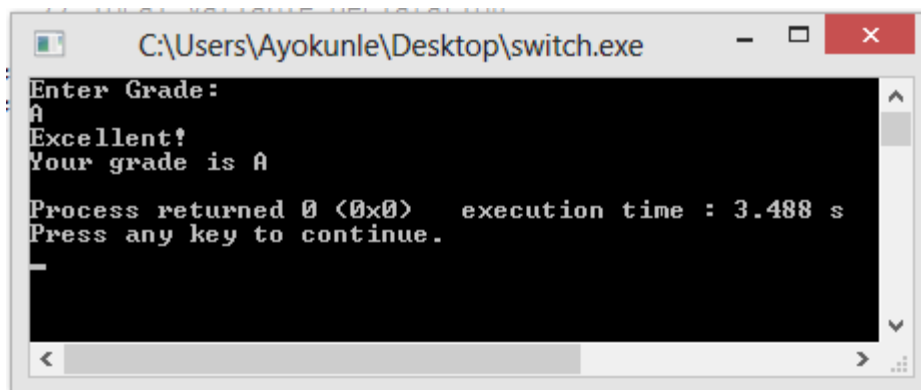


Figure 5.7 The result of the switch statement program in figure 5.6

5.4.2 Nested switch statement

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

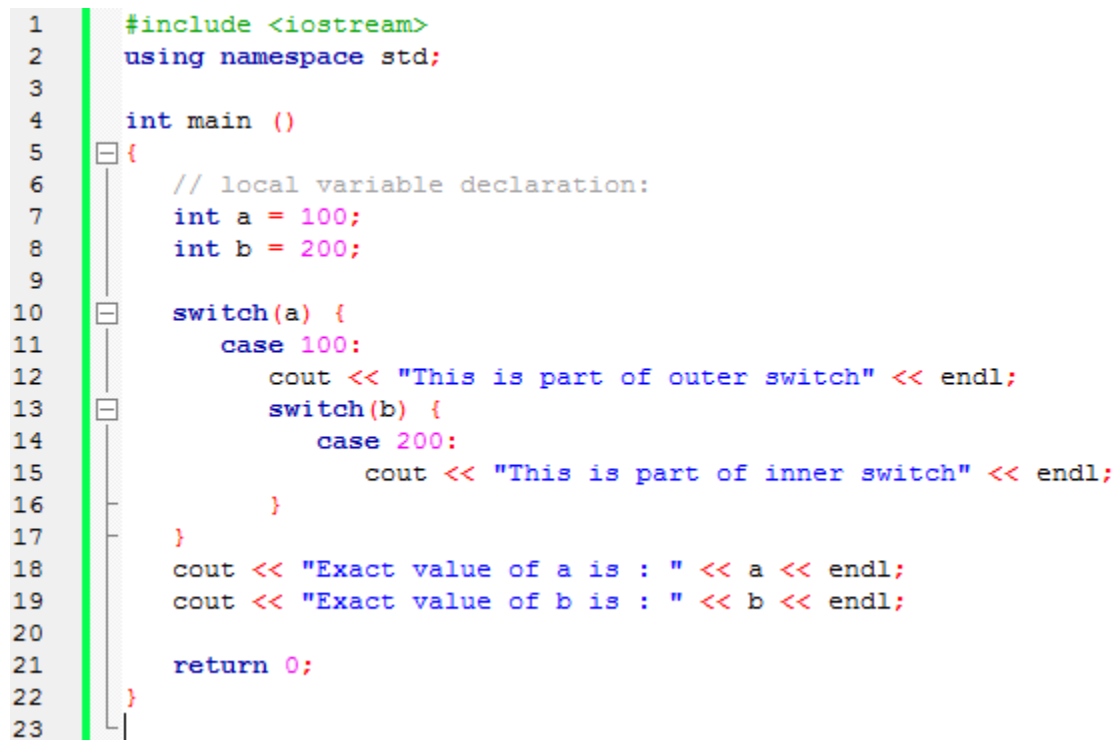
C++ specifies that at least 256 levels of nesting be allowed for switch statements.

Page | 75

The syntax for a nested switch statement is as follows:

```
switch(ch1)
{
    case 'A':
        cout << "This A is part of outer switch";
        switch(ch2) {
            case 'A':
                cout << "This A is part of inner switch";
                break;
            case 'B': // ...
        }
        break;
    case 'B': // ...
}
```

The following is an example of a nested switch statement



```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6      // local variable declaration:
7      int a = 100;
8      int b = 200;
9
10     switch(a) {
11         case 100:
12             cout << "This is part of outer switch" << endl;
13             switch(b) {
14                 case 200:
15                     cout << "This is part of inner switch" << endl;
16             }
17         }
18     cout << "Exact value of a is : " << a << endl;
19     cout << "Exact value of b is : " << b << endl;
20
21     return 0;
22 }
23
```

Figure 5.8 An implementation of the nested switch statement

```

C:\Users\Ayokunle\Desktop\nestedswtich.exe
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.

```

Figure 5.9 Output of the nested switch program in figure 5.8

End of Module Assessment

1. Write a C++ program that does the following: (You are to write **two** versions of this program, one using **if statement** and the other using **switch statement**).

- a. Displays the following menu

MENU	
Blood Group	
1.	A+
2.	AB+
3.	B+
4.	O+

- b. Asks the user for his/her blood group based on the menu in a. above and accepts user input as an integer.
- c. Depending on the user input, the program displays personality characteristics of the user.

Blood Group	Characteristic
A+	Secretive
AB+	Friendly
B+	Goal Getter
O+	Generous

Use this table as a guide

MODULE 6: LOOPS

GENERAL OBJECTIVES

- To learn the basic concept of counter controlled repetition

SPECIFIC OBJECTIVES

- At the end of this module, you will be able to:
 - Use for, nested for, while, nested while, do...while, nested do...while to execute statements in a program respectively

There are certain times when there is need to execute a block of code several times. Loops are used to do such. Generally speaking, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

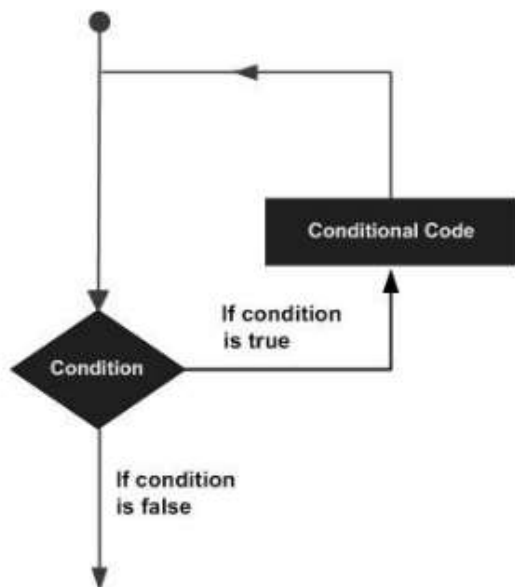


Figure 6.1 General Flow diagram of a loop

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:

C++ programming language provides following types of loop to handle looping requirements.

Table 6.1 A table of the loop types in C++ language

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body.
nested loops	You can use one or more loop inside any another while, for or do..while loop.

6.1 while loop

A while loop statement repeatedly executes a target statement as long as a given condition is true.

The syntax is as follows:

```
While (condition)
{
    Statement(s);
}
```

```

1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6      // Local variable declaration:
7      int a = 1;
8
9      // while loop execution
10     while( a <= 20 )
11     {
12         cout << "value of a: " << a << endl;
13         a++;
14     }
15
16     return 0;
17 }
18
```

Figure 6.2 A program implementing the while loop statement.

Figure 6.3 The result of the while loop statement in figure 5.2

Tutor Marked Assessment

1. Write a C++ program that prints to screen numbers from 10 to 1 using while loop

6.2 For loop

For loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

The syntax of a for loop on C++ is :

```
for ( init; condition; increment )
{
    statement(s);
}
```

Here is the flow of control in a for loop:

- The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

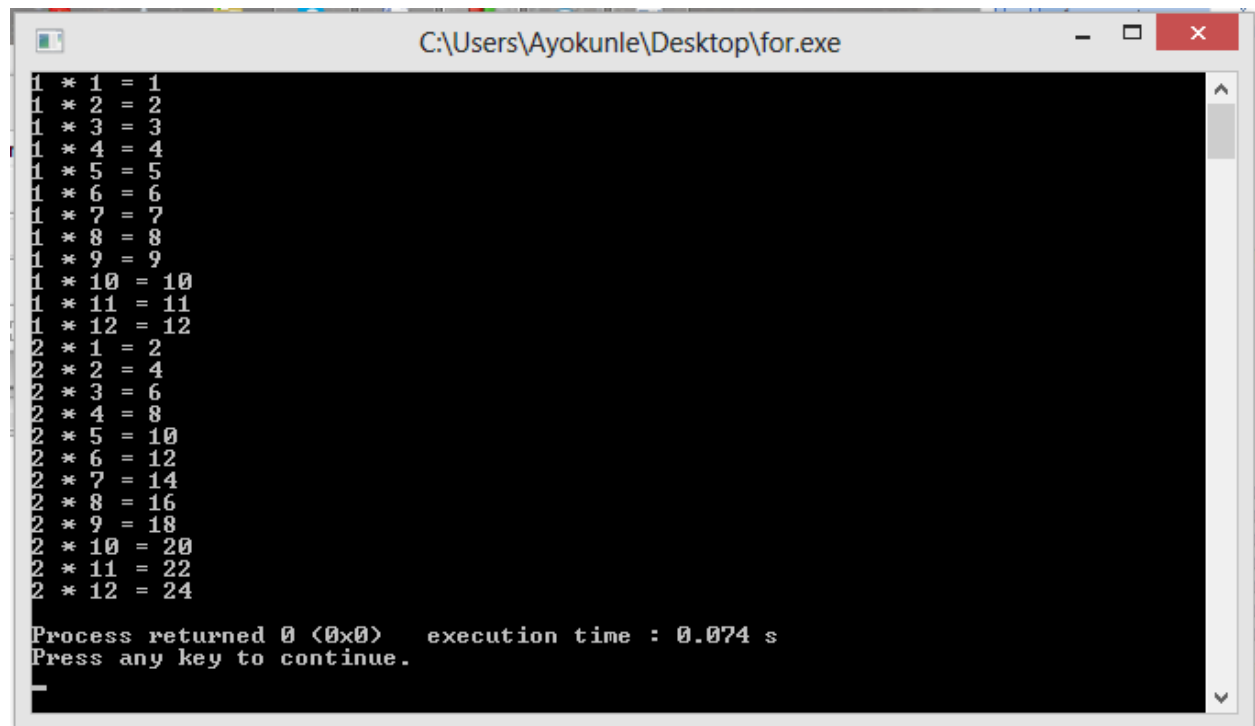
The following program implements the for loop statement

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4  //This is a times table program that terminates at 2
5  int main ()
6  {
7      // for loop execution
8      for( int a = 1; a < 3; a++ )
9      {
10         for(int y = 1; y < 13; y++ )
11             cout << a << " * " << y << " = " << a * y << endl;
12     }
13
14     return 0;
15 }
16

```

Figure 6.4 Implementation of the for loop



```
C:\Users\Ayokunle\Desktop\for.exe
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
1 * 11 = 11
1 * 12 = 12
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
2 * 11 = 22
2 * 12 = 24
Process returned 0 (0x0) execution time : 0.074 s
Press any key to continue.
```

Figure 6.5 Result of the for loop program in figure 5.4

Tutor Marked Assessment

1. Write a program that accepts the score of 100 students using for loop.

6.3 Do...while loop

While for and while loop test conditions at the beginning of the structure, do...while loop tests conditions at the end of the structure.

The syntax of a do...while loop is:

```
do
{
    statement(s);
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  int main ()
6  {
7      // Local variable declaration:
8      int a = 10;
9
10     // do loop execution
11     do
12     {
13         cout << "value of a: " << a << endl;
14         a = a + 1;
15     }while( a < 16 );
16
17     return 0;
18 }
19

```

Figure 6.6 Implementation of do...while loop

```

C:\Users\Ayokunle\Desktop\do...while.exe
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.

```

Figure 6.7 Result of do...while loop program in figure 5.6

6.4 Nested Loops

For all the loops that we have discussed and shown examples of, C++ gives the opportunity to create nested loop. A nested loop can be gotten when it is inside another loop.

6.4.1 Nested for loop

The syntax for nested for loop is:

```
for ( init; condition; increment )
{
    for ( init; condition; increment )
    {
        statement(s);
    }
    statement(s); // you can put more statetments.
}
```

6.4.2 Nested while loop

The syntax for nested while loop is:

```
while (condition)
{
    while(condition)
    {
        statement(s);
    }
    statement(s); // you can put more statetments.
}
```

6.4.3 Nested do...while loop

The syntax for nested do...while loop is:

```
do
{
    statement(s); // you can put more statetments.
    do
    {
        statement(s);
    }while( condition );

}while( condition );
```

The following is a program that uses nested for loop to find the prime numbers from 2 to 100.

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  int main ()
6  {
7      int i, j;
8
9      for(i=2; i<100; i++) {
10         for(j=2; j <= (i/j); j++)
11             if(!(i%j)) break; // if factor found, not prime
12             if(j > (i/j)) cout << i << " is prime\n";
13         }
14     return 0;
15 }
16

```

Figure 6.8 A program that implements nested loop that finds out the prime numbers between 2 and 100

```

11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime

Process returned 0 (0x0)   execution time : 0.066 s
Press any key to continue.

