1. Introduction to Computer System

1.1. Computer System Components

Computer is an electronic device that accepts processes, displays, and stores data.

A **computer** is a device capable of performing computations and making logical decisions at speeds millions (even billions) of times faster than human beings can.

System is a group of components, consisting of subsystems or procedures that work in a coordinated fashion to achieve some objective.

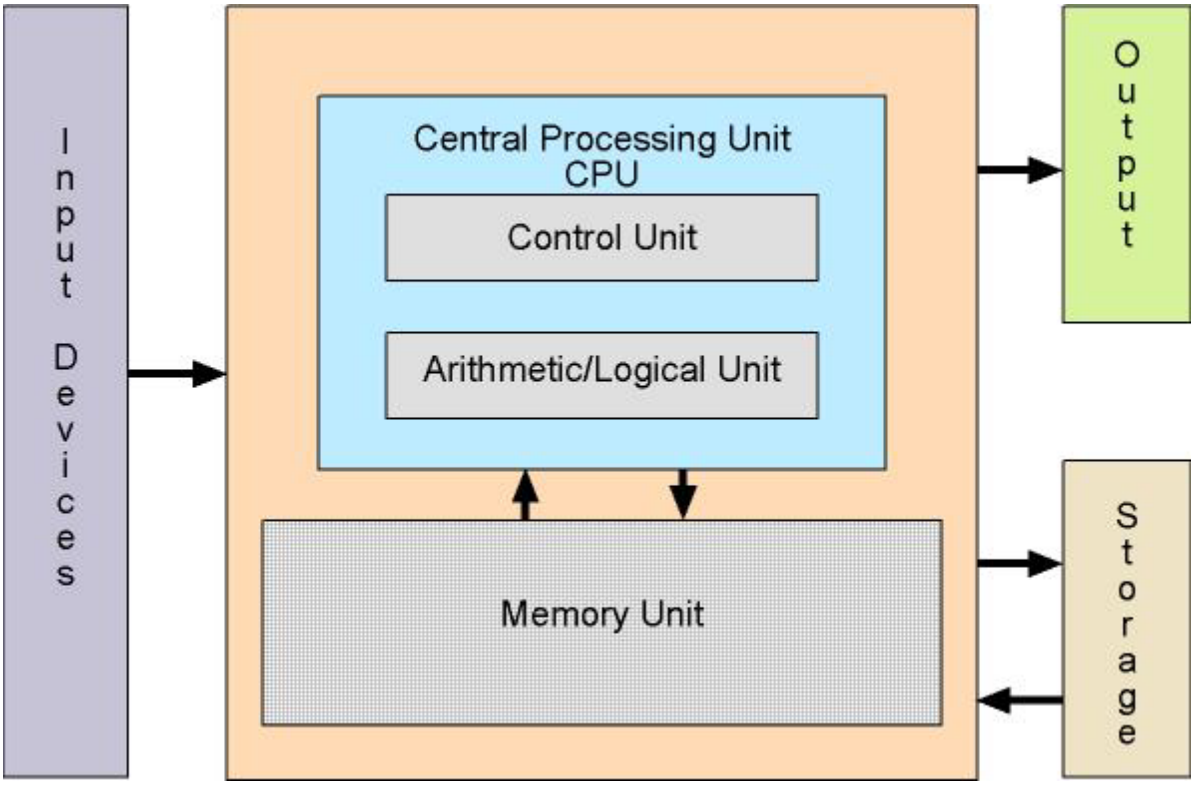
A computer system is composed of both **hardware** and **software** components.

1.1.1. Hardware components

Computer hardware is the physical equipment of the computer you see and touch.

Hardware can be classified as:

- Input devices
- Processing Unit (CPU)
- Output devices
- Storage devices(Primary &Secondary)



Computer block diagram

Input Devices

You use input devices to put information into the computer or convert any format of information in a format that can be processed by computer. Typical input devices are listed below.

- Keyboard
- Mouse
- Touch Sensitive Screen
- Scanner
- Light Pen
- Digital Stills Camera
- Voice Data Entry

Processor – CPU

The CPU of the computer is the brain of the computer which performs all arithmetic, logical and control operations. The function of the CPU is to execute programs stored in the main memory by fetching instructions, examining them, and executing them one after the other.

The CPU consists of a **contro**l unit, an **arithmeti**c logic unit (ALU), and **registers**.



Units of a processor chip (CPU)

Control Unit

Tasks carried out by control unit are listed below:

1. Decoding the instructions within a computer
2. Sequencing the reading and writing of data within the CPU and externally on the data bus
3. Controlling the sequence in which instructions are executed
4. Controlling the operations performed by the ALU

ALU (arithmetic and logic unit)

The arithmetic logical unit (ALU) is responsible for performing arithmetic (+,-, ×, ÷) and logical operations (comparisons like>, <, =, ≠) ondata.

CPU - Registers

The CPU also contains a small high speed memory which is used to store temporary results and control information.They hold various types of information such as data, instructions, addresses and the intermediate results of calculations. Essentially, they hold the information that the CPU is currently working on. Registers can be thought as CPU's working memory. As soon as a particular instruction or piece of data is processed, the next instruction immediately replaces it, and the information that results from the processing is returned to main memory. This memory consists of a number of registers, each performing a specific function

Output Devices

These devices display information that has been held or generated within a computer. Convert information processed by CPU to human understandable format either visually or in any format. Some examples are shown below.

- VDU or Monitor
- Printer
- Plotter
- etc.

Storage Devices

Storage devices are hardware that designed to store information or data. There are two types of storage devices. These are: 1. Primary Storage Devices 2. Secondary Storage Devices.

Primary Memory

The primary memory of a computer is the part used to hold /store/ programs and data.The memory of a computer is constructed from microchips. Memory can be thought of as a set of pigeon holes or cells with each one having a unique address. Each pigeon hole can store 8 bits of data.

A computer contains two types of main memory, Random Access Memory (RAM) and Read Only Memory (ROM).

Random Access Memory

Random Access Memory (RAM) is responsible for storing the instructions and data that the computer is using at that present moment in time.
It is described as volatile memory as the contents of RAM chips can be lost when the computer is turned off or when new data is being written to RAM while other data is being processed. It is Volatile.

Read Only Memory

ROM is responsible for storing permanent data and instructions. It is non-volatile.

Secondary Storage Devices

In many cases the information that has been processed is stored in machine readable format so that it may be accessed at a later time by a computer. This data is stored in binary form in 'bits'. This practice requires the use of storage devices.

Storage Devices:

- Hard Disk
- Floppy Disk
- Magnetic Tape
- Optical Disk

1.1.2. Computer Software

Computer hardware cannot perform alone. To accomplish any task, hardware must be given a serious of instructions called software. The finite set of instructions that the computer follows to do a given job is called program. The group of programs that the computer needs to function is known software.
We can classify software in two major parts
1. System Software
2. Application Software

System software

Refers to programs designed to perform tasks associated with directly controlling and utilizing computer hardware. It does not accomplish specific task to the user, such as creating document or analyzing data. System software constitutes those programs, which increase the facilities of the hardware system. It lays the hardware and the application software.

Systems software was developed in order to automate the following routine tasks:

- Allocation of various programs to main memory
- Loading of programs that perform routines to control peripheral devices.
- Execution of programs.

Systems software provides a means of standardizing the various activities performed during the execution of these tasks.

Systems software has been categorized into **Operating** Systems, **utility** software and **programming** language.

A. Operating Systems: The most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers. It is a program that acts as an interface between the user, the computer software and the hardware resources

Basic functions of OS:

- Process Management
- Memory Management
- Device Management
- File Management
- Security Management
- User interfacing
- Coordinating of communication on the network

The operating system (OS) is loaded from secondary memory into main memory when the computer is turned on, and remains in memory until the computer is turned off.Examples: MacOS, Windows-95, Windows-NT, UNIX, Linux, Solaris,...

B. **Utility programs**: this is another category for system software generally used to support, enhance, or expand existing programs in a computer system. Many operating systems have utility programs built in for common purposes. Additional utility programs are available separately. Some of examples of the utility programs are the following.

- **Backup software**: it is a type of software which facilitate to a user to have a recovery copy if his/her computer suddenly stop.
- **Data recovery**: it is used to restore data that has been physically damaged or corrupted. Data can be damaged by viruses, bad software, and hardware failure and power fluctuation.
- **Virus protection**: if there is an attack of computer viruses on your computer system there may be a need to have antivirus utility software.
- **Data compression**: remove redundant elements, gaps and unnecessary data from the computer storage space so that you will have enough space to work and to transport data.

C. **Language software**: it is the other subdivision of system software which is normally used by the programmer to develop application program.

Application software

Application software refers to those programs developed to meet particular user requirements for specific application.

The common application programs are.

Word processing: it is a type of software which permits the user to create, edit, format, file and print usually text documents. E.g. MS-Word, Word Perfect, WordStar, and WordPad.

Spreadsheet: it is an electronic tabular spreadsheet displayed on the VDU of the computer. It provides the user the opportunity to create, edit, format, file, print, calculate, and analyze data in tabular form. E.g. FoxPro, MS-Excel and so on.

Database management: serve to create computerized datable, to insert records, to make queries, to analyze data, and append data. Eg.MS-Access

Graphics software: these are a type of application software which provides the user the workspace to create, edit, and format image and graphical information. E.g.AutoCAD,

Presentation application software: which provide the user to create edit and present presentations, E.g. MS-power point

1.2. Programming

1.2.1. What is Programming?

A sequence of statements that instruct a computer how to solve a problem is called a program. The act of designing, writing and maintaining a program is called **programming**. In other words,programming is a skill that can be acquired by a computer professional that gives him/her the knowledge of making the computer perform the required operation or task.People who write programs are called **programmers**.

Why do we need to learn computer programming?

Computer programming is critical if one wants to know how to make the computer perform a task. Most users of a computer only use the available applications on the computer. These applications are produced by computer programmers. Thus if someone is interested to make such kind of applications, he/she needs to learn how to talk to the computer, which is learning computer programming.

What is programming language?

Programming Language: is a set of different category of written symbols that instruct computer hardware to perform specified operations required by the designer.

What skills do we need to be a programmer?

For someone to be a programmer, in addition to basic skills in computer, needs to have the following major skills:

• **Programming Language Skill**: knowing one or more programming language to talk to the computer and instruct the machine to perform a task.

• **Problem Solving Skill:** skills on how to solve real world problem and represent the solution in understandable format.

• **Algorithm Development:** skill of coming up with sequence of simple and human understandable set of instructions showing the step of solving the problem. Those set of steps should not be dependent on any programming language or machine.

In every programming Language there are sets of rules that govern the symbols used in a programming language. These set of rules determine how the programmer can make the computer hardware to perform a specific operation. These sets of rules are called **syntax.**

1.2.2. Generations of programming language

Programmers write instructions in various programming languages, some directly understandable by computersand others requiring intermediate translation steps. Hundreds of computer languages are in use today. Thesemay be divided into three general types:

1.2.2.1. Machine Language (ML)

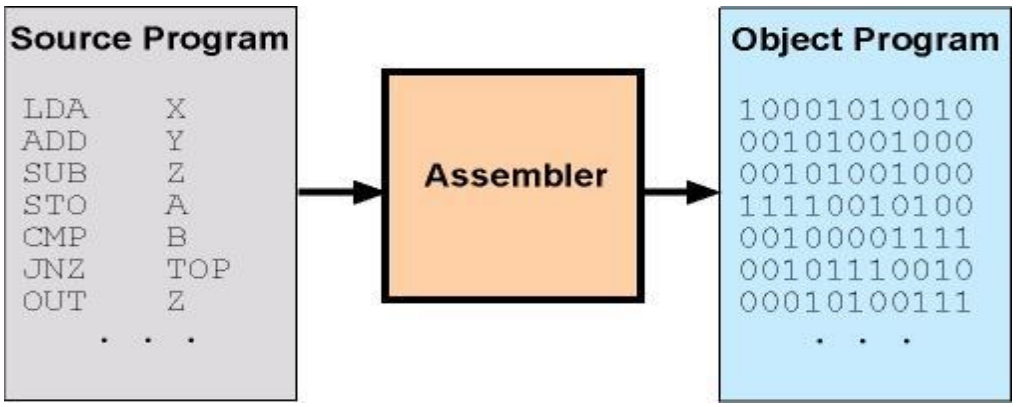
- A language which is directly understood or executed by the computer.
- It is normally developed using binary numbers (a sequences of 0s and 1s)
- Machine language, often referred to as object code, is the "natural language" of a computer and as such is defined by its hardware design.
- Allow the programmer to interact *directly with the hardware*, and it can be executed by the computer without the need for a translator.
- Is more powerful in utilizing resources of the computer.
- Gives power to the programmer.
- They execute very quickly and use memory very efficiently.

The main problems with using machine code directly are:

- ✓ it is very easy to make a mistake
- ✓ very hard to find and correct errors

1.2.2.2. Assembly Languages

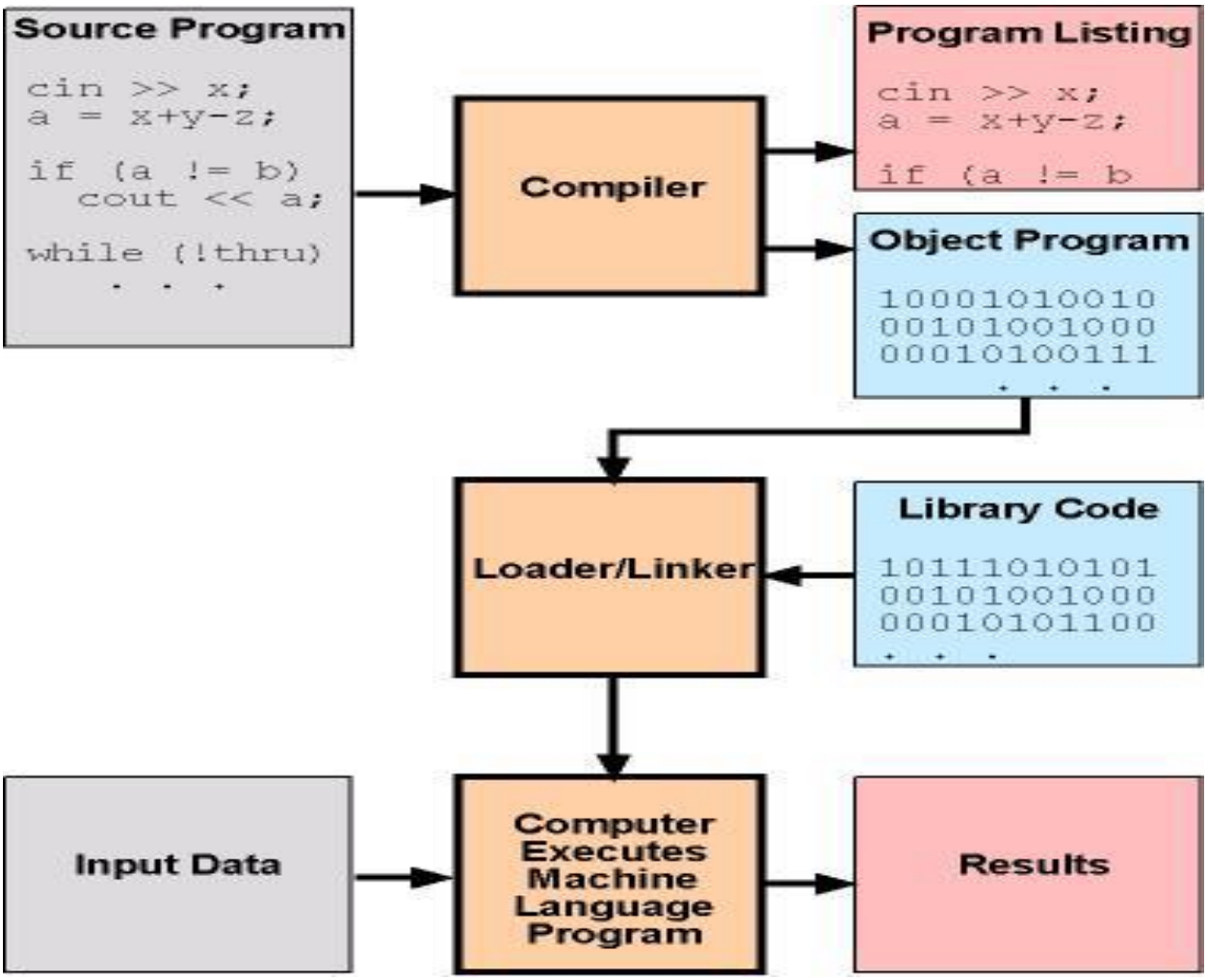
- A low-level programming language in which a mnemonic is used to represent each of the machine language instructions for a specificcomputer.Mnemonics is an alphabetical abbreviations used as a memory aid
- Uses symbolic names for operations and storage locations.
- An **Assemble**r is a computer program that translates an assembly languageprogram into machine code.
- They are highly used in system software development.
- Much easier to read programs
- Much easier to find and fix mistakes
- Programs written in assembly language are not portable. i.e., different computer architectures have their own machine and assembly languages.



1.2.2.3. High Level Languages

- A high level language contains instructions that closely resemble human language and mathematical notation and do not require the programmer to have detail knowledge about the internal operation of the computer.
- Translator programs called **compilers** convert high-level language programs into machine language.
- Much easier to read programs
- Much easier to find and fix mistakes
- Portable from one machine to another (so long as they keep to the language standard).
- Objectives in Programming

From the programmer's standpoint, high-level languages are preferable to machine and assemblylanguage. FORTRAN, Visual Basic.NET, C, C++ and Java are among the most widely used high-levelprogramming languages.



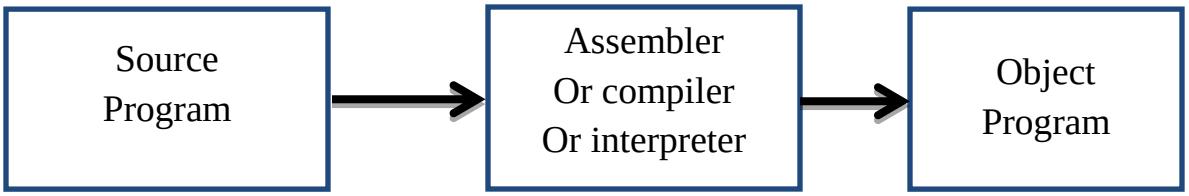
Translation and Execution

The only language that the computer understands is the machine language. Therefore any program that is written in either low-level or high level language must be translated to machine code so that the computer could process it. A program written in high-level or Assembly is called source code or source program and, the translated machine code is called the object code or object program. A program passes through different stages before it carries out its function. First the program will be converted to object code (compilation time), then it will be loaded to the memory and finally it will be executed (run time) or carries out its function.

- **Source code**: high-level language instructions
- **Object code**: translated instructions (a sequence of 0 and 1) ready for computer. For every modern language, there is a program that takes the source code and produces object code in machine language
- **An assembler:**is a software tool for translating assembly language into machine language.
- **Compiler**: is program that converts the instruction of a high level language into machine language as a whole. The compiler translates source code once and produces a complete machine language program.The compiler not only converts the instructions into machine code but also it detects whether the program fulfills the syntax of the 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 levelstatement and executes it and then goes to the next high level language line. An interpreter is a program that converts one statement of a program at a time. It executes this statement before translating the next statement of the source program. If there is an error in the statement, the interpreter will stop working and displays an error message.

  The advantage of interpreters over compilers is that an error is found immediately. So the programmer can make corrections during program development.The disadvantage of interpreter is that it is not very efficient. The interpreter does not produce an object program. It must convert the program each time it is executed. Visual basic uses interpreter.
- **A linker** creates a link between function prototype and function definition and between different modules in modular programming.Linker is a program that combines source code and codes from the library. Linking is the process of bringingtogether source program and library code.



2. Problem solving

A problem is an undesirable situation that prevents the organization from fully achieving its purpose, goals and objectives. Or problem can also be defined as the gap between the existing and the desired situation where problem solving will try to fill this gap.

Problem solving is the process of transforming the description of aproblem into the solution by using our knowledge of the problem domain and by relying on our ability to select and use appropriate problem-solving strategies, techniques, and tools.

2.1. Problem solving approach

There are two approaches of problem solving:

Top down design: A solution method where the larger problem is broken down into smaller sub-problems, which in turn are broken down into smaller sub-problems, continuing until each sub-problem can be solved in a few steps. This approach will try to disintegrate a larger problem into more smaller and manageable problems to narrow the problem domain.

Bottom up design: is the reverse process where the lowest level component are built first and the system builds up from the bottom until the whole process is finally completed.

2.2. Program development

The program we design in any programming language need to be:

Reliable: the program should always do what it is expected to do and handle all types of exception.

Maintainable: the program should be in a way that it could be modified and upgraded when the need arises.

Portable: It needs to be possible to adapt the software written for one type of computer to another with minimum modification.

Efficient: the program should be designed to make optimal use of time, space and other resources of the computer.

2.3. Algorithm development

A computer is a useful tool for solving a great variety ofproblems.The sequence of steps to be performed in order to solve aproblem by the computer is known as an **algorithm**.

- An algorithm is a sequence of steps for solving a problem.
- Engineering problem solutions to real world problems require complex algorithms.
- Development of a good algorithm increases the quality and maintainability of a solution, and reduces the overall time required to implement a correct solution

Generally, an algorithm is a finite set of well-defined instructions for accomplishing some task which, given an initial state, will terminate in a corresponding recognizable end-state.

The algorithm should be:

- Precise and unambiguous
- Simple
- Correct
- Efficient

Example: 1) *Algorithm to **add two** numbers.*

Step 1: start
Step 2: Read two numbers n1 and n2.
Step 3: sum ⟵ n1 + n2
Step 4: Print sum
Step 5: Stop

Example: 2) *Algorithm to find **largest** number from **two** numbers.*

Step 1: start
Step 2: Read two numbers n1 and n2.
Step 3: If n1 > n2 then
Big ⟵n1
else
Big ⟵ n2
Step 4: Print Big
Step 5: Stop

Example: 3) *Algorithm to find **largest** number from **three** numbers.*

Step 1: start
Step 2: Read three numbers n1, n2 and n3.
Step 3: If n1 > n2 and n1 > n3 then

Big← n1
Else
If n2 > n1 and n2 > n3 then
Big ← n2
else
Big ← n3
Step 4: Print Big
Step 5: Stop.

Example 4) Algorithm *to find **sum of N** positive integer numbers*

Step 1: start
Step 2: Read N
Step 3: Sum ←0,
Step 4: Count ← 0
Step 5: Read Num
Step 6: Sum←Sum + Num
Step 7: count ← count +1
Step 8: If Count < N then go to step 5
Step 9: Print Sum
Step 10: Stop

Example 5) Algorithm to find LCM of two given numbers

Step 1: Read M, N
Step 2: A = M
B = N
Step 3: Temp = M % N
Step 4: M = N
Step 5: N = Temp
Step 6: if N > 0 then goto Step 2
Step 7: GCD = M
Step 8: LCM = A*B/GCD
Step 9: Print LCM
Step 10: Stop

2.4. Expressing algorithm: flow charts

Algorithms can be expressed in many different notations,including natural languages, pseudo code, flowcharts andprogramming languages. Natural language expressions of algorithmstend to be verbose and ambiguous, and are rarely used for complex ortechnical algorithms. Pseudo code and flowcharts are structured waysto express algorithms that avoid

many ambiguities common in naturallanguage statements, while remaining independent of a particularimplementation language. Programming languages are primarilyintended for expressing algorithms in a form that can be executed by acomputer, but are often used to define or document algorithms.

