Chapter One

1. Introduction to programming

What is programming?

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.

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?

A programming language is a language for communication between a person and computer. The content of communication is known as a program. Programs are set of instructions, which enable a computer to perform a required operation.

Programming Language: is a set 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.

Generations of Programming Language

Thus, programming languages fit into five generations: First generation languages, second-generation languages, third generation languages, fourth-generation languages and fifth-generation languages. These generations of programming languages can also be categorized into two broad categories: *-low and high-level languages*. Low-level languages are machine-dependent; that is, they are designed to be run on a particular computer. In contrast, high-level languages (for example, COBOL and BASIC) are machine-independent and can be run on a variety of computers.

The first two generations were low-level and the rest high-level of programming languages. The higher-level languages do not provide us with greater programming capabilities, but they do provide a more sophisticated program/computer interaction with simple instructions. In short, the higher the level of the language, the easier it is to understand and use. For example, in a fourth-generation language you need only instruct the computer system *what to do, not necessarily how to do it.*

When programming in one of the first three generations of languages, you have to tell the computer what to do and how to do it. What comprises a new generation is less clear; therefore, languages after the fourth generation are referred to as *very high-level languages.*

With each new level, **fewer instructions** are needed to tell the computer to perform a particular task. A program written in a second-generation language that computes the total sales for each sales representative, and then lists those over quota, may require 100 or more instructions; the same program in a fourth-generation language may have fewer than 10 instructions.
The ease with which the later generations can be used is certainly appealing, but the earlier languages also have their advantages. All generations of languages are in use today.

First generation (Machine languages, 1940's): Difficult to program in; they are dependent on machine languages of the type of computer being used. However, machine language allows the programmer to interact directly with the hardware, and it can be executed by the computer without the need for a translator.
Machine language is the ultimate low-level language. To communicate with the first generation computer programmers had to *write programs in machine language*: - the 0's and 1's of binary code. Programming in 0s and 1s needed reducing statements such as add, subtract and divide into a series of 0's and 1's. Programs written in this way are machine-dependent. There is no universal machine language as different hardware designs dictate different data and memory locations, and hence different bit strings. Programmer must remember exact memory addresses used to store data and instructions. They *execute very quickly and use memory very efficiently, as no conversion from source code is necessary*. *Writing the programs is difficult, tedious, time consuming and hard to debug*. A high *level of understanding of hardware specifics is required*. All programs written using other languages are translated into machine language before they can be executed.

End users who wanted applications have to work with *specialized programmers who could understand, think and work directly in the machine language* of a particular computer. So *programming in machine is slow and labor intensive process.*

Second generation (Assembly languages, appeared in the early 1950's): Use *symbolic names for operations and storage locations*. Assembly language is *easier to use than machine language but it is still difficult to understand*. A systems program called an **assembler** translates it into machine language. Different computer architectures have their own machine and assembly languages, which means that *programs written in these languages are not portable to other, incompatible systems*. Many programmers still use assembly languages because they still give them close control over the hardware.

- Instead of using 0's and 1's programmers substituted language like *acronyms and words such as add, sub(subtract) in programming statements*.

§ *Costly in terms of programmer time*
§ *Difficult to read and debug*, and
§ *Difficult to learn*
§ *Highly used in system software development*

Programs are *easier to create and to debug than machine language*, but are still *prone to errors and tedious and execute faster than high-level languages*.

Third generation (High-level languages, emerged from 1950's - 1970's): *Use English-like instructions* like *print* and mathematicians were able to define variables with statements such as Z=A+B. Many high-level languages are **portable**. Each *has syntax rules that must be followed*. Such languages are *much easier to use than assembly language* but they can take considerable time to learn. Two types of systems programs exist for translating them into machine language: ***interpreters and compilers.***

In High-level languages, *instructions closely resemble human language and mathematical notation*. Programmers do not require detailed knowledge of hardware specifics as these languages extensively use symbolic representations of memory addresses and library functions. The use of common words (**reserved words**) within instructions makes them easier to learn. Assembly language code segments may be added to most high-level programs, where faster execution speed is required by the programmer. There are two types of language-translator programs used to translate into machine code, Interpreters and Compilers.

Fourth-generation (Since late 70's): Have simple, English-like syntax rules; commonly used to access **databases**. As we mentioned earlier, the third-generation programming languages discussed in the previous section are all **procedural languages** because the programmer must list each step and must use logical control structures to indicate the order in which instructions are to be executed. Fourth-generation languages (4GLs), on the other hand, are **nonprocedural languages**. These languages can be compared to the way in which you might instruct someone to cook a meal. The **nonprocedural method is simply** to state the needed output: fix a meal of chicken, rice, and salad. Using the procedural method, on the other hand, involves specifying each step - from preparing the shopping list to washing the dishes.

Obviously, the nonprocedural method is easier to write, but you have less control over how each task is actually performed. For example, the dishes might be washed by hand or in a dishwasher. When using nonprocedural languages, the methods used and the order in which each task is carried out are left to the language itself; the user does not have any control over it. In addition, 4GLs sacrifice computer efficiency in order to make programs easier to write; hence they require more computer power and processing time. As the power and speed of hardware have increased and its cost has decreased, the use of 4GLs has spread.

Because fourth-generation languages have a minimum number of syntax rules, people who have not been trained as programmers can use such languages to write application programs, as they need them. This saves time and fees professional programmers for more complex tasks. There are several categories of 4GLs languages. The most common once are: *query languages, report generators, and application generators and graphic languages.*

Query languages allow the user to retrieve information from databases by following simple syntax rules. For example, you might ask the database to locate all customer accounts that are more than 90 days overdue. Example of query languages is SQL, which has become a de facto standard.

