**Basic Computer programming course module**

(course code EcEg1033)

Developed by

Biniyam Yirsaw (Msc. Information Technology)

Compiled and edited by

January 2023

**Contents**

[1. CHAPTER ONE: BASICS OF COMPUTER AND PROGRAMMING LANGUAGE 7](#_Toc123808423)

[1.1 Computer Fundamentals 7](#_Toc123808424)

[1.1.1 Definition of Computer 7](#_Toc123808425)

[1.1.2 Computer Organization 8](#_Toc123808426)

[1.2 Computer Programming 9](#_Toc123808427)

[1.2.1 Definition of Computer Programming 9](#_Toc123808428)

[1.2.2 Types of Programming Language 9](#_Toc123808429)

[1.3 Programming Paradigms 12](#_Toc123808430)

[1.3.1 Procedural Programming Languages 12](#_Toc123808431)

[1.3.2 Structured Programming Languages 12](#_Toc123808432)

[1.3.3 Object-Oriented Programming Languages 13](#_Toc123808433)

[1.4 Software Development Method 13](#_Toc123808434)

[1.4.1. Problem Definition 14](#_Toc123808435)

[1.4.2. Requirement Analysis 14](#_Toc123808436)

[1.4.3. System Analysis 14](#_Toc123808437)

[1.4.4. Software Design 15](#_Toc123808438)

[1.4.5. Coding 15](#_Toc123808439)

[1.4.6. Testing 15](#_Toc123808440)

[1.4.7. Implementation 15](#_Toc123808441)

[1.4.8. Operation and Maintenance 15](#_Toc123808442)

[1.4.9. Documentation 16](#_Toc123808443)

[1.5. Software crisis 16](#_Toc123808444)

[2 CHAPTER TWO: ALGORITHM REPRESENTATION AND DATA STRUCTURE 17](#_Toc123808445)

[2.1. Algorithm 18](#_Toc123808446)

[2.1.1. Flow chart 19](#_Toc123808447)

[2.1.2. Pseudo code 24](#_Toc123808448)

[2.2. Data structure 26](#_Toc123808449)

[3 CHAPTER THREE : FUNDAMENTALS OF THE C++ PROGRAMMING LANGUAGE 32](#_Toc123808450)

[3.1 Structure of C++ Programs 32](#_Toc123808451)

[3.2 Compilation Process of C++ 33](#_Toc123808452)

[3.3 Error types in Programming 35](#_Toc123808453)

[3.4 Basic Elements of C++ Programming 37](#_Toc123808454)

[3.4.1 Identifiers 37](#_Toc123808455)

[3.4.2 Variables 37](#_Toc123808456)

[3.4.3 Data types 38](#_Toc123808457)

[3.4.4 Keyword 40](#_Toc123808458)

[3.4.5 Working on Variables 41](#_Toc123808459)

[3.4.6 Constants 43](#_Toc123808460)

[3.4.7 Comment in Programming 44](#_Toc123808461)

[3.4.8 Operators and Expressions 45](#_Toc123808462)

[3.4.9 Special Printing Characters. 51](#_Toc123808463)

[3.4.10 Input /Output Statements 52](#_Toc123808464)

[4 CHAPTER FOUR: CONTROL STATEMENTS IN C++ PROGRAMMING LANGUAGE 56](#_Toc123808465)

[4.1. Overview of Control Statement 56](#_Toc123808466)

[4.2. Selection Statements 58](#_Toc123808467)

[4.2.1. The if Statement 58](#_Toc123808468)

[4.2.2. The if….else Statement 61](#_Toc123808469)

[4.2.3. The if...else if Statement: 64](#_Toc123808470)

[4.2.4. Switch Statement 65](#_Toc123808471)

[4.2.5. Nested statement 70](#_Toc123808472)

[4.2.6. Nested if Statement 71](#_Toc123808473)

[4.2.7. Nested if else 72](#_Toc123808474)

[4.2.8. Nested Switch Statements 73](#_Toc123808475)

[4.3. Looping Statements 76](#_Toc123808476)

[4.3.1. The „while‟ Statement 77](#_Toc123808477)

[4.3.2. The *Do* While Loop 80](#_Toc123808478)

[4.3.3. The „for‟ Statement 81](#_Toc123808479)

[4.3.4. Nested Loop 86](#_Toc123808480)

[4.4. Other Statements 88](#_Toc123808481)

[4.4.1. The „continue‟ Statement 88](#_Toc123808482)

[4.4.2. The „break‟ Statement 89](#_Toc123808483)

[4.4.3. The „goto‟ Statement 90](#_Toc123808484)

[4.4.4. The „return‟ Statement 90](#_Toc123808485)

[5. CHAPTER FIVE : ARRAYS AND STRINGS 93](#_Toc123808486)

[5.1. Introduction 93](#_Toc123808487)

[*5.2.*  Array Definition 94](#_Toc123808488)

[5.3. Types of array 95](#_Toc123808489)

[*5.3.1.* One Dimensional Array 95](#_Toc123808490)

[*5.3.2.* Multidimensional arrays 102](#_Toc123808491)

[5.3. String 103](#_Toc123808492)

[5.3.1. String Definition 103](#_Toc123808493)

[5.3.2. String declaration and initialization 104](#_Toc123808494)

[5.3.3. String Manipulation 104](#_Toc123808495)

**Table of Figures**

Figure 1-1 software development concept .................................................................................... 14

Figure 3-1 Compilation process of c++ program .......................................................................... 35

Figure 4-1 Program flow example ................................................................................................ 58

Figure 4-2 Flow diagram for *if* statement .................................................................................. 60

Figure 4-3 Flow diagram for If else statement.............................................................................. 63

Figure 4-4 Flow diagram for the switch statement ....................................................................... 68

Figure 4-5 flow chart for loop statement ...................................................................................... 79

Figure 4-6 flow chart for do while loop ........................................................................................ 81

Figure 4-7 Flow diagram of the for statement .............................................................................. 83

Figure 5-1 Two dimensional array .............................................................................................. 104

**List of Tables**

Table 2-1 Flow chart symbols....................................................................................................... 20

Table 3-1 List of Data Types in c++ ............................................................................................. 40

Table 3-2 List of some key words used in c++ ............................................................................. 41

Table 3-3 operators and their description ..................................................................................... 48

Table 4-1 while loop trace for the example .................................................................................. 81

# 1. CHAPTER ONE: BASICS OF COMPUTER AND PROGRAMMING LANGUAGE

**Objectives**

After completing this chapter, you will be able to:

* Understand the fundamentals of computer
* Understand computer programming and it‘s different paradigms
* Steps involved in software development

### 1.1 Computer Fundamentals

##### 1.1.1 Definition of Computer

Computer is an electronic device that accepts data, performs computations, and makes logical decisions according to instructions that have been given to it; then produces meaningful information in a form that is useful to humans.

Computer is any calculating device or machine, which is electrical, mechanical or electromechanical. But that doesn‘t mean that computer performs only calculation. The name computer comes from a Latin word ―computus", meaning ―to recon‖ or ―to compute‖ and can be applied to abacus or any adding machine as to the modern computer. However, the term ―computer‖ has come to mean a special electronic device having certain definite characteristics.

Computer, in simple terms, is an electronic machine that can be instructed to accept, process, store and present data and information. It processes data (raw facts or figures) into useful information that gives meaning to users.

Storage and retrieval – unlike other machines, computer can store data for indefinite period of time and makes the data available for later use (retrieval).

* Is an Electronic data processing device which is capable of performing **Arithmetic** and **Logical** operations.
* Is an electronic device that‘s designed and organized to automatically accept and store input data, process them and produce the output that results under the direction of a detailed step-by-step stored program.

##### 1.1.2 Computer Organization

The computer system refers to the computer itself and all the components interconnected to it. Basically the computer system is categorized into two components.

1. **Hardware** – the physical architecture of the computer or the physical devices that carry out the activities of capturing, processing, storing and communicating data and information to other computer.
2. **Software** – the program or instructions that control the system or hardware.

1. **Hardware**

The computer hardware falls into four categories:

* + **Input devices**: - are device that feeds data into a computer and used to convert data into electronic machine-readable form. The devices that are commonly used to input data to the computer such as keyboard, mouse, scanner, barcode reader, microphone, digital cameras and so on.
  + **Processors**: - sometimes called CPU (Central Processing Unit) and executes program instructions and performs the computer‘s processing actions. The CPU is like the human brain and it oversee and control all of the activities of the computer. The CPU has two major components: Control Unit (CU) and Arithmetic Logic Unit (ALU). The Control Unit issues, selects and interprets program instructions and supervises that they are executed. It also manages and coordinates the entire computer system. The ALU contains the electronic circuitry that performs the two activities that underline all computing capabilities: arithmetic operations and logic operations.
  + **Storage devices**: - is hardware part of computer that retains data permanently or temporarily. There are two major categories of storage devices in a computer system: Primary storage device (RAM and ROM) and Secondary storage device (Hard disk, Floppy discs, CD and DVD discs).
  + **Output devices**: - are devices that accept result or information generated by the computer and present this information to the user in a variety of different means. I t display in human readable form in soft copy or hard copy. Examples of output devices are monitors, speakers, and printers.

1. **Software**

Software is a term for computer programs. A program is a set of instructions that enables a computer to operate or instructions that tell the computer how to perform a specific task. Computer software has two major categories:

* 1. **System software**: - it includes the computer's basic operating system and language software. The term also usually covers any software used to manage the computer and the network.
  2. **Application software**: - it allows a user to accomplish one or more specific tasks. Typical applications software includes office suites, business software, educational software, databases and computer games. The following are examples of application software:

### 1.2 Computer Programming

##### 1.2.1 Definition of Computer Programming

In order to solve a given problem, computers must be given the correct instruction about how they can solve it. The terms **computer programs**, **software programs**, or just **programs** are the instructions that tells the computer what to do. Computer requires programs to function, and a computer programs does nothing unless its instructions are executed by a CPU. **Computer programming** (often shortened to **programming** or **coding**) is the process of writing, testing, debugging/troubleshooting, and maintaining the source code of computer programs. Writing computer programs means writing instructions, that will make the computer follow and run a program based on those instructions. Each instruction is relatively simple, yet because of the computer's speed, it is able to run millions of instructions in a second. A computer program usually consists of two elements:

* Data – characteristics
* Code – action

##### 1.2.2 Types of Programming Language

A computer program (also known as source code) is often written by professionals known as **Computer Programmers** (simply programmers). Source code is written in one of programming languages. A **programming language** is an artificial language that can be used to control the behaviour of a machine, particularly a computer. Programming languages, like natural language (such as Amharic), are defined by syntactic and semantic rules which describe their structure and meaning respectively. The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics. Many programming languages have some form of written specification of their syntax and semantics; some are defined only by an official implementation. In general, programming languages allow humans to communicate instructions to machines.

A main purpose of programming languages is to provide instructions to a computer. As such, programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, computers do exactly what they are told to do, and cannot understand the code the programmer "intended" to write. So computers need to be instructed to perform all the tasks. The combination of the language definition, the program, and the program's inputs must fully specify the external behaviour that occurs when the program is executed. Computer languages have relatively few, exactly defined, rules for composition of programs, and strictly controlled vocabularies in which unknown words must be defined before they can be used.

Available programming languages come in a variety of forms and types. Thousands of different programming languages have been developed, used, and discarded. Programming languages can be divided in to two major categories: low-level and high-level languages.

***Low level languages***

Computers only understand one language and that is binary language or the language of 1s and 0s. Binary language is also known as **machine language**, one of low-level languages. In the initial years of computer programming, all the instructions were given in binary form. Although the computer easily understood these programs, it proved too difficult for a normal human being to remember all the instructions in the form of 0s and 1s. Therefore, computers remained mystery to a common person until other languages such as assembly language was developed, which were easier to learn and understand. **Assembly language** correspondences symbolic instructions and executable machine codes and was created to use letters (called mnemonics) to each machine language instructions to make it easier to remember or write. For example: ***ADD A, B*** – adds two numbers in memory location A and B

Assembly language is nothing more than a symbolic representation of machine code, which allows symbolic designation of memory locations. However, no matter how close assembly language is to machine code, computers still cannot understand it. The assembly language must be translated to machine code by a separate program called *assembler.* The machine instruction created by the assembler from the original program (source code) is called *object code.* Thus assembly languages are unique to a specific computer (machine). Assemblers are written for each unique machine language.

***High level languages***

Although programming in assembly language is not as difficult and error prone as stringing together ones and zeros, it is slow and cumbersome. In addition it is hardware specific. The lack of portability between different computers led to the development of high-level languages—so called because they permitted a programmer to ignore many low-level details of the computer's hardware. Further, it was recognized that the closer the syntax, rules, and mnemonics of the programming language could be to "natural language" the less likely it became that the programmer would inadvertently introduce errors (called "bugs") into the program. High-level languages are more English-like and, therefore, make it easier for programmers to "think" in the programming language. High-level languages also require translation to machine language before execution. This translation is accomplished by either a compiler or an interpreter. Compilers translate the entire source code program before execution. Interpreters translate source code programs one line at a time. Interpreters are more interactive than compilers. FORTRAN (FORmula TRANslator), BASIC (Bingers All Purpose Symbolic Instruction Code), PASCAL, C, C++, Java are some examples of high-level languages.

The question of which language is best is one that consumes a lot of time and energy among computer professionals. Every language has its strengths and weaknesses. For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organizing large programs. Pascal is very good for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ embodies powerful object-oriented features.

### 1.3 Programming Paradigms

As might be expected in a dynamic and evolving field, there is no single standard for classifying programming languages. Another most fundamental ways programming languages are characterized (categorized) is by programming paradigm. A **programming paradigm** provides the programmer's view of code execution. The most influential paradigms are examined in the next three sections, in approximate chronological order.

##### 1.3.1 Procedural Programming Languages

Procedural programming specifies a list of operations that the program must complete to reach the desired state. Each program has a starting state, a list of operations to complete, and an ending point. This approach is also known as imperative programming. Integral to the idea of procedural programming is the concept of a procedure call.

Procedures, also known as functions, subroutines, or methods, are small sections of code that perform a particular function. A procedure is effectively a list of computations to be carried out. Procedural programming can be compared to unstructured programming, where all of the code resides in a single large block. By splitting the programmatic tasks into small pieces, procedural programming allows a section of code to be re-used in the program without making multiple copies. It also makes it easier for programmers to understand and maintain program structure.

Two of the most popular procedural programming languages are FORTRAN and BASIC.

##### 1.3.2 Structured Programming Languages

Structured programming is a special type of procedural programming. It provides additional tools to manage the problems that larger programs were creating. Structured programming requires that programmers break program structure into small pieces of code that are easily understood. It also frowns upon the use of global variables and instead uses variables local to each subroutine. One of the well-known features of structural programming is that it does not allow the use of the GOTO statement. It is often associated with a "top-down" approach to design. The top-down approach begins with an initial overview of the system that contains minimal details about the different parts. Subsequent design iterations then add increasing detail to the components until the design is complete.

The most popular structured programming languages include C, Ada, and Pascal.