Flowchart is a graphical or symbolic representation of analgorithm. It is the diagrammatic representation of the step-by-stepsolution to a given problem.

A Flowchart is a type of diagram (graphical or symbolic) thatrepresents an algorithm or process.Each step in the process isrepresented by a different symbol and contains a short description ofthe process step.The flow chart symbols are linked together witharrows showing the process flow direction. A flowchart typicallyshows the flow of data in a process, detailing the operations/steps in apictorial format which is easier to understand than reading it in atextual format.

A flowchart describes what operations (and in what sequence)are required to solve a given problem.

The benefits of flowcharts are as follows:

- **Communication**: Flowcharts are better way of communicatingthe logic of a system to all concerned.
- **Effective analysis**: With the help of flowchart, problem can beanalyzed in more effective way.
- **Proper documentation**: Program flowcharts serve as a goodprogram documentation, which is needed for various purposes.
- **Efficient Coding**: The flowcharts act as a guide or blueprintduring the systems analysis and program development phase.
- **Proper Debugging**: The flowchart helps in debugging process.
- **Efficient Program Maintenance**: The maintenance of operating program becomes easy with the help of flowchart. It helps theprogrammer to put efforts moreefficiently on that part.

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

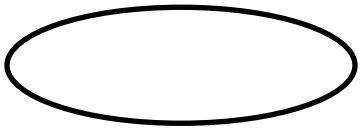
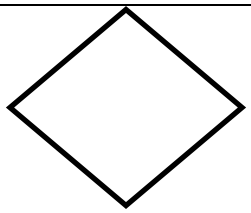
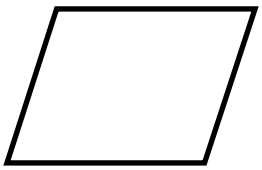
- **Complex logic**: Sometimes, the program logic is quitecomplicated. In that case, flowchart becomes complex and clumsy.

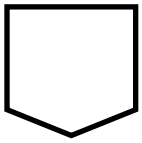
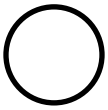
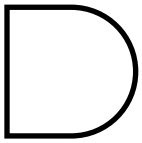
- **Alterations and Modifications**: If alterations are required theflowchart may require re-drawing completely.
- **Reproduction**: As the flowchart symbols cannot be typed,reproduction of flowchart becomes a problem.
- The essentials of what is done can easily be lost in the technicaldetails of how it is done.

Flowchart Symbols& guidelines

Flowcharts are usually drawn using some standard symbols;however, some special symbols can also be developed when required.

Some standard symbols, which are frequently required forflowcharting many computer programs are shown.

Symbol	Name	Description
	Terminal point	used to represent the Start and End of Program
	Decision	it usually contains a question within it. There are typically two output paths: one if the answer to the question is yes (true) , and the other if the answer is no (false)
	Process	Usually encloses operations (arithmetic or logical) or (Command block) a group of operations (a process)
	Input/output	used to represent data input or data output from a computer

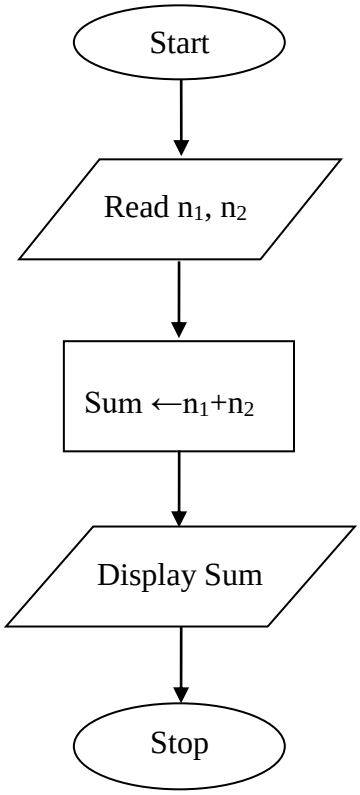
	Flow line	used to indicate the direction of logical flow (A path from one operation to another)
	Inter-page connector	
	On-page connector	Used toindicate a jump in the process flow. Connectors are generally used incomplex or multi-sheet diagrams.
	Delay	Used to indicate a delay or wait in the process for input fromsome other process.

These are the basic symbols used generally. Now, the ***basic guidelines***for drawing a flowchart with the above symbols are that:

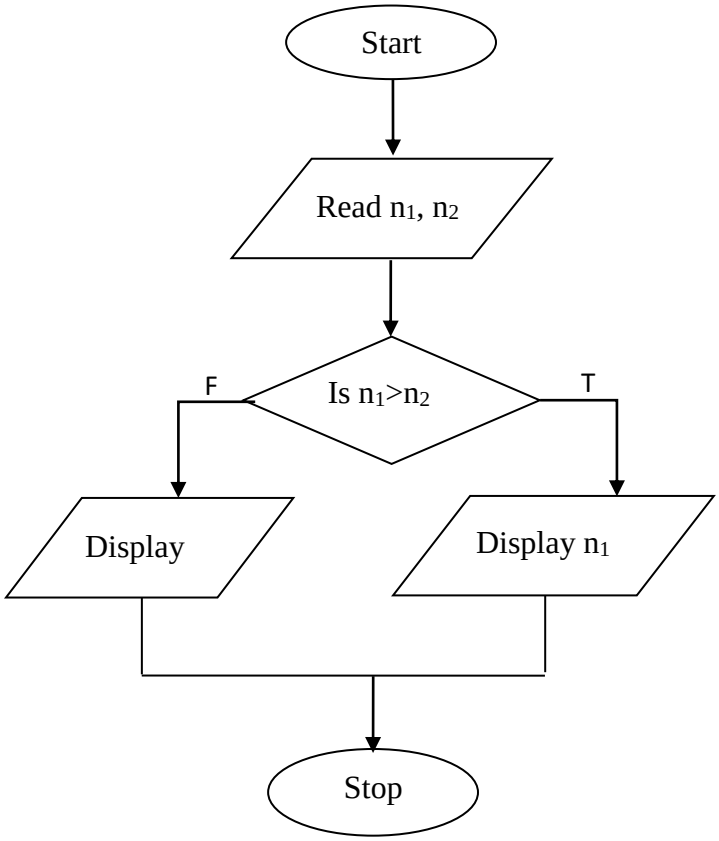
- In drawing a proper flowchart, all necessary requirementsshould be listed out in logical order.
- The flowchart should be neat, clear and easy to follow. There should not be any room for ambiguity in understanding theflowchart.
- The flowchart is to be read left to right or top to bottom.
- A process symbol can have only one flow line coming out of it.
- For a decision symbol, only one flow line can enter it, butmultiple lines can leave it to denote possible answers.
- The terminal symbols can only have one flow line inconjunction with them.

Examples

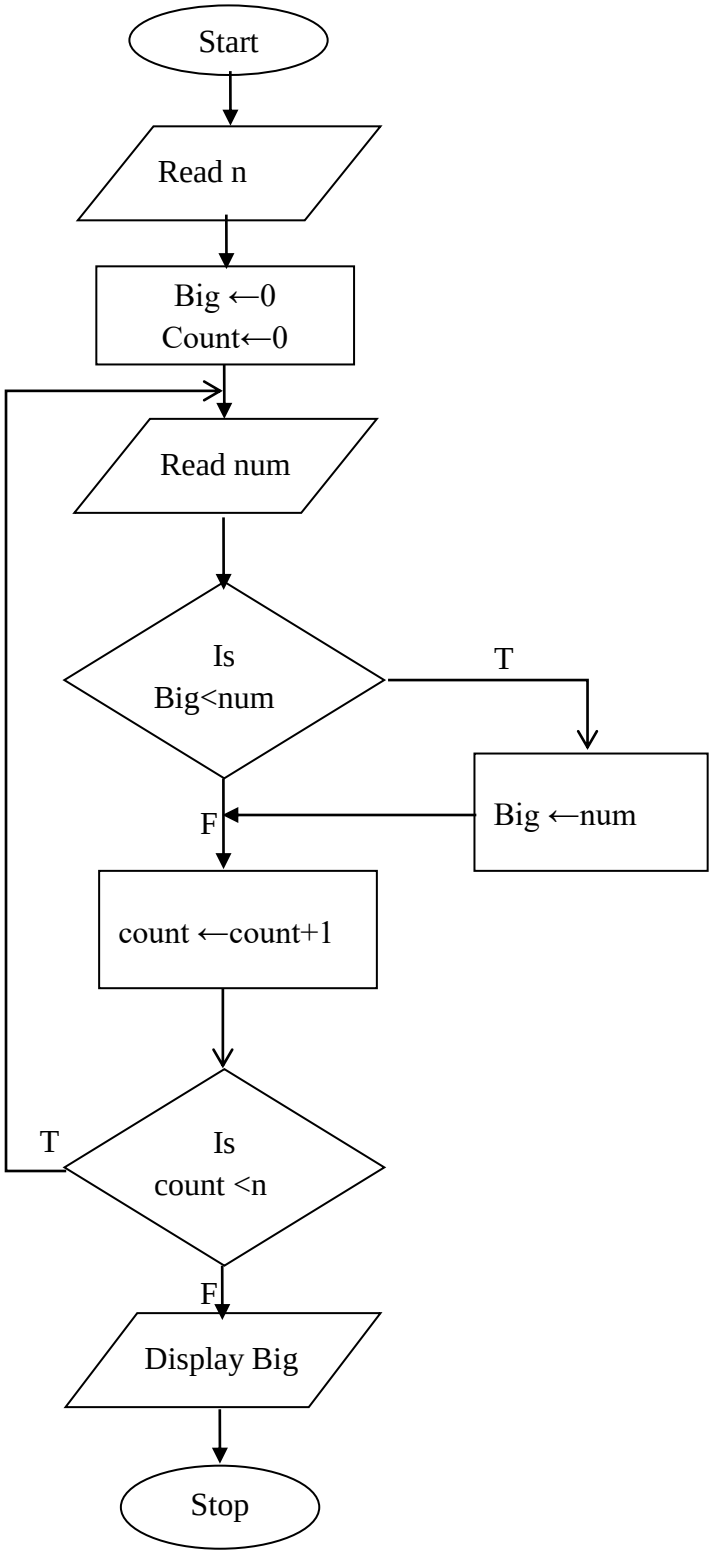
1. Flowchart to add two numbers.



2. A flowchart to find the largest number from two numbers



3. Algorithm and flowchart to find largest of n numbers



Algorithm

Step1: start

Step2: Read n

Step3: Big←0

Step4: count←0

Step5: Read num

Step6: if big<num then

Big←num

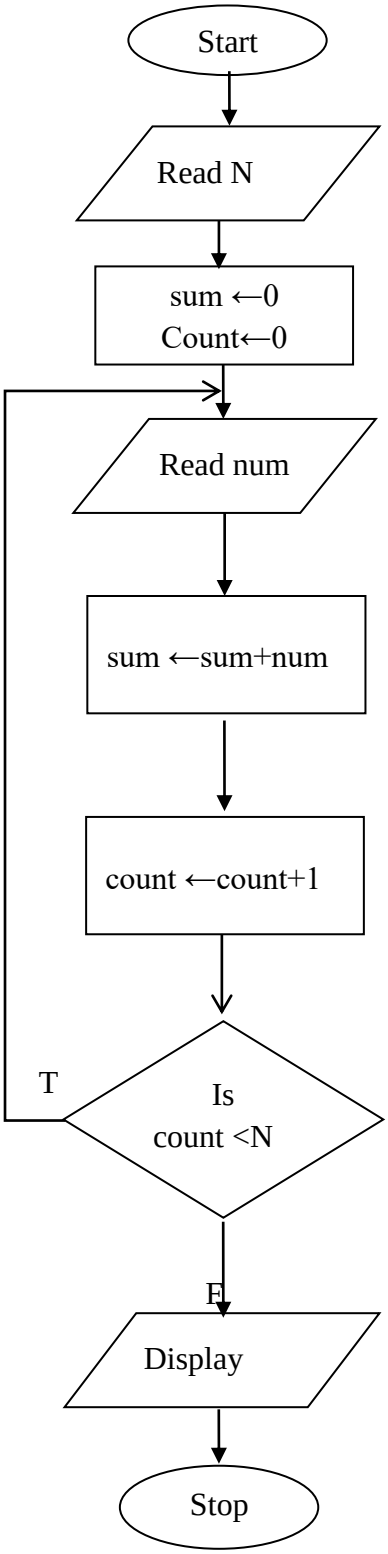
Step7: count←count+1

Step8: if count<n then goto step5

Step9: Display Big

Step10: Stop

4. Flowchart to find the sum of N positive integer numbers



Worksheet 1

For each of the problems below, develop an algorithm and a flow chart

1. Receive a number and determine whether it is odd or even.
2. Receive 3 numbers and display them in ascending order from smallest to largest
3. Add the numbers from 1 to 100 and display the sum
4. Add the even numbers between 0 and any positive integer number given by the user.
5. Find the average of two numbers given by the user.
6. Find the average, maximum, minimum, and sum of three numbers given by the user.
7. Find the area of a circle where the radius is provided by the user.
8. Swap the contents of two variables using a third variable.
9. Swap the content of two variables without using a third variable.
10. Read an integer value from the keyboard and display a message indicating if this number is odd or even.
11. Read 10 integers from the keyboard in the range 0 - 100, and count how many of them are larger than 50, and display this result.
12. Take an integer from the user and display the factorial of that number
13. A program to find the sum of first N even numbers.
14. A program to generate Fibonacci series. The first two digits in the series are 0 and 1 and the rest is computed by adding the preceding two digits. (i.e. 0 1 2 3 5 8 13 . . .)
15. A program to find the sum of digits of given number. If for example the given number is 536, the sum of digits in the number = 5 + 3 + 6 =14
16. A program that Checks a given number is palindrome or not. A palindrome number is the one which is the same when it is read from left to right.
17. A program that displays the number of digits of a given number.

3. C++ Programming Basics

3.1. What is C++?

C++ is a computer programming language.

C++ is a compiled language. This means that before the program is run, it is firsttranslated into a form that the machine can use directly. The C++ files (and a typicalproject contains several) are called the source files. You create the source files bytyping them using a program called a text editor.

The translation of the source files into a program proceeds in two phases. First,the individual source files are translated by a program called the compiler into so-calledobject files. Next, a second program, called a linker (or loader) combinesthe individual object files into an executable file, that is, a program you can execute(run).

C++ systems generally consist of three parts: a program development environment, the languageand the C++ Standard Library. C++ programs typically go through six phases: edit, preprocess,compile, link, load and execute.

Edit/Creating a Program: This consists of editing a file with an editor program (normally known simply as an editor). Youtype a C++ program (typically referred to as source code) using the editor, make any necessarycorrections and save the program on a secondary storage device, such as your hard drive. C++source code file names often end with the .cpp, .cxx, .cc or .C extensions. A simple text editor,such as Notepad in Windows, can be used to write the C++ code. We assume the reader knowshow to edit a program.

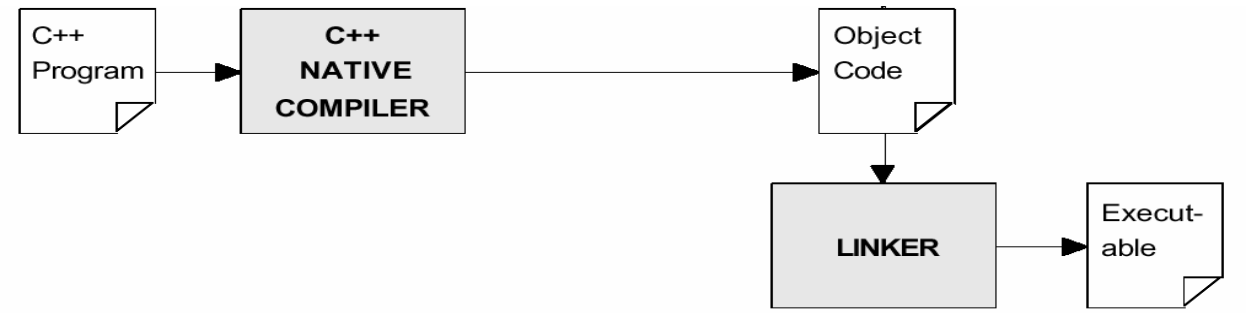
Preprocess: In a C++ system, a *preprocessor* program executes automatically before the compiler's translation phase begins. The C++ preprocessor obeys command called ***preprocessor directives***, which indicate that certain manipulations are to be performed on the program before compilation. The preprocessor is invoked by the compiler before the program is converted to machine language. The C++ **preprocessor** goes over the program text and carries out the instructions specified by the preprocessor 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++ program 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 memory. Additional components from shared libraries that support the program are also loaded.

Execution: 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.

3.2.**Structure of a C++ program**

To understand the basic parts of a simple program in C++, let's have a look at the following code:

```
// my first program in C++
 #include<iostream.h>
       #include<conio.h>
        void main()
       {
       cout<<"\n Hello World!";//this line prints Hello World
       getch();
       }
```

The general structure of a C++ program contains the following components. These are:

A. **Preprocessor directive**: The include instruction is a preprocessor instruction that directs the compiler to include a copy of the file specified in the angle brackets in the source code.The identifier iostream is the name of a file in the Standard C++ Library.The hash sign # is required to indicate that the word "include" is a preprocessor directive; the angle brackets <> are required to indicate that the word "iostream" (which stands for "input/output stream") is the name of a StandardC++ Library file. The expression <iostream> is called a standard header.The first character is the #. This character is a signal to the preprocessor. Each time you start your compiler, the preprocessor runs through the program and looks for the hash (#) symbols and act on those lines before the compiler runs.

The syntax for preprocessor directive statement is

#include<headerfilename.h>

The **.h** extension tells the compiler that the file is header file. The preprocessor directive statement instructs the compiler to bring all the contents of the header file to our program. Some of the header files that we are going to use are:

- **iostream.h**: - contains standard input and output functions.Some of the functions defined here are:

Cin>>: is standard input function that accepts input from the user. **Syntax:**

For single input

Cin>>var1;

For multiple inputs

Cin>>var1>>var2>>var3;

Cin will take value from the keyboard and store it in the memory. Thus the cin statement needs a variable which is a reserved memory place holder.

Cout<<: standard output function that displays its content to the Screen.**Syntax:**

For single output

Cout<<Var1;

For multiple output

Cout<<var1<<", "<<var2<<" and "<<var3;

<<**endl**: standard output function that displays a new line character

- **string.h:** contains standard string operation functions.

- **math.h:** contains common mathematical operations like: sin (x), cos(x)….
- **conio.h**: contains a large set of functions that are used to perform input and output operations. Such as, clrscr()(clears the screen), getch()(extracts a character from the user. Can also beused to pause programs.), etc….

B. **Comments:** comments are not executable statements. They will be ignored by the compiler. The purpose of comment is to clarify the purpose of the program, the need for each variables, statements and functions we use in the program. There are two types of comments:

- **Single Line Comment**: if the comment consists only one line. Anything after // until the end of the line on which it appears is considered a comment.

  Eg:

  cout<<var1; //this line prints the value of var1.

- **Multi line comment**: if a comment consists of more than one line, we use a multi-line comment. Multi line comments are represented by:

  /*

  */

  The comment will be written after the (*).

  Example: -

  /* A program that adds two numbers and
  displays the result to the user
  */

- Comments should be used to enhance (not to hinder) the readability of a program. The following two points, in particular, should be noted:
  - ❖ A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.
  - ❖ 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.
  - ❖ 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.

Note: As a general rule, the overall program should have comments at the beginning, telling you what the program does. Each function should also have comments explaining what the function does and what values it returns. Finally, any statement in your program that is obscure or less than obvious should be commented as well.

Comments at the Top of Each File

It is a good idea to put a comment block at the top of every file you write. The exact style of this block of comments is a matter of individual taste, but every such header should include at least the following information:

- The name of the function or program.
- The name of the file.
- What the function or program does.
- A description of how the program works.
- The author's name.
- A revision history (notes on each change made).
- What compilers, linkers, and other tools were used to make the program.
- Additionalnotes as needed.

C. Function

- In general, a function is a block of code that performs one or more actions.Functions consist of a **header** and a **body**. The header, in turn is made up of the **return** type, the **function name**, and the **parameters** to that function.
- The general form of a function is:

```
return-type function-name(parameter list);
{
body of the function
}
```

Where**, return** type: specifies the type of value the function returns. A function whichreturns nothing should have the return type void.The function **name** is simply a unique identifier.The **parameter list** is a comma-separated list of variablenames and their associated types that receive the values of the arguments when thefunction is

called.Arguments are the actual values passed to the function when it is called.Afunction may bewithout parameters, in which case the parameterlist is empty. However, even if there are no parameters, the parentheses are still required.

- Most functions are called, or invoked, during the course of the program, which is after the program starts execution.
- Program commands are executed line by line, in the order in which they appear in the source code, until a function is reached. The program then branches off to execute the function. When the function finishes, it returns control to the line of code immediately following the call to the function.
- A function may return a value using a return statement; a return statement also causes the function to exit.If there is no return statement at the end of a function, the function will automatically return void.
- A function that adds two integers might return the sum, and thus would be defined to return an integer value. A function that just prints a message has nothing to return and would be declared to return void.

Note:

- ✓ Any C++ program file should be saved with file name extension ".CPP"
- ✓ Every C++ program has a main () function.The main function is the point where program execution starts and contains variable declaration and statement blocks.
- ✓ The end of a single statement ends with semicolon (;).

3.3. Variables and Data Types

In C++ a variable is:

- a symbolic name for a memory location in whichdata can be stored and from which you can later retrieve that data.
- a reserved place in memory to store information in.
- used for holding data values so that they can be utilized in various computations in a program.

A variable will have three components:

- **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**: is a unique identifier for the reserved memory location

- **Value**: a value which can be changed by assigning a new value to the variable.

3.3.1. Data types

When programming, we store the variables in our computer's memory, but the computer must know what we want to store in them since it is not going to occupy the same space in memory to store a simple number, a letter or a large number.

Every name (identifier) in a C++ program has a type associatedwith it. This type determines what operations can be applied to thename (that is, to the entity referred to by the name) and how suchoperations are interpreted.Next you have a list of the existing fundamental data types in C++, as well as the range of values that can be represented with each one of them:

Name	size	Range
unsigned char	8 bits(1 byte)	0 to 255
char	8 bits(1 byte)	-128 to 127
Short int	16 bits(2byte)	-32,768 to 32,767
int	16 bits(2 byte)	-32,768 to 32,767
Unsigned int	16 bits(2 byte)	0 to 65,535
unsigned long	32 bits(4 byte)	0 to 4,294,967,295
long	32 bits(4 byte)	-2,147,483,648 to 2,147,483,647
float	32 bits(4 byte)	-3.4x10-38 to 3.4x10+38
double	64 bits(8 byte)	-1.7x10-308 to 1.7x10+308
long double	80 bits(10 byte)	-1.2x10-4932 to 1.2x10+4932
bool	8 bits(1 byte)	true or false (top 7 bits are ignored)

The **sizeof** operator is used to know the precise size represented for each data type by the computer.