```

Figure 6.9 Result of the nested loop program implemented in figure 3.22

End of Module Assessment

- Find the error in the following code. Correct these errors

```

a. x = 1;
   while ( x <= 10 );
   ++x; }

```

```
b. for ( y = .1; y != 1.0; y += .1 )
    cout << y << endl;
```

```
c. The following code should print the values 1 to 10.
n = 1;
while ( n < 10 )
    cout << n++ << endl;
```

2. Write a program using for loop that prints number from 100 down to 1
3. What is the output of the following program

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

int main()
{
    for ( int i = 1; i <= 3; ++i )
    {
        for ( int j = 1; j <= 3; ++j )
        {
            for ( int k = 1; k <= 3; ++k )
                cout << '*';
            cout << endl;
        } // end inner for
        cout << endl;
    } // end outer for
}
```

4. Write a program that prints the powers of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your while loop should not terminate
5. Write a program that prints the powers of the integer 3, namely 3, 6, 12, 24, 48, 96, etc. Your while loop should terminate at a point determined by the user.

MODULE – 7: C++ FUNCTIONS AND NUMBERS

GENERAL OBJECTIVES

- To be able to write programs more efficiently with the use of both library functions and user-defined functions

Page | 86

SPECIFIC OBJECTIVES

- At the end of this module, you should be able to:
 - Define a function
 - Distinguish between intrinsic function and user-defined function
 - Call/return functions in a program
 - Construct programs using functions

UNIT 7.1 C++ Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function which is `main()`, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

To be part of user-defined A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Depending on whether a function is predefined or created by programmer; there are two types of function:

1. Library/Intrinsic functions
2. User-defined functions

Library/Intrinsic Functions

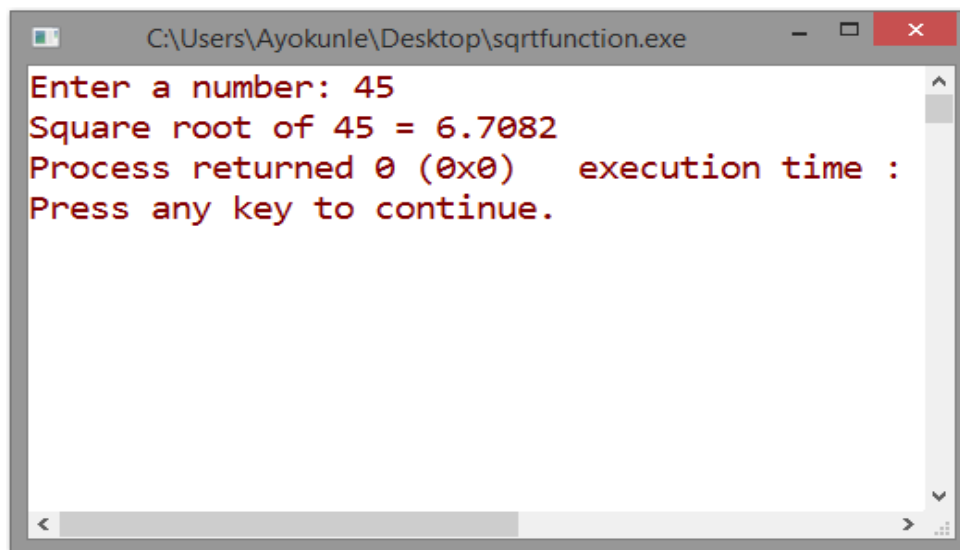
Library functions are the built-in function in C++ programming. Programmer can use library function by invoking function directly; they don't need to write it themselves.

The C++ standard library provides numerous built-in functions that your program can call. For example, function `strcat()` to concatenate two strings, function `memcpy()` to copy one memory location to another location and many more functions.

Another example is shown in figure 7.1.

```
1  # include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  int main()
7  {
8      double number, squareRoot;
9      cout<<"Enter a number: ";
10     cin>>number;
11
12     /* sqrt() is a library function to calculate square root */
13     squareRoot = sqrt(number);
14     cout<<"Square root of "<<number<<" = "<<squareRoot;
15     return 0;
16 }
```

Figure 7.1. An implementation of sqrt library function



```
C:\Users\Ayokunle\Desktop\sqrfunction.exe
Enter a number: 45
Square root of 45 = 6.7082
Process returned 0 (0x0) execution time :
Press any key to continue.
```

Figure 7.2. Output of the sqrt library function in figure 7.1

The example in figure 7.1, `sqrt()` library function is invoked to calculate the square root of a number. The function definition of `sqrt()` (body of that function) is written in the `#include<cmath>`. You can use all functions defined in `cmath` when you include the content of file `cmath` in this program using `#include <cmath>`.

More of these examples are seen in [Module 9: More on Strings](#)

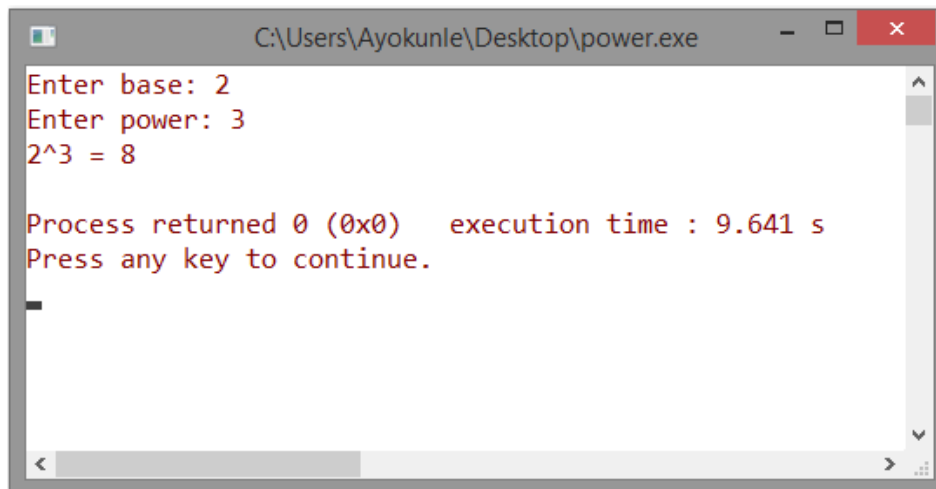
Another example of intrinsic function is seen in the figure 7.3.

```

1  #include<iostream>
2  #include<cmath>
3  using namespace std;
4
5  int main()
6  {
7      int base, power, answer;
8
9      cout <<"Enter base: ";
10     cin >> base;
11     cout <<"Enter power: ";
12     cin >> power;
13     // pow () is an intrinsic function that calculates the power of base raised to the power of power
14     answer = pow(base, power);
15     cout << base <<"^" << power << " = " << answer << endl;
16     return 0;
17 }
18

```

Figure 7.3 An implementation of the pow() library function



```

C:\Users\Ayokunle\Desktop\power.exe
Enter base: 2
Enter power: 3
2^3 = 8

Process returned 0 (0x0)   execution time : 9.641 s
Press any key to continue.

```

Figure 7.4 Output of the pow() library function.

User-defined function

Programmers have the ability to define their own functions in C++. A user-defined function is a group of code to perform a specific task and that group of code is given a name (identifier). When that function is invoked from any part of program, it all executes the codes defined in the body of function.

How user-defined function works in C++ Programming

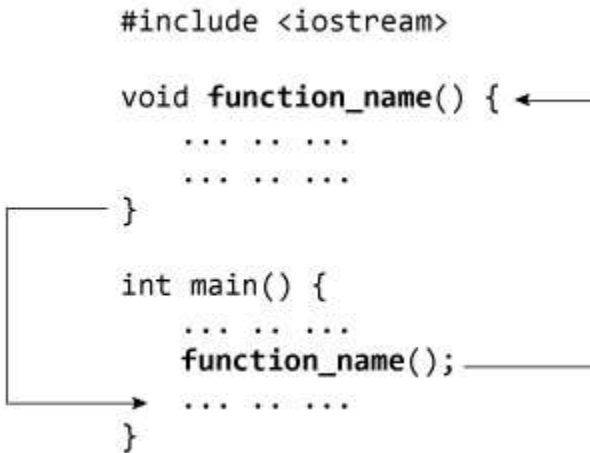


Figure 7.3. How user-defined function works in C++ programming

Consider the figure above. When a program begins running, the system calls the `main()` function, that is, the system starts executing codes from `main()` function. When control of program reaches to `function_name()` inside `main()`, the control of program moves to `void function_name()`, all codes inside `void function_name()` is executed and control of program moves to code right after `function_name()` inside `main()` as shown in figure above.

A function is known with various names like a method or a sub-routine or a procedure etc.

7.1.1 Defining a Function

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
    body of the function
}
```

A C++ function definition consists of a function header and a function body. The following describe the parts of a function:

- **Return Type:** A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the `return_type` is the keyword `void`.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Let's look at more explicit example of a C++ function

```
float distance(float x, float y)
    // Returns the distance of (x, y) from origin
{
    float dist; //local variable
    dist = sqrt(x * x + y * y);
    return dist;
}
```

The function has two input parameters, real values `x` and `y` – declared by the `float` data type, and returns the distance of the point `(x, y)` from the origin. In the function, a local variable `dist` is used to temporarily hold the calculated value inside the function.

We shall now look at the following types of functions:

- Functions with no parameters
- Functions with parameters and no return type
- Functions that return type

7.1.2 Functions with no parameters and no return value

These are of limited use. They will not usually return a value but carry out some operation. Let's consider the following function which skips three lines on output.

```

1  #include <iostream>
2  using namespace std;
3
4  void skipthree(void)  ← Function with no return type
                          and parameter
5      // Function to skip three lines
6  {
7      cout << endl << endl << endl;
8  }
9
10 int main()
11 {
12     //int ....;
13     //float ....;
14     cout << "MAIN TITLE";
15     skipthree();
16     cout << "Sub Title";
17 }
18

```

Figure 7.4 A program that implements a function with no parameters

```

C:\Users\Ayokunle\Desktop\function.exe
MAIN TITLE
Sub Title
Process returned 0 (0x0) execution time : 0.089 s
Press any key to continue.

```

Figure 7.5 Output of the function (without parameters) in figure 7.4

Pay attention to the program. Note that the return-type has been given as void. This informs the computer that the function does not return any value. Because the function does not take any parameters the parameter-list is empty, this is indicated by the void parameter-list. No local variables are required by this function and the function implementation only requires the sending of three successive end-of-line characters to the output stream `cout`. Note the introductory comment that describes what the function does. All functions should include this information as minimal comment.