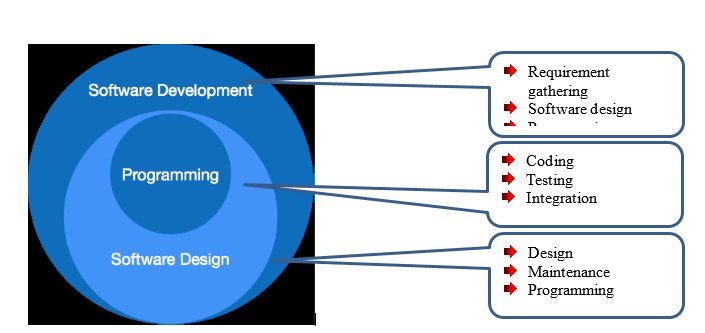
##### 1.3.3 Object-Oriented Programming Languages

Object-oriented programming is one the newest and most powerful paradigms. In object- oriented programs, the designer specifies both the data structures and the types of operations that can be applied to those data structures. This pairing of a piece of data with the operations that can be performed on it is known as an object. A program thus becomes a collection of cooperating objects, rather than a list of instructions. Objects can store state information and interact with other objects, but generally each object has a distinct, limited role.

### 1.4 Software Development Method

Software paradigms refer to the methods and steps, which are taken while designing the software. There are many methods proposed and implemented. But, we will discuss mainly the software development concept.

These can be combined into various categories, though each of them is contained in one another which are described in below figure 1.1



**Figure 1-1 software development concept**

Programming paradigm is a subset of Software design paradigm which is further a subset of Software development paradigm.

##### 1.4.1. Problem Definition

Problem definition is an explicit, written statement of a problem: the gap between the current state and desired state. Can be expressed as:

* A difficulty the users or customers are facing,
* Or as an opportunity that will result in some benefits such as improvement of productivity. The solution to the problem entails developing software. A good problem statement is short and brief.

##### 1.4.2. Requirement Analysis

This step onwards the software development team works to carry on the project. The team holds discussions with various stakeholders from problem domain and tries to bring out as much information as possible on their requirements. The requirements are includes into user requirements, system requirements and functional requirements. The requirements are collected using a number of practices as given:

 Studying the existing or obsolete system and software,

 Conducting interviews of users and developers,

 Referring to the database or

 Collecting answers from the questionnaires.

##### 1.4.3. System Analysis

In this step the developers decide a roadmap of their plan and try to bring up the best software model suitable for the project. System analysis includes understanding of software product limitations, learning system related problems or changes to be done in existing systems beforehand, identifying and addressing the impact of project on organization and personnel etc. The project team analyses the scope of the project and plans the schedule and resources accordingly.

##### 1.4.4. Software Design

Next step is to bring down whole knowledge of requirements and analysis on the desk and design the software product. The inputs from users and information gathered in requirement gathering phase are the inputs of this step. The output of this step comes in the form of two designs; logical design, and physical design. Engineers produce meta-data and data dictionaries, logical diagrams, data-flow diagrams, and in some cases pseudo codes.

##### 1.4.5. Coding

This step is known as programming phase. The implementation of software design starts in terms of writing program code in the suitable programming language and developing error-free executable programs efficiently.

##### 1.4.6. Testing

Software testing is done while coding by the developers and thorough testing is conducted by testing experts at various levels of code such as module testing, program testing, product testing, in-house testing, and testing the product at user‘s end. Early discovery of errors and their remedy is the key to reliable software.

##### 1.4.7. Implementation

This means installing the software on user machines. At times, software needs post-installation configurations at user end. Software is tested for portability and adaptability and integration related issues are solved during implementation.

##### 1.4.8. Operation and Maintenance

This phase confirms the software operation in terms of more efficiency and less errors. If required, the users are trained on, or aided with the documentation on how to operate the software and how to keep the software operational. The software is maintained timely by updating the code according to the changes taking place in user end environment or technology.

This phase may face challenges from hidden bugs and real-world unidentified problems.

##### 1.4.9. Documentation

A well written document provides a great tool and means of information repository necessary to know about software process. Software documentation also provides information about how to use the product.

### 1.5. Software crisis

It is mismatch between what software can deliver and the capacities of computer systems, as well as expectations of their users. Many software projects failed, late, over budget, providing unreliable software that is expensive to maintain. Many software project produced software which did not satisfy the requirement of customer. This all the above attributes are called software crisis. Generally software crisis implies that a problem associated with software development.

The cause of software crises were linked to the overall complexity of hardware and software development process. The crises manifested in several ways:

* Project running over budget
* Project running over time
* Software was very insufficient
* Software was of low quality
* Software project did not meet requirement
* Project were unmanageable and difficult to maintain  Software was never delivered and so on.

**Summary**

* Computer is an electronic device that accepts data, performs computations, and makes logical decisions according to instructions that have been given to it.
* Basically computer is composed of two parts: hardware and software .
* Hardware the physical architecture of the computer while Software is the program or instructions that control the system or hardware.
* The terms **computer programs**, are the instructions that tells the computer what to do.
* A **programming language** is an artificial language that can be used to control the behaviour of a machine, particularly a computer.
* Programming languages can be divided in to two major categories: low-level and highlevel languages.
* A **programming paradigm** provides the programmer's view of code execution. The most influential paradigms are procedural, structural and object-oriented programming paradigms.
* In developing a software we will have problem definition, analysis , design , coding , testing , implementation and maintenance steps.
* While developing a software , there may be a mismatch between in what the software can deliver and the capacity of computer system as well as expectation of the user. Such kind of mismatch is called software crisis.

**Exercises**

1. Explain about the organization of a computer.
2. Discus about software and hardware.
3. What do we mean by computer programing and programming language?
4. Discuses about what programing paradigm means and explains its types.
5. Discus how computer can understand programs.
6. List and discuss Software development method sequentially.
7. What did you the main reason that lead to software crisis?

# 2 CHAPTER TWO: ALGORITHM REPRESENTATION AND DATA STRUCTURE

**Objectives**

After completing this chapter, you will be able to:

* Understand the basics and usefulness of an algorithm,
* Analyse various algorithms,
* Understand a flowchart and its advantages and limitations,
* Steps involved in design algorithm

### 2.1. Algorithm

What are algorithms? Why is the study of algorithms useful? What is the role of algorithms relative to other technologies used in computers? In this chapter, we will answer these questions.  **Definition and characteristics of Algorithm**

**Definition:**

*Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output[1].*

*Formally: An Algorithm is a method of representing the step-by-step procedure for solving a problem. It is a method of finding the right answer to a problem or a different problem by breaking the problem into simple cases[2].*

 **Characteristics of algorithm**

An algorithm can be as

 *Finiteness:* An algorithm should terminate in a finite number of steps.

 *Definiteness*: Each step of the algorithm must be precisely (clearly) stated.

 *Effectiveness:* Each step must be effective. i.e.; it should be easily convertible into a program statement and can be performed exactly in a finite amount of time.

 *Generality:* The algorithm should be complete in itself so that it can be used to solve all problems of a given type for any input data.

 *Input/output:* Each algorithm must take zero, one or more quantities as input data, and give one or more output values. An algorithm can be written in English-like sentences or any standard.

Thus, these are the characteristics that an algorithm should have for its fruitfulness.

 **Algorithm representation techniques**

Design of Algorithms: this consists of the steps a programmer should do before they start coding the program in a specific language. Proper program design helps other programmers to maintain the program in the future. There are different ways to represent an algorithm such as programming language, Natural Language, Pseudo code, and flowchart. The most common representations are flow chart and pseudo code

##### 2.1.1. Flow chart

A flowchart is a graphical or symbolic representation of an algorithm. It is the diagrammatic representation of the step-by-step solution to a given problem.

In computing, there are dozens of different symbols used in flowcharting (there are even national and international flowcharting symbol standards). Basic symbols commonly used in flowcharting of programs are Terminal, Process, and input/output, Decision, Connector, and Predefined Process.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Symbol*** | | | |  | | ***Name*** | ***Representation*** | ***Function*** |
|  | | | | *The most commonly used flow chart* | | | | |
|  | | | |  | | Oval | Start/end | Used to denote start/begun and end/exit of the program |
|  | | | |  | | Parallelogram | Input /Output or Data | Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results |
|  | | | |  | | Diamond | Decision | Used to ask a question that can be answered in a binary format (Yes/No, True/False) |
|  | | | |  | |  | Process | Indicates any type of internal operation inside the Processor |
|  |  | | |
|  | | | | | | Rectangle |  | or Memory |
|  | | | | | | Arrow | Flow Lines | Shows direction of flow |
|  | | | | | |  | Connector | Allows the flowchart to be drawn without intersecting lines or a reverse flow. |
| *Additional flow chart symbols* | | | | | | | |  |
|  | | | | | | Predefined process | |  |
|  | |  |  | |  | Internal storage | |  |
|  |  | |
|  | | |
|  | | | | | | Document | |  |
|  | | | | | | Multi-document | |  |
|  | | | | | | Or operation | |  |
|  | | | | | | Stored data | |  |

**Table 2-1 Flow chart symbols**

**General Rules to draw a flowchart**

* *All boxes of the flowchart are connected with Arrows. (Not lines)*
* *Flowchart symbols have an entry point on the top of the symbol with no other entry points. The exit point for all flowchart symbols is on the bottom except for the Decision symbol.*
* *The Decision symbol has two exit points; these can be on the sides of the bottom and one side.*
* *All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.*
* *All flowcharts end with a terminal or a contentious loop.*

|  |  |
| --- | --- |
|  | ***Practical Exercise to develop a flow chart*** |

1. Draw a flowchart for an algorithm that gets two numbers and prints the sum of their value



Start



Read X,Y



Z



X+Y



Print Z



End

1. Write a flowchart for finding the greater number between two numbers.



False



T

rue



Print

B



Start



Read A,B



If A>B



Print A



End

1. Flowchart for the problem of printing even numbers between 5 and 20



E <= 20



E



E+2



Start



E



6



Write E



End



No

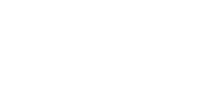


Yes

1. Draw a flowchart to determine a student‘s final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.



Start



Accept

M1,M2,M3,M5



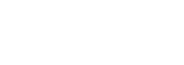
Grade



(

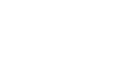
4

M1+M2+M3+M4)/



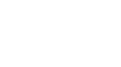
Is Grade<50

?



Print

―Fail‖



Print

―Pass‖



Stop

**Advantages of Using Flowcharts:**

The *benefits of flowcharts* are as follows:

* + Communication: Flowcharts are a better way of communicating the logic of a system to all concerned.
  + Effective analysis: With the help of a flowchart, the problem can be analysed more effectively.
  + Proper documentation: Program flowcharts serve as good program documentation, which is needed for various purposes.
  + Efficient Coding: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
  + Proper Debugging: The flowchart helps in debugging process.
  + Efficient Program Maintenance: The maintenance of the operating program becomes easy with the help of a flowchart. It helps the programmer to put effort more efficiently into that part.

**Limitations of Using Flowcharts:**

Although a flowchart is a very useful tool, there are a few limitations in using flowcharts which are listed below:

* Complex logic: Sometimes, the program logic is quite complicated. In that case, the flowchart becomes complex and awkward.
* Alterations and Modifications: If alterations are required the flowchart may require redrawing completely.
* Reproduction: As the flowchart symbols cannot be typed, reproduction of the flowchart becomes a problem.
* The essentials of what is done can easily be lost in the technical details of how it is done.

**When to Use a Flowchart:**

* A flowchart is generally used when a new project begins to be a plan for the project.
* A flowchart helps to clarify how things are currently working and how they could be improved. It also assists in finding the key elements of a process, while drawing clear lines between where one process ends and the next one starts.
* Developing a flowchart stimulates communication among participants and establishes a common understanding of the process.
* Flowcharts are used to help team members, identify who provides inputs or resources to whom, establish important areas for monitoring or data collection identify areas for improvement or increased efficiency, and generate hypotheses about causes.
* It is recommended that flowcharts be created through group discussion, as individuals rarely know the entire process and the communication contributes to improvement.

**Exercise 2.1(writing a flow chart)**

* 1. Write an algorithm that reads three numbers and prints the value of the largest number.
  2. Draw a flowchart to read an employee name (NAME), overtime hours worked (OVERTIME), hours absent (ABSENT), and determine the bonus payment (PAYMENT).
  3. Write a flowchart that finds the sum, average, and product of the 3 numbers given by the user.
  4. Draw a flowchart to find the sum of the first 10 natural numbers.
  5. Draw flowchart to find Average of 10 Numbers
  6. Design a flowchart to convert a decimal number, n, to binary format?
  7. Draw a flowchart for the problem of determining prime numbers?

##### 2.1.2. Pseudo code

Pseudo codeis one of the methods that can be used to write an initial plan that can be developed into a computer program. Itis a generic way of describing an algorithm without the use of any specific programming language syntax. It is, as the name suggests, *pseudo* code: which means fake code, it cannot be executed on a real computer, but it models and resembles real programming code, and is written in English at roughly the same level of detail. Pseudo code cannot be compiled nor executed, and there are no real formatting or syntax rules.

**How to Write Pseudo code Statements? There are six basic guidelines to develop pseudo code**

1. A computer can receive information

Read (information from a file)

Get (information from the keyboard)

1. A computer can put out information

Write (information to a file)

Display (information to the screen)

1. A computer can perform arithmetic. Use actual mathematical symbols or the words for the symbols

Add a number to the total

Total = total + number

+, -, \*, / Calculate, Compute also used

1. A computer can assign a value to a piece of data. There are three cases to assign value.
   1. to give data an initial value: Initialize, Set
   2. to assign a value as a result of some processing “=”: x=5+y
   3. to keep a piece of information for later use: Save, Store
2. A computer can compare two pieces of information and select one of two alternative actions

IF condition THEN

Some action

ELSE

Alternative action

ENDIF

1. A computer can repeat a group of actions

WHILE condition (is true) Some action

AND WHILE

…………………………………………………………………………………

FOR several times Some action

END

**Examples of Pseudo code**

1. Write an algorithm that obtains two integer numbers from the user. It will print out the sum of those numbers using pseudo code.

Step1: Start

Step2: Get the first Number

Step 3: Get the second Number

Step 4: Compute SUM of two Numbers

Step 5: Display Sum

Step 6: Step

1. Find the average of any three numbers.

Step1: Start

Step2: Read values of A, B, C

Step3: S=A+B+C

Step4: X=S/3

Step5: Write the value of X

Step6: Stop

1. Finding square and cube two numbers algorithm using pseudocode.

Step1: Start

Step2: Read value of N

Step3: S=N\*N

Step4: C=S\*N

Step5: Write values of S, C

Step6: Stop

**Exercise 2.2 (Writing pseudo code)**

1. Write pseudo-code for the following problems

1. S=(A+B+C)/Y
2. Convert from Celsius to Fahrenheit.

i. (Multiply by 9, then divide by 5, then add 32 )

1. Area of Circle()

### 2.2. Data structure

A Data Structure is a way to store and organize data so that it can be used efficiently. The data structure name indicates that the data is being stored in memory. Data Structures are widely used in almost every aspect of Computer Science i.e. operating Systems, Compiler Design, Artificial intelligence, Graphics, and many more.  **Need of data structure**

**Processor speed:** Data is growing day by day to the billions of files per entity, the processor may not be able to deal with that much data in a very fast enough way.