Example:

```
#include<iostream.h>
int main ( )
{
Cout<<” short int” <<sizeof(short int)<<”bytes\n”;
```

```
Cout<<" longt int" <<sizeof(long int)<<"bytes\n";
Cout<<" int" <<sizeof( int)<<"bytes\n";
Cout<<" float" <<sizeof(float)<<"bytes\n";
Cout<<" double" <<sizeof(double)<<"bytes\n";
Return 0;
}
```

3.3.1.1.Signed and Unsigned

- Signed integers are either negative or positive. Unsigned integers are always positive.
- Because both signed and unsigned integers require the same number of bytes, the largest number (the magnitude) that can be stored in an unsigned integer is twice as the largest positive number that can be stored in a signed integer.
- In the example below, in case of unsigned, since all the 4 bits can be used to represent the magnitude of the number the maximum magnitude that can be represented will be 15 as shown in the example.
- If we use signed, we can use the first bit to represent the sign where if the value of the first bit is 0 the number is positive if the value is 1 the number is negative. In this case we will be left with only three bits to represent the magnitude of the number. Where the maximum magnitude will be 7.

E.g.: Lets us have only 4 bits to represent numbers

Unsigned

Binary				Decimal
0	0	0	0	→0
0	0	0	1	→1
0	0	1	0	→2
0	0	1	1	→3
0	1	0	0	→4
0	1	0	1	→5
0	1	1	0	→6
0	1	1	1	→7
1	0	0	0	→8
1	0	0	1	→9
1	0	1	0	→10
1	0	1	1	→11
1	1	0	0	→12
1	1	0	1	→13
1	1	1	0	→14
1	1	1	1	→15

Signed

Binary				Decimal
0	0	0	0	→0
0	0	0	1	→1
0	0	1	0	→2
0	0	1	1	→3
0	1	0	0	→4
0	1	0	1	→5
0	1	1	0	→6
0	1	1	1	→7
1	0	0	0	→0
1	0	0	1	→-1
1	0	1	0	→-2
1	0	1	1	→-3
1	1	0	0	→-4
1	1	0	1	→-5
1	1	1	0	→-6
1	1	1	1	→-7

3.3.2. Declaring Variables

In order to use a variable in C++, we must first declare it specifying which data type we wantit to be.The syntax to declare a new variable is:

Datatype Variable_Name;

For example,

```
int a;
float mynumber;
```

The declaration will instruct the computer to reserve a memory location with the name and size specified during the declaration.Good variable names indicate the purpose of the variable or they should be self –descriptive.

E.g. int myAge; //variable used to store my age

The name of a variable is called an identifier which should be unique in a program.

Identifiers

Identifiers are names given to variables which distinguish them from all other variables.A valid identifier is a sequence of one or more letters, digits or underscores characters (_).The length of

an identifier is not limited.

An identifier must:

- start with a letteror the underscore (_),
- consist only of letters, the digits 0-9, or the underscore symbol _
- not be a reserved word
- not contain arithmetic operators(+, *, -, /), punctuation characters (‘, “ @ $ ….) and any space

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

Very important: The C++ language is a "case sensitive" language. That means that anidentifier written in capital letters is not equivalent to another one with the same name butwritten in small letters. Thus, for example, the **RESULT** variable is not the same as the**result** variable or the **Result** variable. These are three different variable identifiers.

The standard reserved keywords are:

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

3.3.3. Initializing Variables

When a variable is assigned a value at the time of declaration, it is called variable initialization.This is identical with declaring a variable and then assigning a value to the variable immediately after declaration.

The syntax:

DataType variable name = initial value;

Example, int a = 0;

Or: int a;

a=0;

3.3.4. 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.

Global variables: 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.

Local Variables: the scope of the local variable is limited to the code level or block within which they are declared.

Example:
```
#include<iostream.h>
int num1;
int add( int x, int y)
{
int z;
….
}
void main()
{
unsigned short age;
float num2;
cout<<”\n Enter your age:”;
…
}
```

- In the above example, the integer data type num1 is accessible everywhere whereas z is only accessible in the add function and num2 is accessible in main function. This means cout<<z; or any statement involving z is only valid in add function.
- In C++ the scope of a local variable is given by the block in which it is declared.

- If it is declared within a function, it will be a variable with a function scope. If it is declared in a loop, its scope will be only in the loop, etc.

3.4. **Constants**

A constant is any expression that has a fixed value.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 and the programmer can't assign a new value to a constant later.

C++ provides two types of constants: literal and symbolic constants.

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

E.g.: 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.

E.g.: Int studentPerClass =15;

students = classes * studentPerClass;

studentPerClass is a symbolic constant having a value of 15 and 15 is a literal constant directly typed in the program.

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

Defining constants with #define:

The #define directive makes a simple text substitution.

Syntax:#define identifier value

E.g.: #define studentPerClass 15

In our example, each time the preprocessor sees the word studentPerClass, it inserts 15 into the text.

The *#define* directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you

append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

Defining constants with the const key word

With the `const` prefix we can declare constants with a specific type in the same way as we would do with a variable:

Syntax:const <data type of the constant >< name of the constant identifier ><value>;

E.g.: const int length = 15;

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

3.5. Expressions and Statements

Expressions: an expression is any computation which yields a value. The expression may consist of a single entity, such as a constant or variable, or it may consist of some combination of such entities, interconnected by one or more arithmetic operators. Expressions are evaluated to produce a result.

Examples,

```
a + b
x = y
t = u + v
x <= y
++j
```

Statement: is a simple or compound expression that can actually produce some effect.The smallest independent unit in a C++ program is a statement. 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.

e.g.: x = a + b;

Unlike in algebra, this statement does not mean that x equals a+b. This is read, "Assign the value of the sum of a and b to x," or "Assign to x, a+b."

Blocks: A block is a group of zero or more statements between balanced brackets and can be used anywhere a single statement is allowed.A block begins with an opening bracket ({) and ends with a closing bracket (}). Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon.For example

```
{
temp = a;
a = b;
b = temp;
}
```

This block of code acts as one statement and swaps the values in the variables a and b.

3.6. Operators

An operator is a symbol that causes 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

3.6.1. Arithmetic operators

C++ provides five basic arithmetic operators which are summarized in the table below.

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo(remainder)

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators.Modulo (%) is the operation that gives the remainder of a division of two values.

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.

E.g.:

```
int cost = 100;
int volume = 80;
double unitPrice = cost/ (double) volume;
```

3.6.2. Relational operator

C++ provides six relational operators in order to evaluate a comparison between two expressions.The result of a relational operator is a bool value that can only be true or false according to the result of the comparison.

operator	Name	Example
==	Equality	4==4 // gives 1
!=	Inequality	4!=4 //gives 0
<	Less than	4<4 //gives 0
<=	Less than or equal to	4<=4 //gives 1
>	Greater than	4>3 //gives 1
>=	Greater than or equal to	4>=5 //gives 0

The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For example,

'A' < 'F' would return true or 1. It is like (65 < 70)

3.6.3. Logical Operators

C++ provides three logical operators for combining logical expression. These are summarized in the table below.

operator	Name
!	Logical negation
&&	Logical and
||	Logical or

- Logical negation (!) is a unary operator, which negates the logical value of its operand. If its operand is non-zero, it produces 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.

  E.g.:

  !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.

3.6.4. Increment/Decrement Operators

The auto increment (++) and auto decrement (--) operators provide a convenient way of adding and subtracting 1 from a numeric variablerespectively.These are summarized in the table below. Theexamples assume the following variable definition:

int k = 5;

Operator	Name	example
++	Auto increment(prefix)	++k+10 //gives 16
++	Auto increment(postfix)	k++ +10 // give 15
--	Auto decrement(prefix)	--k + 10 // gives 14
--	Auto decrement(postfix)	k-- + 10 // gives 15

Both operators can be used in prefix and postfix form. Thedifference is significant. When used in prefix form, the operator is firstapplied and the outcome is then used in the expression. When used inthe postfix form, the expression is evaluated first and then the operatorapplied.

3.6.5. 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:

- ✓ A literal constant: Eg: x=12;
- ✓ A variable: Eg: x=y;
- ✓ An expression: Eg: x=y+2;

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.

E.g.:

n += 1; is equivalent to n = n + 1;

n -= 1;is equivalent to n = n – 1;

n /= 5; is equivalent to n = n / 5;

n *= m + 1 is equivalent to n = n * (m + 1);And the same is true for the rest.

3.6.6. Conditional Operator (? :)

The conditional operator takes threeexpressions as operands.The first operand is a condition, the second operand isthe value for the entire conditional expression if the condition is true and the third operand is the value for theentire conditional expression if the condition is false.

Its format is:

condition? result1: result2

If condition is true the expression will return result1, if it is not it will return result2.

E.g.: 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.

E.g.2:

```
int m=1,n=2,min;
min = (m < n ? m : n);
```

The value stored in min is 1.

E.g.3:

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

3.6.7. Comma Operator (,)

Multiple expressions can be combined into one expression using the comma operator.The comma operator takes two operands. It first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.The comma operator can be used during multiple declarations, for the condition operator and for function declaration, etc.

For example:

```
int m,n,min;
int mCount = 0, nCount = 0;
min = (m < n ? (mCount++, m): (nCount++, n));
```

Here, when m is less than n, mCount++ is evaluated and the value of m is stored in min otherwise, nCount++ is evaluated and the value of n is stored in min.

3.6.8. 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.

E.g.:

```
a = sizeof(char)
b = sizeof(int)
c = sizeof(1.55) etc
```

3.6.9. Bitwise Operators

C++ provides six bitwise operators for manipulating theindividual bits in an integer quantity. These are summarized in thetable below.

Operator	Name
~	Bitwise Negation
&	Bitwise And
|	Bitwise Or
^	Bitwise Exclusive Or
<<	Bitwise Left Shift
>>	Bitwise Right Shift

Bitwise operators expect their operands to be integer quantities and treat them as bit sequences. Bitwise negation is a unary operator which reverses the bits in its operands. Bitwise and compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise. Bitwise or compares the corresponding bits of its operands and produces a 0 when

both bits are 0, and 1otherwise. Bitwise exclusive or compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0 and 1 otherwise.

Bitwise **left shift** operator and bitwise **right shift** operator both take a bit sequence as their left operand and a positive integer quantity 'n' as their right operand. The former produces a bit sequence equal to the left operand but which has been shifted 'n' bit positions to the left.The latter produces a bit sequence equal to the left operand but which has been shifted 'n' bit positions to the right. Vacated bits at either end are set to 0.

3.6.10. Explicit type casting operators

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

e.g.

```
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

3.6.11. Operator Precedence

The order in which operators are evaluated in an expression is significant and is determined by precedence rules. Operators in higher levels take precedence over operators in lower levels.

Precedence Table:

Level	Operator	Description	Grouping
1	++ --	postfix	Left-to-right
2	++ -- ~ ! sizeof	prefix	Right-to-left
3	* / %	multiplicative	Left-to-right
4	+ -	additive	Left-to-right
5	<<= >>=	relational	Left-to-right
6	== !=	equality	Left-to-right
7	&&	logical AND	Left-to-right

8	||	logical OR	Left-to-right
9	?:	conditional	Left-to-Right
10	= ,+=, -= ,*=, /=,^= ,%= ,&=,/=,>>= <<=	assignment	Right-to-left
11	,	comma	Left-to-right

E.g.

a = = b + c * d

c * d is evaluated first because * has a higher precedence than + and = =.The result is then added to b because + has a higher precedence than = =. And then == is evaluated.

- Precedence rules can be overridden by using brackets.

  E.g. rewriting the above expression as:

  a = = (b + c) * d causes + to be evaluated before *.

- Operators with the same precedence level are evaluated in the order specified by the column on the table of precedence rule.

E.g.

a = b += c the evaluation order is right to left, so the first b += c is evaluated followed by a = b.

3.7. Types of errors in C++

Syntax error: This error occurs due to following reason.

i. Not following the grammatical rules used in declaration of identifier.
ii. Not declaring an identifier used in program.
iii. Not terminating statement by semicolon.
iv. Not providing equal number of opening and closing braces etc.

These errors can be rectified by the user as it is displayed while compiling the program.

Logical error: This error won't be displayed on the screen. However it will lead to display wrong results. Example: An infinite loop. This errorleads to abnormal termination of a program or infinite loop.

Runtime error: This error occurs while running a program by displaying the message listed below.

i. Division by 0.
ii. Overflow
iii. Underflow

Now let's have an example:

Q. Write a corrected code of the following program segment by underlining the error corrected.

```
void main()
{
    int a,b;
    cin<<a;
b=a;
    cout <<"b=",a;
};
```

Answer:

```
#include<iostream.h>
void main()
{
    int a,b;
cin>>a;
b=a;
cout <<"b="<<a;
}
```

Worksheet 2

Course Title: Introduction to Computing Course Code: ECEG 2121

For each of the problems write a C++ code to perform the required task. Your program should be based on the flow chart you drawn in the first worksheet.

1. Receive a number and determine whether it is odd or even.
2. Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
3. Receive 3 numbers and display them in ascending order from smallest to largest
4. Add the numbers from 1 to 100 and display the sum
5. Add the even numbers between 0 and any positive integer number given by the user.
6. Find the average of two numbers given by the user.
7. Find the average, maximum, minimum, and sum of three numbers given by the user.
8. Find the area of a circle where the radius is provided by the user.
9. Swap the contents of two variables using a third variable.
10. Swap the content of two variables without using a third variable.
11. Read an integer value from the keyboard and display a message indicating if this number is odd or even.
12. read 10 integers from the keyboard in the range 0 - 100, and count how many of them are larger than 50, and display this result
13. Take an integer from the user and display the factorial of that number

4. Control statements

The normal flow of execution in a high level language is **sequential,** i.e., each statementis executed in the order of its appearance in the program. However, depending onthe requirements of a problem it might be required to alter the normal sequence ofexecution in a program. The statements which specify the order of execution ofstatements are called **flowcontrol**statements.Flow control is an important concept in programming because it will give all the power to the programmer to decide what to do to execute during a run and what is not, therefore, affecting the overall outcome of the program.

4.1. Selection Statements

Selection statements are statements in a program where there are points at which the program will decide at runtime whether some part of the code should or should not be executed.The term **conditional statement** is oftenused in place of selection statement.There are two types of selection statements in C++, which are the "**if** statement" and the "**switch** statement"

4.1.1. The if Statement

It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The different forms of the 'If" statement will be used to decide whether to execute part of the program based on a condition which will be tested either for TRUE or FALSE result.

The different forms of the "If" statements are:

- The simple if statement
- The if else statement
- The if else if statement

The simple if statement

- The simple if statement will decide only one part of the program to be executed if the condition is satisfied or ignored if the condition fails.
- The General Syntax is:

```
if (expression)
statements;
```

- In any "if" statement, first the "expression" will be evaluated and if the outcome is non-zero (which means TRUE), then the "statements" is executed. Otherwise, nothing happens (the statement will not be executed) and execution resumes to the line immediately after the "if" block.

  E.g.:

```
if(age>18)
cout<<"you are an adult";
```

- To make multiple statements dependent on the same condition we can use a compound statement, which will be implemented by embracing the group of instructions within the left "{"and right "}" French bracket.

  E.g.:

```
if(a<b)
{
cout<<"a is less than b";
a=a+1;
}
```

- Most of the time "expression" will have relational expressions testing whether something is equal, greater, less, or different from something else.
- It should be noted that the output of a relational expression is either True (represented by anything different from zero) or False (represented by Zero).
- Thus any expression, whose final result is either zero or none zero can be used in "expression"

  E.g.:

```
int x;
cin>>x;
if(x)
cout<<"you are an adult";
```

- In the above example, the value of variable x will be an input from the user. The "if" statement will be true if the user gives anything different from zero, which means the message "you are an adult" will be displayed on the screen. If the user gives zero value for x, which will be interpreted as False, nothing will be done as the if statement is not satisfied.
- Thus, expression can be:
  - ✓ Relational expression,
  - ✓ A variable,
  - ✓ A literal constant, or
  - ✓ Assignment operation, as the final result is whatever we have at the right hand side and the one at the left hand side is a variable.

The if else statement

- The "if else if" statement allows us to specify two alternative statements:
  - ✓ One which will be executed if a condition is satisfied and
  - ✓ Another which will be executed if the condition is not satisfied.
- The General Syntax is:

```
if (expression)
statements1;
else
statements2;
```

- First "expression" is evaluated and if the outcome is none zero (true), then "statements1" will be executed. Otherwise, which means the "expression" is false "statements2" will be executed.

E.g.1:

```
if (age>18)
cout<<"you are an adult";
else
cout<<"You are a kid";
```

E.g.2:

```
int x;
cout<<"Enter a number: ";
cin>>x;
if(x%2==0)
cout<<"The Number is Even";
else
cout<<"The Number is Odd";
```

E.g.3:

```
int x;
cout<<"Enter a number: ";
cin>>x;
if(x%2)
cout<<"The Number is Odd";
else
cout<<"The Number is Even";
```

- The above three examples illustrate the "if else" statement.
- The last two examples will do the same thing except that the expression in the Eg3 is changed from relational to arithmetic. We know that the result from the modulus operator is either 0 or 1 if the divider is 2. Thus the final result of the expression in Eg3 will either be 1 for odd numbers or be 0 for even numbers

The "if else if" statement

- The third form of the "if" statement is the "if…else if" statement.
- The "if else if" statement allows us to specify more than twoalternative statements each will be executed based on testing one or more conditions.

The General Syntax is:

```
if (expression1)
statements1;
else if(expression2)
statements2;
    .
    .
    .
else if(expressionN)
statements N;
else
statements;
```

- First "expression1" is evaluated and if the outcome is none zero(true), then "statements1" will be executed. Otherwise, which means the "expression1" then "expression2" will be evaluated: if the output of expression2 is True then "statements2" will be executed otherwise the next "expression" in the else if statement will be executed and so on.
- If all the expressions in the "if" and "else if" statements are Falsethen "statements" under else will be executed.
- The "if…else" and "if…else if" statements are said to be exclusiveselection as if one of the condition (expression) is satisfied only instructions in its block will be executed and the rest will be ignored.

  E.g.:

```
if(score >= 90)
        cout<< "\n your grade is A";
else if(score >= 80)
        cout<< "\n your grade is B";
```

```
else if(score >= 70)
        cout<< "\n your grade is C";
else if(score >= 60)
        cout<< "\n your grade is D";
else
        cout<< "\n your grade is F";
```

- In theabove example,only one of the five coutstatements will beexecuted and the rest will be ignored. But until one of the conditions is satisfied or the else part is reached, the expressions will be tested or evaluated.

Nesting If statements within another if statement

- One or more if statements can be nested within another ifstatement.
- The nestingwill be used to test multiple conditions to perform atask.
- It is always recommended to indent nested if statements to enhancereadability of a program
- The General Syntax might be:

```
if (expression1)
    {
if (expression2)
statementsN;
        else
  statementsM;
        }
   else
{
 if (expression3)
  statementsR;
else
        statementsT;
        }
```

- StatementsN will be executed if and only if “expression1” and“expression2” are evaluated and if the outcome of both is none zero (TRUE).
- StatementsM will be executed if and only if “expression1” is TRUEand “expression2” is FALSE.
- StatementsR will be executed if and only if “expression1” is FALSEand “expression3” is TRUE.
- StatementsT will be executed if and only if “expression1” is FLASEand “expression2” is FALSE

Let us an example on the nested if staement

```
#...
int main()
{
int testScrore, age;
cout<<”\n Enter your test score:”;
cin>>testScore;
cout<<“\n enter your age:”;
cin>>age;
if(testScore >= 70)
{
  if (age<10)
cout<<“\n You did a great job”;
else
cout<<”\ You did pass”;
}
else
cout<<”\ You did not pass”;
getch():
return 0;
}
```

4.1.2.The Switch Statement

- C++ has a multi-way decision statement called switch. The switch statement provides a way of choosing between a set of alternatives based on the value of an expression.
- The switch statement has four components:
  - ✓ Switch

- ✓ Case
- ✓ Default
- ✓ Break

Where Default and Break are Optional

The General Syntax might be:

```
switch (expression)
{
case constant1:
statements;
break;
        .
        .
        .
case constant n:
statements;
break;
default:
statements;
}
```

First expression (called the **switch tag)** is evaluated and the outcome, which is a constant value, will be compared to each of the numeric constants (called the **case labels**), in the order they appear, until a match is found. The statements following the matching case are then executed. Note the plural: each case may be followed by zero or more statements (not just one statement). After one case is satisfied, execution continues until either a break statement is encountered or all intervening statements are executed, which means until the execution reaches the right French bracket of the switch statement.The final default case is optional and is exercised if none of the earlier cases provide a match.

The break statement is most importantly used within a switch statement, which can select among several cases. To interrupt the normal flow of control within a loop, the programmer can use the special statement: **break**;

There are three important things to know about the switch statement:

- ✓ The switch differs from the if in that switch can only test for equality, whereasif can evaluate any type of relational or logical expression.
- ✓ No two case constants in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch may have case constants thatare the same.
- ✓ If character constants are used in the switch statement, they are automaticallyconverted to integers.

The switch statement is often used to process keyboard commands, such as menu selection. As shown here, the function menu () displays a menu for a spelling-checker program and calls the proper procedures:

Now let us see the effect of including a break statement in the switch statement.

Scenario one

```
switch (N){
 case 1: x=10;
case 2: x=20;
case 3: x=30;
}
```

Even if N is 1 or 2 x will have 30

Scenario two

```
switch (N){
 case 1: x=10; break;
case 2: x=20;break;
case 3: x=30;break;
}
```

x will have either 10, 20 or 30 based on the value of N.

- The break terminates the switch statement by jumping to the end of it.
- There are, however, situations in which it makes sense to have a case without a break. For instance:

E.g.:

```
switch(operator)
{ case '+': result = operand1 + operand2;
break;
case '-' : result = operand1 – operand2;
break;
case 'x':
```

```
case '*': result = operand1 * operand2;
case '/': result = operand1 / operand2;
break;
default: cout<<" unknown operator:" <<operator<<"\n";
}
```

- Because case 'x' has no break statement (in fact no statement at all!), when this case satisfied, execution proceeds to the statements of the next case and the multiplication is performed.
- Switch evaluates expression and compares the result to each of the case values.
- Relational and Boolean operators can be used in switch tag if and only if the expected output is either 0 to represent False or 1 to represent true as that is the only possible output from such operators.

4.2. Iteration Statements

- Iteration statements control a block of code to be executed repeatedly for a fixed number of times or until a certain condition fails.It is also called a Repetitive control structure.
- There are three C++ repetition statements:
  - ✓ for statement or loop
  - ✓ while statement or loop
  - ✓ do…while statement or loop

4.2.1.The for statement / loop

- The "for" statement (also called loop) is used to repeatedly execute a block of instructions until a specific condition fails.
- The general syntax is:

```
for (expression1; expression2 ; expression3)
Statements;
```

- The for loop has three expressions:

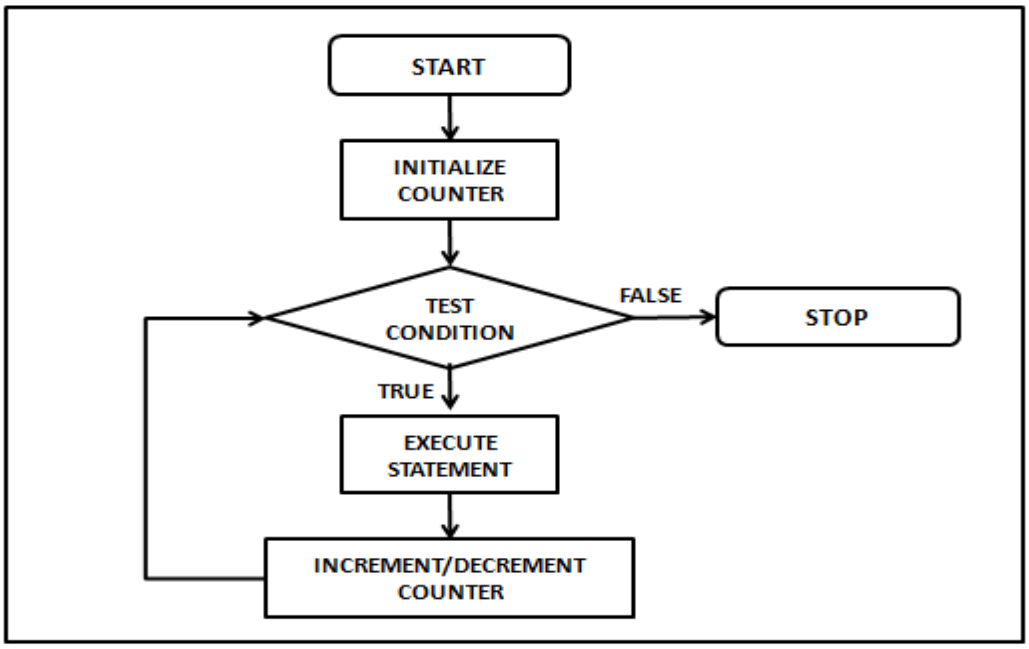
- ✓ **expression**1: is one or more statement/s that will be executed only once and before the looping starts.
- ✓ **expression2**: is the part that decides whether to proceed with executing the instructions in the loop or to stop. Expression2 will be evaluated each time before the loop continues. The output of expression2 should be either zero (to proceed with the loop) or none zero (to stop the loop) to represent false and true output respectively.
- ✓ **expression3**: is one or more statement/s that will be executed after each iteration.

- In most programs, the "for loop" will be used for such expressions where expression1 is initialization, expression2 is condition and expression3 is either increment or decrement.
- The general format can be expressed as follows for the sake of clarity:

```
for (initialization ; condition ; increase/decrease)
statements;
```

- Steps of execution of the for loop:
  - ✓ Initializationis executed. (will be executed only once)
  - ✓ Condition is checked, if it is true the loop continues, otherwise the loop finishes and statement is skipped.
  - ✓ Statement is executed.
  - ✓ Finally, whatever is specified in the increase or decrease field is executed and the loop gets back to step 2.

The flowchart of for loop is:



E.g. guess the output of the following code:

```
int main()
{
for(int i=10;i>0;i--)
{
cout<<i<<",";
}
cout<< "FIRE!";
getch();
return 0;
}
```

- Even though it is not recommended, expression1, expression2 and expression3 can be optional or can be ignored. This means that they can take NULL statement.
- But making expression2 null means that the loop will not terminate. In such cases one can include an "if" statement inside the "for" loop which will test a condition and break out from the loop using the break statement.
- While making one or more of the three expressions null, the semi colon CAN NOT be ignored.

E.g.:

for (;n<10;) //if we want neither initialization nor increase/decrease

for (;n<10;n++) //if no initialization is needed.

for (; ;) //is an infinite loop unless and otherwise there is if statement inside the loop.

- It is declared above that expression1 and expression3 can be one or more statements. The composite statements should be separated by a comma. This means, optionally, using the comma operator (,) we can specify more than one instruction in any of the two fields included in a "for" loop.

  E.g. for (n=0, i=100; n! =i; n++, i--)

  {

  //whatever here

  }

- In the above example, n=0 and i=100 will be part of expression1 and will be executed only once before the loop starts. In addition, n++ and i—will be part of expression3 and will be executed after each looping/iteration.

  Eg:1

```
//the following for statement adds the numbers between 0 and n
int Sum=0;
for(int i=0; i<=n;i++)
Sum=Sum+i;
```

Eg:2

```
//the following for statement adds the even numbers between 0 and n
int Sum=0;
for(int i=0; i<=n;)
{
Sum=Sum+i;
i+=2;
}
```

Eg:3

```
//the following for statement adds the even numbers between 0 and n
//where all the three expressions are null.
```

```
int Sum=0;
int i=0;
for ( ; ; )
{
    if (i<=n)
break;
else
{
Sum=Sum+i;
i+=2;
}
}
```

- In the above example, the initialization is at the top before the looping starts, the condition is put in if statement before the instructions are executed and the increment is found immediately after the statements to be executed in the loop.

NB: even though there is the option of making all the three expressions null in a "for" loop, it is not recommended to make the condition to take null statement.

4.1.2. The while statement

- The while statement (also called while loop) provides a way of repeating a statement or a block as long as a condition holds / is true.
- The general form of the while loop is:

```
while(expression)
statements;
```

- First expression (called the loop condition) is evaluated. If the outcome is non-zero then statement (called the loop body) is executed and the whole process is repeated. Otherwise, the loop is terminated.
- The loop iterates while the condition is true. When the condition becomes false, program control passes tothe line of code immediately following the loop.
- Suppose that we wish to calculate the sum of all numbers from 1 to some integer value n. this can be expressed as :

E.g.1:// adds the numbers between 0 and any given number n

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

E.g.2://adds the numbers between 0 and 100

```
number=1;
sum=0;
while(number <= 100)
{
sum += number;
number++;
}
```

E.g.3:

/*the following while loop will request the user to enter his/her age which should be between 0 and 130. If the user enters a value which is not in the range, the while loop test the value and request the age again until the user enters a valid age.*/

```
cout<<"\n enter your age [between 0 and 130]:";
cin>>age;
while(age < 0 || age > 130)
{
cout<<"\n invalid age. Plz try again!";
cin>>age;
}
```

4.1.3. Do…while loop

- The do statement (also called the do loop) is similar to the while statement, except that its body is executed first and then the loop condition is examined.
- Unlike for and while loops, which test the loop condition at the top of the loop, the do-while loop checks its condition at the bottom of the loop. This means that a do-whileloop always executes at least once.
- The general form of the do-while loop is:

```
do {
```

statements;

}

while(condition);

- Firststatement is executed, and then condition is evaluated. If it is true, control passesback to the beginning of the do statement, and the process repeat itself. When the value ofcondition is false, control passes to the next statement.

E.g1.:

```
//our previous example (Eg3) in the while loop might be changed as:
age=-1;
do
{
cout<<"\n enter your valid age [between 0 and 130]:";
cin>>age;
}
while(age < 0 || age > 130);
```

E.g2. what do you think is the outcome of the following code:

```
unsigned long n;
do
{
cout<<"\n enter number (0 to end):";
cin>>n;
cout<<"\n you entered:"<<n;
}
}while(n != 0);
```

4.1.4. Nested Loops

One loop can be nested inside of another. Nested loops are used to solve a wide variety of problems and are an essential part of programming.

The following program uses a nested for loop to produce the product of the set {1, 2, 3} with itself:

```
for (int i = 1; i <= 3; ++i)
{
```

```
for (int j = 1; j <= 3; ++j)

cout << '(' << i << ',' << j << ")\n";
}
```

A few hints for looping algorithms

1. When the loop is to execute a specific number of times, the countercontrolled For loop is the best choice.
2. If the loop is controlled by an event rather than a counter and the bodymay or may not be executed, the While loop is the best choice.
3. If the loop is controlled by an event rather than a counter and the bodymust be executed at least one time, the Do While loop is the best choice.
4. When in doubt, use the While loop.

4.1.5. Pitfalls in writing repetition statements

There are some pitfalls worth mentioning in repletion statements. These pit falls are the most common programming errors committed by programmers.

- **Infinite loop**: no matter what you do with the while loop (and other repetition statements), make sure that the loop will eventually terminates.

  E.g.1:

```
//Do you know why the following is an infinite loop?
    int product = 0;
    while(product < 50)
    product *= 5;
```

E.g.2:

```
//Do you know why the following is an infinite loop?
    int counter = 1;
    while (counter != 10)
    counter += 2;
```

- In the first example, since product is initialized with zero, the expression “product*=5” will always give us zero which will always be less than 50.

- In the second example, the variable counter is initialized to 1 and increment is 2, counter will never be equal to 10 as the counter only assumes odd values.
- In theory, this while loop is an infinite loop, but in practice, this loop eventually terminates because of an overflow error as counter is an integer it will have a maximum limit.

Off-By-One Bugs (OBOB): another thing for which you have to watch out in writing a loop is the so called Off-By-One Bugs or errors.

Suppose we want to execute the loop body 10 times. Does the following code work?

E.g.:1

```
count = 1;
while(count < 10)
{
…
count++;
}
```

No, the loop body is executed nine times. How about the following?

E.g.:2

```
count = 0;
while(count <= 10)
{
…
count++;
}
```

No this time the loop body is executed eleven times. The correct is

E.g.:3

```
count = 0;
while(count < 10)
{
…
count++;
}
```

OR

```
count = 1;
while(count <= 10)
{
…
count++;
```

```
}
```

4.3. Jump Statements

The jumpstatements are break, continue, goto, and return.

4.3.1. continue Statement

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It is an error to use the continue statement outside a loop.
- In a while and do while loops, the next iteration commences from the loop condition.
- In a "for" loop, the next iteration commences from the loop's third expression.
- The general form of continue is:

  continue;

E.g.:

```
for(int n=10;n>0;n--)
{ if(n==5)
continue; //causes a jump to n--
cout<<n<< ",";

}
```

- 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 statement applies to the "for" loop, and not to the "while" loop.

  E.g.:

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

}

4.3.2. The break statement

The break statement has two uses. You can use it to terminate a case in the switchstatement and can also use it to force immediate termination of a loop, bypassing thenormal loop conditional test.The general form of break is:

break;

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 statement.

E.g.:

```
for(n=10;n>0;n--)
{ cout<<n<< ",";
if(n = = 3)
{ cout<< "count down aborted!!";
break;
}
}
```

4.3.3. goto statement

The goto keyword causes program execution to jump to the label specified in the gotostatement. You should use thisfeature with caution since its execution causes an unconditional jump ignoring any type ofnesting limitations.

The destination point is identified by a label, which is then used as an argument for the gotostatement. A label is made of a valid identifier followed by a colon (:).

The general form of goto is

```
goto label;
.
.
.
Label: statement;
```

4.3.4. return statement

The return statement forces a return from a function and can be used to transfer a valueback to the calling routine. A return may or may not have a value associated with it.If return has a value associated with it, that value becomes the return value of thefunction.It has these two forms:

return;

Or return expression;

Worksheet 3

Course Title: Introduction to Computing Course Code: ECEG 2121

1. Prompt the user for two numbers A & B, computes and displays C=A/B. If the number B is zero,displays a "division by Zero" message.
2. Write a program that reads two integers (in any order) and then print either "multiple" or "not "according to weather 2nd number is multiple of the 1st one or not.
3. Write a program that reads two integers (in any order) and then print either "multiple" or "not"according to weather one of the integer is multiple of other.
4. Write a program that takes two numbers from user and determines that first number is a factor ofsecond number.
5. Write a program that mimics a calculator. The program should take as input two integers and the operation to be performed. It should then output the numbers, the operator, and the result. (For division, if the denominator is zero,output an appropriate message.) Use 1 for Addition, 2 for Subtraction, 3 for Multiplication, and 4 for Division.
6. Write a program that calculate the real roots of a quadratic equation and displays the result.
7. Write a program to display the first 100 odd numbers.
8. Write for, do-while, and while statements to compute the following sums and products.
   a. 1+2+3+…+100

b. 1*2*3*…*20

9. A prime number is an integer greater than one and divisible only by itself and one. The first seven prime numbers are 2, 3, 5, 7, 11, 13, and 17. Write a program that displays all the prime numbers between 1 and 100.
10. Write a C++ program that counts the number of digits in an integer number. For example; 23,498 has five digits.
11. Write a C++ application that can compute the letter grade of a student after accepting the student's mid and final mark. The program should only accept mid result [0-40] and final [0-60]. If the data entered violates this rule, the program should display that the user should enter the mark in the specified range. The program is also expected to run until the user refuses to continue.
12. Write a C++ program that accepts a positive number from the user and displays the factorial of that number. Use for loops to find the factorial of the number.
13. write a program in C++ to read a number n and a single digit d. The program then determines whether the digit presents in the number or not. If it presents the program should determine how many times it appear and the position of the digit in the number.

    E.g. N = 12567

    d = 7

    Out put

    7 presents once in 12567 at the position 3 from left to right.

    N = 12246

    D = 2

    Out put

    2 presents twice in12246 at the positions 3 and 4 from left to right.
14. Using the ASCII table numbers, write a program to print the following output, using a nested for loop. (Hint: the outer loop should loop from 1 to 5, and the inner loop's start variable should be 65, the value of ASCII "A").

    A

    AB

    ABC

    ABCD

ABCDE

15. Write a C++ program that will print the following shapes.

A.

```
*
**
***
****
*****
```

B.

```
*****
****
***
**
*
```

C.

```
  *
 ***
*****
*******
*********
```

D.

```
  *
 ***
*****
 ***
  *
```

16. A program to generate Fibonacci series. The first two digits in the series are 0 and 1 and the rest is computed by adding the preceding two digits. (i.e. 0 1 2 3 5 8 13 . . .)
17. Write a program that accepts a number from the key board and reversesthe digits of a given number.

18. write a program that generates the following series of numbers

a.
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

b.
```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

c.
```
9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
7 6 5 4 3 2 1
6 5 4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

d.
```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

19. Use either a switch or an if-else statement and display whether a vowel or a consonant character is entered by the user. The program should work for both lower case and upper case letters.
20. Write a C++ code to display only even numbers found between 0 and 20.
21. Write a program that will accept a mark and assign letter grade according tothe following table.

Grade	Mark
A	> = 90
B	> = 80 but < 90
C	> = 70 but < 80
D	> = 60 but < 70
F	< 60

5. Functions

5.1. What is a function?

- A function is a group of statements that is executed when it is called from some point of the program.
- Functions contain a piece of code to be executed whenever the function is called in the program.
- Functions are used to enclose a section of code that provides a specific functionality to the program. When the function is called from the main program its statements are executed and a value can be returned to the main program upon completion.
- A function is a block of code designed to tackle a specific problem.
- One of the best ways to tackle a problem is to start with the overall goal, then divide this goal into several smaller tasks. You should never lose sight of the overall goal, but think also of how individual pieces can fit together to accomplish such a goal.
- If your program does a lot, break it into several functions. Each function should do only one primary task.

There are three benefits of using functions

1. They make the program code easy to understand/ maintain.
2. Tried or tested functions can be used by other programs.
3. Several programmers can divide the work load in large project by working on different functions.

Functions come in two varieties: user-defined and built-in. built-in functions are part of your compiler package. They are supplied by the manufacturer for your use.

5.2. Declaring function

- A function prototype (also called a function declaration) tells the compiler the name of a function, the type ofdata returned by the function, the number of parameters the function expects to receive, the types of thoseparameters and the order in which the parameters of those types are expected.

- The interface of a function (also called its prototype) specifies how it may be used. It consists of three entities:
  - ✓ The **function return type**. this specifies the type of value the function returns. A function which returns nothing should have a return type void.
  - ✓ The **function name**. this is simply a unique identifier
  - ✓ The **function parameters** (also called its signature). This is a set of zero or more typed identifiers used for passing values to and from the function.

- Each function should be declared early in a program code as a prototype. The syntax for function prototyping is:

  return_type function_name ([type parameterName]...);

5.3. **Defining a function**:

- A function definition consists of two parts: interface (prototype) & body.
- The function body is a pair of braces surrounding the statements to be executed whenever this statement is called.
- The definition consists of a line called the declarator, followed by the function body.
- The return type, the number, order and data type of all the arguments must be the same for both function definition and function prototype. Otherwise it creates a compiler error. But the name of the arguments used in the definition may be different from that used in prototype.
- If function definition is done before the main function, then there is no need to put the prototype, otherwise the prototype should be scripted before the main function starts.
- All statements within the body of the function must be terminated with semicolons, but the function itself is not ended with a semicolon; it ends with aclosing brace.
- Function Definition Syntax is :

  return_type function_name ([type parameterName]...)

  {

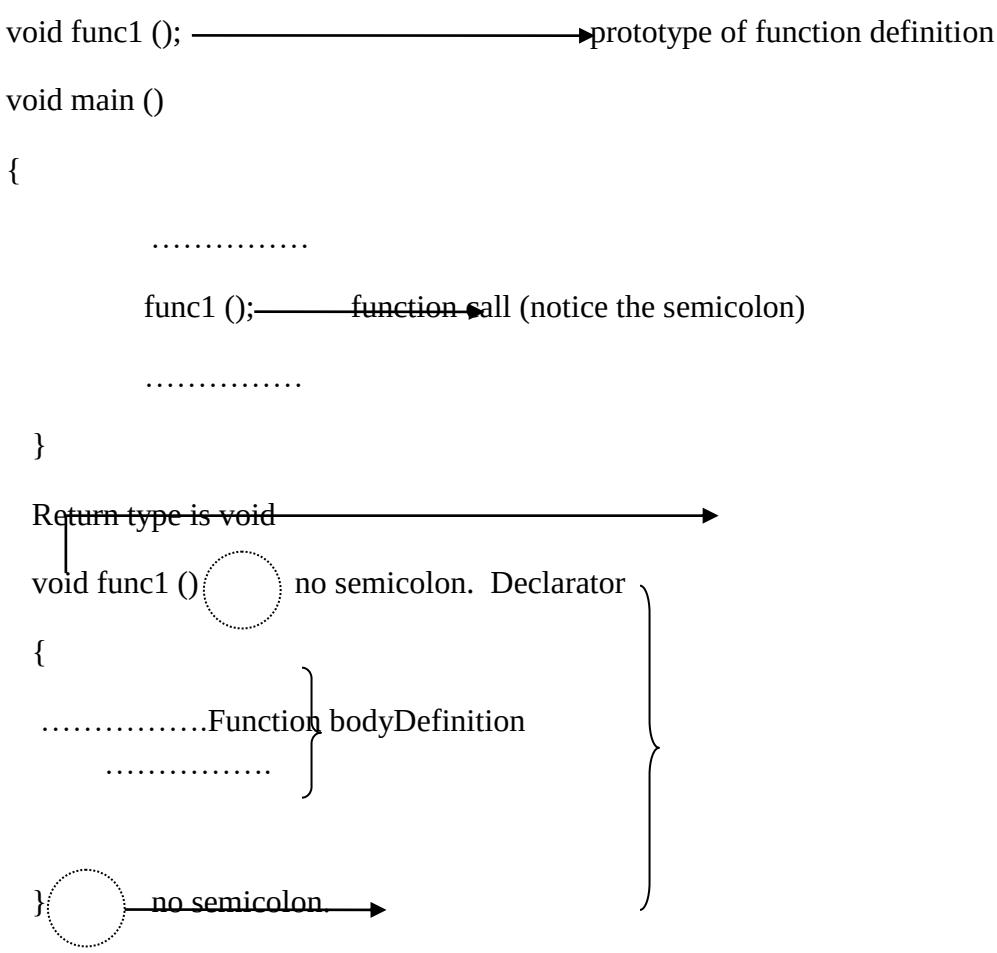
  statements;

  }

5.4. Calling the function

- Calling a function means making the instruction of the function to be executed.
- A function call consists of the function name followed by the call operator brackets '()', inside which zero or more comma-separated arguments appear. The number and type of arguments should match the number of function parameters. Each argument is an expression whose type should match the type of the corresponding parameter in the function interface
- When a function call is executed, the arguments are first evaluated and their resulting values are assigned to the corresponding parameters. The function body is then executed. Finally the return value (if any) is passed to the caller.
- The syntax of the call is very similar to that of the declaration, except that the return type is not used. The call is terminated by a semicolon. Executing the call statement causes the function to execute; that is, control is transferred to the function, the statements in the function definition are executed, and then control returns to the statement following the function call.
- A function call in C++ is like a detour on a highway. Imagine that you are traveling along the "road" of the primary function called main (). When you run into a function-calling statement, you must temporarily leave the main () function and execute the function that was called. After that function finishes (its return statement is reached), program control returns to main (). In other words, when you finish a detour, you return to the "main" route and continue the trip. Control continues as main () calls other functions.

- the syntax of thefunction declaration, function call, and function definition:



Example

```
#include <iostream.h>
float Convert(float);
int main()
{float TempFer;
        float TempCel;
        cout << "Please enter the temperature in Fahrenheit: ";
        cin >> TempFer;
        TempCel = Convert(TempFer);
        cout << "\nHere's the temperature in Celsius: ";
         cout << TempCel << endl;
        return 0;
}
```

```
float Convert(float TempFer)
        {
        float TempCel;
        TempCel = ((TempFer - 32) * 5) / 9;
        return TempCel;
        }
```

Example2

Write a float function add () that sums two numbers using its two formal parameters.

```
#include<iostream.h>
#include<conio.h>
float add(float, float);
int main()
{
clrscr();
float a,b,c;
cout<<"enter the numbers\n";
cin>>a>>b;
c= add(a,b);
cout<<"the sum is:"<<c;
getch ();
return 0;
}
float add(float a,float b)
{
float n;
n=a+b;
return n;
}
```

5.5. Passing Arguments to Functions

An argument is a piece of data (an int value, for example) passed from a program to the function.Arguments allow a function to operate with different values, or even to do differentthings, depending on the requirements of the program calling it.

The arguments of function calling can be using either of the two supported styles in C++: passing by value or passing by reference.Any valid C++ expression can be a function argument, including constants, mathematical and logical expressions, and other functions that return a value.

5.5.1. Passing by value

- A value parameter receives a copy of only the value of the argument passed to it. As a result, if the function makes any changes to the parameters, this will not affect the argument.
- When arguments are passed to a function it is important to recognize that it is only the value that is passed, not the variable itself.
- When passing by value the function is not operating on the original value rather on a copy of it.

Example

```
# include<iostream.h>
void swap (intx, int y);
int main ( )
{
int x = 4 ; int y = 2 ;
cout << " main. Before swap, x: "<<x<<"and y is "<<y<<endl;
swap ( x , y );
 cout << " main. After swap, x: "<<x<<"and y is "<<y<<endl;
}
void swap (int x, int y)
{
 int temp ;
```

```
cout << "Swap. Before swap, x: " << x << " y: " << y <<endl;
 temp = y ;
y = x ;
x = temp;
cout << "Swap. After swap, x: " << x << " y: " << y <<endl;
 }
```

Output: Main. Before swap, x: 4 y: 2

Swap. Before swap, x: 4 y: 2

Swap. After swap, x: 2 y: 4

Main. After swap, x: 4 y: 2

- This program initializes two variables in main () and then passes them to the swap ()function, which appears to swap them. When they are examined again in main (), however, they areunchanged!Passing arguments in this way, where the function creates copies of the arguments passed to it is called passing by value.

5.5.2. Passing by reference

A reference provides an alias—a different name—for a variable. One of the most importantuses for references is in passing arguments to functions.