7.1.3 Functions with no parameters but return value

```
1  #include <iostream>
2  using namespace std;
3
4  int prime();
5
6  int main() {
7      int num, i, flag = 0;
8      num = prime();    /* No argument is passed to prime() */
9      for (i = 2; i <= num/2; ++i) {
10         if (num%i == 0) {
11             flag = 1;
12             break;
13         }
14     }
15     if (flag == 1) {
16         cout<<num<<" is not a prime number.";
17     }
18     else {
19         cout<<num<<" is a prime number.";
20     }
21     return 0;
22 }
23 // Return type of function is int
24 int prime() {
25     int n;
26     cout <<"Enter a positive integer to check: ";
27     cin>>n;
28     return n;
29 }
30
```

Figure 7.6 Function with no parameter but return type

7.1.4 Functions with parameters but no return value

```
1  #include <iostream>
2  using namespace std;
3
4  void prime(int n);
5
6  int main() {
7      int num;
8      cout<<"Enter a positive integer to check: ";
9      cin>>num;
10
11     prime(num); // Argument num is passed to function.
12     return 0;
13 }
14 // There is no return value to calling function. Hence, return type of function is void. */
15 void prime(int n) ← Function prime has a parameter but the return type is void
16 {
17     int i, flag = 0;
18     for (i = 2; i <= n/2; ++i)
19     {
20         if (n%i == 0)
21         {
22             flag = 1;
23             break;
24         }
25     }
26     if (flag == 1) {
27         cout<<n<<" is not a prime number.";
28     }
29     else {
30         cout<<n<<" is a prime number.";
31     }
32 }
```

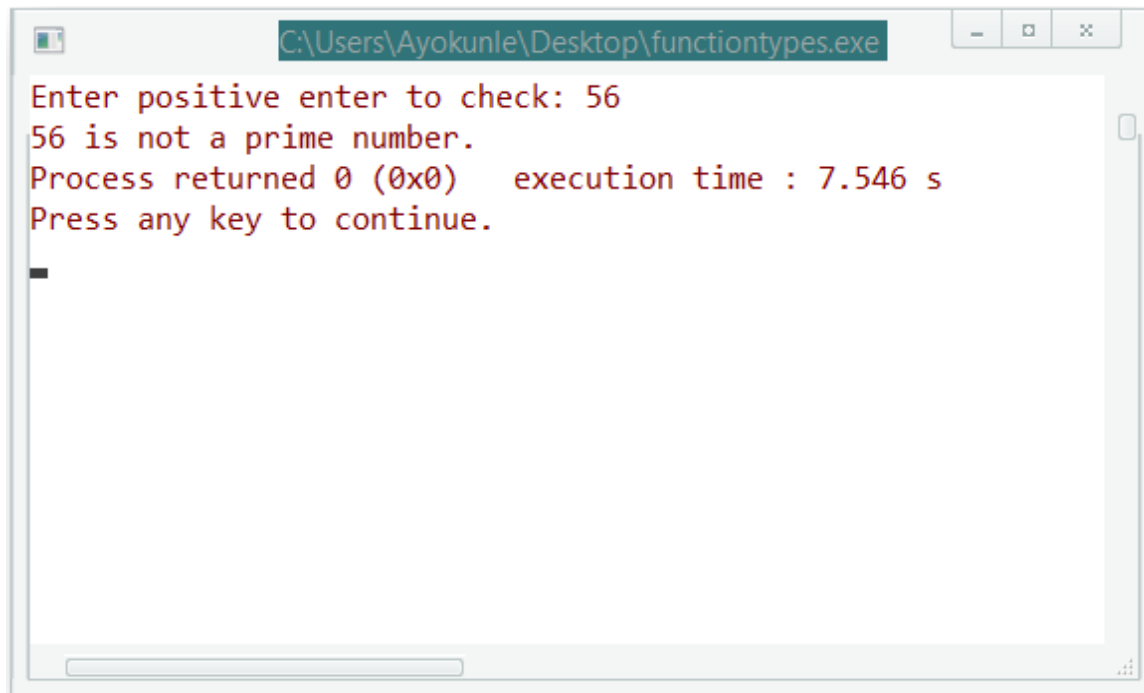
Figure 7.7 Function with parameter with no return type

7.1.5 Functions with parameters that return values

This is the most useful form of function as it returns a value that is a function of its parameters.

```
1  #include <iostream>
2  using namespace std;
3
4  int prime(int n);
5
6  int main()
7  {
8      int num, flag = 0;
9      cout<<"Enter positive enter to check: ";
10     cin>>num;
11     flag = prime(num); /* Argument num is passed to prime() function. */
12     if(flag == 1)
13         cout<<num<<" is not a prime number.";
14     else
15         cout<<num<<" is a prime number.";
16     return 0;
17 }
18
19 /* This function has parameter n returns integer value. */
20 int prime(int n)
21 {
22     int i;
23     for(i = 2; i <= n/2; ++i)
24     {
25         if(n%i == 0)
26             return 1;
27     }
28     return 0;
29 }
30
```

Figure 7.8 Function with parameter and return type



```
C:\Users\Ayokunle\Desktop\functiontypes.exe
Enter positive enter to check: 56
56 is not a prime number.
Process returned 0 (0x0) execution time : 7.546 s
Press any key to continue.
█
```

Figure 7.9 Output of the programs in figures 7.6, 7.7, and 7.8.

N.B. There is no hard and fast rule on which method should be chosen. The particular method is chosen depending upon the situation and how a programmer want to solve that problem.

Tutor Marked Assessment

1. Here is a function, double numbers (int x), what is the name of this function?
A. double
B. int x
C. numbers
2. From question 1, what data type will this function return?
A. Int
B. Double
C. char
3. From question 2, what data type will this function take in?
A. int
B. double
C. char

Page | 96

UNIT 7.2 Numbers in C++

This explains another consolidated way to define various types of number. We shall look at the following example.

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6      // number definition:
7      short shrt;
8      int integer;
9      long lng;
10     float flt;
11     double dble;
12
13     // number assignments;
14     shrt = 10;
15     integer = 1000;
16     lng = 1000000;
17     flt = 230.47;
18     dble = 30949.374;
19
20     // number printing;
21     cout << "short s :" << shrt << endl;
22     cout << "int i :" << integer << endl;
23     cout << "long l :" << lng << endl;
24     cout << "float f :" << flt << endl;
25     cout << "double d :" << dble << endl;
26
27     return 0;
28 }
```

Figure 7.10 A Consolidated example to define various types of numbers in C++

When the code in figure 7.10 is compiled and executed, the following results are generated

```

C:\Users\Ayokunle\Desktop\numbers.exe
short s :10
int i :1000
long l :1000000
float f :230.47
double d :30949.4

Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.

```

Figure 7.11 Output of the number definition program in figure 4.3

7.6 Math Operations in C++

In addition to the various functions you can create, C++ also includes some useful functions you can use. These functions are available in standard C and C++ libraries and called built-in functions. These are functions that can be included in your program and then use.

C++ has a rich set of mathematical operations which can be performed on various numbers. Following table list down some useful built-in mathematical functions available in C++.

To utilize these functions you need to include the math header file `<cmath>`, that is,

```
#include <cmath> or
#include <math.h>
```

Table 7.1 Some mathematical functions in C++

Function	Purpose
<code>double cos(double);</code>	This function takes an angle (as a double) and returns the cosine.
<code>double sin(double);</code>	This function takes an angle (as a double) and returns the sine.
<code>double tan(double);</code>	This function takes an angle (as a double) and returns the tangent.
<code>double log(double);</code>	This function takes a number and returns the natural log of that number.
<code>double pow(double, double);</code>	The first is a number you wish to raise and the second is the power you wish to raise it to.
<code>double hypot(double, double);</code>	If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse.
<code>double sqrt(double);</code>	You pass this function a number and it gives you this square root.
<code>int abs(int);</code>	This function returns the absolute value of an integer that is passed to it.
<code>double fabs(double);</code>	This function returns the absolute value of any decimal number passed to it.
<code>double floor(double);</code>	Finds the integer which is less than or equal to the argument passed to it.

Refer to Appendix B for a more comprehensive list of inbuilt mathematical functions

The following program shows the implementation of some of the mathematical operations in the table 7.1.

```

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main ()
6  {
7      // number definition:
8      short s = 10;
9      int i = 100;
10     long l = 100000;
11     float f = 230.47;
12     double d = -200.374;
13
14     // mathematical operations;
15     //Returns the sin value defined in d
16     cout << "sin(d): " << sin(d) << endl;
17     //Returns the absolute value defined in d
18     cout << "abs(d): " << abs(d) << endl;
19     //Returns the integer which is less than or equal to valude defined in d
20     cout << "floor(d): " << floor(d) << endl;
21     //Returns the integer which is greater than or equal to the value defined in d
22     cout << "ceil(d): " << ceil(d) << endl;
23     //Returns the square root of the value defined in i
24     cout << "sqrt(i): " << sqrt(i) << endl;
25     //The value in d is raised to the power of 2. This displays the result
26     cout << "pow(d,2): " << pow(d, 2) << endl;
27     //Returns the cos value defined in d
28     cout << "cos(d): " << cos(d) << endl;
29     //Returns the tan value defined in d
30     cout << "tan(d): " << tan(d) << endl;
31     //Returns the naturatl log of the value defined in l
32     cout << "log(l): " << log(l) << endl;
33
34     return 0;
35 }

```

Figure 7.12 Program to implement some of the mathematical operators

```

C:\Users\Ayokunle\Desktop\mathoperators.exe
sin(d): 0.634939
abs(d): 200.374
floor(d): -201
ceil(d): -200
sqrt(i): 10
pow(d,2): 40149.7
cos(d): 0.772562
tan(d): 0.821862
log(l): 11.5129

Process returned 0 (0x0) execution time : 0.172 s
Press any key to continue.

```

Figure 7.13
Output of the
program in figure
7.12

7.7 Random Numbers in C++

There are many cases where you will wish to generate random numbers. There are actually two functions you will need to know about random number generation. The first is `rand()`, this function will only return a pseudo random number. The way to fix this is to first call the `srand()` function.

The following is a simple example to generate few random numbers. This example makes use of `time()` function to get the number of seconds on your system time, to randomly seed the `rand()` function:

```

1  #include <iostream>
2  #include <ctime>
3  #include <cstdlib>
4
5  using namespace std;
6
7  int main ()
8  {
9      int i,j;
10
11      // set the seed
12      srand( (unsigned)time( NULL ) );
13
14      /* generate 10 random numbers. */
15      for( i = 0; i < 10; i++ )
16      {
17          // generate actual random number
18          j= rand();
19          cout << " Random Number : " << j << endl;
20      }
21
22      return 0;
23  }
```

Figure 7.14 A program that randomize numbers based on your system time

```

C:\Users\Ayokunle\Desktop\randomnumbers.exe
Random Number : 12672
Random Number : 16353
Random Number : 19702
Random Number : 31866
Random Number : 11089
Random Number : 24982
Random Number : 17818
Random Number : 22407
Random Number : 12417
Random Number : 24385

Process returned 0 (0x0)   execution time : 0.434 s
Press any key to continue.
```

Figure 7.14 Output of the randomization program in figure 7.

End of Module Assessment

1. Write a C++ program that accepts three integers and a function that returns the largest and the smallest of the integers.
2. Show the value of x after each of the following statements is performed:
 - a) `x = fabs(7.5)`
 - b) `x = floor(7.5)`
 - c) `x = fabs(0.0)`
 - d) `x = ceil(0.0)`
 - e) `x = fabs(-6.4)`
 - f) `x = ceil(-6.4)`
 - g) `x = ceil(-fabs(-8 + floor(-5.5)))`
3. Write a program that inputs a series of integers and passes them one at a time to function `isEven`, which uses the modulus operator to determine whether an integer is even. The function should take an integer argument and return `true` if the integer is even and `false` otherwise.

MODULE – 8: ARRAYS

GENERAL OBJECTIVES

- To use the array data structure to represent a set of related data items

SPECIFIC OBJECTIVES

- At the end of the module, you will be able to:
 - Declare arrays
 - Initialize arrays
 - Access the elements of an array
 - Declare and manipulate two dimensional arrays

8.1 Arrays

This is one of the data structures that C++ provides. The array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more constructive to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

8.2 Declaring Arrays

An array declaration is very similar to a variable declaration. First a type is given for the elements of the array, then an identifier for the array and, within square brackets, the number of elements in the array. The number of elements must be an integer. The general form for declaring an array is:

```
type arrayName [ arraySize ];
```

For example data on the average temperature over the year in Britain for each of the last 100 years could be stored in an array declared as follows:

```
float annual_temp[100];
```

This declaration will cause the compiler to allocate space for 100 consecutive float variables in memory. The number of elements in an array must be fixed at compile time. It is best to make

the array size a constant and then, if required, the program can be changed to handle a different size of array by changing the value of the constant,

```
const int NE = 100;
float annual_temp[NE];
```

then if more records come to light it is easy to amend the program to cope with more values by changing the value of NE. This works because the compiler knows the value of the constant NE at compile time and can allocate an appropriate amount of space for the array. It would not work if an ordinary variable was used for the size in the array declaration since at compile time the compiler would not know a value for it.

8.3 Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces {} cannot be larger than the number of elements that we declare for the array between square brackets []. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
Balance	1000.0	2.0	3.4	7.0	50.0

8.4 Accessing Array Elements

To explain how one can access array elements, we will go back to the illustration in 8.2, the declaration of a 100 element array instructs the compiler to reserve space for 100 consecutive

floating point values and access these values using an index/subscript that takes values from 0 to 99. The first element in an array in C++ always has the index 0. And if the array has n elements, the last element will have the index n-1.

An array element is accessed by writing the identifier of the array followed by the subscript in square brackets. Thus to set the 15th element of the array above to 1.5 the following assignment is used:

```
annual_temp[14] = 1.5;
```

Note that since the first element is at index 0, then the ith element is at index i-1. Hence in the above the 15th element has index 14.

An array element can be used anywhere an identifier may be used. Here are some examples assuming the following declarations:

```
const int NE = 100,  
        N = 50;  
int i, j, count[N];  
float annual_temp[NE];  
float sum, av1, av2;
```

A value can be read into an array element directly, using cin

```
cin >> count[i];
```

The element can be increased by 5,

```
count[i] = count[i] + 5;  
or, using the shorthand form of the assignment  
count[i] += 5;
```

Array elements can form part of the condition for an if statement, or indeed, for any other logical expression:

```
if (annual_temp[j] < 10.0)  
    cout << "It was cold this year "  
<< endl;
```

for statements are the usual means of accessing every element in an array. Here, the first NE elements of the array annual_temp are given values from the input stream cin.

```
for (i = 0; i < NE; i++)  
    cin >> annual_temp[i];
```

The following code finds the average temperature recorded in the first ten elements of the array.

```
sum = 0.0;  
for (i = 0; i < 10; i++)  
    sum += annual_temp[i];  
av1 = sum / 10;
```

Notice that it is good practice to use named constants, rather than literal numbers such as 10. If the program is changed to take the average of the first 20 entries, then it's all too easy to forget to change a 10 to 20. If a `const` is used consistently, then changing its value will be all that is necessary.

For example, the following example finds the average of the last `k` entries in the array. `k` could either be a variable, or a declared constant. Observe that a change in the value of `k` will still calculate the correct average (provided $k \leq NE$).

```
sum = 0.0;
for (i = NE - k; i < NE; i++)
    sum += annual_temp[i];
av2 = sum / k;
```

```
1  #include <iostream>
2  using namespace std;
3
4  #include <iomanip>
5  using std::setw;
6
7  int main ()
8  {
9      int n[ 10 ]; // n is an array of 10 integers
10
11     // initialize elements of array n to 0
12     for ( int i = 0; i < 10; i++ )
13     {
14         n[ i ] = i + 100; // set element at location i to i + 100
15     }
16     cout << "Element" << "\t""\t" << "Value" << endl;
17
18     // output each array element's value
19     for ( int j = 0; j < 10; j++ )
20     {
21         cout << setw(5)<< j << "\t""\t" << n[ j ] << endl;
22     }
23
24     return 0;
25 }
26
```

Figure 8.1 A array program

This program makes use of `setw()` function to format the output. When the above code is compiled and executed, it produces the following result:


```

C:\Users\Ayokunle\Desktop\arrays.exe
Element      Value
0            100
1            101
2            102
3            103
4            104
5            105
6            106
7            107
8            108
9            109

Process returned 0 (0x0)   execution time : 0.066 s
Press any key to continue.

```

Figure 8.2 Output of the array program in figure 8.1

Tutor Marked Assessment

1. double num [7]; what is the name of this array?

A. double	B. num
C. Doesn't have a name	

2. How many values will the array in 4 hold?

A. 6	B. 7
C. 8	

3. What is the greatest index number in the array in 4

A. 6	B. 7
C. 8	

4. What is the most efficient way to assign or print out arrays?

A. Functions	B. Pointers
C. loops	

8.5 More on Arrays

8.5.1 Multi-dimensional Array

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration. The general form of multi-dimensional array is as follows:

```
type name[size1][size2]...[sizeN];
```

Page | 106

Two-dimensional array

This is the simplest form of a multi-dimensional array.

A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where type can be any valid C++ data type and arrayName will be a valid C++ identifier.

A two dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimensional array which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multi-dimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3} , /* initializers for row indexed by 0 */  
    {4, 5, 6, 7} , /* initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

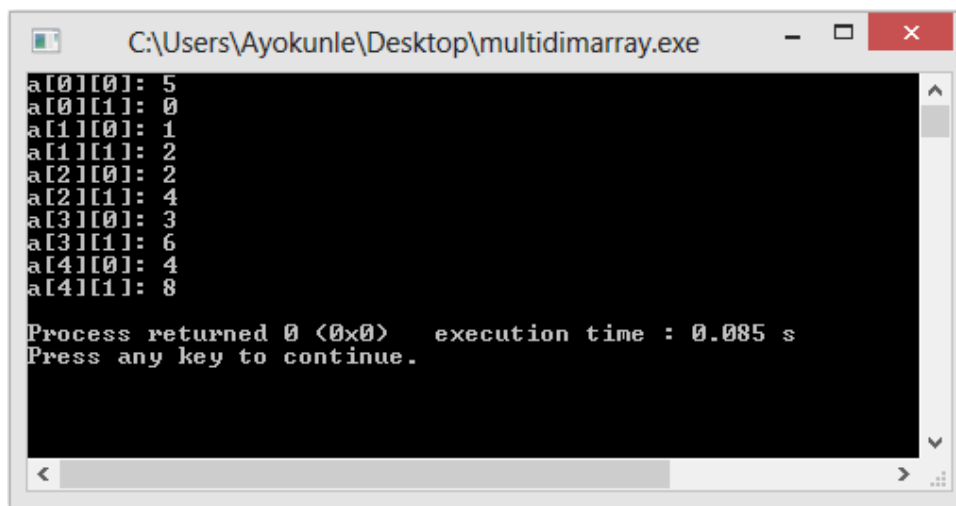
```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Arrays

An element in 2-dimensional array is accessed by using the subscripts ie. row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. This can be verified it in the above diagram.



```
C:\Users\Ayokunle\Desktop\multidimarray.exe
a[0][0]: 5
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8

Process returned 0 (0x0)   execution time : 0.085 s
Press any key to continue.
```

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6      // an array with 5 rows and 2 columns.
7      int a[5][2] = { {5,0}, {1,2}, {2,4}, {3,6}, {4,8} };
8
9      // output each array element's value
10     for ( int i = 0; i < 5; i++ )
11         for ( int j = 0; j < 2; j++ )
12         {
13             cout << "a[" << i << "][" << j << "]: ";
14             cout << a[i][j] << endl;
15         }
16
17     return 0;
18 }
19
```

Figure 8.3 An implementation of a two-dimensional array

Figure 8.4 Output of the two-dimensional array in figure 8.3

8.5.2 Pointer to an Array

An array name is a constant pointer to the first element of the array. Therefore, in the declaration: Page | 108

```
double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p the address of the first element of balance:

```
double *p;  
double balance[10];  
  
p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on. Below is the example to show all the concepts discussed above:

```

1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  int main ()
6  {
7      // an array with 5 elements.
8      double balance[5] = {1000.05, 2200.50, 3500.99, 4000.30, 550.01};
9      double *p;
10
11     p = balance;
12
13     // output each array element's value
14     cout << "Array values using pointer " << endl;
15     for ( int i = 0; i < 5; i++ )
16     {
17         cout << "*(p + " << i << " ) : ";
18         cout << *(p + i) << endl;
19     }
20
21     cout << "Array values using balance as address " << endl;
22     for ( int i = 0; i < 5; i++ )
23     {
24         cout << "*(balance + " << i << " ) : ";
25         cout << *(balance + i) << endl;
26     }

```

Figure 8.5 A program that uses C++ pointer to an Array

```

C:\Users\Ayokunle\Desktop\pointer.exe
Array values using pointer
*(p + 0) : 1000.05
*(p + 1) : 2200.5
*(p + 2) : 3500.99
*(p + 3) : 4000.3
*(p + 4) : 550.01
Array values using balance as address
*(balance + 0) : 1000.05
*(balance + 1) : 2200.5
*(balance + 2) : 3500.99
*(balance + 3) : 4000.3
*(balance + 4) : 550.01
Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.

```

Figure 8.6 Output of the pointer program in figure 8.6

8.5.3 Passing Arrays as Function Arguments in C++

C++ does not allow passing an entire array as an argument to a function. However, you can pass a pointer to an array by specifying the array's name without an index.

Page | 110

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way 1: Formal parameters as a pointer

```
void myFunction(int *param)
{
    .
    .
    .
}
```

Way 2: Formal parameters as a sized array

```
void myFunction(int param[10])
{
    .
    .
    .
}
```

Way 3: Formal parameters as an unsized array

```
void myFunction(int param[])
{
    .
    .
    .
}
```

We shall now consider the following function which takes an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:

```
double getAverage(int arr[], int size)
```

```
{
    int    i, sum = 0;
    double avg;

    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    avg = double(sum) / size;

    return avg;
}
```

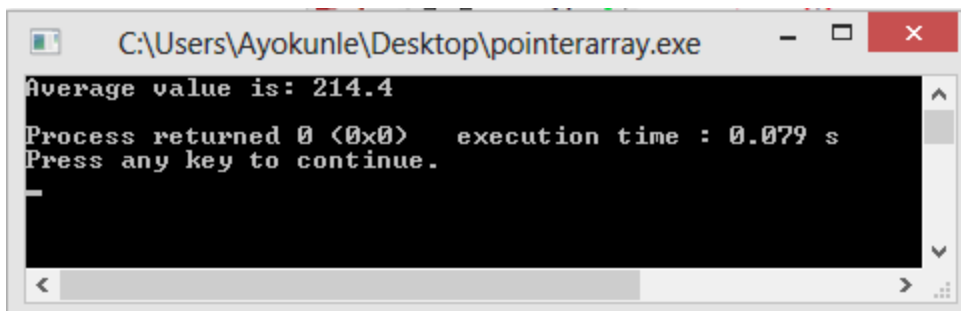
Now let's look at the following program

```

1  #include <iostream>
2  using namespace std;
3
4  // function declaration:
5  double getAverage(int arr[], int size);
6
7  double getAverage(int arr[], int size)
8  {
9      int i, sum = 0;
10     double avg;
11
12     for (i = 0; i < size; ++i)
13     {
14         sum += arr[i];
15     }
16
17     avg = double(sum) / size;
18
19     return avg;
20 }
21
22 int main ()
23 {
24     // an int array with 5 elements.
25     int balance[5] = {1000, 2, 3, 17, 50};
26     double avg;
27
28     // pass pointer to the array as an argument.
29     avg = getAverage( balance, 5 ) ;
30
31     // output the returned value
32     cout << "Average value is: " << avg << endl;
33
34     return 0;
35 }

```

Figure 8.7 A program that passes an array as Function Arguments in C++



```

C:\Users\Ayokunle\Desktop\pointerarray.exe
Average value is: 214.4
Process returned 0 (0x0) execution time : 0.079 s
Press any key to continue.

```

Figure 8.8 Output of the program in figure 8.7

Return array from functions in C++

C++ does not allow returning an entire array as an argument to a function. However, You can return a pointer to an array by specifying the array's name without an index.

Page | 113

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()  
{  
    .  
    .  
    .  
}
```

Second point to remember is that C++ does not advocate to return the address of a local variable to outside of the function so you would have to define the local variable as static variable.