**Data Search:** Consider an inventory size of 106 items in a store; if our application needs to search for a particular item, it needs to traverse 106 items every time, which results in slowing down the search process.

**Multiple requests:** If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process to solve the above problems, data structures are used.

**Operations on data structure**

 **Traversing:** Every data structure contains a set of data elements. Traversing the data means visiting each element of the data structure. If we want to calculate the average of a student's marks in 6 subjects, we need to traverse the complete array of marks and calculate the total sum.

 **Insertion:** Insertion can be defined as the process of adding elements to the data structure at any location. If the size of the data structure is **n** then we can only insert **n-1** data elements into it.

 **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location. If we try to delete an element from an empty data structure then **underflow** occurs.

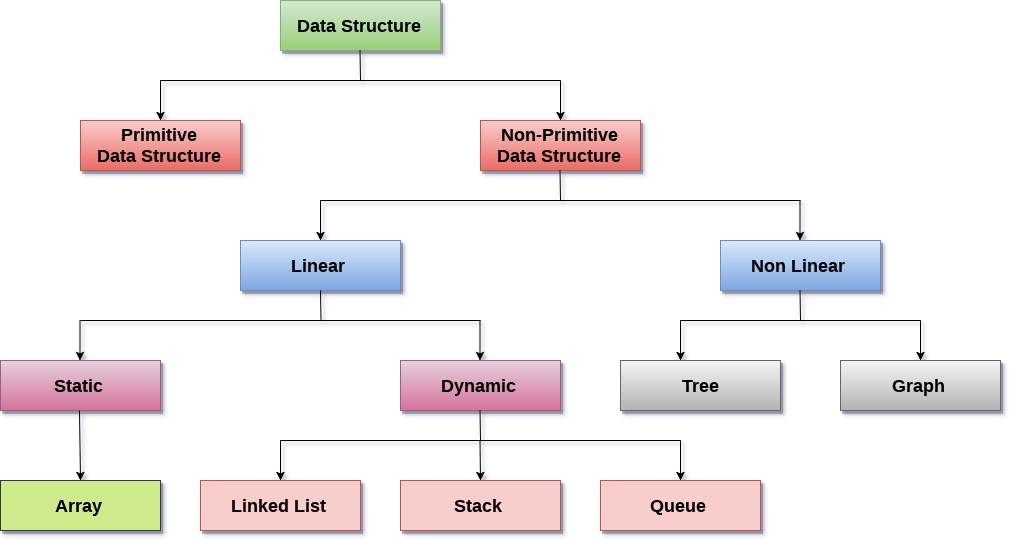
 **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

 **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. Many algorithms can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

 **Merging:** When two lists List A and List B of size M and N respectively, of similar types of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

**Types of data structure**

The primitive data structure is a fundamental type of data structure that stores the data of only one type whereas the non-primitive data structure is a type of data structure that is user-defined and stores the data of different types in a single entity.



**Linear Data Structures:** A data structure is called linear if all of its elements are arranged in linear order. In linear data structures, the elements are stored in a non-hierarchical way where each element has successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

* **Arrays:** It is a collection of similar types of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float, or double. The individual elements of the array age are: age[0], age[1], age[2], age[3],......... age[98], age[99].
* **Linked List:** is a linear data structure that is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.
* **Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.
* **The queue** is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called a **front**. The queue is opened at both ends therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

**Non-Linear Data Structures:** This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non-Linear Data Structures are given below:

* **Trees:** are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf nodes** while the topmost node is called the **root node**
* **Graphs:** can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges.

\

**Summary**



*The a*

*lgorithm*

is the sequence of steps to be performed to solve a problem by the

computer.

 Three reasons for using algorithms are *efficiency, abstraction,* and *reusability.*

 Algorithms can be expressed in many *different notations*,including natural languages, pseudo code, flowcharts, and programming languages.

 The practical goal of *algorithm analysis* is to predict the performance of different algorithms in order to guide program design decisions.

 *Flowchart* is a graphical or symbolic representation of an algorithm. It is the diagrammatic representation of the step-by step solution to a given problem.

 *Flowcharts* are used in analysing, designing, documenting or managing a process or program in various fields.

 *Benefits of using flowcharts* include ease of communication, effective and efficient analysis and coding, proper documentation and maintenance.

 *Limitations of using flowcharts* include complex logic and multiple modifications.

**Exercises**

Writing programs that solve the Programming Projects helps to get hard your understanding of the material and demonstrates how the chapter‘s concepts are applied.

1. Write an algorithm to find the sum of the first 50 natural numbers and also draw the corresponding flowchart.
2. Write an algorithm to read a number N from the user and print all its divisors.
3. Write an algorithm to find the sum of given N numbers and also draw the corresponding flowchart.
4. Draw a flow chart to log in to a Facebook account.
5. Write an algorithm to compute the sum of the squares of integers from 1 to 50 and also draw the corresponding flowchart.
6. Draw a flowchart explaining the process of waking up in the morning. [Hint: Use steps Alarm Ringing, Ready to get up, climbing out of bed]
7. Write and algorithm that prints ―Hello Engineering‖ five times and draw flow chart. 8) Describe the following flow chart by pseudo code. What it do?



Start



In X



Sum=0



X>=0



Out: sum



End



Sum=sum+X



X= X

-

1

# 3 CHAPTER THREE : FUNDAMENTALS OF THE C++ PROGRAMMING LANGUAGE

**Objectives**

After completing this chapter, you will be able to:

* Understand the basics of C++ programming language
* Identify the basic elements of C++ programming language
* Working with C++ programming language

### 3.1 Structure of C++ Programs

To understand the structure of a simple program in C++, let‘s have a look at the following code:

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

cout<<”\n Hello World!”; return 0;

}

Any C++ program file should be saved with file name extension ― .CPP ‖ . Type the program directly into the editor (such as Dev-C++, Code:: Blocks , etc.), and ***save the file as*** **filename.cpp**, ***compile it*** and ***then run it***. It will print the words ***Hello World!*** on the computer screen.

The first character is the **#**. This character is a signal to the ***pre-processor***. ***Each time you start your compiler, the pre-processor*** runs through the program and looks for the pound (#) symbols and act on those lines before the compiler runs.

The **include** instruction is a pre-processor ***instruction that directs the compiler to include*** a copy of the file specified in the angle brackets in the source code.

If the path of the file is not specified, the pre-processor looks for the file under ***c:\include\*** folder or in include folder of the location where the editor is stored.

The effects of **line 1**, i.e. ***include<iostream>*** is to include the file ***iostream*** into the program as if the programmer had actually typed it.

The effects of **line 2**, i.e ***using namespace std;*** is to define namespace standard used in this program.

When the program starts, **main()** is called automatically. Every C++ program has a **main()** function. The return value type for ***main()*** here is **int**, which means main function will return a value to the caller (which is the operating system). The main function can be made to return a value to the operating system.

The ***Left French brace*** **“{“**signals the beginning of the main function body and the corresponding ***Right French brace*** **“}”** signals the end of the main function body. Every Left French Brace needs to have a corresponding Right French Brace. The lines we find between ***the braces*** are ***statements*** or **said to be the body of the main function**. The ***statement found between the opening and closing brace or the body of the main function*** is a ***computation step*** which *may produce a value or interact with input and output streams*. The end of a single statement ends with semicolon **(;)**. The statement in the above example causes the string ―Hello World!‖ to be sent to the ―cout‖ stream which will display it on the computer screen.

### 3.2 Compilation Process of C++

**What is a compilation?**

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

Any program written in a language other than machine language needs to be translated to machine language. The set of instructions that do this task are known as translators. There are different kinds of translator software, among which compilers and interpreters are of interest for most programmers. There are different kinds of translator software, among which compilers and interpreters are of interest for most programmers.

**Compilers**: a compiler is a computer program that translates a series of statements written in source code (a collection of statements in a specific programming language) into a resulting object code (translated instructions of the statements in a programming language). A compiler changes or translates the whole source code into executable machine code (also called object code) which is output to a file for latter execution. E.g. C++, Pascal, FORTRAN, etc.

**Interpreters**: is a computer program that translates a single high level statement and executes it and then goes to the next high level language line etc. E.g. QBASIC, Lisp etc.

It is fundamental to know how C++ compilation works to understand how programs are compiled and executed. C++ programs typically go through five edit, pre-process, compile, link, and load.

**Edit:** this is accomplished with an editor program. The programmer types C++ statements with the editor and makes corrections if necessary. The programs source file is then stored on secondary storage device such as a disk with a ―.cpp‖ file name.

After the program is edited, C++ is principally compiled in three phases: pre-processing, translation to object code, and linking (the last two phases are what is generally thought of as the "compilation" process).

**Pre-process**: In a C++ system, a pre-processor program executes automatically before the compiler‘s translation phase begins. The C++ pre-processor obeys command called preprocessor directives, which indicate that certain manipulations are to be performed on the program before compilation. The pre-processor is invoked by the compiler before the program is converted to machine language. The C++ pre-processor goes over the program text and carries out the instructions specified by the pre-processor directives (e.g., **#include**). The result is a modified program text which no longer contains any directives.

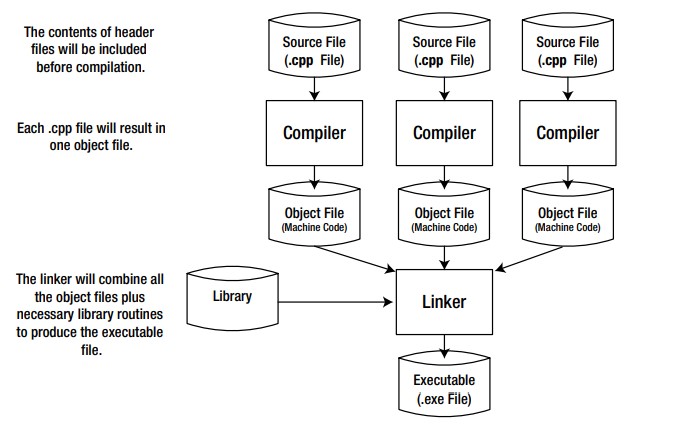
**Compile:** Then, the C++ compiler translates the program code. The compiler may be a true C++ compiler which generates native (assembly or machine) code. The outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program.

For example, the **<<** operator which is actually defined in a separate IO library.

**Linking**: C++ programs typically contain references to functions and data defined elsewhere, such as in the standard libraries. The object code produced by the C++ compiler typically contains ―holes‖ due to these missing parts. A linker links the object code with the code for the missing function to produce an executable image (with no missing pieces). Generally, the linker completes the object code by linking it with the object code of any library modules that the program may have referred to. The final result is an executable file.

**Loading:** the loader takes the executable file from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded. Finally, the computer, under the control of its CPU, executes the program.

In practice all these steps are usually invoked by a single command and the user will not even see the intermediate files generated.



**Figure 3-1 Compilation process of c++ program**

### 3.3 Error types in Programming

When developing programs there are three types of error that can occur:

1. **Syntax errors:** errors due to the fact that the syntax of the language is not respected.
2. **Semantic errors:** errors due to an improper use of program statements.
3. **Logical errors**: errors due to the fact that the specification is not respected.

From the point of view of when errors are detected, we distinguish:

1. **Compile time errors**: syntax errors and static semantic errors indicated by the compiler.
2. **Runtime errors**: dynamic semantic errors, and logical errors, that cannot be detected by the compiler (debugging).

**Syntax errors/ Compile time errors**

A syntax error occurs when the code given does not follow the syntax rules of the programming language.

Examples of syntax errors include:

* ***misspelling a statement***, e.g. writing pint instead of print using a variable before it has been declared
* ***missing brackets***, e.g. opening a bracket, but not closing it

A program cannot run if it has syntax errors. Any such errors must be fixed first. A good integrated development environment (IDE) usually points out any syntax errors to the programmer.

*Example 1:* Missing semicolon:

int a = 5 // semicolon is missing

*Example 2:* Errors in bracket or expressions:

x = (3 + 5; // missing closing parenthesis ')' y = 3 + \* 5; // missing argument between '+' and '\*' *Example 3:* using a variable before it has been declared:

*The lecturer should explain this under variable declaration and usage topic*

**Runtime/Logical errors**

A logic error is an error in the way a program works. The program can run but does not do what it is expected to do.

Logic errors can be caused by the programmer:

* incorrectly using logical operators, e.g. expecting a program to stop when the value of a variable reaches 5, but using <5 instead of <=5
* incorrectly using Boolean operators
* unintentionally creating a situation where an infinite loop may occur
* incorrectly using brackets in calculations
* unintentionally using the same variable name at different points in the program for different purposes
* using incorrect program design

Unlike a syntax error, a logic error does not usually stop a program from running. The program will run, but not function as expected. This type of error is not caught during compilation, but causes an exception to be thrown at runtime.

### 3.4 Basic Elements of C++ Programming

##### 3.4.1 Identifiers

It is a **unique name** which is given to an entity to distinctly identify it meanwhile the execution of the source-code. Both an identifier and a variable are the names allotted by users to a particular entity in a program. The identifier is only used to identify an entity uniquely in a program at the time of execution whereas, a variable is a name given to a memory location that is used to hold a value.

A valid identifier is a sequence of one or more letters, digits or underlined symbols. The length of an identifier is not limited. Neither space nor marked letters can be part of an identifier.

##### 3.4.2 Variables

A Variable is a name that is assigned to a memory location, which is used to contain the corresponding value in it. Identifiers are not variables, nor are variables identifiers.

* A variable is a reserved place in memory to store information in.
* A variable will have three components:
* Variables are used for holding data values so that they can be used in various computations in a program.
* All variables have three important properties:
  + **Data Type**: a type which is established when the variable is defined. (E.g. integer, real, character etc.). Data type describes the property of the data and the size of the reserved memory
  + **Name**: a name which will be used to refer to the value in the variable. A unique identifier for the reserved memory location. o **Value:** a value which can be changed by assigning a new value to the variable.

##### 3.4.3 Data types

Data types define the type of data a variable can hold, for example an integer variable can hold integer data, a character type variable can hold character data etc.

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.

Data types in C++ are categorised in three groups:

1. **Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. Example: int, char , float, bool etc.