When arguments are passed byvalue, the called function creates a new variable of the same type as the argument and copiesthe argument's value into it. As we noted, the function cannot access the original variable inthe calling program, only the copy it created. Passing arguments by value is useful when thefunction does not need to modify the original variable in the calling program. In fact, it offersinsurance that the function cannot harm the original variable.

Passing arguments by reference uses a different mechanism. Instead of a value being passed tothe function, a reference to the original variable, in the calling program, is passed. (It's actuallythe memory address of the variable that is passed.)The reference variable contains the symbolampersand (&).

An important advantage of passing by reference is that the function can access the actual variablesin the calling program. Among other benefits, this provides a mechanism for

passingmore than one value from the function back to the calling program.

Examples:

```
# include<iostream.h>
Void swap (int &x, int &y);
int main ( )
{
float x = 22.2 ; int y = 44.4 ;
cout << " main. Before swap, x: "<<x<< "and y is "<<y<<endl;
swap ( x , y );
 cout << " main. After swap, x: "<<x<< "and y is "<<y<<endl;
}
Void swap (int &rx, int&ry)
{
 int temp ;
cout << "Swap. Before swap, rx: " <<rx << " ry: " <<ry <<endl;
 temp = ry ;
ry = rx ;
rx = temp;
cout << "Swap. After swap, rx: " <<rx << " ry: " <<ry <<endl;
 }
```

Output: Main. Before swap, x: 22.2ry: 44.4
Swap. Before swap, rx: 22.2ry: 44.4
Swap. After swap, rx: 44.4ry: 22.2
Main. After swap,x: 44.4 y: 22.2

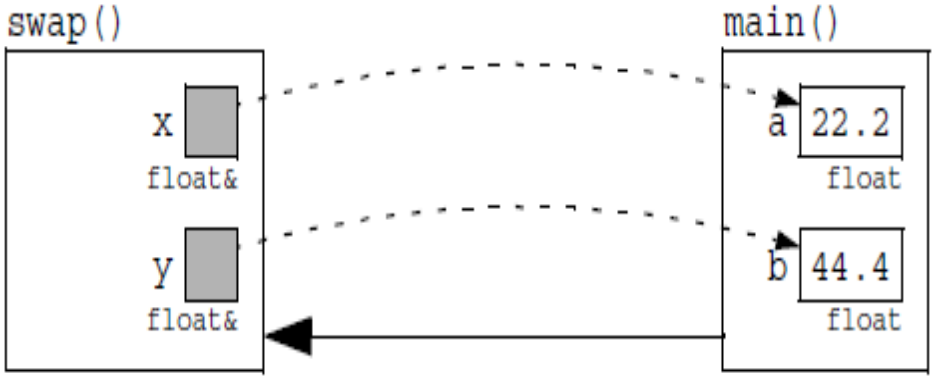
Analysis: This program initializes two variables in main () and the function swap () is called, but note that x and y, not their addresses, are passed. The calling function simply passes the variables.

When swap () is called, program execution jumps to the function definition, where the variables are identified as references. These are aliases for the original values, and can be used as such. The values are swapped. Program execution jumps back to the calling function and the values are

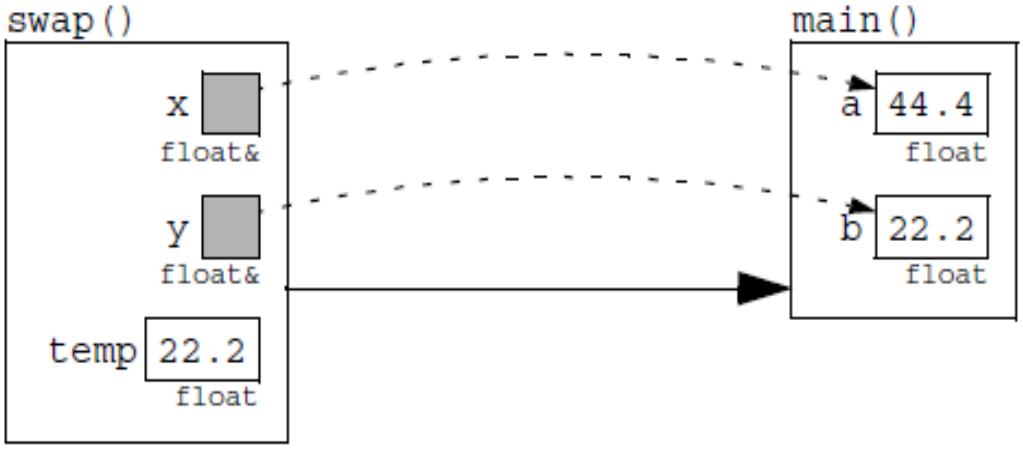
printed in main (). Because the parameters to swap () are declared to be references, the values from main () are passed by reference, and thus are changed in main () as well.

When the call swap(x, y) executes, the function creates its local references rx and ry, so that rx is the function's local name for x, and ry is the function's local name for y. Then the function's three statementsexecute: the local variable temp is declared and initialized with the value of ry (which is y); then ry (whichis y) is assigned the value of rx (which is x); then rx (which is x) is assigned the value of temp. So x endsup with the value 44.4, and y ends up with the value 22.2:

Upon the call swap (x, y):



Upon the return:



Example2:

#.....

```
#.....
void order(int &, int &);
int main()
{
int n1 = 99, n2=11;
int n3 = 22, n4=88;
order (n1,n2);
order (n3,n4);
cout<< "n1="<<n1<<endl;
cout<< "n2="<<n2<<endl;
cout<< "n3="<<n3<<endl;
cout<< "n4="<<n4<<endl;
return 0;
}
void order(int & num1,int & num2)
{
if(num1 > num2)
 {
int temp=num1;
num1 = num2;
num2 = temp;
}
}
```

- In main () there are two pairs of numbers-the first pair is not ordered and the second pair is ordered. The order () function is called once for each pair, and then all the numbers are printed out. The output revels that the first pair has been swapped while the second pair has not. Here it is:

```
N1 = 11
N2 = 99
N3 = 22
```

N4 = 88

- In the order () function the first variable is called num1 and the second is num2. If num1 is greater than num2, the function stores num1 in temp, puts num2 in num1, and finally puts temp back in num2.
- Using reference arguments in this way is a sort of remote control operation. The calling program tells the function what variables in the calling program to operate on, and the function modifies these variables without ever knowing their real names.

5.6.Overloaded Functions

C++ enables you to create more than one function with the same name. This is called function overloading. The arguments to each of these functions must differ in number, data type or both number and data type. The compiler will match a function call to the correct function according to the arguments specified in the call. Overloaded functions enable you to have similar functions that work on different types of data.

Example

```
#include<iostream.h>
#include<iomanip.h>
void FtoC ( int temp) ;
void FtoC (float temp) ;
void FtoC ( double temp) ;
int main ( )
{
     int itemp , level ;
    float ftemp ;
    double dtemp ;
    Cout<<“CONVERTING FARANHEIT TO CELIUS \N “ ;
     cout<< “ select the required level of precision \n “ ;
    cout<< “ integer (1) , float (2) , double (3)\n” ;
     cin>>level ;
    cout<< “ enter faranheit temprature: “ ;
```

```
switch ( level )
      {
      case 1: cin>>itemp ;
      FtoC ( itemp ) ;
     break ;
     case 2:
     cin>>ftemp ;
     FtoC ( ftemp ) ;
     break ;
     case 3: cin>>dtemp ;
     FtoC ( dtemp ) ;
      break ;
     default : cout<< " invalid selection \n " ;
     }
     return 0 ;
      }
void FtoC ( int itemp )
     {
     int temp = ( itemp – 32 ) * 5/9 ;
     cout<< "intiger precision: " ;
      cout<<itemp<< " F is"<<temp<< "C\n" ;
     }
void FtoC (float ftemp )
     {
     float temp = ( ftemp – 32 ) * 5.0/9.0 ;
     cout<< setprecision (8 )<<"float precision:" ;
     cout<<ftemp<< " F is"<<temp<< "C\n" ;
     }
void FtoC (double dtemp )
     {
     double temp = ( dtemp – 32 ) * 5.0/9.0 ;
```

```
cout<< setprecision (12) <<"double precision:"
 cout<<dtemp<< " F is"<<temp<< "C\n" ;
}
```

N.B: if two or more functions differ only in their return types, C++ can't overload them. Two or more functions that differ only in their return types must have different names and can't be overloaded

5.7. Recursive functions

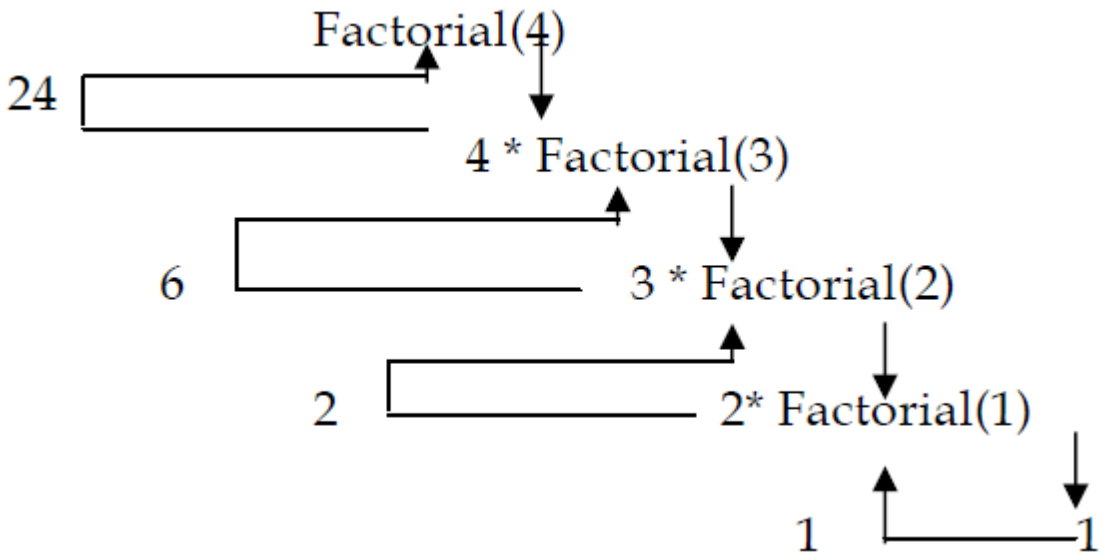
A function whose body contains a function call to itself is known as recursive function. A **recursive function** is a function that calls itself, either directly or indirectly (through another function). It is a function being executed where one of the instructions is to "repeatthe process". It sounds similar to a loop. Recursion is a general programming technique applicable to problems which can be defined in terms of themselves. Take the factorial problem, for instance which is defined as:

- ✓ factorial of 0 is 1
- ✓ factorial of a positive number n is n time the factorial of n-1.

The second line clearly indicates that factorial is defined in terms of itself and hence can be expressed as a recursive function.

```
int Factorial(unsigned int n )
{
return n = = 0 ? 1: n * factorial (n-1);
}
```

For n set to 4, the following figure shows the recursive call:



- The stack frames for these calls appear sequentially on the runtime stack, one after the other.
- A recursive function must have at least one termination condition which can be satisfied. Otherwise, the function will call itself indefinitely until the runtime stack overflows.
- The three necessary components in a recursive method are:
    1. A test to stop or continue the recursion
    2. An end case that terminates the recursion
    3. A recursive call(s) that continues the recursion

Use recursion if:

1. A recursive solution is natural and easy to understand
2. A recursive solution doesn't result in excessive duplicate computation.
3. the equivalent iterative solution is too complex and
4. Of course, when you are asked to use one in the exam!!

5.8. Inline Functions

When a function is declared inline, the function is expanded at the calling block. The inlinespecifier forces the compiler to substitute the body of code where function invocation occurs.As a result of which the control of a program is not shifted to the function being called. Inlinefunctions are used to increase program's speed and performance. They are declared using thefollowing syntax:

Syntax:

inline return_type function_name (list of arguments)
{
function_body
}

- Suppose that a program frequently requires finding the absolute value of an integer quantity. For a value denoted by n, this may be expressed as:

  (n > 0 ? n : -n)

- However, instead of replicating this expression in many places in the program, it is better to define it as a function:

```
int Abs(int n)
{
return n > 0 ? n : -n;
}
```

- The function version has a number of advantages.
  - ✓ It leads to a more readable program.
  - ✓ It is reusable.

The disadvantage of the function version, however is that

- Its frequent use can lead to considerable performance penalty due to overheads associated with calling a function.
- The overhead can be avoided by defining Abs as an inline function.

```
inline int Abs(int n)
{
return n > 0 ? n: -n;
}
```

- The effect of this is that when Abs is called, the compiler, instead of generating code to call Abs, expands the program body and substitutes the body of Abs in place of the call. While essentially the same computation is performed, no function call is involved and hence no stack frame is allocated.

Example1,

```
#include<iostream>
using namespace std;
inline int F( int n )
{ if (n < 0) return 0;
if (n<=1) return 1;
else
return n* F(n–1);}
int main ()
{
cout << “Factorial(6) = “ <<F(6)<<endl;
cout <<“Factorial (5) = “<<F(5)<<endl;
return 0;
 }
```

Example2,

```
#include<iostream>
using namespace std;
inline int Power(int x, int n) // inline function
{ int P = 1; // definition
for (int i =0; i<n ; i++)
P *=x;
return P; }
int main()
{ int A = 2, B = 3, C =5;
cout<< “A square = ”<< Power(A,2); // function call
cout << “\nB to the power 4 = ”<< Power(B,4); // function call
cout<<“\nC to the power 3 = ”<< Power(C,3)<<endl;//function call
return 0;

    }
```

- The "inline" keyword is merely a hint to the compiler or development environment.
- Not every function can be inlined. Some typical reasons why in lining is sometimes not done include:
  - ✓ the function calls itself, that is, is recursive
  - ✓ the function contains loops such as for(;;) or while()
  - ✓ the function size is too large
- Another good reason to inline is that you can sometimes speed up your program by inlining the right function. Instead of calling the function every time it is invoked, the compiler will replace the function call with a copy of the function body. If it's a small function which gets called a lot, this can sometimes speed things up.
- Most of the advantage of inline functions comes from avoiding the overhead of calling an actual function. Such overhead includes saving registers, setting up stack frames, and so on. But with large functions the overhead becomes less important.
- Concerning inline functions, the compiler is free to decide whether a function qualifies to be an inline function. If the inline function is found to have larger chunk (amount) of code, it will not be treated as an inline function, but as like other normal functions.
- Then, Why not inline everything? :Since the compiler will copy the entire function body every time the function is called, if it is a large function (more than three or four lines), inlining can increase the size of your executable program significantly.

5.9. Default Arguments

When declaring variables, C++ also allows us to assign default values for arguments. Theseparameters are used if this particular parameter is left blank when the function is called.Although if a value is passed with the function call then the parameters default value isoverwritten by the value passed in function call.

Example,

```
# include<iostream.h>
# include<conio.h>
int product (int x, int y=3)
{
int multiply;
multiply = x* y;
return multiply;
}
void main()
{
clrscr();
// First call to function 'product'
cout<<product(10)<<endl;
// Second call to function 'product'
cout<<product(20,12);
}
```

In this program, function 'product' has been called twice by function 'main'. In the first callto function 'product', we have only specified one argument '10', so the argument 'x' in thisfunction obtains the value '10' where as argument 'y' takes its default value '3'. As a result ofwhich we obtain the result '30' on the screen. In the second call to the function 'product' twoparameters have been passed to the function as a result of which the parameters value '12'overwrites the default value of arguments 'y' resulting in returning a value '240' from thefunction.

Use of a default argument saves writing this defaultvalue at each invocation.The default argument is placed in the function header by using initializing syntaxwithin the argument list for one or more of its right-most parameters.

- Default argument is a programming convenience which removes the burden of having to specify argument values for all function parameters.

  Eg. You might pass a function an error message that is stored in a character array, and the function displays the error for a certain period of time. The prototype for such a function can be this:

  Void pr_msg(char note[]);

- Therefore, to request that pr_msg() display the line 'Turn printer on', you call it this way:

  Pr_msg("Turn printer on");

- As you write more of the program, you begin to realize that you are displaying one message-for instance, the 'Turn printer on' msg-more often than any other message.
- Instead of calling the function over and over, typing the same message each time, you can set up the prototype for pr_msg() so that it defaults to the 'turn printer on' message in this way:

  Void pr_msg(char note[] = "Turn printr on");

- This makes your programming job easier. Because you would usually want pr_msg() to display 'turn printer on', the default argument list takes care of the message and you don't have to pass the message when you call the function.
- For functions having more than one argument with default, the parameters with possible default value should be declared in theright side of the function definition and prototype.
- E.g.: Let us develop part of the program with default arguments.

```
#include…..
void Add_Display(int x=10, int y=20, int z=30)
{
        cout<< (x+y+z);
}
void Mult_Dispaly (int x, int y=70)
        {
        cout<< (x*y);
        }
void main()
```

```
        {
        int a=40, b=50, c=60;
        Add_Display(a,b,c); //will print 150 (ie 40+50+60)
        Add_Display(a,b); //will print 120 (ie 40+50+30)
        Add_Display(a);//will print 90 (ie 40+20+30)
        Add_Display(); //will print 60 (ie 10+20+30)
        Mult_Display(a,b) //will print 2000 (40*50)
        Mult_Display(a) //will print 2800 (40*70)
        //Mult_Display() //is invalid as there is no default for x
        getch();void main()
        }
        //the following function definition is invalid as z is
        //a parameter without a default and written after y
        //parameters with default should always be at the
        //right side of function declaration
void Mult_Dispaly (int x, int y=70, int z)
        {
        cout<< (x*y*z);
        }
        //thus the following function definition will be correct
void Mult_Dispaly (int x, int z, int y=70)
        {
        cout<< (x*y*z);
        }
```

Worksheet 4

Course Title: Introduction to Computing Course Code: ECEG 2121

1. Write a float function triangle() that computes the area of a triangle using its two formal parameters h and w, where h is the height and w is the length of the bases of the triangle.
2. Write a float function rectangle() that computes and returns the area of a rectangle using its two float formal parameters h and w, where h is the height and w is the width of the rectangle
3. The formula for a line is normally given as y = mx + b. Write a function Line() that expects three float parameters, a slop m, a y-intercept b, and an x-coordinate x. the function computes the y-coordinate associated with the line specified by m and b at x-coordinate.
4. Write a program that accepts a positive integer from the user and displays the factorial of the given number. You should use a recursive function called factorial () to calculate the factorial of the number.
5. Develop a program that uses the function factorial () of exercise 4 to compute an approximation of e (Euler's number). Base your approximation on the following formula for e:

   1 + 1/1! + 1/2! + 1/3! + …
6. Write a function called isPrime() that accepts a number and determine whether the number is prime or not.
7. Write a function called isEven() that uses the remainder operator(%) to determine whether an integer is even or not

6. Arrays and Strings

6.1. What is An Array

- A collection of identical data objects, which are stored in consecutive memory locations under a common heading or a variable name. In other words, an array is a group or a table of values referred to by the same name. The individual values in array are called elements. Array elements are also variables.
- An array is a collection of data storage locations, each of which holds the same type of data. Eachstorage location is called an element of the array.
- A block of memory representing a collection of many simple data variables stored in a separate array element, and the computer stores all the elements of an array consecutively in memory.

6.2.Properties of arrays

- Arrays in C++ are zero-bounded; that is the index of the first element in the array is 0 and the last element is N-1, where N is the size of the array.
- It is illegal to refer to an element outside of the array bounds, and your program will crash or have unexpected results, depending on the compiler.
- Array can only hold values of one type

6.3.Array declaration

Declaring the name and type of an array and setting the number of elements in an array is called dimensioning the array. The array must be declared before one uses in like other variables. In the array declaration one must define:

- The type of the array (i.e. integer, floating point, char etc.)
- Name of the array,
- The total number of memory locations to be allocated or the maximum value of each subscript. i.e. the number of elements in the array.

So the general syntax for the declaration is:

DataType arrayname [array size];

The expression array size, which is the number of elements, must be a constant such as 10 or a symbolic constant declared before the array declaration, or a constant expression such as 10*sizeof (int), for which the values are known at the time compilation takes place.

Note: array size cannot be a variable whose value is set while the program is running.

For example,

To declare an integer with size of 10 having a name of num is:

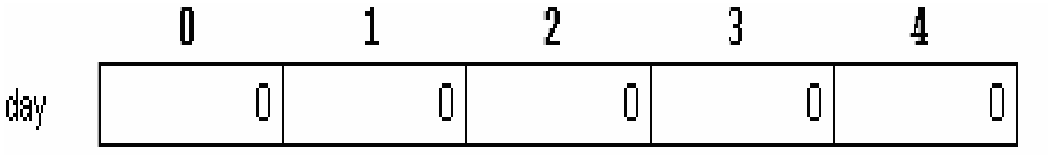
int num [10];

This means: Ten consecutive two byte memory locations will be reserved with the name num and we can store 10 values of type int without having to declare 10 different variables each one with a different identifier. Instead of that, using an array we can store 10 different values of the same type, int for example, with a unique identifier.

6.4.Initializing Arrays

- When declaring an array of local scope (within a function), if we do not specify the array variable will not be initializedto any value by default, so its content is undetermined until we store some values in it.
- If we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare:

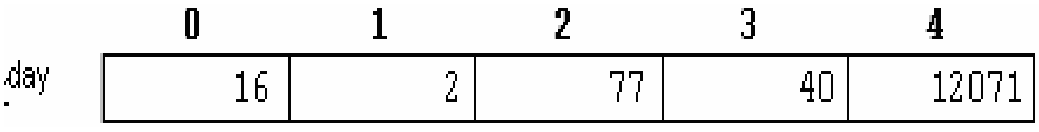
  int day [5];
- every element of day will be set initially to 0:



- But additionally, when we declare an Array, we have the possibility to assign initial values to each one of its elements using curly brackets { }.

  For example:

  int day [5] = { 16, 2, 77, 40, 12071 };
- The above declaration would have created an array like the following one:



- The number of elements in the array that we initialized within curly brackets { } must be equal or less than the length in elements that we declared for the array enclosed within square brackets [ ]. If we have less number of items for the initialization, the rest will be filled with zero.

- For example, in the example of the day array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element. If we ignore the last initial value (12071) in the above initialization, 0 will be taken automatically for the last array element.
- Because this can be considered as useless repetition, C++ allows the possibility of leaving empty the brackets [ ], where the number of items in the initialization bracket will be counted to set the size of the array.

  int day [] = { 1, 2, 7, 4, 12,9 };

  The compiler will count the number of initialization items which is 6 and set the size of the array day to 6 (i.e.: day [6])
- You can use the initialization form only when defining the array. You cannot use it later, and cannot assign one array to another once. i.e.

  int arr [] = {16, 2, 77, 40, 12071};

  int ar [4];

  ar[]={1,2,3,4};//not allowed

  arr=ar;//not allowed

Note: when initializing an array, we can provide fewer values than the array elements.

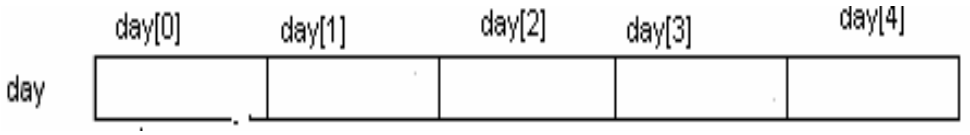
E.g. int a [10] = {10, 2, 3}; in this case the compiler sets the remaining elements to zero.