Now consider the following function which will generate 10 random numbers and return them using an array and call this function as follows:

```

1  #include <iostream>
2  #include <ctime>
3  #include <cstdlib>
4  using namespace std;
5
6  // function to generate and retrun random numbers.
7  int * getRandom( )
8  {
9      static int r[10];
10
11      // set the seed
12      srand( (unsigned)time( NULL ) );
13      for (int i = 0; i < 10; ++i)
14      {
15          r[i] = rand();
16          cout << r[i] << endl;
17      }
18
19      return r;
20  }
21
22  // main function to call above defined function.
23  int main ()
24  {
25      // a pointer to an int.
26      int *p;
27
28      p = getRandom();
29      for ( int i = 0; i < 10; i++ )
30      {
31          cout << "(p + " << i << ") : ";
32          cout << *(p + i) << endl;
33      }
34
35      return 0;

```

Figure 8.9 A program that returns array from functions

```

C:\Users\Ayokunle\Desktop\rtrnarrayfromfunction.exe
1986
5069
101
31400
13610
3437
8764
11323
4457
783
*(p + 0) : 1986
*(p + 1) : 5069
*(p + 2) : 101
*(p + 3) : 31400
*(p + 4) : 13610
*(p + 5) : 3437
*(p + 6) : 8764
*(p + 7) : 11323
*(p + 8) : 4457
*(p + 9) : 783
Process returned 0 (0x0)   execution time : 0.100 s
Press any key to continue.

```

Figure 8.10 Output of the program that returns array from functions in figure 8.9

Tutor Marked Assessment

1. The names of the four elements of array p (`int p[4];`) are _____, _____, and _____.
2. Naming an array, stating its type and specifying the number of elements in the array is called _____ the array.
3. By convention, the first subscript in a two-dimensional array identifies an element's _____ and the second subscript identifies an element's _____.
4. An m -by- n array contains _____ rows, _____ columns and _____ elements.
5. The name of the element in row 3 and column 5 of array d is .

End of Module Assessment

1. The syntax for creating an array is _____

2. Which is the correct way to initialize values in an array?

A. <code>my_array [5]= (1,2,3,4,5);</code>	B. <code>my_array (5) = (1,2,3,4,5);</code>
C. <code>my_array {5}={1,2,3,4,5};</code>	

3. Write a C++ program that does the following:

- a. Asks for the score of 100 students.
- b. Determines the grade of each score.
- c. Computes the average of the score of the 100 students
- d. Determines the average grade of the 100 Students

A: 80 – 100; B: 60 – 79; C: 50 – 59; D: 45 – 49; E: 40 – 44; F: 0 – 39

4. Declare an array to be an integer array and to have 3 rows and 3 columns. Assume that the constant variable `arraySize` has been defined to be 3.

- a. How many elements does the array contain?
- b. Use a for statement to initialize each element of the array to the sum of its subscripts.
- c. Assume that the integer variables `i` and `j` are declared as control variables.
Write a program segment to print the values of each element of array table in tabular format with 3 rows and 3 columns. Assume that the array was initialized with the declaration

5. Write a program that asks for a number and then prints the square and cube (the number multiplied by itself three times) of the number you input, if that number is more than 1. Otherwise, the program does not print anything.

MODULE – 9: MORE ON STRINGS

GENERAL OBJECTIVES

- To be able to use variables that store non-numerical values that are longer than a single character.

Page | 116

SPECIFIC OBJECTIVES

- At the end of this module, you will be able to:
 - Assign, concatenate, and compare strings using the C-Style character string
 - Call and use functions that manipulate null-terminated strings
 - Write programs that use the C++ string class
 - Understand the differences between C-style character string and C++ string class type

The term *string* generally means an ordered sequence of characters, with a first character, a second character, and so on, and in most programming languages such strings are enclosed in either single or double quotes. In C++ the enclosing delimiters are double quotes. In this form the string is referred to as a *string literal* and we often use such string literals in output statements when we wish to display text on the screen for the benefit of our users.

Unit 9.1 Character String in C

There are two types of string representation provided by C++ language. They are”

- The C-style character string
- The string class type introduced with Standard C++

C-style character string

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Good". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Good".

```
char a[5] = {'G', 'o', 'o', 'd', '\\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows:

```
char a[] = "Good";
```

Page | 117

Following is the memory presentation of above defined string in C/C++:

G	o	o	d	\\0'
---	---	---	---	------

Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\\0' at the end of the string when it initializes the array. Let us try to print above mentioned string:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main ()
6  {
7      char a[6] = {'G', 'o', 'o', 'd', '\\0'};
8
9      cout << "How do you do? ";
10     cout << a << endl;
11
12     return 0;
13 }
14
```

Figure 9.1 A program that prints a string

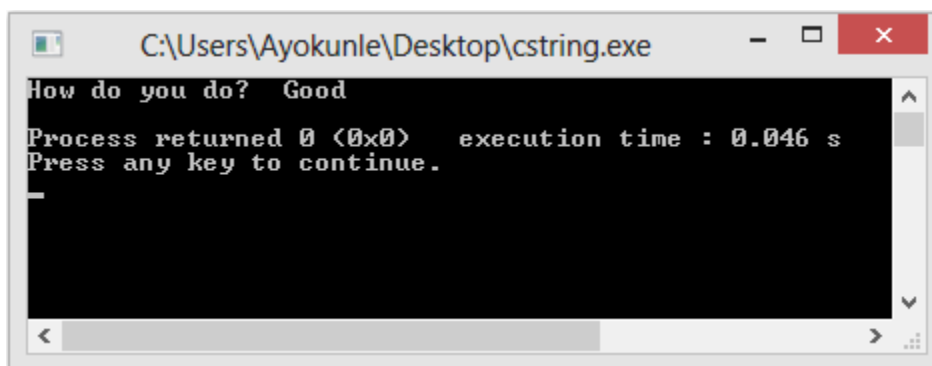


Figure 9.2 Output

Unit 9.2 Functions that manipulate null-terminated strings

C-Style string or simply C-string supports a wide range of functions that manipulate null-terminated strings. These are listed below:

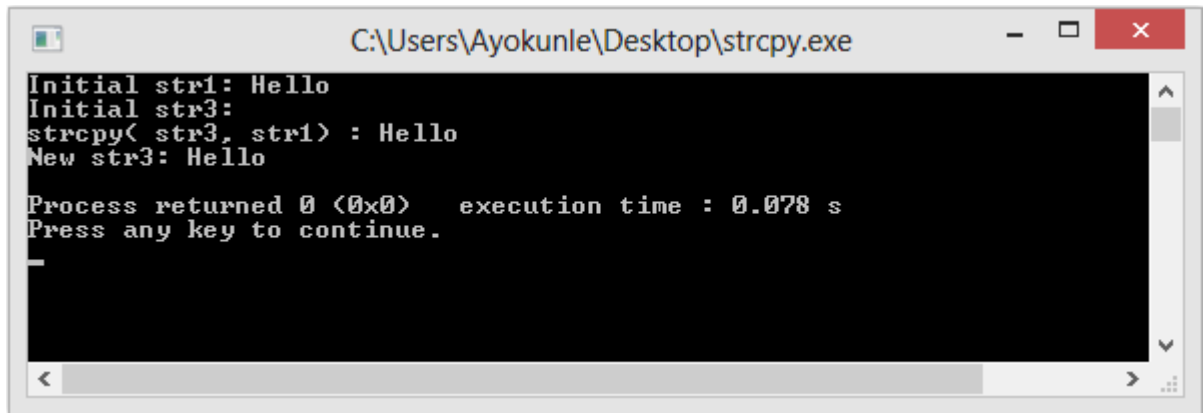
- strcpy function
- strcat function
- strlen function

9.2.1 strcpy function

This is used to copy contents from one string to another. The syntax is as follows:

```
strcpy(s1, s2);  
1  #include <iostream>  
2  #include <cstring>  
3  
4  using namespace std;  
5  
6  int main ()  
7  {  
8      char str1[10] = "Hello";  
9      char str2[10] = "World";  
10     char str3[10] = "";  
11     int len ;  
12  
13     // output initial str1 and str3  
14     cout <<"Initial str1: " << str1 << endl;  
15     cout <<"Initial str3: " << str3 << endl;  
16     // copy str1 into str3  
17     strcpy( str3, str1);  
18     cout << "strcpy( str3, str1) : " << str3 << endl;  
19     // output the new string str3  
20     cout << "New str3: " << str3 << endl;  
21     return 0;  
22 }  
23
```

Figure 9.3 Program that copies the content of a string to another using the strcpy function (NB - <cstring> header file is used)



```
C:\Users\Ayokunle\Desktop\strcpy.exe
Initial str1: Hello
Initial str3:
strcpy( str3, str1) : Hello
New str3: Hello

Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

Figure 9.4 Output of the strcpy function in figure 9.3

9.2.2 strcat function

This function is used to concatenate strings onto others. The syntax is as follows:

```
strcat(s1, s2);
```

```
1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5
6  int main ()
7  {
8      char str1[10] = " Hello ";
9      char str2[10] = " World ";
10     char str3[10];
11     int len ;
12
13     // concatenates str1 and str2
14     strcat( str1, str2);
15     cout << "strcat( str1, str2): " << str1 << endl;
16
17
18     return 0;
19 }
20
```

Figure 9.5 Program that implements the strcat function

```

C:\Users\Ayokunle\Desktop\strcat.exe
strcat( str1, str2): Hello World
Process returned 0 (0x0) execution time : 0.257 s
Press any key to continue.

```

Figure 9.6 Output of the strcat program in figure 9.5

9.2.3 strlen function

This function is used to return the length of a string. The syntax is as follows:

```
Strlen(s1);
```

```

1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5
6  int main ()
7  {
8      char str1[10] = "Hello ";
9      char str2[10] = "World";
10     char str3[10];
11     int len ;
12
13     // concatenates str1 and str2
14     strcat( str1, str2);
15     cout << "strcat( str1, str2): " << str1 << endl;
16
17     // total length of str1 after concatenation
18     len = strlen(str1);
19     cout << "strlen(str1) : " << len << endl;
20
21     return 0;
22 }
23

```

Figure 9.7 strlen program to calculate the length of a string


```

C:\Users\Ayokunle\Desktop\strlen.exe
strcat< str1, str2>: Hello World
strlen(str1) : 11

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.

```

Figure 9.8 Output of the strlen program in figure 9.7

9.3 The String Class in C++

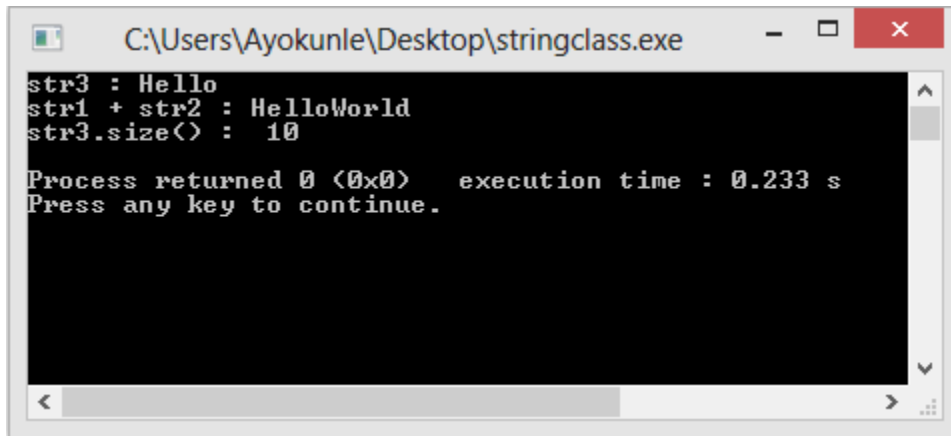
The standard C++ library provides a string class type that supports all the operations mentioned in 9.2, additionally much more functionality. Let us check following example:

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main ()
7  {
8      string str1 = "Hello";
9      string str2 = "World";
10     string str3;
11     int len ;
12
13     // copy str1 into str3
14     str3 = str1;
15     cout << "str3 : " << str3 << endl;
16
17     // concatenates str1 and str2
18     str3 = str1 + str2;
19     cout << "str1 + str2 : " << str3 << endl;
20
21     // total length of str3 after concatenation
22     len = str3.size();
23     cout << "str3.size() : " << len << endl;
24
25     return 0;
26 }

```

Figure 9.9 A program that implements the string class in C++



```
C:\Users\Ayokunle\Desktop\stringclass.exe
str3 : Hello
str1 + str2 : HelloWorld
str3.size() : 10

Process returned 0 (0x0)   execution time : 0.233 s
Press any key to continue.
```

Figure 9.10 Output of the string class program in figure 9.9

9.4 Differences between C-Style character string `<cstring>` and C++ string class type `<string>`

As we have discussed earlier, a string generally is an ordered sequence of character enclosed in double quotes and it is referred to as a string literal. This is most useful in situations when we just wish to display text on the screen. Without string variables, i.e. named memory locations to manipulate strings all that can be done is to display string literals to the screen.