Primitive data types available in C++ are:

* + Integer : variable data‘s can be expressed as negative or positive integer num
  + Character : variable data‘s can be expressed using single quotes
  + Boolean : variable data‘s can be expressed as true or false
  + Floating Point : variable data‘s can be expressed as fraction
  + Double Floating Point : variable data‘s can be expressed as fraction
  + Valueless or Void
  + Wide Character and string

1. [**Derived Data Types:**](https://www.geeksforgeeks.org/derived-data-types-in-c/) The data-types that are derived from the primitive or built-in data types are referred to as Derived Data Types. These can be of four types namely:
   * Function
   * Array
   * Pointer
   * Reference
2. [**Abstract or User-Defined Data Types**:](https://www.geeksforgeeks.org/user-defined-derived-data-types-in-c/) These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined data types:
   * Class
   * Structure
   * Union
   * Enumeration
   * Typedef defined DataType

In this course we will discuss all **primitive data types and from derived data types we will discuss array.**

The following table shows the data type, how much memory it takes to store the value in memory, and what is the maximum and minimum value which can be stored in such type of variables. (Here: remember the definition of variables)

|  |  |  |
| --- | --- | --- |
| **Type** | **Typical Bit Width** | **Typical Range** |
| Char | 1byte | -127 to 127 or 0 to 255 |
| unsigned char | 1byte | 0 to 255 |
| signed char | 1byte | -127 to 127 |
| Int | 4bytes | -2147483648 to 2147483647 |
| unsigned int | 4bytes | 0 to 4294967295 |
| signed int | 4bytes | -2147483648 to 2147483647 |
| short int | 2bytes | -32768 to 32767 |
| unsigned short int | 2bytes | 0 to 65,535 |
| signed short int | 2bytes | -32768 to 32767 |
| long int | 8bytes | -2,147,483,648 to 2,147,483,647 |
| signed long int | 8bytes | same as long int |
| unsigned long int | 8bytes | 0 to 4,294,967,295 |
| long long int | 8bytes | -(2^63) to (2^63)-1 |
| unsigned long long int | 8bytes | 0 to  18,446,744,073,709,551,615 |
| float | 4bytes |  |
| double | 8bytes |  |
| long double | 12bytes |  |
| wchar\_t | 2 or 4 bytes | 1 wide character |

**Table 3-1 List of Data Types in c++**

##### 3.4.4 Keyword

Keywords are the reserved keywords that are defined by the compiler to perform the internal operation, written in lowercase.

Keywords have some meaning which is defined by the compiler to accomplish a task in code; they cannot be used as a variable in programming. C++ provides different keywords such as– ***int,*** ***float***, ***double***, ***char***, **main**, ***bool*** etc. Primitive data types are key words used to define or declare a variable value. The following table indicates some of the key words used in c++ programming

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| auto | break | case | char | const | Continue | default | Do |
| double | else | enum | extern | float | For | goto | If |
| int | long | register | return | short | Signed | sizeof | Static |
| struct | switch | typedef | union | unsigned | Void | volatile | While |

**Table 3-2 List of some key words used in c++**

##### 3.4.5 Working on Variables

*3.4.5.1 Declaring Variables*

Variables can be created in a process known as declaration/definition. A variable definition tells the compiler where and how much storage to create for the variable. A typical variable declaration takes the following form or syntax:

Syntax: Datatype *Variable\_Name*;

Example 1: **int** ***myage***; (i.e. The [**Datatype**:](https://www.geeksforgeeks.org/c-data-types/) indicate type of data that can be stored in this variable is ***integer***; and ***Variable\_Name***: Name given to the variable is ***myage***.

Example 2: Declaring float variable

float *simpleInterest*;

Example 3: Declaring multiple variables

Syntax: Datatype *Variable\_Name,* *variable2\_name,* *variable3\_name*; int myage,myphonenumber,weight;

**Consideration should be taken while declaring variable**

* Good variable names indicate the purpose of the variable or they should be selfdescriptive.
* The name of a variable sometimes is called an identifier which should be unique in a program.
* Certain words are reserved by C++ for specific purposes and cannot be used as identifiers.
* A variable name can consist of alphabets (both upper and lower case), numbers and the underscore ‗\_‘ character. However, the name must not start with a number.
* C++ is not case sensitive. Small letter and capital letters are different for C++. E.g.:

variable **Age** is not identical with variable **age**

**Difference between variable declaration and definition**

The **variable declaration** refers to the part where a variable is first declared or introduced before its first use. A **variable definition** is a part where the variable is assigned a memory location and a value. Most of the times, variable declaration and definition are done together. See the following example 4 for better clarification:

Example 4 **: int myage;** // variable declaration **int myage=30;** // variable definition

*3.4.5.2 Initializing Variables*

When a variable is assigned a value at the time of declaration, it is called variable initialization.

**The syntax**: **DataType** ***variable name*** **=** ***initial value***;

This is identical with declaring a variable and then assigning a value to the variable immediately after declaration. (i.e. **The syntax**: **DataType** ***variable name*** ; ***variable name*** **=** ***initial value***; ) See the following example 5 for better clarification:

E.g. int a = 0; or: int a; a=0; ***Example 6: How define / Initilize / assign value for variable during declaration***

int myNum = 5; // Integer (whole number without decimals)

double myFloatNum = 5.99;// Floating point number (with decimals)

char myLetter = 'D'; // Character string myText = "Hello"; // String (text) bool myBoolean = true; // Boolean (true or false)

*3.4.5.3* Scope *of Variables*

Scope of a variable is the boundary or block in a program where a variable can be accessed. The boundary or block is identified by the left and right French brackets. In C++, we can declare variables anywhere in the source code. But we should declare a variable before using it no matter where it is written. There are mainly two types of variable scopes: Local Variables and Global Variables.

Example 1 : Simple c++ program to declare variable as local variable and global variable

1: #include<iostream>

2: using namespace std;

3: int global\_variable;

4: int main()

5: {

6: int local\_variable;

7: return 0;

8: }

**Local Variables**

* The scope of the local variable is limited to the code level or block within which they are declared.
* Anything between ‗{‗ and ‗}‘ is said to inside a block.
* Local variables do not exist outside the block in which they are declared, i.e. they **can not** be accessed or used outside that block.
* As illustrated example 1 : line 6 : the scope of the variable is local; int local\_variable

**Global variables**:

* There are variables that can be referred/accessed anywhere in the code, within any function, as long as it is declared first.
* A variable declared before any function immediately after the include statements are global variables
* As illustrated example 1 : line 3 : the scope of the variable is global; int global\_variable

##### 3.4.6 Constants

A constant is any expression that has a fixed value. A constant value is the one which does not change during the execution of a program. Constants can be of any of the basic data types. The way each constant is represented depends upon its type. Constants are also called **literals**.

Like variables, constants are data storage locations in the computer memory. But, constants, unlike variables their content cannot be changed after the declaration.

Constants must be initialized when they are created by the program, and the programmer can‘t assign a new value to a constant later. In C++ two types of constants can be implemented: literal and symbolic constants.

**Literal constant**: is a value typed directly into the program wherever it is needed.

Example 1: int num = 43;

43 is a literal constant in this statement:

**Symbolic constant**: is a constant that is represented by a name, similar to that of a variable. But unlike a variable, its value can‘t be changed after initialization.

In C++, we have two ways to declare a symbolic constant. These are using the **#define** and the **const** key word.

***Using #define pre-processor directive:***

* This directive is used to declare an alias name for existing variable or any value.
* The #define directive makes a simple text substitution.
* The define directive can define only integer constants

Syntax: #define identifierName value

Example 2: #define age 30

* In our example, each time the pre-processor sees the word **age**, it inserts 30 into the text.

***Using const key word***

Here, the constant has a type, and the compiler can ensure that the constant is used according to the rules for that type.

Syntax: **const** identifierName value;

Example 2: **const** age 30;

##### 3.4.7 Comment in Programming

* A comment is a piece of descriptive text which explains some aspect of a program.
* Comments are normally used to annotate code for future reference.
* Program comments are text totally ignored by the compiler and are only intended to inform the reader how the source code is working at any particular point in the program.
* Comments can be used to explain C++ code, and to make it more readable.
* Over-use of comments can lead to even less readability.
* A program which contains so much comment that you can hardly see the code can by no means be considered readable.
* It can also be used to prevent execution when testing alternative code.
* Use of descriptive names for variables and other entities in a program, and proper indentation of the code can reduce the need for using comments.
* C++ provides two types of comment delimiters:
  + **Single Line Comment**: Anything after **//** {double forward slash} (until the end of the line on which it appears) is considered a comment. o The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment.

Example 1:

#include <iostream> using namespace std; int main() { int x = 11; // x is a variable with integer type cout<<x<<"\n";

}

* + **Multiple Line Comment**: Anything enclosed by the pair **/\*** and **\*/** is considered a comment
  + The /\* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the \*/ characters. Example 2

#include <ostream> using namespace std; int main() {

/\* declare and print variable in C++. \*/ int x = 35; cout<<x<<"\n";

}

##### 3.4.8 Operators and Expressions

*3.4.8.1 Operators*

Once introduced to variables and constants, we can begin to operate with them by using *operators*. An operator is a symbol that makes the machine to take an action. Different Operators act on one or more operands and can also have different kinds of operators.

C++ provides several categories of operators, including the following:

* Assignment operator
* Arithmetic operator
* Relational operator
* Logical operator
* Increment/decrement operator
* Conditional operator
* Comma operator
* The size of operator
* Explicit type casting operators, etc.

**Assignment operator (=).** The assignment operator causes the operand on the left side of the assignment statement to have its value changed to the value on the right side of the statement.

* **Syntax**: Operand1=Operand2;
* Operand1 is always a variable
* Operand2 can be one or combination of:

o **A literal constant**: E.g.: x=12; o **A variable**: E.g.: x=y; o **An expression**: E.g.: x=y+2;

**Arithmetic operators (+, -, \*, /, % ).** Except for remainder or modulo (%), all other arithmetic operators can accept a mix of integers and real operands. Generally, if both operands are integers then, the result will be an integer. However, if one or both operands are real then the result will be real.

When both operands of the **division operator** (/) are integers, then the division is performed as an integer division and not the normal division we are used to.

* Integer division always results in an integer outcome.
* Division of integer by integer will not round off to the next integer
  + - * E.g.: 9/2 gives 4 not 4.5
      * -9/2 gives -4 not -4.5
* To obtain a real division when both operands are integers, you should cast one of the operands to be real.

o E.g.: int cost = 100; o Int volume = 80; o Double unitPrice = cost/**(double)**volume;

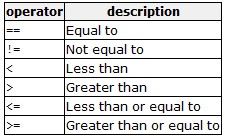
**The module(%)** is an operator that gives the remainder of a division of two integer values.

For instance, 13 % 3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

E.g.: a = 11 % 3; ,then a is 2

**Relational operator** **(==, !=, > , <, >=, <=).**

* In order to evaluate a comparison between two expressions, we can use the relational operator.
* The result of a relational operator is a bool value that can only be true or false according to the result of the comparison.
* Two expressions can be compared using relational and equality operators. For example, to know if two values are equal or if one is greater than the other.
* The result of such an operation is either true or false (i.e., a Boolean value).



**Table 3-3 operators and their description**

(7 == 5) *// evaluates to false*

(5 > 4) *// evaluates to true*

(3 != 2) *// evaluates to true*

(6 >= 6) *// evaluates to true*

(5 < 5) *// evaluates to false*

*‘A’ < ‘F’ //would return true or 1. It is like (65 <*

*70)*

**Logical Operators (!, &&, ||):**

* Logical negation (!) is a unary operator, which negates the logical value of its operand. If its operand is non-zero, it produce 0, and if it is 0 it produce 1
* Logical AND (&&) produces 0 if one or both of its operands evaluate to 0 otherwise it produces 1.
* Logical OR (||) produces 0 if both of its operands evaluate to 0 otherwise, it produces 1. Example

!20 //gives 0

10 && 5 //gives 1

10 || 5.5 //gives 1

10 && 0 // gives 0

**N.B.** In general, any non-zero value can be used to represent the logical true, whereas only zero represents the logical false.

**Increment/Decrement Operators: (++) and (--)**

* The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable.
* Example: if **a** was **10** and if **a++** is executed then a will automatically changed to **11**.
* Example: if **b** was **10** and if **b++** is executed then a will automatically changed to **9**.

**Prefix and Postfix:**

* The prefix type is written before the variable. E.g. **(++ myAge),** whereas the postfix type appears after the variable name **(myAge ++).**
* Prefix and postfix operators cannot be used at once on a single variable: E.g.: ++age-- or -age++ or ++age++ or - - age - - is invalid
* In a simple statement, either type may be used. But in complex statements, there will be a difference.
* The prefix operator is evaluated before the assignment, and the postfix operator is evaluated after the assignment. #
* Example int k = 5;

(auto increment prefix) y= ++k + 10; //gives 16 for y

(auto increment postfix) y= k++ + 10; //gives 15 for y

(auto decrement prefix) y= --k + 10; //gives 14 for y

(auto decrement postfix) y= k-- + 10; //gives 15 for y

**Conditional Operator (?:)**

* The conditional operator takes three operands. It has the general form:

o **Syntax**: operand1 **?** operand2 **:** operand3

* First operand1 is a relational expression and will be evaluated. If the result of the evaluation is non-zero (which means TRUE), then operand2 will be the final result.

Otherwise, operand3 is the final result.

* Example 1: 
  + - General Example Z=(X<Y? X : Y)
    - This expression means that if X is less than Y the value of X will be assigned to Z otherwise (if X>=Y) the value of Y will be assigned to Z.
* Example 2:

o int m=1,n=2,min; o min = (m < n ? m : n); o The value stored in min is 1.

* Example 2:

o (7 = = 5 ? 4: 3) returns 3 since 7 is not equal to 5

**Comma Operator (,):**

* Multiple expressions can be combined into one expression using the comma operator.
* The comma operator takes two operands. Operand1,Operand2
* The comma operator can be used during multiple declarations, for the condition operator and for function declaration, etc.
* It the first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.
* Example 1: int m,n,min; int mCount = 0, nCount = 0;

**The sizeof() Operator:**

* This operator is used for calculating the size of any data item or type.
* It takes a single operand (e.g. 100) and returns the size of the specified entity in bytes.

The outcome is totally machine dependent.

* Example 1:

o a = sizeof(char) o b = sizeof(int) o c = sizeof (1.55) etc.

**Explicit type casting operators:**

Type casting operators allows you to convert a datum of a given type to another data type.

Example 1: int i; float f = 3.14; i = (int)f;  equivalent to i = int(f); Then variable i will have a value of 3 ignoring the decimal point.

**Compound assignment operators (+=, -=, \*=, /=, %=, >>=, <<=, &=, ^=):**

* Compound assignment operator is the combination of the assignment operator with other operators like arithmetic and bit wise operators.
* The assignment operator has a number of variants, obtained by combining it with other operators.o **Example 1:** 
  + - * value += increase; is equivalent to value = value + increase; o a -= 5; is equivalent to a = a – 5; o a /= b; is equivalent to a = a / b; price \*= units + 1 is equivalent to price = price \* (units + 1); ƒ
      * And the same is true for the rest.

**Expressions and Statements**

* An expression statement is an expression followed by a semicolon.
* In C++, a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement)
* All C++ statements end with a semicolon. 
  + - * Example: x = a + b;
      * The meaning is: assign the value of the sum of a and b to x.
* *Statements* are the ―commands‖ or ―line of code‖ that can be executed whereas *expressions* are not executed by themselves.
* **White spaces**: white spaces characters (spaces, tabs, new lines) can‘t be seen and generally ignored in statements. White spaces should be used to make programs more readable and easier to maintain.
* Blocks: a block begins with an opening French brace ({) and ends with a closing French brace (}).
* Expressions: an expression is a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value).
* Example: the statement 3+2; returns the value 5 and thus is an expression.