6.5.Accessing and processing array elements

- The items in an array are called elements.As we noted, all the elements in an array are of the same type; only the values vary.Array elements are counted from zero.
- In any point of the program in which the array is visible we can access individually anyone of its elements for reading or modifying it as if it was a normal variable. To access individual elements, index or subscript is used. The format is the following:

  name [index];
- In c++ the first element has an index of 0 and the last element has an index, which is one less the size of the array (i.e. arraysize-1). Thus, from the above declaration, day[0] is the first element and day[4] is the last element.
- Following the previous examples where day had 5 elements and each element is of type int,

the name, which we can use to refer to each element, is the following one:



- For example, to store the value 75 in the third element of the array variable day a suitable sentence would be:

  day[2] = 75; //as the third element is found at index 2

- And, for example, to pass the value of the third element of the array variable day to the variable a , we could write:

  a = day[2];

- Therefore, for all the effects, the expression day[2] is like any variable of type int with the same properties. Thus an array declaration enables us to create a lot of variables of the same type with a single declaration and we can use an index to identify individual elements.
- Notice that the third element of day is specified day[2], since first is day[0] , second day[1] , and therefore, third is day[2] . By this same reason, its last element is day [4]. Since if we wrote day [5], we would be acceding to the sixth element of day and therefore exceeding the size of the array. This might give you either error or unexpected value depending on the compiler.
- In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can cause certain detectable problems, since they do not cause compilation errors but they can cause unexpected results or serious errors during execution. The reason why this is allowed will be seen ahead when we begin to use pointers.
- At this point it is important to be able to clearly distinguish between thetwo uses the square brackets [ ] have for arrays.
  - ✓ One is to set the size of arrays during declaration
  - ✓ The other is to specify indices for a specific array element when accessing the elements of the array
- We must take care of not confusing these two possible uses of brackets [ ] with arrays:

  Eg1:

  int day[5]; // declaration of a new Array (begins with a type name)

  day[2] = 75; // access to an element of the Array.

Other valid operations with arrays in accessing and assigning:

```
int a=1;
day [0] = a;
day[a] = 5;
b = day [a+2];
day [day[a]] = day [2] + 5;
```

Eg: Arrays example, display the sum of the numbers in the array

```
#include <iostream.h>
int day [ ] = {16, 2, 77, 40, 12071};
int n, result=0;
void main ()
{
for ( n=0 ; n<5 ; n++ )
{
result += day[n];
 }
 cout << result;
getch();
}
```

6.6.Arrays as parameters

- At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass by value a complete block of memory as a parameter, even if it is ordered as an array, to a function, but it is allowed to pass its address, which has almost the same practical effect and is a much faster and more efficient operation.
- In order to admit arrays as parameters the only thing that we must do when declaring the function is to specify in the argument the base type for the array that it contains, an identifier and a pair of void brackets [] . For example, the following function:

  void procedure (int arg[])
- admits a parameter of type "Array of int " called arg . In order to pass to this function an array declared as:

  int myarray [40];
- it would be enough with a call like this:

procedure (myarray);

Here you have a complete example:

```
// arrays as parameters
#include <iostream.h>
#include <conio.h>
void printarray (int arg[], int length)
{
for (int n=0; n<length; n++)
cout << arg[n] << " ";
cout << "\n";
}
void mult(int arg[], int length)
{
 for (int n=0; n<length; n++)
arg[n]=2*arg[n];
}
void main ()
{
 int firstarray[] = {5, 10, 15};
 int secondarray[] = {2, 4, 6, 8, 10};
 printarray (firstarray,3);
printarray (secondarray,5);
mult(firstarray,3);
cout<<"first array after being doubled is\n";
printarray (firstarray,3);
getch()
}
```

- As you can see, the first argument (int arg[]) admits any array of type int , whatever its length is, for that reason we have included a second parameter that informs the function the

length of each array that we pass to it as the first parameter so that the for loop that prints out the array can have the information about the size we are interested about. The function mult doubles the value of each element and the firstarray is passed to it. After that the display function is called. The output is modified showing that arrays are passed by reference.

- To pass an array by value, pass each element to the function

6.7.Multidimensional arrays

The arrays we have used previously have all been *one-dimensional*. This means that they are*linear*; *i.e.*, sequential. But the element type of an array can be almost any type, including anarray type. An array of arrays is called a *multidimensional array*. A one-dimensional array ofone-dimensional arrays is called a two-dimensional array; a one-dimensional array oftwo-dimensional arrays is called a three-dimensional array; *etc.*

2D Array (Matrix)

Two-dimensional arrays consist of values arranged in rows and columns.

Definition:

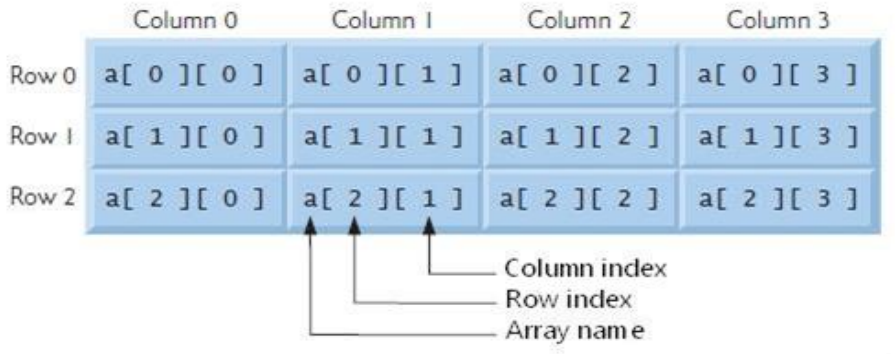
data type arrayName[RowSize][ColumnSize];

For example:

```
int   a[3][4];
float b[10][20];
```

Accessing 2D array elements:

arrayName[RowIndex] [ColumnIndex]



For example:

- a[3][4] = 60;

- cin >> mark[3][1];
- for(int i=0; i<10; i++)
  for(int j=0; j<10; j++)
  cin >> a[i][j]; // input values to the 2D array

- for(int m=0; m<10; m++)

{

for(int n=0; n<10; n++)

cout<< a[m][n]<<"\t"; //print values of 2D array

cout<<endl;

}

2D Array initialization:

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

int a[3][4] = { {1 , 2 , 3 , 4} , {5 , 6 , 7 , 8} , {3 , 4 , 1 , 2} };

Example: Write a C++ program that adds two initialized 3×4 matrices A and B and then stores the result in a matrix C.

```
#include <iostream.h>
int main()
{
 int A[3][4] = { {1, 4, 3, 2}, {5, 6, 7, 8}, {9, 10, 11, 12} };
 int B[3][4] = { {3, 4, 3, 1}, {8, 7, 5, 6}, {12, 9, 11, 8} };
 int C[3][4];
 for (int i=0; i<3; i++)
  {
   for (int j=0; j<4; j++)
    {
     C[i][j] = A[i][j] + B[i][j];
     cout << C[i][j] << "\t";
    }
  cout << endl;
 }
```

```
return 0;
}
```

Multidimensional arrays are nothing else than an abstraction, since we can simply obtain the same results with a simple array by putting a factor between its indices:

int matrix [3][5]; is equivalent to

int matrix [15]; (3 * 5 = 15)

With the only difference that with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bi-dimensional array and the other one uses a simplearray:

multidimensional array	pseudo-multidimensional array

#include <iostream.h>
#define WIDTH 5
#define HEIGHT 3
int matrix [HEIGHT][WIDTH];
int n,m;
int main ()
{
for (n=0;n<HEIGHT;n++)
for (m=0;m<WIDTH;m++)
{
matrix [n][m]=(n+1)*(m+1);
}
return 0;
}
	#include <iostream.h>
	#define WIDTH 5
	#define HEIGHT 3
	int matrix [HEIGHT][WIDTH];
	int n,m;
	int main ()
	{
	for (n=0;n<HEIGHT;n++)
	for (m=0;m<WIDTH;m++)
	{
	matrix [n*WIDITH+m]=(n+1)*(m+1);
	}
	return 0;
	}

None of the programs above produce any output on the screen, but both assign values to the memory block called matrix in the following way:

matrix

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

We have used defined constants (#define) to simplify possible future modifications of the program, for example, in case that we decided to enlarge the array to a height of 4 instead of 3 it

would be enough by changing the line:

#define HEIGHT 3

to

#define HEIGHT 4with no need to make any other modifications to the program.

Examples on arrays

1. Write C++ program that loads an integer array with the numbers 0 through 9 and display the array values.

```
#include <iostream.h>
#include<conio.h>
int main()
{int a[10];
for (int i=0;i<10;i++)
a[i] =i;
cout<<”Array is:”<<endl;
for (int i=0;i<10;i++)
cout<<a[i] <<” ”;
getch ();
return 0;
}
```

2. Write C++ program that read 10 numbers and print their sum, average, maximumand minimum.

```
#include <iostream.h>
#include<conio.h>
int main()
{
const int n=10,min, max, sum;
float average;
int numbers[n];
cout<<"enter the 10 numbers \n";
for(int i=0;i<n;i++)
cin>>numbers[i];
min=max=sum=numbers[0]; /* the minimum equals the maximum equals the summation
equals the first number at the start */
```

```
for(int i=1;i<n;i++) // we consider starting from the next element in the array
{
 if(numbers[i]<min)
      min=numbers[i];
   if(numbers[i]>max)
       max=numbers[i];
sum+=numbers[i];
}
   average=((float) sum)/n;
   cout<<"sum =" <<sum<<“, average= "<< average<<"\n";
    cout<<" maximum =" <<max<<“, minimum= "<<min;
   getch ();
return 0;
}
```

3. Write a C++ program that computes the number of eveninteger numbers in an array entered by the user.

```
#include <iostream.h>
#include<conio.h>
int main()
{
 const int size = 10;
 int a[size] , count = 0;
 cout<<"Enter ten integer numbers: ";
 for(int i=0 ; i<10 ; i++)
   {
    cin >> a[i];
    if(a[i] % 2 == 0)
      count++;
   }
 cout<<"The number of even numbers is " << count;
   getch ();
return 0;
}
```

4. Write a C++ program that inputs an integer array a[10] andarranges it in an ascending order.

```
#include <iostream.h>
#include<conio.h>
int main()
{  const int size = 10;
  int a[size];
  cout<<"Enter ten integer array values: ";
  for (int i=0; i<size; i++)
     cin>>a[i];
  for (i=0; i<size-1; i++)
   for(int j=i+1; j<size; j++)
    if(a[i] > a[j])
     {
       int temp = a[i];
       a[i] = a[j];
       a[j] = temp;
     }
  cout << "Array in ascending order: " <<endl;
  for(i=0; i<size ;i++)
    cout<<a[i]<<" ";
    getch ();
return 0;
}
```

5. Write a program that accepts elements of a maximum 3x3 matrix from the key board and display it.

```
#include<iostream.h>
#include<conio.h>
int main()
{int matrix[3][3];
  int i, j;
cout<<"enter elements of the matrix\n";
 for(i=0;i<3;i++)
```

```
for(j=0;j<3;j++)
cin>>matrix[i][j];
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
    cout<<matrix[i][j]<<" ";
      cout<<endl;
}
getch ();
return 0;
}
```

6. Write a program that performs matrix addition using 3x3 matrixes.

```
#include<iostream.h>
#include<conio.h>
 int main()
{int A[3][3];
int B[3][3];            //declaration
int C[3][3];
  int i, j;
cout<<"enter elements of the first matrix\n";
 for(i=0;i<3;i++)
for(j=0;j<3;j++)
cin>>A[i][j];                        //accepting inputs
cout<<"enter elements of the second matrix\n";
 for(i=0;i<3;i++)
for(j=0;j<3;j++)
cin>>B[i][j];
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
C[i][j]=A[i][j]+B[i][j];   //matrix addition
}
```

```
cout<<" The result of matrix addition is:\n";
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
    cout<<C[i][j]<<” “;  //display addition result
      cout<<endl;
}
getch ();
return 0;
}
```

7. Write a C++ program that exchanges row3 with row1 in a 4× 4integer matrix input by the user.

```
#include <iostream.h>
int main()
{
 int a[4][4];
 cout<<"Enter 4x4 integer matrix: ";
 for (int i=0; i<4; i++)
  for (int j=0; j<4; j++)
    cin>>a[i][j];
 for (i=0; i<4; i++)
  {
   int temp = a[0][i];
   a[0][i] = a[2][i];
   a[2][i] = temp;
  }
 cout<<"Matrix after exchanging row3 with row1:"<<endl;
 for (i=0; i<4; i++)
  {
   for (j=0; j<4; j++)
     cout<<a[i][j]<<" ";
   cout<<endl;
  }
return 0;
}
```

8. Write a program that displays an identity matrix of 3x3.

```
#include<iostream.h>
```

```
#include<conio.h>
int matrix[3][3];
 int main()
{
  int i, j;
   for(i=0;i<3;i++)
 {
for(j=0;j<3;j++)
 if(i==j)
matrix[i][j]=1;
}
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
     cout<<matrix[i][j]<<" ";
     cout<<endl;
}
getch ();
return 0;
}
```

9. Write a C++ program that multiplies 3×4 matrix by 4×3 matrix both are entered by the user. Then the program should store the result in athird matrix.

```
#include <iostream.h>
void main()
{
  const int row_a=3 , col_a=4 ,  row_b=4, col_b=3;
  int a[row_a][col_a];
```

```
int b[row_b][col_b];
int c[row_a][col_b];

cout<<"Enter "<<row_a<<"x"<<col_a<<" integer matrix: " << endl;
for (int i=0; i<row_a; i++)
 for (int j=0; j<col_a; j++)
   cin>>a[i][j];
cout<<"Enter "<<row_b<<"x"<<col_b<<" integer matrix: " << endl;
for (i=0; i<row_b; i++)
 for (j=0; j<col_b; j++)
   cin>>b[i][j];
for(i=0; i<row_a; i++)
 for(j=0; j<col_b ; j++)
  {
    c[i][j] = 0;
    for(int k=0; k<col_a ; k++)
 c[i][j] += a[i][k] * b[k][j];
    }
cout<<"Resulted Matrix is " << endl;
for (i=0; i<row_a; i++)
 {
  for (j=0; j<col_b; j++)
    cout<<c[i][j]<<" ";
  cout<<endl;
 }
}
```

6.8.Strings of Characters

6.8.1. What are Strings?

In all programs and concepts we have seen so far, we have used only numerical variables, used to express numbers exclusively. But in addition to numerical variables there also exist strings of

characters that allow us to represent successive characters, like words, sentences, names, texts, etc. Until now we have only used them as constants, but we have never considered variables able to contain them.

A string is a series of characters stored in consecutive bytes of memory. This implies that, a string can be stored in an array of char, with each character kept in its own array elements. Strings provide a convenient way to store text information such as messages to the user.

A string can also be defined as a character array that is terminated by a null. A null character is specified using '\0', and is zero.

Because of the null terminator, it is necessary to declare a character array to be one character longer than the largest string that it will hold.

For example, if you want to declare an array name that could hold a 20-character string, you would write:

char name [20];

You may imagine it thus:

name



This maximum size of 20 characters is not required to be always fully used. For example, name could store at some moment in a program either the string of characters "Hello" or the string "studying C++". Therefore, since the array of characters can store shorter strings than its total length, there has been reached a convention to end the valid content of a string with a null character, whose constant can be written as '\0'.

We could represent name (an array of 20 elements of type char) storing the strings of characters "Hello" and "Studying C++" in the following way:

H	e	l	l	o	\0														

S	t	u	d	y	i	n	g		C	+	+	\0							

Notice how after the valid content it is included a null character ('\0') in order to indicate the end of string. The empty cells (elements) represent indeterminate values.

6.8.2. Initialization of Strings

Since strings of characters are ordinary arrays they fulfill same rules as any array. For example, if we want to initialize a string of characters with predetermined values we can do it in a similar way to any other array:

```
char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

In this case we would have declared a string of characters (array) of 6 elements of type char initialized with the characters that compose Hello plus a null character '\0'.

String of characters can be initialized using constant strings.To initialize a string, follow the char array declaration with the equal character and a string of characters enclosed in the double quotes (“ ”). For example,

```
char promptMsg[] = “Enter a positive number: ”;
char errMsg[] = “Value must be positive. “;
```

promptMsg is stored in 25 locations, one for each character.

Unlike single quotes (‘) which allow to specify single character constants, double quotes (“) are constants that specify a succession of characters. These strings enclosed between double quotes have always a null character (‘\0’) automatically appended at the end.Therefore we could initialize the string **mystring** with values by any of these two ways:

```
char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char mystring [] = "Hello";
```

In both cases the Array or string of characters mystring is declared with a size of 6 characters (elements of type char): the 5 characters that compose Hello plus a final null character ('\0') which specifies the end of the string and that, in the second case, when using double quotes (") it is automatically appended.

Before going further, you should note that the assignation of multiple constants like double-quoted constants (“ ”) to arrays are only valid when initializing the array, that is, at the moment when declared.

The following expressions within a code are not valid for arrays

mystring="Hello";

mystring[] = "Hello";

neither would be: mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };

So remember: We can "assign" a multiple constant to an array only at the moment of initializing it. The reason will be more comprehensible when you know a bit more about pointers, since then it will be clarified that an array is simply a constant pointer pointing to an allocated block of memory. And because of this constant feature, the array itself cannot be assigned any value, but we can assign values to each of the elements of the array.

At the moment of initializing an Array it is a special case, since it is not an assignation, although the same equal sign (=) is used. Anyway, have always present the rule previously underlined.

6.8.3. Reading a String from the Keyboard

The object cin of <istream> along with extraction operator >> may be used for input of stringsif there is no white space between the characters because cin takes the white space as the end ofstring.

In the following program only cin is used. The input string consists of two words Abebe Alemu, but cin extracts only the first word Abebeand stops reading on encountering white space.

```
#include <iostream.h>
#include<conio.h>
int main()
{
char S[15] ;
cout<<“Write a short name : ” ;
cin >> S ;
cout <<“You have written : ”<< S<<endl;
return 0 ;
}
```

The expected output is below.

Write a short name : Abebe Alemu

You have written: Abebe

From the output it is clear that cin stopped reading after encountering the white space,

i.e.after Abebe. Now if the same string is extracted by cin character by character as an array, thecomplete name is read but the white space is not read as space. This is illustrated in the followingprogram. The two words Abebe and Alemu are merged together.

```
#include <iostream.h>
int main()
{char S[15];
cout<<“Write a short name : ”;
for (int i =0; i<=15;i++) // Use of for loop for reading
{
cin>>S[i] ; // It does not read the white space
cout<<S[i] ;
}
return 0 ;
}
```

The expected output is given below.

Write a short name: Abebe Alemu

AbebeAlemu

From the 2nd line of the output it is clear that the white space has not been read and includedin the string. So in the output the two parts of name are merged together.

The above mentioned problem is not there if we use function **cin.getline().**The function cin.getline() reads all the characters in a line until the limiting numberof characters is reached or it encounters a delimiting character. It takes maximum of threearguments. The syntax for function call is as below.

cin.getline(identifier, int length, ch);

The “**identifier**” is the name of string we want to put in. The integer**length** isused to limit the number of characters to be read or the maximum length of the size of the array. The “**ch**” is the character used to determine the end of the user input, which by default - if we do not include that parameter - will be the newline character (‘\n').

The function stops reading when

i. n–1characters have been read or

ii. a new line or end of file is encountered or

iii. the delimiting character ch is encountered.

Example1:

```
#include <iostream>
using namespace std;
int main()
{
char B[50];
cout <<"Enter a long name: ";
cin.getline(B,50) ; // It specifies to read up to read 50–1
// characters from string. The last character is Null (\0).
cout<<"You have written = "<<B <<"\n" ;
return 0 ;
}
```

The expected output is given below.

Enter a long name: I like studying C++.

You have written = I like studying C++.

Example2:

```
#include <iostream>
using namespace std;
int main()
{
char B[20];
cout <<"Enter a name: ";
cin.getline(B,5) ;              // read first 5 characters of B
cout<<"You have written = "<<B <<"\n" ;
return 0 ;
}
```

The name entered was Abebe Alemu but the function has read only first five characters, i.e. Abebeas given in the output given below, because, it was specified so in getline () function.

Enter a name: Abebe Alemu

You have written = Abeb

In the following program a delimiting character has been put in the cin.getline() function tostop reading on encountering the character.

Example3:

```
#include <iostream>
using namespace std;
int main()
{
char B[20];
cout <<"Enter a name: ";
char ch = 'm';
cin.getline(B,20, ch) ; //read up to character 'S'
cout<<"You have written = "<<B <<"\n" ;
return 0 ;
}
```

The output is given below. It stopped reading on occurrence of letter'm'. So only Abebe Ale isread as given in the output below.

```
Enter a name: Abebe Alemu
You have written = Abebe Ale
```

6.8.4. Functions to manipulate strings

The standard C++ library contains several functions to perform some manipulation operations with C-like strings. The prototypes for these functions are contained in the file string.h. Here you have a brief with the most usual:

a. String length:

- Returns the length of a string, not including the null character (\0).

strlen (const char* string);

Example

The following program takes your name and tells the length of your name.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
void main()
{
  char name[25];
  cout<<"Enter your name: ";
  cin>>name;
  cout<<"your name is "<<strlen(name)<<" characters long";
  getch();
 }
```

Example2:

The following C++ program reads your name from keyboard and display it in reverse order.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<iomanip.h>
void main ( )
{ char name[20];
  cout<<"Enter your name \t";
  cin>>setw(20)>>name;
  for(int i=(strlen(name)-1);i>=0;i--)
                    cout<<name[i];
  getch();
}
```

b. **String Copy**:

- The string copy can have two forms, where the first one is to copying the whole content of the source to the destination and the other will copy only part of the source to the destination.
- Overwrites the content of the dest string by the src string. Returns dest.
  - Copy the whole content of the source

strcpy (**char*** dest , **const char*** src);

- Appending part of the source

strncpy (**char*** dest , **const char*** src, int size);

Where size is the number characters to be copied

Eexample1:

```
#include <iostream.h>
#include <string.h>
int main ()
{
char src[10],dest[10];
cout<<"\n enter your name :";
cin>>src;
strcpy (dest,src);
cout<<"\n your name after copying :"<<dest;
return 0 ;
 }
```

Eexample2:

```
#include <iostream.h>
#include <string.h>
int main ()
{
char src[10],dest[10];
cout<<"\n enter your name :";
cin>>src;
strncpy (dest,  src, 3);  // look strncpy not strcpy
cout<<"\n your name after copying :"<<dest;
return 0 ;
 }
```

c. **String Compar**e:

- Compares the two string string1 and string2.

- The string compare can have two forms, where the first one is to compare the whole content of the two strings and the other will compare only part of the two strings.
  - Compare the whole content of the source

strcmp (const char* string1 , const char* string2);

  - Appending part of the source

strncmp (const char* string1 , const char* string2, int size);

Where size is the number characters to be compared

- Both string compare functions returns three different values:
  - Returns 0 if the strings are equal
  - Returns negative value if the first is less than the second string
  - Returns positive value if the first is greater than the second string

Example1:

```
#include<iostream>
# include <string.h>
using namespace std;
int main()
{
char Str1[] = “XYZ” ;
char Str2[] = “ABC”;
char Str3[] = “XYZ”;
cout << strcmp(Str1, Str2)<<endl;
cout << strcmp(Str2, Str1)<<endl;
cout << strcmp(Str3, Str1)<<endl;
cout << strcmp(Str2, Str3)<<endl;
return 0 ;
}
```

The expected output is given below. Str1 is lexicographically greater than Str2 and Str1 islexicographically equal to that Str3. So the following output is obvious.