C++ is backward compatible with C, its predecessor which means some functions which include the string function inherent in C may also be used in C++. However, the way C deals with strings is very much different from the way C++ deals with string and as such, one must exercise caution.

One then must be recognize the difference among the following three things:

1. A simple array of characters that have no special properties
2. A C-string that consists of an array of characters terminated by the null character `'\0'`. This makes it different from the simple array that has no special characteristics. Strings that are represented in this form are handled by a library of functions that are included into the program using the `<cstring>` header. In some implementations this library may be automatically included when you include other libraries such as the `<iostream>` library. (This is evident in the program in figure 9.1. Take note that the `<cstring>` header was not included in the program.)
3. A C++ string object, which is an instance of a "class" data type whose actual internal representation you need not know or care about, as long as you know what you can and can't do with variables (and constants) having this data type. There is a library of C++ string functions as well, available by including the `<string>` header file.

Both the C-string library functions and the C++ string library functions are available to C++ programs. But, don't forget that these are two **different** function libraries, and the functions of the first library have a different notion of what a string is from the corresponding notion held by the functions of the second library. There are two further complicating aspects to this situation: first, though a function from one of the libraries may have a counterpart in the other library (i.e., a function in the other library designed to perform the same operation), the functions may not be used in the same way, and may not even have the same name; second, because of backward compatibility many functions from the C++ string library can be expected to work fine and do the expected thing with C-style strings, but not the other way around.

The last statement above might seem to suggest we should use C++ strings and forget about C-strings altogether, and it is certainly true that there is a wider variety of more intuitive operations available for C++ strings. However, C-strings are more primitive, you may therefore find them simpler to deal with (provided you remember a few simple rules, such as the fact that the null character must always terminate such strings), and certainly if you read other, older programs you will see lots of C-strings. So, use whichever you find more convenient, but if you choose C++ strings and occasionally need to mix the two for some reason, be extra careful. Finally, there are certain situations in which C-strings *must* be used.

In an attempt to clarify the differences between C-style strings and C++ strings, we will use a program to draw the distinction in the following areas:

- Declaration of string
- Initialization of string
- Assigning to a string
- Concatenation of two strings
- Copying a string
- Accessing a single character
- Comparing two strings
- Finding the length of a string
- Outputting a string

The strings in C-style string is are referred to as ***string variables*** while strings in C++ string are referred to as ***string objects***.

Table 9.1. A table showing the major difference between C-style string and C++ string

C-style String	C++ Strings
Importing Header File	
#include<cstring>	#include<string>

Declaring a string	
<code>char str[15];</code>	<code>string str;</code>
Initializing a string	
<code>char str1[5] = "Hello";</code> <code>char str2[] = "How are you?";</code>	<code>string str1 ("Hello");</code> <code>string str2 = "How are you?";</code>
Assigning to a string	
<code>char str[10];</code> <code>str = "Hello";</code>	<code>string str;</code> <code>str = "Hello";</code>
Concatenating strings	
<code>char str1[5] = "Hello";</code> <code>char str2[5] = "World";</code> <code>strcpy(str1, str2);</code> <code>strcpy(str, strcat(str1, str2));</code>	<code>string str1 = "Hello";</code> <code>string str2 ("World");</code> <code>str1 += str2;</code> <code>str = str1 + str2;</code>
Copying a String	
<code>char str[20];</code> <code>string str;</code> <code>strcpy(str, "Hello!");</code>	<code>str = "Hello";</code> <code>strcpy(str, otherString);</code> <code>str = otherString;</code>
Accessing a single character	
<code>str[index]</code>	<code>str[index]</code> <code>str.at(index)</code> <code>str(index, count)</code>
Comparing two Strings	
<code>if (strcmp(str1, str2) < 0)</code> <code>cout << "str1 comes 1st.";</code> <code>if (strcmp(str1, str2) == 0)</code> <code>cout << "Equal strings.";</code> <code>if (strcmp(str1, str2) > 0)</code> <code>cout << "str2 comes 1st.";</code>	<code>if (str1 < str2)</code> <code>cout << "str1 comes 1st.";</code> <code>if (str1 == str2)</code> <code>cout << "Equal strings.";</code> <code>if (str1 > str2)</code> <code>cout << "str2 comes 1st.";</code>
Getting the length of a string	
<code>strlen(str);</code>	<code>str.length()</code>

Output of a string	
<code>cout << str;</code> <code>cout << setw(width) << str;</code>	<code>cout << str;</code> <code>cout << setw(width) << str;</code>

End of Module Assessment

1. Write a C++ program that does the following using the C-Style string:
 - a. Asks for user's first name
 - b. Asks for user's middle name
 - c. Asks the user's last name
 - d. Displays the length of the first name
 - e. Displays the length of the middle name
 - f. Displays the length of the last name
 - g. Concatenates all the names so as to have it in the format:
`<firstname><middlename><lastname>`
2. Write a program to store and print the names of your two favorite television programs. Store these programs in two character arrays. Initialize one of the strings (assign it the first program's name) at the time you declare the array. Initialize the second value in the body of the program with the strcpy() function.
3. Repeat the question in 1 above using the C++ string

MODULE – 10: C++ BASIC INPUT/OUTPUT

GENERAL OBJECTIVES

- To be able to understand and use the extensive input/output capabilities that C++ provides

SPECIFIC OBJECTIVES

- At the end of this module, you will be able to:
 - Write a C++ program that uses the standard output stream – `cout`
 - Write a C++ program that uses the standard input stream – `cin`
 - Write a C++ program that uses the standard error stream – `cerr`
 - Write a C++ program that uses the standard log stream – `clog`

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc, this is called **output operation**.

10.1 I/O Library Header Files

These are the following header file important to C++ Programs

Table 10.1 Important header files and their function

Header File	Function
<iostream>	This file defines the cin , cout , cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
<iomanip>	This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision .
<fstream>	This file declares services for user-controlled file processing.

10.2 The standard output stream (cout)

The predefined object `cout` is an instance of `ostream` class. The `cout` object is said to be "connected to" the standard output device, which usually is the display screen. The `cout` is used in conjunction with the stream insertion operator, which is written as `<<` which are two less than signs as shown in the following example.

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      char oruko[] = "Omotunde Ayokunle A";
7      cout << "Hello my name is " << oruko;
8      return 0;
9  }
```

Figure 10.1 A program to illustrate the standard output stream (cout)

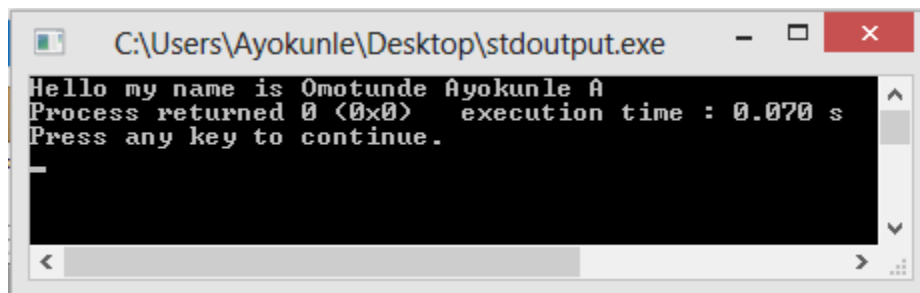


Figure 10.2 Output of the `cout` program in figure 10.1

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The `<<` operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator `<<` may be used more than once in a single statement as shown above and `endl` is used to add a new-line at the end of the line.

10.3 The standard input stream (cin)

The predefined object `cin` is an instance of `istream` class. The `cin` object is said to be attached to the standard input device, which usually is the keyboard. The `cin` is used in conjunction with the

stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main( )
6  {
7      char name[50];
8
9      cout << "Please enter your surname: ";
10     cin >> name;
11     cout << "Your name is: " << name << endl;
12
13 }
14
```

Figure 10.3 A program to illustrate the standard input stream (cin)

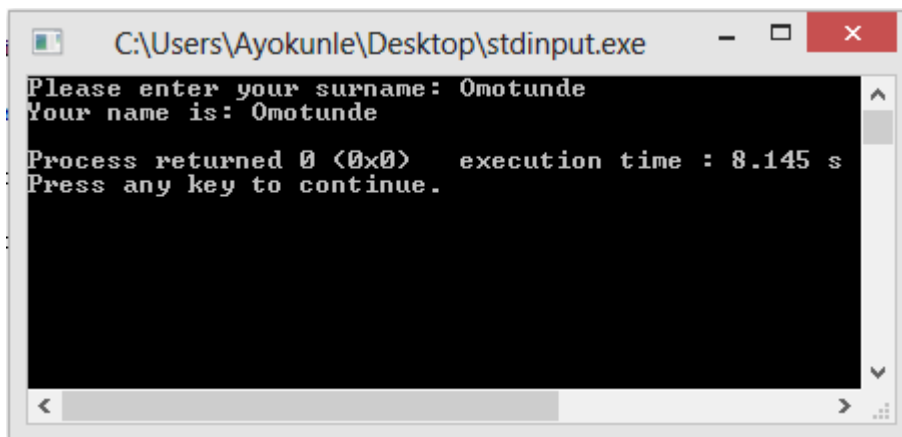


Figure 10.4 Output of the cin program in figure 10.3

10.4. Inputting white spaces

Let us rerun the program in figure 10.3 and respond to the request (Please enter your name) in a slightly different manner.

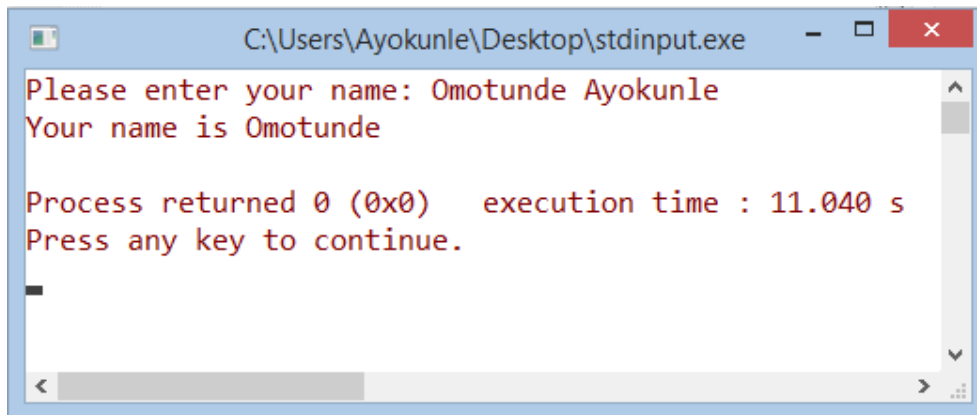


Figure 10.5 Output of the cin program in figure 10.3 with white space.