##### 3.4.9 Special Printing Characters.

In C++, there are some special characters used for formatting. These are:

* **\n** new line
* **\t** tab
* **\b** backspace
* **\”** double quote
* **\‟** single quote
* **\?** Question mark
* **\\** backslash

##### 3.4.10 Input /Output Statements

The two instances **cout in C++** and **cin in C++** of iostream class are used very often for printing outputs and taking inputs respectively. This article mainly discusses the objects defined in the header file *iostream* like the cin and cout.

 **Standard output stream (cout)**: Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(**<<**).

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

char sample= „G‟; cout << sample; return 0;

}

In the above program, the insertion operator(**<<**) inserts the value of the character variable **sample** in the standard output stream **cout** which is then displayed on the screen.

**Standard input stream (cin)**: Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard.

The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.

#include <iostream> using namespace std;

int main()

{

int age;

cout << "Enter your age:";

cin >> age; cout << "\nYour age is: " << age;

return 0;

}

**Output:**

Enter your age:

Your age is: 18

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator(**>>**) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

**Summary**

* Any C++ program file should be saved with file name extension ― .CPP ‖ .
* When the program starts, **main()** is called automatically
* The compilation is a process of converting the source code into object code. It is done with the help of the compiler.
* When developing programs there are three types of error that can occur: Syntax errors: semantic errors and logical errors:
* Identifier is a unique name which is given to an entity to distinctly identify it meanwhile the execution of the source-code
* A Variable is a name that is assigned to a memory location, which is used to contain the corresponding value in it
* Depending on the type of data a variable can hold data types can be categorized as primitive data types, derived data types and abstract or user defined data types
* Keywords are the reserved keywords that are defined by the compiler to perform the internal operation, written in lowercase.
* Scope of a variable is the boundary or block in a program where a variable can be accessed. There are mainly two types of variable scopes: Local Variables and Global Variables.
* A constant is any expression that has a fixed value. A constant value is the one which does not change during the execution of a program. Constants can be of any of the basic data types.
* A comment is a piece of descriptive text which explains some aspect of a program. Comments are normally used to annotate code for future reference.
* An operator is a symbol that makes the machine to take an action. C++ provides several categories of operators, including the following: Assignment operator, Arithmetic operator, Relational operator, Logical operator, Increment/decrement operator, Conditional operator, Comma operator, The size of operator and Explicit type casting operators.
* In C++, there are some special characters used for formatting. These are: **\n** new line, **\t** tab , **\b** backspace, **\”** double quote, **\‟** single quote, **\?** Question mark, and **\\** backslash.
* The two instances **cout in C++** and **cin in C++** of iostream class are used very often for printing outputs and taking inputs respectively

**Exercise**

1. Write a c++ program which display the size of each data types in c++;
2. Write a c++ program which read two number from the user and display the sum and the product of the two numbers.
3. Write a program in C++ to check the upper and lower limits of integer.
4. Write a program in C++ to swap two variables value without additional third variable.
5. Write a program in C++ which calculate the area of a square.
6. Write a program in C++ to convert temperature in Celsius to Fahrenheit.
7. Write a program in C++ to find the third angle of a triangle.
8. Write a program in C++ to convert distance in kilometre to meter.
9. Write a program in C++ which accept two number from the user and compute quotient and remainder.
10. Write a C++ program to display the current date and time.

# 4 CHAPTER FOUR: CONTROL STATEMENTS IN C++ PROGRAMMING LANGUAGE

**Objectives**

After completing this chapter, you will be able to:

* Understand the definition of control statement
* Understand the types of control statements in C++ programming
* Understand the usage of the control statements in C++ programming

### 4.1. Overview of Control Statement

A running program spends all of its time executing statements. The order in which statements are executed is called **flow control** (or control flow). This term reflect the fact that the currently executing statement has the *control* of the CPU, which when completed will be handed over (*flow*) to another statement.

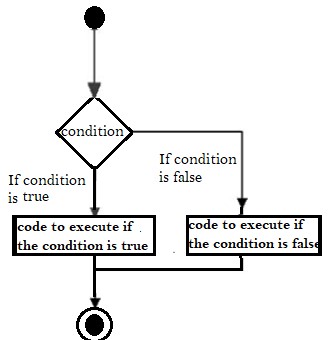
Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements. Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program. Not many programs execute all their statements in strict order from beginning to end. Most programs (like many humans) decide what to do in response to changing circumstances. The flow of control jumps from one part of the program to another, depending on calculations performed in the program. Program statements that cause such jumps are *called control statements.*

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the c++ programming language.

There are two major categories *of control statements* such as

* Branching or selection statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition.
* Loop statements are used for specifying computations, which need to be repeated until a certain logical condition is satisfied.

Flow control statements are used to divert the execution path to another part of the program. They are also known as **decision making statements**. 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. Following is the general from of a typical decision making structure found in most of the programming languages:



**Figure 4-1 Program flow example**

C++ programming language provides following types of decision making statements. These are c*onditional Statements, Looping Statements and* others. Both selection and loop statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition. Such statement requires that you specify a conditional expression. Any of the following comparison operators can be used in C++ conditions:

*==* : equal to

*!=* : not equal to

*>* : greater than *<* : less than

*>=* : greater than or equal to

*<=* : less than or equal to

You can also combine conditions using the standard Boolean operators:

*&&* : and

*||* : or

*!* : not

**For example**, the following are all valid conditional expressions in C++:

*((x > 0) && (x <= 10))*

*((x <= 0) || (x > 10))*

*(!((a == 1) && (b == 1)) || (c != 0))*

### 4.2. Selection Statements

##### 4.2.1. The if Statement

The *If-Else* statement allows the programmer to specify a *condition* for the execution of a statement, or group of statements. The common form / **Syntax** for this is as follows:

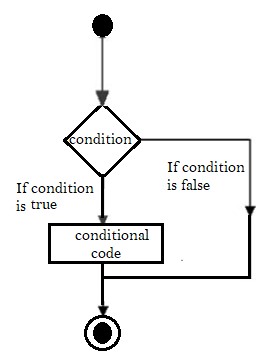
if(boolean\_expression)

{

// statement(s) will execute if the Boolean expression is true

}

* If the Boolean expression evaluates to **true**, then the block of code inside the if block will be executed.
* If Boolean expression evaluates to **false**, then the first set of code after the end of the if block (after the closing curly brace) will be executed.
* If we have only one statement inside the if block so that we are not expected to use opening and closing curly brace. However if we have more than one number of statement we are expected to write those statements inside curly brace.



**Figure 4-2 Flow diagram for *if* statement**

**Example 1:**

The following program will check if the value of a is less than 20 and if it is less than 20 it will display a text ―a is less than 20‖ and ―value of a is : 20‖ otherwise it only display the text ―value of a is ‖ combined with the a‘s values.

#include <iostream.h> int main ()

{ int a = 10; // local variable declaration: if( a < 20 ) // check the Boolean condition

{

// if condition is true then print the following cout << "a is less than 20;" << endl;

} cout << "value of a is:" << a << endl; // this line executed

always return 0; // return the integer value 0

}

When the above code is compiled and executed, it produces the following result:

a is less than 20; value of a is : 10

**Example 2**:- the following program will read a temperature value from the user and it will display the text ―warning- overheating ‖ if the temperature value is greater than 60.

#include <iostream.h> int main ()

{ float temperature; // declaring a float type variable cin>> temperature; // accept the temperature value

// compare if the temperature value is greater than 60 or not if (temperature > 60) cout<<"Warning-overheating\n";//display the text if the //above condition is true return 0; // return the integer value 0 }

**Example 3:** The following program will read income value from the user and compute the tax if the income is greater than 15000. The tax will be 20% of the income.

#include <iostream.h>

int main ()

{ float income; // declaring a float type variable named cin>> income; // accept the income value if (income < 15000) { cout << “Tax rate = 20%\n”; tax = income \* 0.2; cout<<”the tax is”<< tax;

} return 0;

}

*Notice that you can use curly brackets to cause more than one statement to be conditionally executed. This is called creating a compound statement.*

##### 4.2.2. The if….else Statement

An **if statement** can be followed by an optional **else** statement, which executes when the Boolean expression is false. It allows you to specify an alternative statement that is executed if the expression is not true. The *If-Else* statement allows the programmer to specify a *condition* for the execution of a statement, or group of statements.

**Syntax:**

The syntax of an if...else statement in C++ is:

if (boolean\_expression)

{

// statement(s) will execute if the boolean expression is true } else

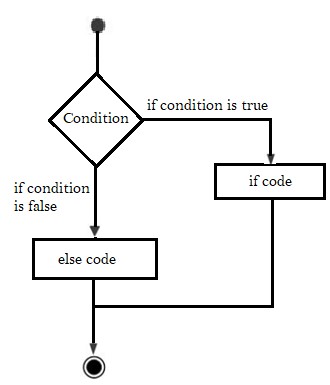
{

// statement(s) will execute if the boolean expression is false

}

If the Boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

**Flow Diagram:**



**Figure 4-3 Flow diagram for If else statement**

**Example 1**:- The following program will identify if the value a is greater than 20 or not and display its value.

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

{

// local variable declaration: int a = 100;

// check the boolean condition if( a < 20 ) // if condition is true then print the

//following

{

cout << "a is less than 20;" << endl;

} else // if condition is false then print the following

{ cout << "a is not less than 20;" << endl;

} cout << "value of a is : " << a << endl; return 0;

}

When the above code is compiled and executed, it produces the following result:

a is not less than 20; value of a is : 100

**Example 2:-** The following program will accept the value of balance from the user and compute and display the interest and the reaming balance based on the credit and debit rate.

|  |  |
| --- | --- |
| **Alterative one** | **Alterative Two** |
| float balance; cin>>balance; if (balance > 0) { interest = balance \* creditRate; balance += interest;  } else { interest = balance \* debitRate; balance += interest;  }  cout<< balance<<endl; cout<< interest<<endl; | float balance; cin>>balance;  if (balance > 0) interest = balance \* creditRate;  else interest = balance \* debitRate;  balance += interest; |
| **Alterative Three** | **Alterative Four** |

float balance; cin>>balance;

interest = balance \* (balance > 0 ?

creditRate : debitRate);

balance += interest; cout<< balance<<endl; cout<< interest<<endl; float balance; cin>>balance;

balance += balance \* (balance > 0 ? creditRate : debitRate); cout<< balance<<endl; cout<< interest<<endl;

##### 4.2.3. The if...else if Statement:

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind.

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.
* The final else block will be execute If all if conditions are not true.

**Syntax:**

The syntax of an if...else if...else statement in C++ is:

|  |
| --- |
| if(boolean\_expression 1) {  // Executes when the boolean expression 1 is true  } else if( boolean\_expression 2) {  // Executes when the boolean expression 2 is true  } else if( boolean\_expression 3) {  // Executes when the boolean expression 3 is true  } else  {  // executes when the none of the above condition is true. } |

**Example 1:**

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

{ int a = 100; // local variable declaration: if( a == 10 ) // check the boolean condition {// if condition is true then print the following cout << "Value of a is 10" << endl;

} else if( a == 20 )

{ // if else if condition is true cout << "Value of a is 20" << endl;

} else if( a == 30 )

{

// if else if condition is true cout << "Value of a is 30" << endl;

} else

{// if none of the conditions is true cout << "Value of a is not matching" << endl;

} cout << "Exact value of a is : " << a << endl; return 0;

}

When the above code is compiled and executed, it produces the following result:

Value of a is not matching

Exact value of a is: 100

##### 4.2.4. Switch Statement

Sometimes you may have many *If-Else* statements which all use conditions based on the same variable. It is not incorrect to use *If-Else* statements in this case, but it can make your code difficult to read and understand, and more prone to having errors. A preferable technique is to use a *Switch* statement. This offers an easy way of coding situations where the same variable needs to be checked against a number of different values.

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

**Syntax:**

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

|  |
| --- |
| switch(expression){  //check if expression and constant-expression1 are matched case constant-expression1:  // executed if expression and constant-expression1 matched statement(s); break;    //check if expression and constant-expression1 are matched case constant-expression2 :  // executed if expression and constant-expression2 are matched statement(s); break;  // you can have any number of case statements. default : //Optional  // executed if all cases are not matched statement(s); break;  } |

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-expressions** 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.
* Note that the *Switch* statement is used only for equality tests – you cannot use it for other comparisons (e.g. >, <, etc.). The effect of the *break* statement is to transfer control to the statement immediately following the *switch* statement.

**Flow Diagram:**

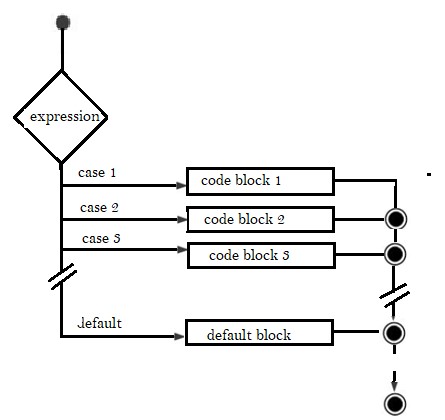


Figure 4-4 Flow diagram for the switch statement

**Example 1:**

The following program will read the day number and display the name of that day.

#include <iostream> using namespace std; int main () { int day; cin>>day; switch (day) { case 1: cout << “Segno\n”; break; case 2: cout << “Maksegno\n”; break; case 3: cout << “Rob\n”; break; case 4: cout << “Hamus\n”; break; case 5: cout << “Arb\n”; break; case 6: cout << “Kidamey\n”; break; case 7: cout << “Ihood\n”; break; default: cout << “Number out of range!\n”; break;

};

}

**Example 2**: The following program will accept the letter grade from the user and display based on the following criteria.

|  |  |
| --- | --- |
| Grade | Remark |
| A | Excellent |
| B | Well done |
| C | Well done |
| D | You passed |
| F | try again |
| Other | Invalid grade |

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

{

// local variable declaration:

char grade = 'D'; switch(grade)

{ case 'A' : cout << "Excellent!" << endl; break;

case 'B' :

case 'C' : cout << "Well done" << endl; break;

case 'D' : cout<<"You passed" << endl; break;

case 'F' : cout<<" try again"<< endl; break;

default :

cout<<"Invalid grade"<< endl;

} cout << "Your grade is " << grade << endl; return 0;

}

This would produce the following result:

You passed

Your grade is D

**Example 3**. The following program will receive the operator type from the user and apply the selected operator o the given operands. if we extend the above statement to also allow x to be used as a multiplication operator, we will have:

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

{

char operator; cin>>operator; int operand1,operand2; cin>operand1 cin>>operand2;

switch (operator) { case '+':

result = operand1 + operand2;

break; case '-':

result = operand1 - operand2;

break; case 'x': case '\*':

result = operand1 \* operand2;

break; case '/':

result = operand1 / operand2;

break; default:

cout << "unknown operator: " << ch << '\n';

break;

}

}

*Because case 'x' has no break statement (in fact no statement at all!), when this case is satisfied, execution proceeds to the statements of the next case and the multiplication is performed.*

##### 4.2.5. Nested statement

In selection statement we may want to check additional conditions even if the previous one is correct or not; so that we have to include additional selection statement inside the other. Such kind of statement is called selection statement.