```
1
-1
0
```

-1

d. **String Concatenation**:

- Appends src string at the end of dest string. Returns dest.
- The string concatenation can have two forms, where the first one is to append the whole content of the source to the destination the other will append only part of the source to the destination.
  - Appending the whole content of the source

    strcat (**char*** dest , **const char*** src);
  - Appending part of the source

    strncat (**char*** dest , **const char*** src, **int** size);

    Where size is the number characters to be appended

Example2:

```
#include<iostream>
# include <string.h>
using namespace std;
int  main()
{
char Str1 [10] = " Abebe" ;
char Str2 [14] = " Alemu Kebede";
char Str3 [10] = "Belay";
char Str4 [10] = "Dereje";
strncat(Str1, Str2, 5);
cout<<Str1<<endl;
cout<<Str2<<endl;
strcat( Str3, Str4);
cout << Str3<<endl;
cout<<Str4 <<endl;
return 0;
}
```

The expected output is given below.

```
Abebe Alemu
```

Alemu Kebede

Belay Dereje

Dereje

The first line of output is of Str1 after the operation of strncat(Str1,Str2, 5);.The first 5 characters of S2 (Alemu) are appended to end of Str1. The string Str2 is not affectedby this operation. The third line of output is the result of strcat(Str3,Str4);. Thecomplete string Str4 is appended to the end of string Str3. The string Str4 remains intact.

6.8.5. Converting strings to other types

Due to that a string may contain representations of other data types like numbers it might be useful to translate that content to a variable of a numeric type. For example, a string may contain "**1977**", but this is a sequence of 5 chars not so easily convertible to a single integer data type. **The cstdlib** (**stdlib.h**) library provides three useful functions for this purpose:

- **atoi**: converts string to **int** type.
- **atol**: converts string to **long** type.
- **atof**: converts string to **float** type.

All of these functions admit one parameter and return a value of the requested type (int, long or float). These functions combined with getline method of **cin** are a more reliable way to get the user input when requesting a number than the classic **cin**>> method:

Example1:

```
#include <iostream>
# include <cstdlib>
using namespace std;
void main()
{
char ch1 []= “6444”;
char ch2[] = “21.66”;
char ch3[] = “Absent45”;
char ch4[] = “45Absent”;
char chL1[] = “Delhi45676548”;
char chL2[] = “45676548Delhi”;
```

```
cout<<"Number represented by ch1 divided by 2 = " <<atoi(ch1)/2 <<endl;
cout<<"Number ch2 /3= " << atof (ch2)/3<<endl;
cout<<"Number ch3 = " <<atoi (ch3)<<endl;
cout<<"Number represented by ch4 = " <<atoi (ch4)<<endl;
cout<<"Number represented by chL1 = " <<atol (chL1)<<endl;
cout<<"Number represented by chL2 = " <<atol (chL2)<<endl;
}
```

The expected output is as below.

Number represented by ch1 divided by 2 = 3222
Number ch2 /3= 7.22
Number ch3 = 0
Number represented by ch4 = 45
Number represented by chL1 = 0
Number represented by chL2 = 45676548

Six strings ch1, ch2, ch3, ch4, chL1 and chL2 are declared and initialized. The functionsatoi(), atof () and atol() are called to convert the strings into corresponding digits. The stringswhich begin with non-digit characters are not converted and 0 is returned while those whichbegin with digit characters are converted. To show that the converted digits represent number,the numbers obtained in first two cases have been divided by 2 and 3 respectively. The trailingends of strings consisting of non-digital characters are ignored.

Example2:

```
// cin and ato* functions
        #include <iostream.h>
        #include <stdlib.h>
        #include<conio.h>
        int main()
        {
         clrscr();
        char mybuffer[100];
        float price;
        int quantity;
```

```
cout << "Enter price: ";
cin.getline (mybuffer, 100);
price = atof (mybuffer);
cout << "Enter quantity: ";
cin.getline (mybuffer, 100);
quantity = atoi (mybuffer);
cout<<"\nafter conversion :\n";
cout<<"\nprice is : "<<price;
cout<<"\nquantity is : "<<quantity;
cout << "\nTotal price: " << price*quantity;
getch();
return 0;
}
```

Worksheet 5

Course Title: Introduction to Computing Course Code: ECEG 2121

1. Write a program to store the ages of six of your friends in a single array. Store each of the six ages using the assignment operator. print the ages on the screen
2. Write a C++ program that accepts 10 integers from the user and finally displays the smallest value and the largest value.
3. Write a program that accepts ten different integers from the user and display these numbers after sorting them in increasing order.
4. Write a program to store six of your friend's ages in a single array. Assign the ages in a

random order. print the ages, from low to high, on-screen

5. Write a C++ program that calculates the letter grades of 20 students. The program should accept the mid result and the final result from the students. Use the appropriate validity control mechanism to prevent wrong inputs.
6. Write a C++ program that has two functions toBinary and toDecimal. The program should display a menu prompting the user to enter his choice. If the user selects toBinary, then the function should accept a number in base ten and displays the equivalent binary representation. The reverse should be done if the user selects toDecimal.
7. Write a C++ program that accepts a word from the user and then displays the word after reversing it.
8. Develop a C++ program that accepts the name of a person and then counts how many vowels the person's name have.
9. Write a program in C++ which read a three digit number and generate all the possible permutation of numbers using the above digits. For example n = 123 then the permutations are:

    9.6.1. 123, 213, 312, 132, 231, 321

10. Write a program which read a set of lines until you enter #.
11. Write a program which read two matrixes and then print a matrix which is addition of these two matrixes.
12. Write a program which reads two matrix and multiply them if possible
13. Write a program which reads a 3 x 2 matrix and then calculates the sum of each row and store that in a one dimension array.

7. Pointers

7.1. Introduction

- We have already seen how variables are memory cells that we can access by an identifier. But these variables are stored in concrete places of the computer memory. For our programs, the computer memory is only a succession of 1 byte cells (the minimum size for a datum), each one with a unique address.
- A pointer is a variable which stores the address of another variable. The only difference between pointer variable and regular variable is the data they hold.

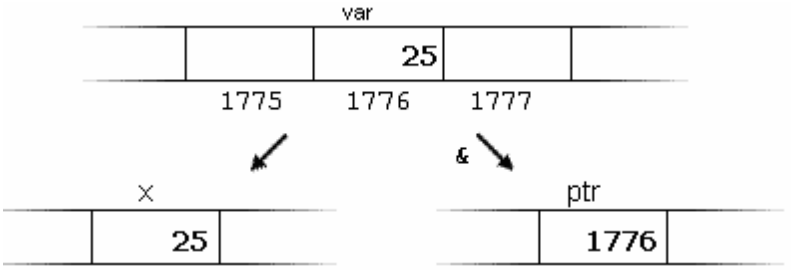
- There are two pointer operators in C++:
  - ✓ & the address of operator
  - ✓ * the dereference operator
- Whenever you see the & used with pointers, think of the words “address of.” The & operator always produces the memory address of whatever it precedes. The * operator, when used with pointers, either declares a pointer or dereferences the pointer’s value. The dereference operator can be literally translated to "**value pointed by**”.
- **A pointer** is simply the address of an object in memory. Generally, objects can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to an object via a pointer to it is called **dereferencing** the pointer. Pointer variables are defined to point to objects of a specific type so that when the pointer is dereferenced, a typed object is obtained.
- At the moment in which we declare a variable this one must be stored in a concrete location in this succession of cells (the memory). We generally do not decide where the variable is to be placed - fortunately that is something automatically done by the compiler and the operating system on runtime, but once the operating system has assigned an address there are some cases in which we may be interested in knowing where the variable is stored.
- This can be done by preceding the variable identifier by an ampersand sign (&), which literally means, "**address of**”. For example:

  ptr= &var;
- This would assign to variable ptr the address of variable var , since when preceding the name of the variable var with the ampersand (&) character we are no longer talking about the content of the variable, but about its address in memory.
- We are going to suppose that var has been placed in the memory address 1776 and that we write the following:

  var=25;

  x=var;

  ptr = &var;
- The result will be the one shown in the following diagram:



- We have assigned to x the content of variable var as we have done in many other occasions in previous sections, but to ptr we have assigned the address in memory where the operating system stores the value of var , that we have imagined that it was 1776 (it can be any address). The reason is that in the allocation of ptr we have preceded var with an ampersand (&) character.
- The variable that stores the address of another variable (like ptr in the previous example) is what we call a pointer.

7.2.Declaring Pointers

- Is reserving a memory location for a pointer variable in the heap.

  Syntax:

  type * pointer_name ;

- to declare a pointer variable called p_age, do the following:

  int * p_age;

- Whenever the dereference operator, *, appears in a variable declaration, the variable being declared is always a pointer variable.

7.3.Assigning values to pointers

- p_age is an integer pointer. The type of a pointer is very important. p_age can point only to integer values, never to floating-point or other types.
- To assign p_age the address of a variable, do the following:

  int age = 26;

  int * p_age;

p_age = &age;

OR

int age = 26;

int * p_age = & age;

- Both ways are possible.
- If you wanted to print the value of age, do the following:

  cout<<age;//prints the value of age

  Or by using pointers you can do it as follows

  cout<<*p_age;//dereferences p_age;
- The dereference operator produces a value that tells the pointer where to point. Without the *, (i.e cout<<p_age), a cout statement would print an address (the address of age). With the *, the cout prints the value at that address.
- You can assign a different value to age with the following statement:

  age = 13; //assigns a new value to variable age

  *p_age = 13; //assigns 13 as a value to the memory p_age points at.
- N.B: the * appears before a pointer variable in only two places: when you declare a pointer variable and when you dereference a pointer variable (to find the data it points to).
  - The following program is one you should study closely. It shows more about pointers and the pointer operators, & and *, than several pages of text could do.

    #...

    #...

    void main()

    {

    int num = 123; // a regular integer variable

    int *p_num; //declares an integer pointer

    cout<< "num is "<<num<<endl;

    cout<< "the address of num is "<<&num<<endl;

    p_num = #// puts address of num in p_num;

    cout<< "*p_num is "<<*p_num<<endl; //prints value of num

    cout<< "p_num is "<<p_num<<endl; //prints value of P_num

```
getch();
}
```

7.4 Pointer to void

- note that we can't assign the address of a float type variable to an integer pointer variable and similarly the address of an integer variable cannot be stored in a float or character pointer.

```
float y;
int x;
int *ip;
float *fp;
ip = &y; //illegal statement
fp = &x; //illegal statement
```

- That means, if a variable type and pointer to type is same, then only we can assign the address of variable to pointer variable. And if both are different type then we can't assign the address of variable to pointer variable but this is also possible in C++ by declaring pointer variable as a void as follows:

```
void *p;
```

- Let us see an example:

```
void *p;
int x;
float y;
p = &x; //valid assignment
p = &y; //valid assignment
```

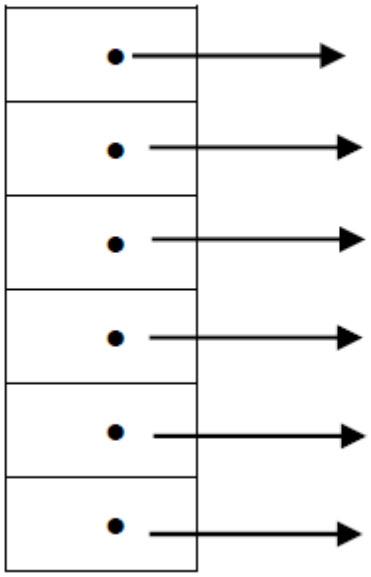
- The difficulty on void pointers is that, void pointers cannot be de referenced. They are aimed only to store address and the dereference operator is not allowed for void pointers.

7.5 Arrays of Pointers

- If you have to reserve many pointers for many different values, you might want to declare an array of pointers.
- The following reserves an array of 6 integer pointer variables:

```
int *iptr[6]; //reserves an array of 6 integer pointers
```

- The above statement will create the following structure in RAM



iptr[4] = &age;// makes iptr[4] point to address of age.

7.6 Pointer and arrays

When we declare an array, its identifier (name) is a **const pointer** to the first element of array. Thepointer to array also carries the address of first element of array. Thus the pointer to an array hassame value as the name of the array. The difference is that the name of an array is a **const pointer**,i.e. it cannot be incremented or changed while the pointer to an array which carries the address offirst element of the array may be incremented. If pointer to an array is incremented by 1 it wouldpoint to the next element of the array. Thus the value of the pointer would increase by the numberof bytes allocated to one array element.

The declaration of a pointer to an array is illustrated below.

```
int A[] = {12, 25, 36 ,50};      // A is the name of array
int *ptrA ;
ptrA = A ;    // Notice there is no & operator before A
```

The above definition may also be written as below.

```
int *ptrA = A;
```

and also as,

```
int* ptrA = &A[0];
```

It is so because the address of A[0] is same as value of A. The elements of array are A[0], A[1], A[2], and A[3] have respective values as 12, 25, 36 and 50 which may also be obtained from pointers as illustrated below.

A[0] = *ptrA;

A[1] = *(ptrA+1);

A[2] = *(ptrA+2) ;

A[3] = *(ptrA+3);

The above discussion shows that array subscript and pointer offset are equal. If we call array A without a subscript it will give the address of first element of array. A +1 gives the address ofsecond element, A+2 gives address of third elements and so on. Therefore, we can also get thevalues stored at these addresses by using dereference operator (*). Thus for the above declaredarray we may get the values of array elements as given below.

*A = 12

*(A+1) = 25

*(A+2) = 36

*(A+3) = 50

Note that expression *A++ is not legal because A is a **constant pointer** to array and a constantcannot be incremented. Also note that array subscripts are equal to pointer offset in traversingof array elements.

- Consider the following character array:

  char name[] = “C++ Programming”;

  What output do the following cout statements produce?

  cout<<name[0]; // ____C__

  cout<<*name; // _____C__

  cout<<*(name+3); //_____

  cout<<*(name+0); //____C____

  Program 1: illustrates declaration of pointer to an array

  #include <iostream>

  using namespace std;

  int main()

  { int A [] = {80,40,20,10}; // A is an array of integers

```
int *ptrA = A; //ptrA is pointer to A. Note that & is not used.
cout <<"*ptrA = "<<*ptrA <<", \tptrA = "<<ptrA<<endl;
cout <<"*(ptrA+1) = " << *(ptrA+1)<< " ,\tptrA+1 = " <<ptrA+1<<endl;
cout<<"*(ptrA+2) = "<<*(ptrA+2)<< " , \tptrA + 2 = " <<ptrA+2 <<endl;
cout<<"*(ptrA+3) = "<<*(ptrA+3)<< " , \tptrA+3= " << ptrA+3<<"\n\n";
cout << "A = " << A<<endl;
cout<< ptrA[0]<<"\t"<<ptrA[1]<<"\t"<<ptrA[2]<<"\t"<<ptrA[3]<<endl;
cout << *(A+0) <<"\t"<<*(A+1) << "\t"<<*(A+2) << "\t" << *(A+3)<<endl;
// these will display the values of array elements.
return 0;
}
```

Program2: illustrates traversing an array by pointers.

```
#include <iostream>
using namespace std;
int main()
{ int kim [ ] = {50, 86, 90, 12} ;
int* pkim = kim; // defining the pointer to array kim
for ( int i=0 ; i<4;i++) // for loop for output of array
cout<<"kim["<<i<<"] = " << *(pkim +i) <<" , ";
//*(pkim +i)is the value of kim[i]element of array
return 0;
}
```

The expected output of the program is given below.

kim[0] = 50 , kim[1] = 86 , kim[2] = 90 , kim[3] = 12 ,

7.7.Pointer Advantage

- You can't change the value of an array name, because you can't change constants. This explains why you can't assign an array a new value during a program's execution: eg: if Cname is array of characters then:

```
Cname = "Football"; //invalid array assignment;
```

- Unlike arrays, you can change a pointer variable. By changing pointers, you can make them point to different values in memory. Have a look at the following code:

```
#...
#...
void main()
{
clrscr();
float v1 = 679.54;
float v2 = 900.18;
float * p_v;
p_v = &v1;
cout<< "\nthe first value is "<<*p_v;
p_v = &v2;
cout<< "\nthe second value is "<<*p_v;
getch();
}
```

- You can use pointer notation and reference pointers as arrays with array notation. Study the following program carefully. It shows the inner workings of arrays and pointer notation.

```
void main()
{
clrscr();
int ctr;
int iara[5] = {10,20,30,40,50};
int *iptr;
iptr = iara; //makes iprt point to array's first element. Or iprt = &iara[0]
cout<< "using array subscripts:\n"
cout<< "iara\tiptr\n";
for(ctr=0;ctr<5;ctr++)
{
cout<<iara[ctr]<< "\t"<< iptr[ctr]<< "\n";
}
```

```
cout<< "\nUsing pointer notation\n";
for(ctr=0;ctr<5;ctr++)
{
cout<< *(iara+ctr) << "\t" << *(iptr+ctr)<< "\n";
}
getch();
}
```

- Suppose that you want to store a person's name and print it. Rather than using arrays, you can use a character pointer. The following program does just that.

```
void main()
{
clrscr();
char *c = "Meseret Belete";
cout<< "your name is : "<<c;
}
```

- Suppose that you must change a string pointed to by a character pointer, if the person's name in the above code is changed to Meseret Alemu: look at the following code:

```
void main()
{
char *c = "Meseret Belete";
cout<< "youe name is : "<<c;
c = "Meseret Alemu";
cout<< "\nnew person name is : "<<c;
getch();
}
```

If c were a character array, you could never assign it directly because an array name can't be changed.

7.8. Pointer Arithmetic

- To conduct arithmetical operations on pointers is a little different than to conduct them on other integer data types. To begin, only addition and subtraction operations are allowed to be

conducted, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point to.

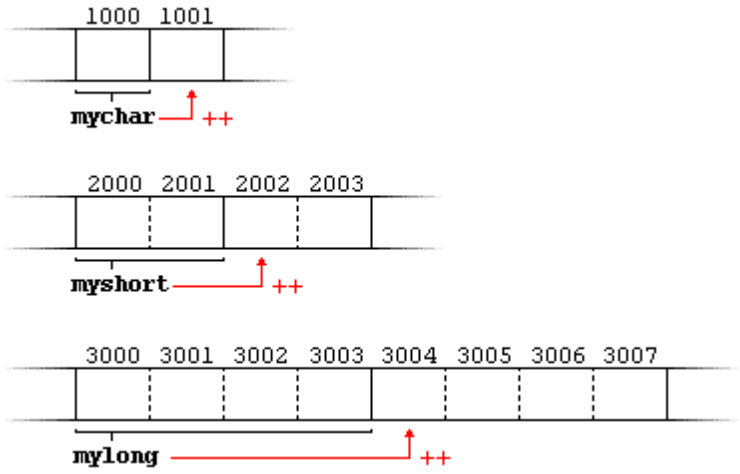
- When we saw the different data types that exist, we saw that some occupy more or less space than others in the memory. For example, in the case of integer numbers, char occupies 1 byte, short occupies 2 bytes and long occupies 4.
- Let's suppose that we have 3 pointers:

```
char *mychar;
short *myshort;
long *mylong;
```

- And that we know that they point to memory locations 1000 , 2000 and 3000 respectively. So if we write:

```
mychar++;
myshort++;
mylong++;
```

- mychar , as you may expect, would contain the value 1001 . Nevertheless, myshort would contain the value 2002, and mylong would contain 3004. The reason is that when adding 1 to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer.



- This is applicable both when adding and subtracting any number to a pointer.
- It is important to warn you that both increase (++) and decrease (--) operators have a

greater priority than the reference operator asterisk (*), therefore the following expressions may lead to confusion:

```
*p++;
 *p++ = *q++;
```

- The first one is equivalent to *(p++) and what it does is to increase p (the address where it points to - not the value that contains). The second, because both increase operators (++) are after the expressions to be evaluated and not before, first the value of *q is assigned to *p and then they are both q and p increased by one. It is equivalent to:

```
*p = *q;
p++;
q++;
```

- Now let us have a look at a code that shows increments through an integer array:

```
void main()
{
int iara[] = {10,20,30,40,50};
int * ip = iara;
cout<<*ip<<endl;
ip++;
cout<<*ip<<endl;
ip++;
cout<<*ip<<endl;
ip++;
cout<<*ip<<endl;
}
```

7.9. Pointer and String

- If you declare a character table with 5 rows and 20 columns, each row would contain the same number of characters. You can define the table with the following statement.
- char names[5][20] ={{“George”},{“Mesfin”},{“John”},{“Kim”},{“Barbara”}};
- The above statement will create the following table in memory:

G	e	o	r	g	e	\0												

M	e	s	f	i	n	\0													
J	o	h	n	\0															
K	i	n	\0																
B	a	r	b	a	r	a	\0												

- Notice that much of the table is waster space. Each row takes 20 characters, even though the data in each row takes far fewer characters.
- To fix the memory-wasting problem of fully justified tables, you should declare a single-dimensional array of character pointers. Each pointer points to a string in memory and the strings do not have to be the same length.
- Here is the definition for such an array:

  char *name [5] = {{“George”},{“Mesfin”},{“John”}, {“Kim”},{“Barbara”}};
- This array is a single-dimension array. The asterisk before names makes this array an array of pointers. Each string takes only as much memory as is needed by the string and its terminating zero. At this time, we will have this structure in memory:
- To print the first string, we should use:

  cout<<*names; //prints George.
- To print the second use:

  cout<< *(names+1); //prints Mesfin
- Whenever you dereference any pointer element with the * dereferencing operator, you access one of the strings in the array

7.10 Pointer to pointer

- As the memory address where integer, float or character is stored in can be stored into a pointer variable, the address of a pointer can also be stored in another pointer. This pointer is said to be pointer to a pointer.
- An array of pointer is conceptually same as pointer to pointer type. The pointer to pointer type is declared as follows:

  Data_type ** pointer_name;

- Note that the asterisk is double here.

  int **p; //p is a pointer which holds the address another pointer.

  E.g.:

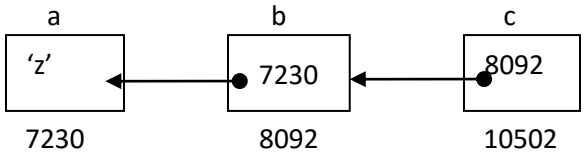
  char a; char * b;

  char ** c;

  a = 'z';

  b = &a; c = &b;

- This, supposing the randomly chosen memory locations of 7230 , 8092 and 10502 , could be described as:



(inside the cells there is the content of the variable; under the cells its location)

- Have a look at the following code:

```
#...
#...
void main()
{
        clrscr();
        int data;
        int *p1;
        int **p2;
        data = 15;
        cout<< "data = "<<data<<endl;
        p1 = &data;
        p2 = &p1;
        cout<< "data through p1 = "<<*p1<<endl;
        cout<< "data through p2 = "<< **p2<<endl;
        getch();
```

}

7.11. Dynamic Memory

Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them to be fixed before the execution of the program.

In addition to the program stack (which is used for storing global variables andstack frames for function calls), another memory area, called the **heap**, is provided.

The heap is used for dynamically allocating memory blocks during programexecution. As a result, it is also called **dynamic memory**. Similarly, the programstack is also called **static memory**.

Two operators are used for allocating and de-allocating memory blocks on theheap. The new operator takes a type as argument and allocated a memory blockfor an object of that type. It returns a pointer to the allocated block.

a. The New Operator

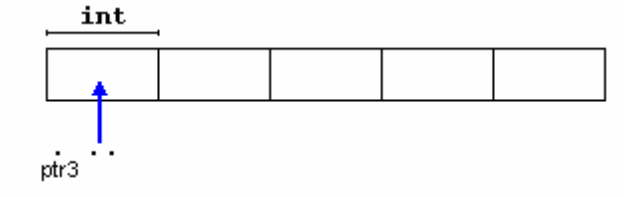
In C++ new operator can create space dynamically i.e at run time.When the space is created for a variable at compile time this approach is called static. If space is created at run time for a variable, this approach is called dynamic. See the following two lines:

```
int a[10];//creation of static array
int *a;
a = new int[10];//creation of dynamic array
```

Let's have another example:

```
int * ptr3; ptr3 = new int [5];
```

In this case, the operating system has assigned space for 5 elements of type **int** in the heap and it has returned a pointer to its beginning that has been assigned to **ptr3.** Therefore, now, **ptr3** points to a valid block of memory with space for 5 **int** elements.



You could ask what is the difference between declaring a normal array and assigning memory to a pointer as we have just done. The most important one is that the size of an array must be a constant value, which limits its size to what we decide at the moment of designing the program before its execution, whereas the dynamic memory allocation allows assigning memory during the execution of the program using any variable, constant or combination of both as size.

The dynamic memory is generally managed by the operating system, and in the multi-task interfaces can be shared between several applications, so there is a possibility that the memory exhausts. If this happens and the operating system cannot assign the memory that we request with the operator **new,** a null pointer will be returned. For that reason it is recommendable to always verify if after a call to instruction **new** the returned pointer is null:

*int * ptr3; ptr3 = new int [5];*

if (ptr3 == NULL) { // error assigning memory. Take measures. *};*

If ptr3 is **NULL**, it means that there is no enough memory location in the heap to be given for ptr3

b. Operator delete

Since the necessity of dynamic memory is usually limited to concrete moments within a program, once this one is no longer needed it shall be freed so that it become available for future requests of dynamic memory. For this exists the operator delete, whose form is:

delete pointer ;

or

delete [] pointer ;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for multiple elements (arrays).

In most compilers both expressions are equivalent and can be used without distinction, although indeed they are two different operators and so must be considered for operator overloading.

In the following simple example, a program that memorizes numbers, does not have a limited amount of numbers that can be introduced, thanks to the concept and power of pointer that we request to the system as much space as it is necessary to store all the numbers that the user wishes to introduce.