Notice that the name entered upon request has a white space in between. When the name is value of the variable name is displayed, the part that comes after the white space is truncated.

It can therefore be concluded that `cin` ignores whitespaces when reading a string. In order to cater for white space:

`getline(cin, variablename);` is used.

This is implemented in the following program

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  int main()
6  {
7      string name;
8      cout << "What is your name? ";
9      getline(cin, name);
10     cout << name;
11
12     return 0;
13 }
14

```

Figure 10.6 A program that accepts white space

NB: the header `<string>` in this program...`include<string>`

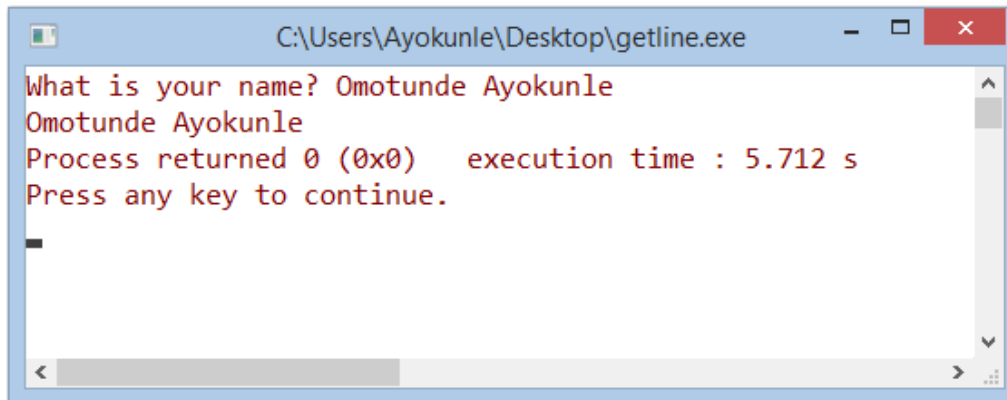


Figure 10.7 Output of the program in figure 10.6 (A program that accepts white space)

10.5 C++ Standard error stream

This is an instantiation of the `ostream` class. The `cerr` object is usually attached to the standard error device, which is also a display screen but the object `cerr` is un-buffered and each stream insertion to `cerr` causes its output to appear immediately. The following program uses `cerr` in conjunction with the stream insertion operator.

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main( )
6  {
7      char str[] = "Corrupt File...";
8
9      //the following line uses cerr, an instance of ostream class
10     cerr << "Error message: " << str << endl;
11 }
12

```

Figure 10.8 Use of standard error stream (cerr) in a program

The output of the `cerr` program in figure 10.5 is given in figure 10.6

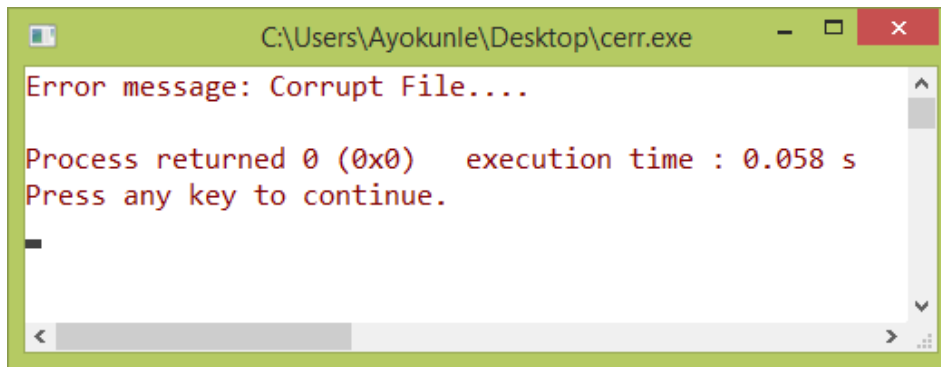


Figure 10.9 Output of the cerr program in figure 10.5

10.6 C++ Standard log stream (clog)

The predefined object **clog** is an instance of **ostream** class. The **clog** object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to **clog** could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main( )
6  {
7      char str[] = "Corrupt File...";
8
9      //the following line uses clog, an instance of ostream class
10     clog << "Error message: " << str << endl;
11 }
12

```

Figure 10.7 Use of standard error stream (clog) in a program

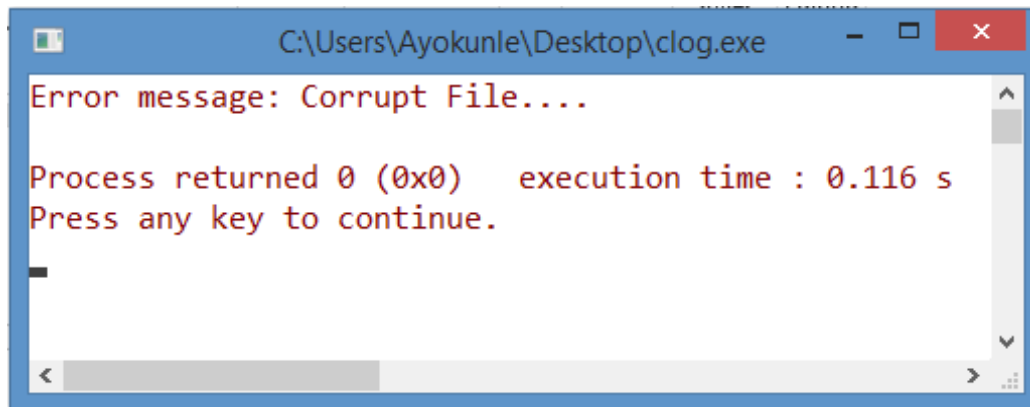


Figure 10.10 Output

End of Module Assessment

1. Write a C++ to generate the following conversation between the computer and a user
Computer: Hello. How are you doing?
Computer: What is your name?
User: My name is <user's name>
Computer: <user's name>! That is a lovely name
Computer: I know some people who bear <user's name>. They are usually nice people.
Computer: <user's name>. How old are you?
User: <user's age>.
Computer: Nice. So you'll be (<user's age> + 5) in the next five years.

MODULE 11: INPUT/OUTPUT WITH FILES

GENERAL OBJECTIVE

- To be able to process files in C++ environment

SPECIFIC OBJECTIVES

- At the end of this module, you will be able to:
 - Create, read, write and update files
 - Check state flags
 - Get and put stream pointers

To perform input and output of characters to and from file, C++ provides 3 classes that are derived from the class `istream` and `ostream`. Objects of the classes `istream` and `ostream` have already been used in module 10 (`cin` and `cout`), however the difference between the ones in module 10 and the ones we will discuss in the module is that we have to associate these streams with physical files. For example

`cout` which is used to write/display text to screen is replaced by `myfile` which is used to write into a file. Compare the following codes.

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  int main ()
5  {
6      ofstream file_mi;
7      file_mi.open ("apere.txt");
8      file_mi << "Writing this to a file.\n";
9      file_mi << "Hello, my name is Omotunde Avokunle A.\n";
10     file_mi << "This is i/o file.\n";
11     file_mi.close();
12     return 0;
13 }
```

1. Program that uses `file_mi` object.
`file_mi` is an instantiation of the class `ostream`

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  int main ()
5  {
6      cout << "Writing this to a file.\n";
7      cout << "Hello, my name is Omotunde Avokunle A.\n";
8      cout << "This is i/o file.\n";
9      return 0;
10 }
11
```

2. Program that uses `cout`

Figure 11.1 A Comparison between `cout` and `myfile` objects of the `ostream` class.

Program 1 in figure 11.1 writes strings in lines 7, 8, and 9 into a text file called `example.txt`.

When the file is opened, the following is displayed

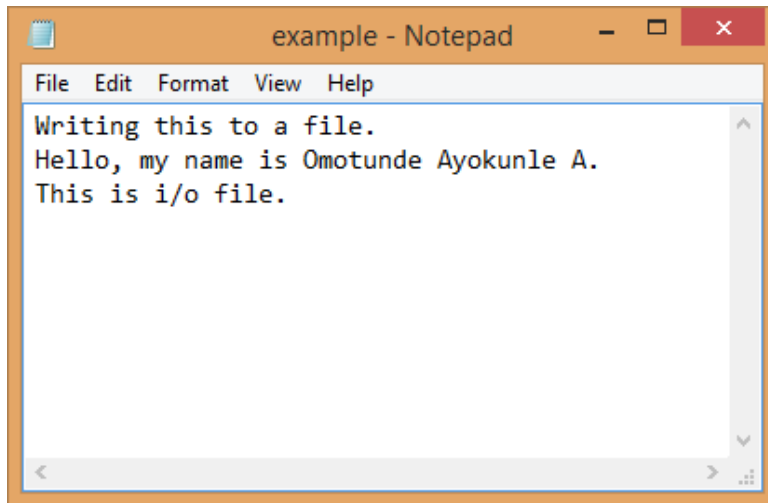


Figure 11.2 Output of program 1 in figure 11.1

Program 2 on the other hand displays the strings in lines 5, 6, and 7 to screen. The result is seen in figure 11.3.

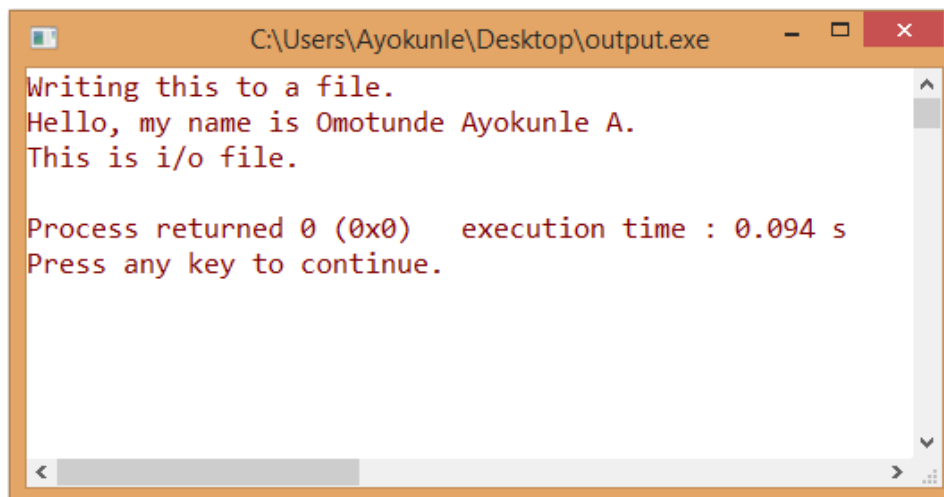


Figure 11.3 Output of program 2 in figure 11.1

Tutor Marked Assessment

1. Using `myfile` stream object, write a program that describes you and write that description into a file called `MySelf.txt`.

11.1 Opening/Creating a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to **open a file**. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the example in program 1, figure 11.1 this was done using `file_mi`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()` :

```
open(filename, mode);
```

Where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

Table 11.1 Flags used with stream objects (these will be applied to the physical file associated with it)

Flag	Description
ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file `apere.txt` in binary mode to add data we could do it by the following call to member function `open()`:

```
ofstream file_mi;
file_mi.open("apere.bin", ios::out | ios::app | ios::binary);
```

Now we will try using some of these flags for the example used in figure 11.1.

Flag `ios::out`

The following program uses the `ios::out` flag. This flag is simply used to write to file.

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  int main ()
5  {
6      ofstream file_mi;
7      file_mi.open("apere.txt", ios::out);
8      file_mi << "Writing this to a file.\n";
9      file_mi << "Hello, my name is Omotunde Ayokunle A.\n";
10     file_mi << "This is i/o file.\n";
11     file_mi.close();
12     return 0;
13 }
14
```

Figure 11.4 Program that uses the `ios::out` flag with stream object `file_mi` and filename `apere.txt` (Line 7)