Nested selection structures are used when more than one decision must be made before carrying out a task. Nesting is a programming activity, in which one program block is placed inside other program block of the same operation type. Nesting processes are mostly used implemented in the selection control structures.

**Advantages and Disadvantages of Nested Selection Advantages**:

* Nested selections give provisions to specify one condition inside another condition.
* Nested selections have many advantages when compared to labels that are used in programming.

**Disadvantages**:

* When a greater number of conditions are mentioned inside the nested selection, the code complexity will increase significantly.
* It becomes a difficult task for the programmers to find out each programming path separately if there is more nesting in the program.

##### 4.2.6. Nested if Statement

It is always legal to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

**Syntax:**

The syntax for a **nested if** statement is as follows:

if( boolean\_expression 1)

{

|  |
| --- |
| // Executes when the boolean expression 1 is true if(boolean\_expression 2)  {  // Executes when the boolean expression 2 is true  }  } |

You can nest **else if...else** in the similar way as you have nested *if* statement.

**Example:**

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

{

// local variable declaration:

int a = 100; int b = 200;

// check the boolean condition if( a == 100 )

{

// if condition is true then check the following if( b == 200 )

{

// if condition is true then print the following cout << "Value of a is 100 and b is 200" << endl;

}

} cout << "Exact value of a is : " << a << endl; cout << "Exact value of b is : " << b << endl; return 0;

}

When the above code is compiled and executed, it produces the following result:

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

##### 4.2.7. Nested if else

Nested if-else statement will have one if-else statement inside another if-else statement.

**Example**:- The following program will the percentage values of Chris and Harry compare the two values.

|  |
| --- |
| #include <iostream> using namespace std; int main () {  float g1, g2;  cout<<"Enter the percentage of Chris "; cin>> g1;  cout<<“"Enter the percentage of Harry"; cin>>g2; // Outer if else if (g1!= g2) {  cout<<“"Chris and Harry are not of equal grades\n";  //Nested if else  if (g1 > g2){  cout<<“"Chris is a better performer than Harry\n";  } else {  cout<<“"Harry is a better performer than Chris\n";  } // Nested if else ends  }  else // Outer if else ends  {  cout<<“"Chris and Harry have secured equal grades\  n";  } return 0;  } |

Output

Enter the percentage of Chris 78

Enter the percentage of Harry 54

Chris and Harry are not of equal grades

Chris is a better performer than Harry

##### 4.2.8. Nested Switch Statements

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.

**Syntax:**

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 is part of inner switch"; break; case 'B': // ...  } break; case 'B': // ...  } |

**Example : I**n the following program the value if a and b are displayed however in addition to those two values the text ―this is part of outer switch‖ will also be displayed only if the values of a is 100 and b is 200.

#include <iostream> using namespace std; int main () { int a = 100; int b = 200; switch(a) { case 100:

cout << "This is part of outer switch" << endl; switch(b) { case 200: cout << "This is part of inner switch" << endl;

brake;

}

brake;

} cout << "Exact value of a is : " << a << endl; cout << "Exact value of b is : " << b << endl; return 0;

}

The output of the above program is

This is part of outer switch

This is part of inner switch

Exact value of a is : 100

Exact value of b is : 200

**Exercise**

1. Write a C++ program to check whether a given number is even or odd.
2. Write a C++ program to accept the height of a person in centimetre and categorize the person according to their height.

hint

Test Data : 135

*Expected Output* : The person is Dwarf.

1. Write a C++ program to find the largest of three numbers
2. Write a C++ program to accept a coordinate point in a XY coordinate system and determine in which quadrant the coordinate point lies.

Hint:

Test Data : 7 9

*Expected Output* :

The coordinate point (7,9) lies in the First quadrant.

1. Write a C++ program to calculate the root of a Quadratic Equation.
2. Write a C++ program to read temperature in centigrade and display a suitable message according to temperature state below : Temp < 0 then Freezing weather

Temp 0-10 then Very Cold weather

Temp 10-20 then Cold weather

Temp 20-30 then Normal in Temp

Temp 30-40 then Its Hot

Temp >=40 then Its Very Hot

Test Data : 42

*Expected Output* :Its very hot.

1. Write a C++ program to check whether an alphabet is a vowel or consonant.
2. Write a program in C++ to read any Month Number in integer and display Month name in the word

Test Data : 4

*Expected Output* : April

1. Write a program in C++ which is a Menu-Driven Program to perform a simple calculation.

### 4.3. Looping Statements

Loop statements are useful when you want the same piece of code to be executed a number of times. For example: Suppose we want to print ―Hello World‖ 10 times. This can be done in two ways as shown below:

// C++ program to illustrate need of loops

#include <stdio.h>

int main() { cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”; cout<< “Hello World\n”;

return 0;

}

In Loop, the statement needs to be written only once and the loop will be executed 10 times. In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

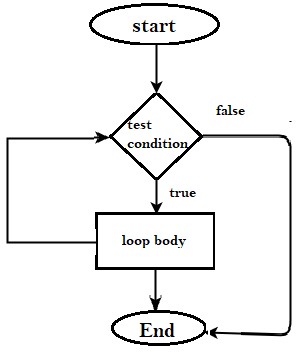
* An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.
* **Counter not Reached:** If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
* **Counter reached:** If the condition has been reached, the next instruction ―falls through‖ to the next sequential instruction or branches outside the loop.

There are several different loop statements; each one is designed for slightly different programming situations. There are mainly two types of loops:

1. **Entry Controlled loops**: In this type of loops the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.
2. **Exit Controlled Loops**: In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. **do – while loop** is exit controlled loop.

##### 4.3.1. The „while‟ Statement

The while statement (also called **while loop**) provides a way of repeating a statement while a condition holds. It is one of the three flavours of **iteration** in C++. The general form of the while statement is:



**Syntax**;

while (*expression*)

{

*statement*;//body

}

###### Figure 4-5 flow chart for loop statement

First *expression* (called the **loop condition**) is evaluated. If the outcome is nonzero then *statement* (called the **loop body**) is executed and the whole process is repeated. Otherwise, the loop is terminated.

**Example: the following program will display the text” hello world” 10 times**

// C++ program to illustrate while loop

#include <iostream> using namespace std;

int main()

{

// initialization expression int i = 1;

// test expression while (i < 6)

{ cout << "Hello World\n";

// update expression i++;

} return 0;

}

**For example**, suppose we wish to calculate the sum of all numbers from 1 to some integer denoted by *n*. This can be expressed as:

i = 1; sum = 0; while (i <= n)

sum += i;

For n set to 5, the following table provides a trace of the loop by listing the values of the variables involved and the loop condition.

**While loop trace**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Iteration** | **i** | **n** | **i <= n** | **sum += i++** |
| First | 1 | 5 | 1 | 1 |
| Second | 2 | 5 | 1 | 3 |
| Third | 3 | 5 | 1 | 6 |
| Fourth | 4 | 5 | 1 | 10 |
| Fifth | 5 | 5 | 1 | 15 |
| Sixth | 6 | 5 | 0 |  |

###### Table 4-1 while loop trace for the example

It is not unusual for a while loop to have an empty body (i.e., a null statement). The following loop, for example, sets n to its greatest odd factor. while (n % 2 == 0 && n /= 2)

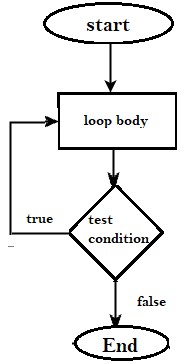
;

Here the loop condition provides all the necessary computation, so there is no real need for a body. The loop condition not only tests that n is even, it also divides n by two and ensures that the loop will terminate should n be zero.

##### 4.3.2. The *Do* While Loop

The Do loop is similar to the While loop – it should also be used when you do not know how many times the loop should be executed. However, with the Do loop the statement is guaranteed to be executed at least once, as the expression is only checked after the first iteration.

do



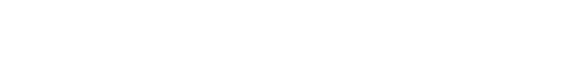
The general form of a

*Do*

loop is:

statement;// body

while (expression);



**Figure**

**4**

**-**

**6**

**flow chart for do while loop**

First *statement* is executed and then *expression* is evaluated. If the outcome of the latter is nonzero then the whole process is repeated. Otherwise, the loop is terminated. The do loop is less frequently used than the while loop. It is useful for situations where we need the loop body to be executed at least once, regardless of the loop condition.

For example, suppose we wish to repeatedly read a value and print its square, and stop when the value is zero.

This can be expressed as the following loop:

do { cin >> n;

cout << n \* n << '\n'; } while (n != 0);

For example, examine the following piece of code – can you see what it does?

float i = 1.0; do {

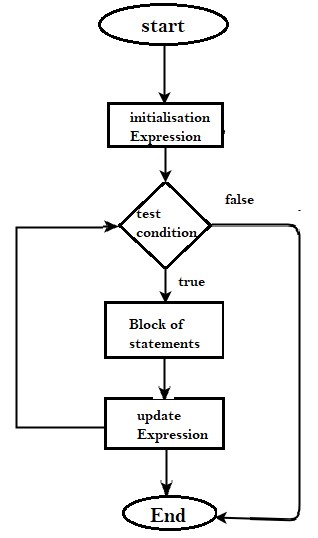
cout << i << endl; i /= 2;

} while (i > 0.01);

Unlike the while loop, the do loop is never used in situations where it would have a null body. Although a do loop with a null body would be equivalent to a similar while loop, the latter is always preferred for its superior readability.

##### 4.3.3. The „for‟ Statement

The for statement (also called **for loop**) is similar to the while statement, but has two additional components: an expression which is evaluated only once before everything else, and an expression which is evaluated once at the end of each iteration.



**Figure 4-7 Flow diagram of the for statement**

The general form of the for statement is: ***Syntax*** *for (init-stmt; expr-1; expr-2) statement;*

**For loop flow diagram**

Based on the above syntax the program will be executed as follow

* First *expression1* / init-stmnt is evaluated. it is used to initialise a control variable.
* expr-1 is used as the loop control (the loop will be executed until this expression becomes false), and
* expr-2 is generally used to update the control variable‘s value. Each time round the loop, *expr-1* is evaluated. If the outcome is nonzero then statement is executed and *expr-2* is evaluated. Otherwise, the loop is terminated

.

In general In for loop, a loop variable is used to control the loop. First initialize this loop variable to some value, then check whether this variable is less than or greater than counter value. If statement is true, then loop body is executed and loop variable gets updated . Steps are repeated till exit condition comes.

* **Initialization Expression**: In this expression we have to initialize the loop counter to some value. for example: int i=1;
* **Test Expression**: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;
* **Update Expression**: After executing loop body this expression increments/decrements the loop variable by some value. for example: i++;

**Example: the following program wil display the text “hello world” 10 times**

// C++ program to illustrate for loop

#include <iostream> using namespace std;

int main() { for (int i = 1; i <= 10; i++)

{ cout << "Hello World\n"; } return 0;

}

**Example**, the following piece of code prints out the numbers from 1 to 10:

for (i = 1; i <= 10; i++) cout << i << endl;

Here, init-stmnt assigns the value 1 to the variable *i*, expr-1 (the loop control) checks to see if the value of *i* is greater than 10 at the end of each iteration of the loop, and expr-2 increments the value of *i* after each iteration

The general for loop is equivalent to the following while loop:

*expression1*; while (*expression2*) { *statement*; *expression3*;

}

The most common use of for loops is for situations where a variable is incremented or decremented with every iteration of the loop. The following for loop, for example, calculates the sum of all integers from 1 to n.

sum = 0;

for (i = 1; i <= n; ++i) sum += i;

This is preferred to the while-loop version we saw earlier. In this example, i is usually called the **loop variable**. C++ allows the first expression in a for loop to be a variable definition. In the above loop, for example, i can be defined inside the loop itself:

for (int i = 1; i <= n; ++i) sum += i;

Contrary to what may appear, the scope for i is *not* the body of the loop, but the loop itself.

Scope-wise, the above is equivalent to:

int i;

for (i = 1; i <= n; ++i)

sum += i;

Any of the three expressions in a for loop may be empty. For example, removing the first and the third expression gives us something identical to a while loop:

for (; i != 0;) // is equivalent to: while (i != 0) something; // something;

Removing all the expressions gives us an infinite loop. This loop's condition is assumed to be always true:

for (;;) // infinite loop something;

For loops with multiple loop variables are not unusual. In such cases, the comma operator is used to separate their expressions:

for (i = 0, j = 0; i + j < n; ++i, ++j) something;

Because loops are statements, they can appear inside other loops. In other words, loops can be nested. For example,

for (int i = 1; i <= 3; ++i) for (int j = 1; j <= 3; ++j) cout << '(' << i << ',' << j << ")\n";

produces the product of the set {1,2,3} with itself, giving the output:

(1,1)

(1,2)

(1,3)

(2,1)

(2,2)

(2,3)

(3,1)

(3,2)

(3,3)

##### 4.3.4. Nested Loop

**Nested Statements**

Any of the above control structure statements can be *nested*. This simply means putting one statement inside another one. We have already seen nested statements in the examples given above.

As another example, examine the following piece of code and see if you can work out what it does:

int i, n, f; while (cin >> n) { f = 1;

for (i = 2; i <= n; i++) f \*= i; cout << f << endl;

}

Here we have a *for* loop nested inside a *while* loop. The *while* loop reads in a sequence of numbers from the user, terminated by Ctrl-D. Then, for each number, the *for* loop computes its factorial.

**Nested for loop**

// C++ program to display 7 days of 3 weeks

#include <iostream> using namespace std;

int main() { int weeks = 3, days\_in\_week = 7;

for (int i = 1; i <= weeks; ++i) { cout << "Week: " << i << endl;

for (int j = 1; j <= days\_in\_week; ++j) { cout << " Day:" << j << endl;

}

} return 0;

}

|  |
| --- |
| Output |
| Week: 1  Day:1  Day:2  Day:3  ... .. ...  Week: 2  Day:1  Day:2  Day:3  ... ... .. |

**Example 2**

// C++ program to display a pattern

// with 5 rows and 3 columns

#include <iostream> using namespace std;

int main() { int rows = 5; int columns = 3;

for (int i = 1; i <= rows; ++i) { for (int j = 1; j <= columns; ++j) { cout << "\* ";

} cout << endl;

}

return 0;

}

|  |
| --- |
| Output |
| * \* \* * \* \* * \* \* * \* \* * \* \* |

### 4.4. Other Statements

##### 4.4.1. The „continue‟ Statement

The continue statement terminates the current iteration of a loop and instead jumps to the next iteration. It applies to the loop immediately enclosing the continue statement. It is an error to use the continue statement outside a loop.

In while and do loops, the next iteration commences from the loop condition. In a for loop, the next iteration commences from the loop‘s third expression. For example, a loop which repeatedly reads in a number, processes it but ignores negative numbers, and terminates when the number is zero, may be expressed as:

do { cin >> num; if (num < 0) continue; // process num here...

} while (num != 0);

This is equivalent to:

do { cin >> num; if (num >= 0) {

// process num here...

}

} while (num != 0);

A variant of this loop which reads in a number exactly *n* times (rather than until the number is zero) may be expressed as:

for (i = 0; i < n; ++i) { cin >> num; if (num < 0) continue; // causes a jump to:

++i

// process num here...