```
#include <iostream.h>
#include <stdlib.h>
#include<conio.h>
int main ()
{
char input [100];
int i,n;
long * num;// total = 0;
cout << "How many numbers do you want to type in? ";
cin.getline (input, 100);
i=atoi (input);
num= new long[i];
if (num == NULL)
{
cout<<"\nno enough memory!";
getch();
exit (1);
}
for (n=0; n<i; n++)
{
cout << "Enter number: ";
cin.getline (input, 100);
num[n]=atol (input);
}
cout << "You have entered: ";
for (n=0; n<i; n++)
cout << num[n] << ", ";
delete[] num;
getch();
return 0;
        }
```

NULL is a constant value defined in C++ libraries specially designed to indicate null pointers. In case that this constant is not defined you can do it yourself by defining it to 0:

8. Structures, Unions and Enumerations

8.1. Structures

8.1.1. Introduction

A structure is a collection of one or more variableswith different data types grouped together that can be referred using a single name (group name) and a member name.The variables that make up the structure are called members. (Structure members are also commonly referred to as elements or fields.)A structure empowersus to create our own data types by using the collection of fundamental and derived datatypes available in C++ which serves as an efficient way of

grouping several pieces ofrelated information together. Just the same way as random lying papers are grouped in afile and words are put together to form sentences.

We can refer to a structure as a single variable, and we can also initialize, read and change the parts of a structure (the individual variables that make it up).

Till now, we have learnt about simple data types like int, char, and float. Structure allowsus to declare user defined data types thus increasing the number of data types.

The General Syntax of structures is:

```
Struct [structure tag]
{
Member definition;
Member definition;
…
Member definition;
}[one or more structure variables];
```

The structure definition tells us about how the structure is organized. It informs thecompiler about the various data members present inside the structure.

Let us see an example:

```
struct StudentStructure tag
{
char ID[8];
char FName[15];
char LName[15];members of the structure
char Sex;
int age;
float CGPA;
};// all structures end with semicolon
```

The above "Student" structure is aimed to store student record with all the relevant details.

The important thing to understand at this point is that this structure definition alone doesnot actually create any variables in other words no memory has been assigned to thesevariables. With this declaration one has declared a type of structure which we will usefurther to create structure variables. They are used to declare new data types.

After the definition of the structure, one can declare a structure variable using the structure tag. If we need a variable to have the above structure property then the declaration would be:

Student stud1;

This statement would declare a variable 'stud1' of type student. We could easily understandthis by comparing it to a normal declaration that is declaring a variable 'stud1' of say, typeinteger.

int stud1;

The only difference being in the two declarations is that in the 1st case stud1 is a variable oftype 'student' and in the second case stud1 is a variable of type int.

Now let me ask you a very interesting question what is the size of this structure variable 'stud1' defined above?

The answer to this question is very simple, as we know that a structure is nothing but aheterogeneous collection of data hence the size of a structure variable can be calculatedeasily by adding up individually the size of all the structure members present in thestructure. Using this logic we can easily calculate the size of the structure variable 'stud1'which comes out to be 45 bytes.

Example 1:

a. Declare a structure 'date' having the following members:
   Day -- A string of size[3]
   Represents the day part of the date
   month -- A string of size[20]
   Represents month name of a date

year -- A string of size [4]
Represents the year part of the date

b. What is size of a variable of type date hence created?

Solution

a.

```
struct date
{
char day[3];
char month[20];
char year[4];
};
```

b. The size of a structure variable of type 'date' will occupy 27 bytes of computers memory.

Note: The size of structure might differ from one machine to another also the size of the structure is also determined by a concept known as 'word alignment'

Example 2:

b. Declare a structure 'Employee' having the following members:
Emp_Name -- A string of size[15]
Represents the name of the employee
Emp_IdNo. -- A Long int variable
Represents the employees Id number
Emp_Designation -- A string of size[20]
Represents the designation of the employee
Emp_salary -- A float variable
Represents the salary of the employee

c. Declare C++ statement to declare two structure variables 'Emp1', 'Emp2' of type 'Employee

Solution

a.

```
struct Employee
{
char Emp_Name[15];
long int Emp_IdNo;
char Emp_Designation[20];
float Emp_salary;
};
```

b. Employee Emp1,Emp2;

We can also define the structure variables after the structure, in such a case the tag namebecomes optional but then one cannot declare any more structure variables in the mainfunction.

For example, consider the structure 'Employee', for this structure we can also declare thestructure variables 'Emp1', 'Emp2'without giving a name to the structure as shownbelow.

```
struct <optional tag/name>
{
char Emp_Name[15];
long int Emp_IdNo;
char Emp_Designation;
float Emp_salary;
} Emp1, Emp2;
```

The only disadvantage using this feature is that now we cannot create any more structurevariables of this type.

8.1.2. Accessing members of a structure

We can access members of a structure with the help of two operators in C++. The dotoperator **(.)** and the structure pointer operator **(->)** also known as the arrowoperator.

The General syntax to access members of a structure variable would be:

VarName.Member

Where VarName is the variable name of the structure variable and member is variable name of the members of the structure.

Example 1:

For the above student structure:

```
struct Student Stud; //declaring Stud to have the property of the Student structure
strcpy(Stud.FName,"Abebe"); //assigned Abebe as First Name
Stud.CGPA=3.21; //assignes 3.21 as CGPA value of Abebe
cout<<Stud.FName; //display the name
cout<<Stud.CGPA; // display the CGPA of Abebe
```

The important thing to note here is that the part of the statement before the (.) operator isthe structure variable name 'Stud' and the part after the dot operator 'CGP, FName' are thename of the structure member to be accessed.

Structures members can be treated just like any other ordinary variables available in C++.

That is all the various operations which can be done on any ordinary variables are alsovalid with structure members.

Example 2:

```
#include<iostream.h>
#include<conio.h>
# include <string.h>
struct student
{
int roll_no;
char student_name[15];
float avg;
};
int main()

{
 student s1;// Declaring structure variable s1 of type student
 cout<<"Enter Student Name:\n";
 cin.getline(s1.student_name,15); // Accessing the structure member 'student_name'
 cout<<" Enter Roll No: \n";
    cin>>s1.roll_no; // Accessing the Structure member 'roll_no'
cout<<"Enter Student Average\n";
    cin>>s1.avg; // Accessing the structure member 'avg'
cout<<"PRINTING DERAILS\n";
cout<<"\n Roll No:"<<s1.roll_no;
cout<<"\t"<< " Student Name:"<<s1.student_name;
cout<< " Student Avg:"<<s1.avg;
getch ();
return 0;
}
```

When this program is executed at first the compiler creates a structure ‘student’ with theabove mentioned structure members. Then the compiler declares a structure variable ‘s1’of type student. In this program, the user enters the details of a student which includestudents roll number, name and average and then the program display these details onthe screen.

PROGRAMING TIPS

- Tag Names for structures should always be initialized to increase the scope of creating structure variables at time ofstructure declaration.
- All structure members should have different names toprevent logical errors.

Example 3:

Using the structure definition for structure 'Library' having the following members writea c++ program which declares a structure variable L1:

Book_Name -- A string of size[15]

Represents the name of the book

Book_IdNo -- A int variable

Represents the books Id number

Book_Shelf -- A string of size[5]

Represents the Book shelf number wherethe book is stored.

The user should then be asked to enter the details for the structure variable. The programshould also check whether the book name is "Cleopatra" if so, the program should print asuitable message on the screen.

Solution

```
// Header Files
# include <iostream.h>
# include <conio.h>
# include <string.h>
struct Library// Structure Library
{
char Book_Name[15];
int Book_IdNo;
char Book_Shelf[5];
};
int main()// Main Function

{
Library L1;
char str[]="Cleopatra";
cout<<"Enter Respective Details:";
```

```
cout<<endl<<"Book Name";
gets(L1.Book_Name);
cout<<endl<<"Book_IdNo";
cin>>L1.Book_IdNo;
cout<<endl<<"Book Shelf";
gets(L1.Book_Shelf);
if (strcmp(str,L1.Book_Name)==0)
{
cout<<endl<<"Condition Satisfied";
}
else
{
cout<<endl<<"Condition not Satisfied";
}
getch();
return 0;

}
```

Common programming errors

- Not ending the structure definition with a (;) semicolon is a compilation error.
- Not using the dot operator (.) to reference to structure members is a compilationerror.

8.1.3. Structure Arrays

Before we learn about arrays of structures let's compare arrays and structures. Both arraysand structures are collection of data types. The major difference in them being that anarray is a collection of data of the same type, where as a structure helps to group togetherdata that may vary in type. Let us look at the following comparison between array andstructures.

Let us look at the following comparison between array andstructures.

```
struct Employee char Emp_Name [15];
{charEmp_Designation[15];
charEmp_Name[15];
int Emp_IdNo ;
char Emp_Designation[15];
};
```

Structures Arrays

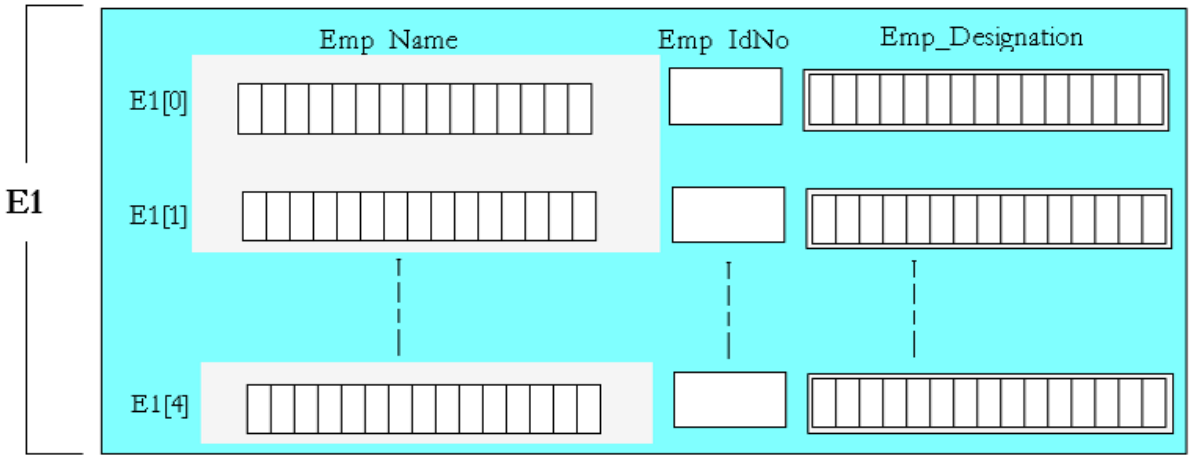
Now as we can see the structure ‘employee’ groups data of three different data types:char, int and float whereas the array ‘name[15]’ is a collection only of character type ofdata. Structures allow us to build data bases, especially if one considers the possibility ofbuilding arrays of them. Arrays of structure are declared by declaring thestructure variables as arrays.

Example:

```
# include <iostream.h>
# include <conio.h>
# include <stdio.h>
struct Employee// Structure Employee
{
char Emp_Name[15];
int Emp_IdNo;
char Emp_Designation[15];
};
void  main()// Main Program
{
Employee E1[10]; // Array of structures
for (int i = 0; i<10;i++)
{
cout<<" Enter Respective Details for Employee"<<i+1;
cout<<endl<<"1. Name";
gets(E1[i].Emp_Name);
cout<<endl<<"2. Employee Id No.";
cin>>E1[i].Emp_IdNo;
cout<<endl<<"3. Employee Designation";
cin>>E1[i].Emp_Designation;
}
cout<<endl<< " Printing Employee database";
for (i=0;i<10;i++)
{
cout<<endl<<"1. Name"<<E1[i].Emp_Name;
```

```
cout<<endl<<"2. Employee Id No."<<E1[i].Emp_IdNo;
cout<<endl<<"3. Employee Designation"<<E1[i].Emp_Designation;
}
getch ();
}
```

Arrays of structure have the properties same as a normal array. The only difference beingis that this time they are collection of a ‘struct’ data type. All operations valid on ordinaryarrays are also valid with arrays of structure. For example, in the program above we havedeclared an array ‘E1’ with index value 10 of type ‘Employee’. This array of structurehelps us to create a database for a set of 10 employees with the name, designation and Id No for each employee. Each index value in this array of structure is accessed in a similarmanner with the help of the (.) operator.



Memory Allocation for Structure variable E1

8.1.4. Passing Structures to Functions

8.1.4.1. Passing Structure Members to Functions

When you pass a member of a structure to a function, you are actually passingthe value of that member to the function. Therefore, you are passing a simplevariable (unless, of course, that element is compound, such as an array). Forexample, consider this structure:

```
struct fred
{
char x;
```

```
        int y;
        float z;
        char s[10];
    } mike;
```

Here are examples of each member being passed to a function:

```
    func(mike.x); /* passes character value of x */
    func2(mike.y); /* passes integer value of y */
    func3(mike.z); /* passes float value of z */
    func4(mike.s); /* passes address of string s */
    func(mike.s[2]); /* passes character value of s[2] */
```

If you wish to pass the address of an individual structure member, put the & operatorbefore the structure name. For example, to pass the address of the members of thestructure mike, write

```
    func(&mike.x); /* passes address of character x */
    func2(&mike.y); /* passes address of integer y */
    func3(&mike.z); /* passes address of float z */
    func4(mike.s); /* passes address of string s */
    func(&mike.s[2]); /* passes address of character s[2] */
```

Remember that the & operator precedes the structure name, not the individualmember name. Note also that s already signifies an address, so no & is required.

8.1.4.2. Passing Entire Structures to Functions

When a structure is used as an argument to a function, the entire structure is passedusing the standard call-by-value method. Of course, this means that any changesmade to the contents of the structure inside the function to which it is passed do notaffect the structure used as an argument.

When using a structure as a parameter, remember that the type of the argumentmust match the type of the parameter. For example, in the following program both theargument arg and the parameter parm are declared as the same type of structure.

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

```
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
cout<< parm.a;
}
```

As this program illustrates, if you will be declaring parameters that are structures,you must make the declaration of the structure type global so that all parts of yourprogram can use it. For example, had struct_type been declared inside main() (forexample), then it would not have been visible to f1().

8.1.5. Structure Pointers

C++ allows pointers to structures just as it allows pointers to any other typeof variable. However, there are some special aspects to structure pointers thatyou should know.

8.1.5.1. Declaring a Structure Pointer

Like other pointers, structure pointers are declared by placing * in front of a structurevariable's name. For example, assuming the previously defined structure addr, thefollowing declares addr_pointer as a pointer to data of that type:

```
struct addr *addr_pointer;
```

Remember, in C++ it is not necessary to precede this declaration with the keywordstruct.

8.1.5.2. Using Structure Pointers

There are two primary uses for structure pointers:

- to pass a structure to a functionusing call by reference,
- to create linked lists and other dynamic data structures thatrely on dynamic allocation.

There is one major drawback to passing all but the simplest structures to functions:the overhead needed to push the structure onto the stack when the function call isexecuted. (Recall that arguments are passed to functions on the stack.) For simplestructures with few members, this overhead is not too great. If the structure containsmany members, however, or if some of its members are arrays, run-time performancemay degrade to unacceptable levels. The solution to this problem is to pass only apointer to the function.

When a pointer to a structure is passed to a function, only the address of thestructure is pushed on the stack. This makes for very fast function calls. A secondadvantage, in some cases, is when a function needs to reference the actual structureused as the argument, instead of a copy. By passing a pointer, the function canmodify the contents of the structure used in the call.

To find the address of a structure variable, place the **&** operator before thestructure's name.

For example, given the following fragment:

```
struct bal {
float balance;
char name[80];
} person;
struct bal *p; /* declare a structure pointer */
then
```

```
p = &person;
```

places the address of the structure person into the pointer p.

To access the members of a structure using a pointer to that structure, you mustuse the –> operator. For example, this references the balance field:

```
p->balance
```

The –> is usually called the arrow operator, and consists of the minus sign followedby a greater-than sign. The arrow is used in place of the dot operator when you areaccessing a structure member through a pointer to the structure.

8.1.6. Nested structure

When a structure is a member of another structure, it is called a nested structure.

For example, the structure Distance is nested inside Room in this example:

```
Example
struct Distance
{
int feet;
float inches;
};
struct Room
{
char name[20];
Distance length;
Distance width;
};
```

To access the nested structure members, we have to apply dot operator twice.

Example

Room Dining;

```
strcpy(Dining.name,"dining room");
Dining.length.feet=10;
Dining.length.inches=6;
Dining.width.feet=20;
```

Initializing Nested Structures

In the previous example we can initialize the nested structure suchthat each structure of type Distance, which is embedded in Room,is initialized separately which involves surrounding the valueswith braces and separating them with commas.

Example;

```
Room dining = {"Dining room", {13, 6.5},{20,10} };
struct Distance
{
int feet;
float inches;
};
struct Room
```

```
{
char name[20];
Distance length;
Distance width;
};
void main( )
{
Room dining= {“Dining room”,{13,3.2},{10,0.0} };
cout<<dining.name<<endl;
cout<<dining.length.feet<<endl;
cout<< dining.length.inches<<endl;
cout<< dining.width.feet<<endl;
cout<<dining.width.inches<<endl;
}
```

As you can see, the members of each structure are referenced from outermost toinnermost. Standard C specifies that structures may be nested to at least 15 levels.Standard C++ suggests that at least 256 levels of nesting be allowed.

8.2. Unions

A union is a memory location that is shared by two or more different variables,generally of different types, at different times. Declaring a union is similar todeclaring a structure. Its general form is

```
union union-type-name {
        type member-name;
        type member-name;
        type member-name;
        .
        .
        .
        } union-variables;
```

For example:

```
union u_type {
int i;
char ch;
};
```

This declaration does not create any variables. You may declare a variable eitherby placing its name at the end of the declaration or by using a separate declarationstatement.

When declaring union variables in C++, you need use only the type name—you don't need to precede it with the keyword union. For example, this is howcnvt is declared in C++:

u_type cnvt;

In C++, preceding this declaration with the keyword union is allowed, but redundant.In C++, the name of a union defines a complete type name.

8.3. Enumerations

An enumeration is a set of named integer constants that specify all the legal valuesa variable of that type may have. Enumerations are common in everyday life. Forexample, an enumeration of the coins used in the United States is

penny, nickel, dime, quarter, half-dollar, dollar

Enumerations are defined much like structures; the keyword enum signals the startof an enumeration type. The general form for enumerations is

enum enum-type-name { enumeration list } variable_list;

Here, both the type name and the variable list are optional. (But at least one mustbe present.) The following code fragment defines an enumeration called coin:

enum coin { penny, nickel, dime, quarter,half_dollar, dollar};

The enumeration type name can be used to declare variables of its type.

In C++, an enumeration name specifies a complete type.

9. Files and Graphics

9.1. Files

9.1.1. Introduction

All the programs we have looked at so far use input only from the keyboard, and output only to the screen. If we were restricted to use only the keyboard and screen as input and output devices, it would be difficult to handle large amounts of input data, and output data would always be lost as soon as we turned the computer off. To avoid these problems, we can store data in some secondary storage device, usually magnetic tapes or discs. Data can be created by one program, stored on these devices, and then accessed or modified by other programs when necessary. To achieve this, the data is packaged up on the storage devices as data structures called *files*.

Applications generally involve tremendous amount of data to be read and written on to the auxiliary storage device. Hence, to save such information on to the disk we use a concept called data files. File is a collection of numbers, symbols and text placed onto the disk. Thus, files allow us to store information permanently on to the disk and then access them when needed. Further, its information can be altered depending upon the needs. This process leads to the concept of data files.

A file can be considered as a stream of characters. It is a set of records that can be accessed through the set of library functions. These functions are available in stdio.h. the data can be stored in secondary storage devices such as floppy or hard disk.

9.1.2. Streams

We can think of a stream as a channel or conduit on which data is passed from senders to receivers. As far as the programs we will use are concerned, streams allow travel in only one direction. Data can be sent out from the program on an *output stream*, or received into the program on an *input stream*. For example, at the start of a program, the standard input stream "cin" is connected to the keyboard and the standard output stream "cout" is connected to the screen.

In fact, input and output streams such as "cin" and "cout" are examples of (stream) *objects*. So learning about streams is a good way to introduce some of the syntax and ideas behind the

object-oriented part of C++. The header file which lists the operations on streams both to and from files is called "fstream". We will therefore assume that the program fragments discussed below are embedded in programs containing the "include" statement

#include<fstream>

As we shall see, the essential characteristic of stream processing is that data elements must be sent to or received from a stream one at a time, i.e. in *serial* fashion.