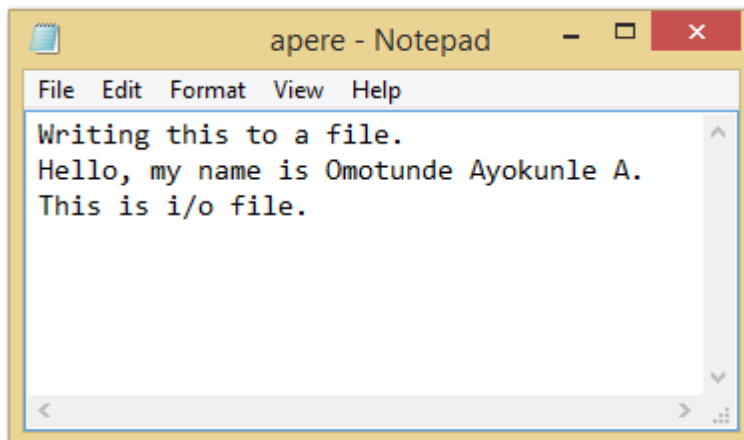


Figure 11.5 Output of the `ios::out` flag on the stream object `file_mi`

Flag `ios::binary`

The following program uses the `ios::out` flag. This flag is simply used to write to file in binary mode.


```

1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  int main ()
5  {
6      ofstream file_mi;
7      file_mi.open("apere.txt", ios::binary);
8      file_mi << "Writing this to a file.\n";
9      file_mi << "Hello, my name is Omotunde Ayokunle A.\n";
10     file_mi << "This is i/o file.\n";
11     file_mi.close();
12     return 0;
13 }
14

```

Figure 11.6 Program that uses the `ios::binary` flag with stream object `file_mi` and filename `apere.txt` (Line 7)

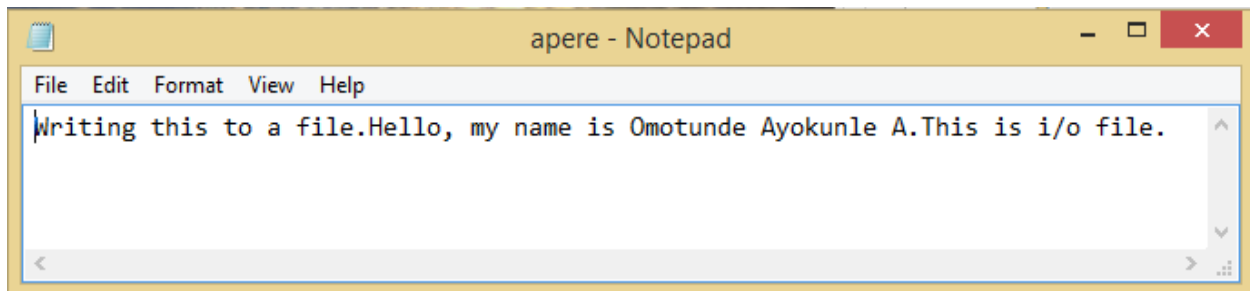


Figure 11.7 Output of the `ios::binary` flag on the stream object `file_mi`

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

Class	Default mode parameter
ofstream	<code>ios::out</code>
ifstream	<code>ios::in</code>
fstream	<code>ios::in ios::cout</code>

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

This is why the output of program 1 in figure 11.1 is the same as the output of the program in figure 11.4

Tutor Marked Assessment

1. Modify the program in figure 11.6 such that all output operations are performed at the end of the file, appending the content to the current content of the file.

11.2 Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function `close()`. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
file_mi.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`.

11.3 Text files

Text file streams are those that the `ios::binary` flag is not include in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with `cout`:

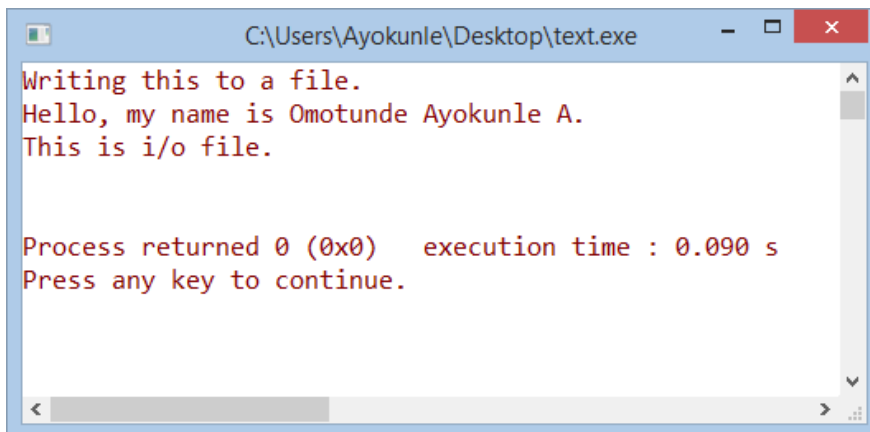
Consider the following program that read the text off a text file and displays it.

```

1  /* this program reads a text file
2  and displays it when it is run*/
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7  int main ()
8  {
9      string line;
10     ifstream file_mi ("apere.txt");
11     if (file_mi.is_open())
12     {
13         while (! file_mi.eof() )
14         {
15             getline (file_mi,line);
16             cout << line << endl;
17         }
18         file_mi.close();
19     }
20     else cout << "Unable to open file";
21     return 0;
22 }
23

```

Figure 11.8 A program that reads text from a text file.



```

C:\Users\Ayokunle\Desktop\text.exe
Writing this to a file.
Hello, my name is Omotunde Ayokunle A.
This is i/o file.

Process returned 0 (0x0)   execution time : 0.090 s
Press any key to continue.

```

Figure 11.9 Output of the program in figure 11.8

In the program in figure 11.8, the text in the file `apere.txt` are read and displayed in the output.

Now let's amend the program in Figure 8 to read from a file that doesn't exist. In line 10, we will replace the existing file (`apere.txt`) with a non-existing file (`asdf.txt`). `Apere.txt` exists because we created the file in figure 11.1 (the 1st program)

```

1  /* this program reads a text file
2  and displays it when it is run*/
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7  int main ()
8  {
9      string line;
10     ifstream file_mi ("asdf.txt");
11     if (file_mi.is_open())
12     {
13         while (! file_mi.eof() )
14         {
15             getline (file_mi,line);
16             cout << line << endl;
17         }
18         file_mi.close();
19     }
20     else cout << "Unable to open file";
21     return 0;
22 }
23

```

Figure 11.10 A program that tries to read text from a non-existing file

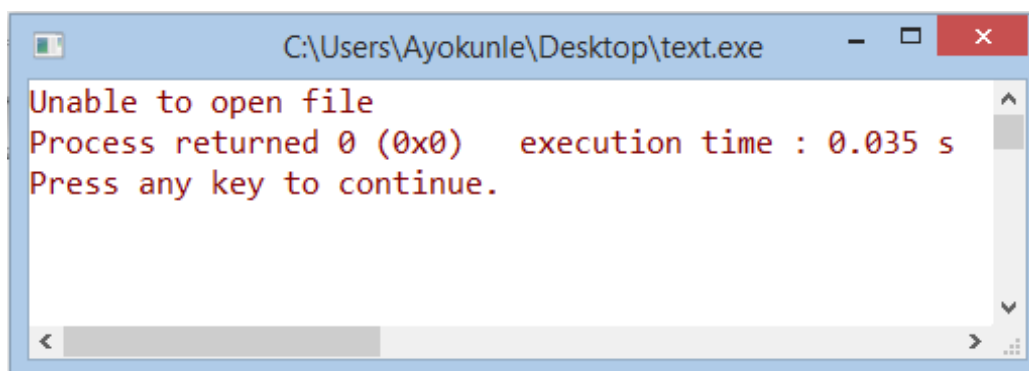


Figure 11.11

11.4 State Flags

C++ provides a set of member functions to check the state of a stream.

In addition to `eof()`, which checks if the end of file has been reached, other member functions exist to check the state of a stream (all of them return a bool value):

Member function	Description
bad()	Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()	Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
eof()	Returns true if a file open for reading has reached the end.
good()	It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would reTturn true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function clear(), which takes no parameters.

Tutor Marked Assessment

1. Modify the program in figure

End of Module Assessment

1. What is the output of the following program

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

int main()
{
    int integerValue;
    cout << "Before a bad input operation:"
         << "\n cin.rdstate(): " << cin.rdstate()
         << "\n      cin.eof(): " << cin.eof()
         << "\n      cin.fail(): " << cin.fail()
         << "\n      cin.bad(): " << cin.bad()
         << "\n      cin.good(): " << cin.good()
         << "\n\nExpects an integer, but enter a character: ";

    cin >> integerValue;
    cout << endl;

    cout << "After a bad input operation:"
         << "\n cin.rdstate(): " << cin.rdstate()
         << "\n      cin.eof(): " << cin.eof()
         << "\n      cin.fail(): " << cin.fail()
         << "\n      cin.bad(): " << cin.bad()
         << "\n      cin.good(): " << cin.good() << endl << endl;

    cin.clear();

    cout << "After cin.clear()"
         << "\n      cin.fail(): " << cin.fail()
         << "\n      cin.good(): " << cin.good() << endl;
}
```

2. Write a C++ program that opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen

Appendix A: C++ Standard Library File Header

Standard Library File Header	Function
<algorithm>	Contains functions for manipulating data in C++ Standard Library containers.
<cassert>	Contains macros for adding diagnostics that aid program debugging.
<cctype>	This defines functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa
<cfloat>	Contains the floating-point size limits of the system.
<climits>	Contains the integral size limits of the system.
<cmath>	Contains function prototypes for math library functions.
<cstdio>	Contains function prototypes for the C-style standard input/output library functions.
<cstdlib>	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions.
<cstring>	This header file defines the C-style string-processing functions.
<ctime>	Contains function prototypes and types for manipulating the time and date.
<fstream>	Contains function that declares services for user-controlled file processing.
<functional>	Contains classes and functions used by C++ Standard Library algorithms.
<iomanip>	Contains function that declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision
<iostream>	Contains function that defines the cin , cout , cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
<iterator>	Contains classes for accessing data in the C++ Standard Library containers.
<limits>	Contains classes for defining the numerical data type limits on each computer platform.
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different

	languages (e.g., monetary formats, sorting strings, character presentation, etc.).
<memory>	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers.
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory
<string>	This defines the class string from the C++ Standard Library
<typeinfo>	Defines classes for runtime type identification that is, determining data types at execution time.
<utility>	Contains classes and functions that are used by many C++ Standard Library headers.
<vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset>	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution.

Table adopted from: (Deitel & Deitel, 2012)

Appendix B: Math Library Functions

Function	Description	Example
ceil(a)	Rounds a to the smallest integer not less than a	<code>ceil(4.6) = 5.0</code> <code>ceil(-4.6) = 4.0</code>
cos(a)	Trigonometric cosine of a (a in radians)	<code>cos(0.0) = 1.0</code>
exp(a)	Exponential function e^a	<code>exp(1.0) = 2.718282</code> <code>exp(2.0) = 7.389056</code>
fabs(a)	absolute value of a	<code>fabs(6.1) = 6.1</code> <code>fabs(0.0) = 0.0</code> <code>fabs(-9.50) = 9.50</code>
floor(a)	Rounds a to the largest integer not greater than a	<code>floor(9.2) = 9.0</code> <code>floor(-9.8) = -10.0</code>
fmod(a, b)	remainder of a/b as a floating point number	<code>fmod(2.6, 1.2) = 0.2</code>
log(a)	natural logarithm of a (base e)	<code>log(2.718282) = 1.0</code> <code>log(7.389056) = 2.0</code>
log10(a)	logarithm of a (base 10)	<code>log10(10.0) = 1.0</code> <code>log10(100.0) = 2.0</code>
pow(a, b)	a raised to power b (a^b)	<code>pow(2, 7) = 128</code> <code>pow(9, .5) = 3</code>
sin(a)	trigonometric sine of a (a in radians)	<code>sin(0.0) = 0</code>
sqrt(a)	square root of a (where a is a nonnegative value)	<code>sqrt(9.0) = 3.0</code>
tan(x)	trigonometric tangent of a (a in radians)	<code>tan(0.0) = 0</code>

Table adopted from: (Deitel & Deitel, 2012)