}

When the continue statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops. For example, in the following set of nested loops, the continue applies to the for loop, and not the while loop:

while (more) { for (i = 0; i < n; ++i) { cin >> num; if (num < 0) continue; // causes a jump to: ++i

// process num here...

}

//etc...

}

##### 4.4.2. The „break‟ Statement

A break statement may appear inside a loop (while, do, or for) or a switch statement. It causes a jump out of these constructs, and hence terminates them. Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it. It is an error to use the break statement outside a loop or a switch.

For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts:

for (i = 0; i < attempts; ++i) { cout << "Please enter your password: "; cin >> password;

if (Verify(password)) //check password for correctness break; // drop out of the loop cout << "Incorrect!\n";

}

Here we have assumed that there is a function called Verify which checks a password and returns true if it is correct, and false otherwise.

Rewriting the loop without a break statement is always possible by using an additional logical variable (verified) and adding it to the loop condition:

verified = 0;

for (i = 0; i < attempts && !verified; ++i) { cout << "Please enter your password: "; cin >> password; verified = Verify(password)); if (!verified) cout << "Incorrect!\n";

}

The break version is arguably simpler and therefore preferred.

##### 4.4.3. The „goto‟ Statement

The goto statement provides the lowest-level of jumping. It has the general form:

goto *label*;

where *label* is an identifier which marks the jump destination of goto. The label should be followed by a colon and appear before a statement within the same function as the goto statement itself. For example, the role of the break statement in the for loop in the previous section can be emulated by a goto:

for (i = 0; i < attempts; ++i) { cout << "Please enter your password: "; cin >> password;

if (Verify(password)) // check password for correctness

goto out; // drop out of the loop cout << "Incorrect!\n";

} out:

//etc...

Because goto provides a free and unstructured form of jumping (unlike break and continue), it can be easily misused. Most programmers these days avoid using it altogether in favor of clear programming. Nevertheless, goto does have some legitimate (though rare) uses.

##### 4.4.4. The „return‟ Statement

The return statement enables a function to return a value to its caller. It has the general form:

return *expression*;

where *expression* denotes the value returned by the function. The type of this value should match the return type of the function. For a function whose return type is void, *expression* should be empty:

return;

The only function we have discussed so far is main, whose return type is always int. The return value of main is what the program returns to the operating system when it completes its execution. Under UNIX, for example, it its conventional to return 0 from main when the program executes without errors. Otherwise, a non-zero error code is returned. For example:

int main (void)

{

cout << "Hello World\n"; return 0;

}

When a function has a non-void return value (as in the above example), failing to return a value will result in a compiler warning. The actual return value will be undefined in this case (i.e., it will be whatever value which happens to be in its corresponding memory location at the time).

**Summary**

Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements.

* In c++ we have two main categories of flow control programs. One is selection statement and the other is loop statement.
* Selection statement is a statement which allow us ot execute set of statements based on a given criteria.
* We have for types of selection statement, such as if , if else, if else if, switch statement.
* If statement used to specify a set of activities to be executed under only if a given condition is true.
* If else statement is used to specify a set of two block in which the first block to be executed is the give condition is true and the other will be executed if the condition is false.
* If else if statement used to specify more than one conditions to be checked in the program
* Switch statement is a selection statement which includes a number of cases to be matched with the given expression. The block under the matched case will be executed otherwise if none of the given cases are not matched each to the expression so that the final default black will be executed.
* In switch statement default block is optional
* In c++ if we want to execute a given activity for number of times we could implement such statements by using loop statements.
* Loop statements are statements which allow us to execute a given set of statements multiple times based on the given condition.
* In c++ we have 3 main loop statements such as for loop , switch statement , and do while loop statements.

**Exercise**

* 1. Write a C++ Program to display the pattern like pyramid using the alphabet.
  2. Write a program in C++ to find the number and sum of all integer between 100 and 200 which are divisible by 9.
  3. Write a program in C++ to display the n terms of harmonic series and their sum. 1 + 1/2 + 1/3 + 1/4 + 1/5 ... 1/n terms
  4. Write a program in C++ to display the pattern like a pyramid using asterisk and each row contain an odd number of asterisks.
  5. Write a program in C++ to display the n terms of odd natural number and their sum
  6. Write a program in C++ to convert an octal number into binary
  7. Write a program in C++ to convert a decimal number to hexadecimal.
  8. Write a program in C++ to print a string in reverse order
  9. Write a program in C++ to find LCM of any two numbers using HCF
  10. Write a program in C++ to display the such a pattern for n number of rows using a number which will start with the number 1 and the first and a last number of each row will be 1

# 5. CHAPTER FIVE : ARRAYS AND STRINGS

**Objectives**

On completion of this chapter, students should be able to:

* Understand and explain the concept of array and string
* Understand how to use array and string to solve problems in C++ programming language

### 5.1. Introduction

Variables in a program have values associated with them. During program execution these values are accessed by using the identifier associated with the variable in expressions etc. In none of the programs written so far have very many variables been used to represent the values that were required. Thus even though programs have been written that could handle large lists of numbers it has not been necessary to use a separate identifier for each number in the list. This is because in all these programs it has never been necessary to keep a note of each number individually for later processing. For example in summing the numbers in a list only one variable was used to hold the current entered number which was added to the accumulated sum and was then overwritten by the next number entered. If that value were required again later in the program there would be no way of accessing it because the value has now been overwritten by the later input.

If only a few values were involved a different identifier could be declared for each variable, but now a loop could not be used to enter the values. Using a loop and assuming that after a value has been entered and used no further use will be made of it allows the following code to be written. This code enters six numbers and outputs their sum:

sum = 0.0;

for (i = 0; i < 6; i++)

{ cin >> x; sum += x; }

This of course is easily extended to n values where n can be as large as required. However if it was required to access the values later the above would not be suitable. It would be possible to do it as follows by setting up six individual variables: float a, b, c, d, e, f;

and then handling each value individually as follows:

sum = 0.0; cin >> a; sum += a; cin >> b; sum += b; cin >> c; sum += c; cin >> d; sum += d; cin >> e; sum += e; cin >> f; sum += f;

which is obviously a very tedious way to program. To extend this solution so that it would work with more than six values then more declarations would have to be added, extra assignment statements added and the program re-compiled. If there were 10000 values imagine the tedium of typing the program (and making up variable names and remembering which is which)!

To get round this difficulty all high-level programming languages use the concept of a data structure called an **Array.**

### *5.2.* Array Definition

An **array** is a data structure which allows a collective name to be given to a group of elements which ***all have the same type***. An individual element of an array is identified by its own unique ***index*** (or **subscript**).

An array can be thought of as a collection of numbered boxes each containing one data item. The number associated with the box is the index of the item. To access a particular item the index of the box associated with the item is used to access the appropriate box. The index **must** be an integer and indicates the position of the element in the array. Thus the elements of an array are **ordered** by the index.

### 5.3. Types of array

#### *5.3.1.* One Dimensional Array

*5.3.1.1. Declaration of 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**.

For example data on the average temperature over the year in Ethiopia 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.

*5.3.1.2. Initialization of arrays*

The initialization of simple variables in their declaration has already been covered. An array can be initialized in a similar manner. In this case the initial values are given as a list enclosed in curly brackets. For example initializing an array to hold the first few prime numbers could be written as follows:

int primes[] = {1, 2, 3, 5, 7, 11, 13};

Note that the array has not been given a size, the compiler will make it large enough to hold the number of elements in the list. In this case primes would be allocated space for seven elements. If the array is given a size then this size must be greater than or equal to the number of elements in the initialization list. For example:

int primes[10] = {1, 2, 3, 5, 7}; would reserve space for a ten element array but would only initialize the first five elements.

Example Program: Printing Outliers in Data

The requirement specification for a program is:

A set of positive data values (200) are available. It is required to find the average value of these values and to count the number of values that are more than 10% above the average value.

Since the data values are all positive a negative value can be used as a sentinel to signal the end of data entry. Obviously this is a problem in which an array must be used since the values must first be entered to find the average and then each value must be compared with this average. Hence the use of an array to store the entered values for later re-use.

An initial algorithmic description is:

initialize. enter elements into array and sum elements. evaluate average.

scan array and count number greater than 10% above average. output results.

This can be expanded to the complete algorithmic description:

set sum to zero. set count to zero. set nogt10 to zero. enter first value.

while value is positive

{ put value in array element with index count. add value to sum. increment count. enter a value.

} average = sum/count.

for index taking values 0 to count-1 if array[index] greater than 1.1\*average then increment nogt10. output average, count and nogt10.

In the above the variable nogt10 is the number greater than 10% above the average value. It is easy to argue that after exiting the while loop, count is set to the number of positive numbers entered. Before entering the loop count is set to zero and the first number is entered, that is count is one less than the number of numbers entered. Each time round the loop another number is entered and count is incremented hence count remains one less than the number of numbers entered. But the number of numbers entered is one greater than the number of positive numbers so count is therefore equal to the number of positive numbers.

A **main()** program written from the above algorithmic description is given below:

void main() {

const int NE = 200; // maximum no of elements in array float sum = 0.0; // accumulates sum int count = 0; // number of elements entered int nogt10 = 0; // counts no greater than 10%

// above average float x; // holds each no as input float indata[NE]; // array to hold input float average; // average value of input values int i; // control variable

// Data entry, accumulate sum and count // number of +ve numbers entered

cout << "Enter numbers, -ve no to terminate: " << endl; cin >> x; while (x >= 0.0)

{

sum = sum + x; indata[count] = x; count = count + 1; cin >> x;

}

// calculate average average = sum/count;

// Now compare input elements with average for (i = 0; i < count; i++)

{

if (indata[i] > 1.1 \* average) nogt10++;

}

// Output results

cout << "Number of values input is " << n; cout << endl

<< "Number more than 10% above average is "

<< nogt10 << endl;

}

Since it was assumed in the specification that there would be less than 200 values the array size is set at 200. In running the program less than 200 elements may be entered, if n elements where n < 200 elements are entered then they will occupy the first n places in the array indata. It is common to set an array size to a value that is the maximum we think will occur in practice, though often not all this space will be used.

Example Program: Test of Random Numbers

The following program simulates the throwing of a dice by using a random number generator to generate integers in the range 0 to 5. The user is asked to enter the number of trials and the program outputs how many times each possible number occurred.

An array has been used to hold the six counts. This allows the program to increment the correct count using one statement inside the loop rather than using a switch statement with six cases to choose between variables if separate variables had been used for each count. Also it is easy to change the number of sides on the dice by changing a constant. Because C++ arrays start at subscript 0 the count for an i occurring on a throw is held in the i-1th element of this count array. By changing the value of the constant die\_sides the program could be used to simulate a die\_sides-sided die without any further change.

#include <iostream.h>

#include <stdlib.h> // time.h and stdlib.h required for

#include <time.h> // random number generation

void main() {

const int die\_sides = 6; // maxr-sided die int count[die\_sides]; // holds count of each // possible value int no\_trials, // number of trials roll, // random integer i; // control variable float sample; // random fraction 0 .. 1

// initialize random number generation and count

// array and input no of trials srand(time(0)); for (i=0; i < die\_sides; i++) count[i] = 0; cout << "How many trials? "; cin >> no\_trials;

// carry out trials for (i = 0; i < no\_trials; i++)

{

sample = rand()/float(RAND\_MAX); roll = int ( die\_sides \* sample);

// returns a random integer in 0 to die\_sides-1 count[roll]++; // increment count

}

// Now output results for (i = 0; i < die\_sides; i++)

{

cout << endl << "Number of occurrences of "

<< (i+1) << " was " << count[i];

} cout << endl;

*5.3.1.3. Array manipulation*

**Accessing Array Elements**

Given the declaration above of a 100-element array the compiler reserves space for 100 consecutive floating point values and accesses 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 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<=NE**).

sum = 0.0;

for (i = NE - k; i < NE; i++) sum += annual\_temp[i]; av2 = sum / k;

Important - C++ does not check that the subscript that is used to reference an array element actually lies in the subscript range of the array. Thus C++ will allow the assignment of a value to annual\_temp[200], however the effect of this assignment is unpredictable. For example it could lead to the program attempting to assign a value to a memory element that is outside the program's allocated memory space. This would lead to the program being terminated by the operating system. Alternatively it might actually access a memory location that is within the allocated memory space of the program and assign a value to that location, changing the value of the variable in your program which is actually associated with that memory location, or overwriting the machine code of your program. Similarly reading a value from annual\_temp[200] might access a value that has not been set by the program or might be the value of another variable. It is the programmer's responsibility to ensure that if an array is declared with n elements then no attempt is made to reference any element with a subscript outside the range 0 to n-1. Using an index, or subscript, that is out of range is called Subscript Overflow. Subscript overflow is one of the commonest causes of erroneous results and can frequently cause very strange and hard to spot errors in programs.

**Copying Arrays**

The assignment operator *cannot* be applied to array variables:

const int SIZE=10 int x [SIZE] ; int y [SIZE] ;

x = y ; // Error - Illegal

Only individual elements can be assigned to using the index operator, e.g., **x[1] = y[2];.**

To make all elements in 'x' the same as those in 'y' (equivalent to assignment), a loop has to be used.

// Loop to do copying, one element at a time for (int i = 0 ; i < SIZE; i++) x[i] = y[i];

This code will copy the elements of array y into x, overwriting the original contents of x. A loop like this has to be written whenever an array assignment is needed.

Notice the use of a constant to store the array size. This avoids the literal constant '10' appearing a number times in the code. If the code needs to be edited to use different sized arrays, only the constant needs to be changed. If the constant is not used, all the '10's would have to be changed individually - it is easy to miss one out.

#### *5.3.2.* Multidimensional arrays

An array may have more than one dimension. Each dimension is represented as a subscript in the array. Therefore a two dimensional array has two subscripts, a three dimensional array has three subscripts, and so on.

Arrays can have any number of dimensions, although most of the arrays that you create will likely be of one or two dimensions.

A chess board is a good example of a two-dimensional array. One dimension represents the eight rows, the other dimension represents the eight columns.

*5.3.2.1. Declaration of arrays*

The declaration of array named board that represents would be two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a twodimensional 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 **a**, which contains three rows and four columns can be shown as below −

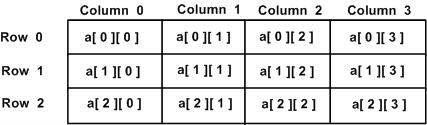


Figure 5-1 Two dimensional array

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.

**For example:**

Square board[8][8];

The program could also represent the same data with a one dimensional, 64-square array. For example, it could include the statement

Square board[64];

Such a representation does not correspond as closely to the real-world object as the two dimensional array, however.

Suppose that when the game begins. The king id located in the fourth position in the first row. Counting from zero that position corresponds to board[0][3] in the two dimensional array, assuming that the first subscript corresponds to the row, and the second to the column.

*5.3.2.2. Initializing Multidimensional Arrays*

To initialize a multidimensional arrays , you must assign the list of values to array elements in order, with last array subscript changing while the first subscript while the first subscript holds steady. Therefore, if the program has an array **int theArray[5][3],** the first three elements go **int theArray[0];** the next three into **theArray[1];** and so forth. The program initializes this array by writing

int theArray[5][3] ={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,

13, 14, 15};

for the sake of clarity, the program could group the initializations with braces, as shown below.

int theArray[5][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10,

11, 12}, {13, 14,15} };

The compiler ignores the inner braces, which clarify how the numbers are distributed.

Each value should be separated by comma, regardless of whither inner braces are include. The entire initialization must set must appear within braces, and it must end with a semicolon.

## 5.3. String

#### 5.3.1. String Definition

String in C++ is nothing but a sequence of character in which the last character is the null character ‗\0‘. The null character indicates the end of the string. Any array of character can be converted into string type in C++ by appending this special character at the end of the array sequence.

In C++ strings of characters are held as an **array of characters**, one character held in each array element. In addition a special **null character**, represented by `\0', is appended to the end of the string to indicate the end of the string. Hence if a string has n characters then it requires an n+1 element array (at least) to store it. Thus the character `a' is stored in a single byte, whereas the single-character string "a" is stored in two consecutive bytes holding the character `a' and the null character.

#### 5.3.2. String declaration and initialization

A string variable s1 could be declared as follows:

char s1[10];

The string variable s1 could hold strings of length up to nine characters since space is needed for the final null character. Strings can be initialized at the time of declaration just as other variables are initialized. For example:

char s1[] = "example"; char s2[20] = "another example" would store the two strings as follows:

s1 |e|x|a|m|p|l|e|\0|

s2 |a|n|o|t|h|e|r| |e|x|a|m|p|l|e|\0|?|?|?|?|

In the first case the array would be allocated space for eight characters, that is space for the seven characters of the string and the null character. In the second case the string is set by the declaration to be twenty characters long but only sixteen of these characters are set, i.e. the fifteen characters of the string and the null character. Note that the length of a string does not include the terminating null character.

#### 5.3.3. String Manipulation

**String Output**

A string is output by sending it to an output stream, for example:

cout << "The string s1 is " << s1 << endl; would print

The string s1 is example

The **setw(width)** I/O manipulator can be used before outputting a string, the string will then be output right-justified in the field width. If the field width is less than the length of the string then the field width will be expanded to fit the string exactly. If the string is to be leftjustified in the field then the **setiosflags** manipulator with the argument **ios::left** can be used.

**String Input**

When the input stream **cin** is used space characters, **newline** etc. are used as separators and terminators. Thus when inputting numeric data **cin** skips over any leading spaces and terminates reading a value when it finds a white-space character (**space, tab, newline** etc. ). This same system is used for the input of strings, hence a string to be input cannot start with leading spaces, also if it has a space character in the middle then input will be terminated on that space character. The null character will be appended to the end of the string in the character array by the stream functions. If the string s1 was initialized as in the previous section, then the statement

cin << s1;

would set the string s1 as follows when the string "**first**" is entered (without the double quotes)

|f|i|r|s|t|\0|e|\0|

Note that the last two elements are a relic of the initialization at declaration time. If the string that is entered is longer than the space available for it in the character array then C++ will just write over whatever space comes next in memory. This can cause some very strange errors when some of your other variables reside in that space!

To read a string with several words in it using cin we have to call cin once for each word. For example to read in a name in the form of a Christian name followed by a surname we might use code as follows:

char christian[12], surname[12]; cout << "Enter name "; cin >> christian; cin >> surname;

cout << "The name entered was "

<< christian << " "

<< surname;

The name would just be typed by the user as, for example, **Ian Aitchison** and the output would then be

The name entered was Ian Aitchison

Enter a string: Law is a bottomless pit.

You entered: Law

Where did the rest of the phrase go?

It turns the insertion operator >> consider a space to be a terminating character.

Thus it will read strings consisting of a single word, but anything typed after a space is thrown away.

To read text containing blanks we use another function, **cin::get()**.

#include<iostream.h> void main() { const int max=80; char str[max]; cout<<"\n Enter a string;";

cin.get(str,max); // max avoid buffer overflow cout<<"\n You entered : "<<str;

}

Reading multiple lines

We have solved the problem of reading strings with embedded blanks, but what about strings with multiple lines? It turns out that the **cin::get()** function can take a third argument to help out in this situation.

This argument specifies the character that tells the function to stop reading. The default value of this argument is the **newline('\n')**character, but if you call the function with some other character for this argument, the default will be overridden by the specified character.

In the next example, we call the function with a dollar sign (**'$'**) as the third argument

//reads multiple lines, terminates on '$' character

#include<iostream.h> void main(){ const int max=80; char str[max];

cout<<"\n Enter a string:\n";

cin.get(str, max, '$'); //terminates with $ cout<<\n You entered:\n"<<str; }

now you can type as many lines of input as you want. The function will continue to accept characters until you enter the terminated character **$** (or untill you exceed the size of the array. Remember, you must still press Enter key after typing the **'$'** character .

**Avoiding buffer over flow**

The strings in the program invites the user to type in a string. What happens if the user enters a string that is longer than the array used to hold it? There is no built-in mechanism in C++ to keep a program from inserting array elements outside an array.

However, it is possible to tell the >> operator to limit the number of characters it places in an array.

//avoids buffer overflow with cin.width

#include<iostream.h>

#include<iomanip.h> //for setw

void main(){ const int MAX=20; char str[MAX]; cout<<"\n Enter a string: "; cin>>setw(MAX)>>str; cout<<"\n You entered :"<<str;

}

**String constants**

You can initialize a string to a constant value when you define it. Here's an example'

#include<iostream.h> void main(){

char str[] = "Welcome to C++ programming language"; cout<<str;

}

if you tried to the string program with strings that contain more than one word , you may have unpleasant surprise. Copying string the hard way

The best way to understand the true nature of strings is to deal with them character by character

#include<iostream.h>

#include<string.h> //for strlen() void main() {

const int max=80;

char str1[]='' Oh, Captain, my Captain!" our fearful trip is done"; char str2[max];

for(int i=0; i<strlen(str1);i++) str2[i]=str1[1]; str2[i]='\0'; cout<<endl; cout<<str2;

}

**Copying string the easy way**

Ofcourse you don't need to use a **for** loop to copy a string. As you might have guesses, a library function will do it for you. You can copy strings using **strcpy** or **strncpy** function. We assign strings by using the string copy function**strcpy**. The prototype for this function is in **string.h.**

strcpy(destination, source);

**strcpy** copies characters from the location specified by source to the location specified by destination. It stops copying characters after it copies the terminating null character.

o The return value is the value of the destination parameter.

You must make sure that the destination string is large enough to hold all of the characters in the source string (including the terminating null character).

Example:

#include <iostream.h> #include <string.h> void main(){ char me[20] = "David"; cout << me << endl; strcpy(me, "YouAreNotMe"); cout << me << endl ; return;

}

There is also another function **strncpy**, is like **strcpy**, except that it copies only a specified number of characters.

strncpy(destination, source, int n);

It may not copy the terminating null character. Example

#include <iostream.h> #include <string.h> void main() { char str1[] = "String test"; char str2[] = "Hello"; char one[10]; strncpy(one, str1, 9); one[9] = '\0'; cout << one << endl; strncpy(one, str2, 2); cout << one << endl; strcpy(one, str2); cout << one << endl;

}

**Concatenating strings**

In C++ the + operator cannot normally be used to concatenate string, as it can in some languages such as BASIC; that is you can't say

Str3 = str1 + str2;

You can use strcat() or strncat

The function strcat concatenates (appends) one string to the end of another string.

strcat(destination, source);

o The first character of the source string is copied to the location of the terminating null character of the destination string. o The destination string must have enough space to hold both strings and a terminating null character.

**Example**:

#include <iostream.h>

#include <string.h>

void main() { char str1[30]; strcpy(str1, "abc"); cout << str1 << endl; strcat(str1, "def"); cout << str1 << endl;

char str2[] = "xyz"; strcat(str1, str2); cout << str1 << endl; str1[4] = '\0'; cout << str1 << endl;

}

The function strncat is like strcat except that it copies only a specified number of characters.

strncat(destination, source, int n);

It may not copy the terminating null character.

Example:

#include <iostream.h> #include <string.h> void main() { char str1[30]; strcpy(str1, "abc"); cout << str1 << endl; strncat(str1, "def", 2); str1[5] = '\0'; cout << str1 << endl; char str2[] = "xyz"; strcat(str1, str2); cout << str1 << endl; str1[4] = '\0'; cout << str1 << endl;

}

**Comparing strings**

Strings can be compared using strcmp or strncmp functions The function strcmp compares two strings.

strcmp(str1, str2); strcmp returns: < 0 if str1 is less than str2

= 0 if str1 is equal to str2

> 0 if str1 is greater than str2

**Example**:

#include <iostream.h> #include <string.h> void main() { cout << strcmp("abc", "def") << endl; cout << strcmp("def", "abc") << endl; cout << strcmp("abc", "abc") << endl; cout << strcmp("abc", "abcdef") << endl; cout << strcmp("abc", "ABC") << endl;

}

The function **strncmp** is like **strcmp** except that it compares only a specified number of characters.

strncmp(str1, str2, int n);

**strncmp** does not compare characters after a terminating null character has been found in one of the strings.

**Example:**

#include <iostream.h> #include <string.h> void main()

{ cout << strncmp("abc", "def", 2) << endl; cout << strncmp("abc", "abcdef", 3) << endl; cout << strncmp("abc", "abcdef", 2) << endl; cout << strncmp("abc", "abcdef", 5) << endl; cout << strncmp("abc", "abcdef", 20) << endl;

}

**Summary**

An array is a data structure which allows a collective name to be given to a group of elements which all have the same type.

* Thus the elements of an array are ordered by the index
* An array can be one dimensional or multi-dimensional
* An array may have more than one dimension. Each dimension is represented as a subscript in the array. Therefore a two dimensional array has two subscripts, a three dimensional array has three subscripts, and so on.
* String in C++ is nothing but a sequence of character in which the last character is the null character ‗\0‘.
* In C++ strings of characters are held as an array of characters, one character held in each array element.
* A special null character, represented by `\0', is appended to the end of the string to indicate the end of the string.

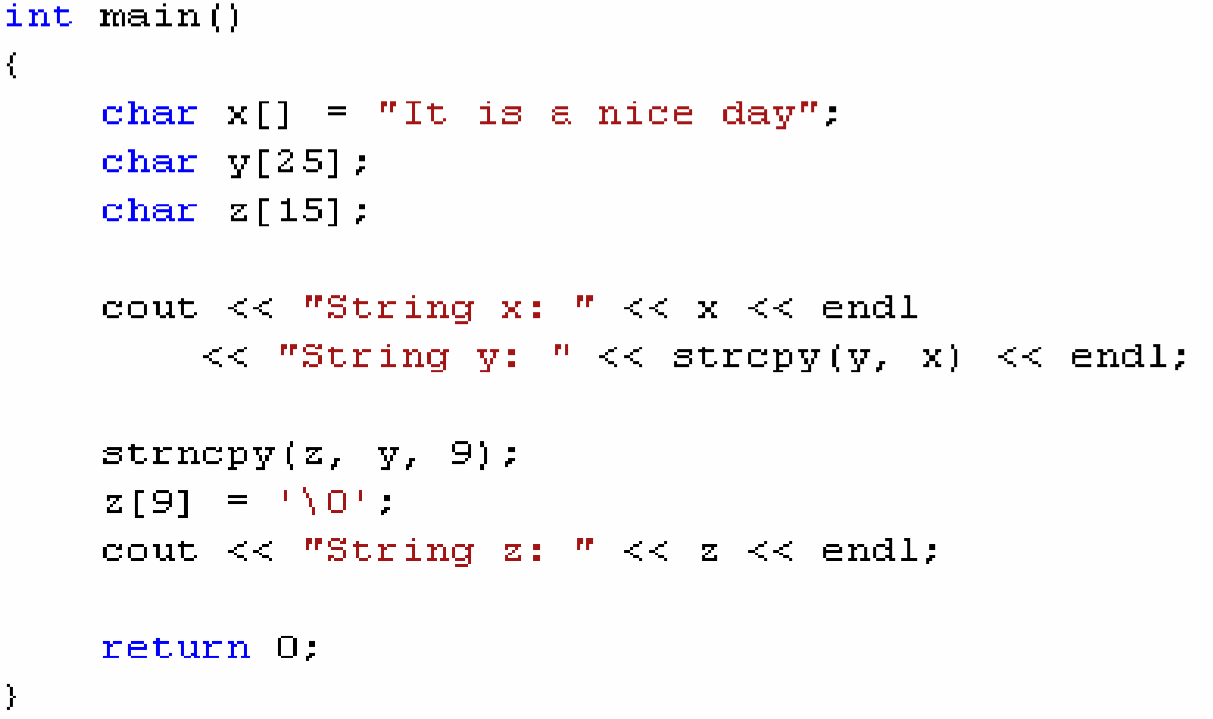
**Exercise**

1. Write an array declaration for the following
   1. A list of 100 floating\_point voltage
   2. A list of 100 integer years
   3. A list of 30 characters
   4. A list of 32 floating-point velocities
2. Write a program to input the following values into an array named volts: 10.95, 16.32, 8.22, 15.98, 26.22, 13.54, 8.22, 6.45, and 173.86. After the data has been entered, have your program output the values.
3. Write a program to input eight integer numbers into an array named temp. As each number is input, add the number into a total. After all numbers are input, display the number and their average.
4. Determine the output produced by the following program #include <iostream.h> int main()

{

int i, j,val[3][4]={8,16,9,52,3,15,27,6,14,25,2,10}; for (i=0;i<3;i++) for(j=0;j<4;j++) cout<<‖ ―<<val[i][j]; return 0;}

1. Write a c++ program to select the values in a four by five array of integers in increasing order and store the selected values in a single dimensional array named sort. Use the following values to initialize the array 16, 22, 99, 4, 18, -258, 4, 101, 5, 98, 105, 6, 15, 2, 45, 33, 88, 72, 16.
2. What is the output of the following program?



1. Write a C++ program that takes two words from a user and compares their alphabetical precedence.

**5 Reference**

1. *Walter Savitch, Problem Solving with C++( 6th ed), USA, Addison Wesley,2006*
2. *Dromey, R.G., How to solve it by computer, UK, Prentice Hall Inc. ,1982*
3. *GaddisTony, Starting out with C++, USA , Scott/Jones Inc. Publishers, 2001*
4. *Schildt Herbert, C++ - The Complete Reference(4th ed), USA, McGraw Hill Inc. 2001*
5. *Lafore, Robert, Object Oriented Programming in C++ (2nd ed),2001*
6. *Dietel&Dietel, ―****C How To Program****, Third Edition, Prentice – Hall, 2003*
7. *Robert Lafore, ―****The Waite Group‘s programming Using Turbo C++*** *Techmedia, 1993*
8. *John R. Hubrard, ―****Fundamentals of Computing with C++****,‖ Shuam‘s Outline, 1997*
9. *Jess Liberry, ―****An Introduction to C++*** *1995*
10. *Robert Lafore, ―****The Wait Group Object Oriented programming With C++*** *1994*