Fifth-generation (1990's): Used in artificial intelligence and expert systems; also used for accessing databases. Fifth-generation languages (5GLs) are also nonprocedural languages and are also commonly used to query databases. Because these languages are still in their infancy, only a few are currently commercially available. They are closely tied to artificial intelligence and expert systems.

Fifth-generation languages are the "natural" languages whose instructions closely resemble human speech "Get me Debela's sales figures for the 1997 financial year" is a typical instruction. Very powerful hardware and software are required to execute such programs because of the complexity involved in interpreting commands entered in human language. **This means computers can in the future have the ability to think for themselves and draw their own inferences using programmed information in large databases. Complex processes like understanding speech would appear to be trivial using these fast inferences and would make the software seem highly intelligent.**

1.1.Problem solving using computers

Problems: Undesirable situations that prevent an organization from fully achieving its purpose, goals and objectives

- True problem situations, either real or anticipated, that require corrective action
- Unexploited Opportunities to improve a situation despite the absence of complaints
- Directives to change a situation regardless of whether anyone has complained about the current situation

Problem solving is the process of transforming the description of a problem into the solution of that problem 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.

Problem Solving: is basic intellectual process that has been refined and systemized for the various challenges people face.

There are two approaches of problem solving:

Top down design: is a systematic approach based on the concept that the structure of the problem should determine the structure of the solution and what should be done in lower level. 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.

1.2.Basics of program development

The vehicle for the computer solution to problem is a set of explicit and unambiguous instructions called programs expressed in programming language.

Quality programming is necessary for the economic and correct solution of problems. This only comes from good program design. The programs we design need to be:

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

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.

Algorithm development

An algorithm is procedure for solving a problem in terms of

- the action to execute (what to do) and
- the order in which these actions are executed(done)

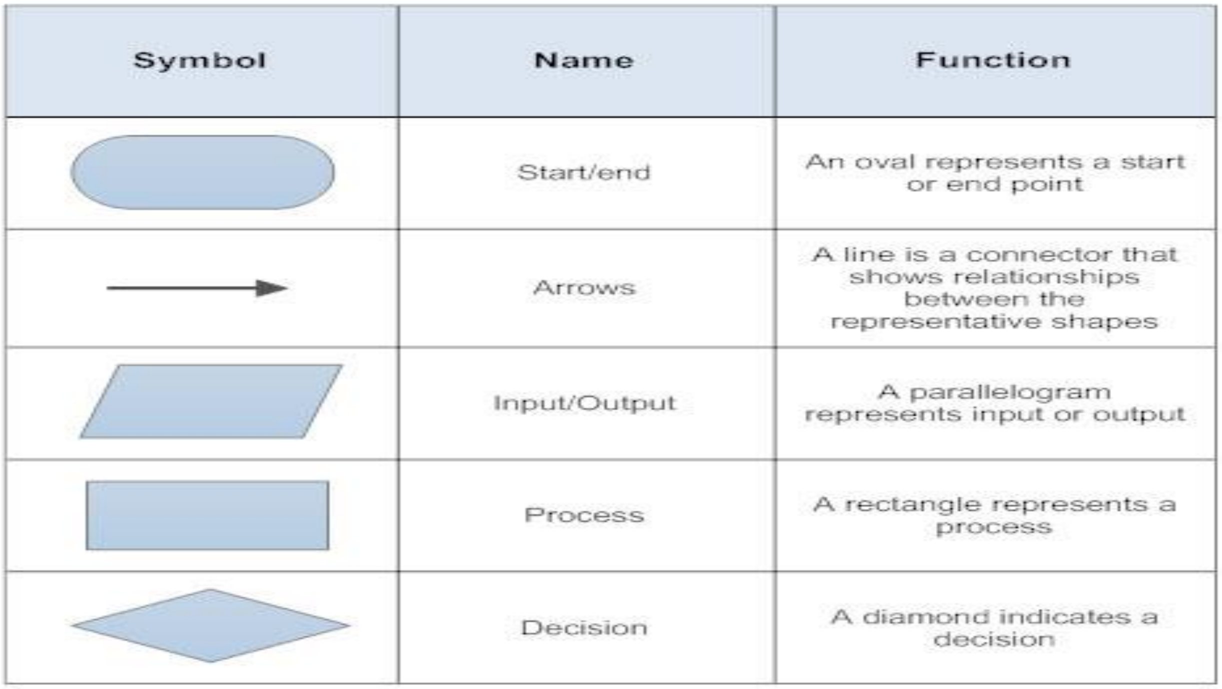
An algorithm needs to be

- Precise and unambiguous (no ambiguity in any instruction and in the order of execution)
- Simple
- Correct
- Finite (has to have an end)
- Produce expected output
- Efficient: in time, memory and other resources

An algorithm can be expressed in many ways. Some of these methods are ***narrative, flowchart and pseudo-code:*** ***Narrative***: often used to narrate the algorithm, can be understood by any user who may not have any knowledge of computer programming. Too wordy, too unambiguous and can be interpreted in different ways.Example: Accept salary of the employee. Calculate bonus as 10% of salary and add it to salary. Accept service year of employee. If the service year is greater than 10, give additional 100 birr as bonus. Display the bonus of the employee.

Flowcharts: a diagram consisting of labeled symbols, together with arrows connecting one symbols to another.

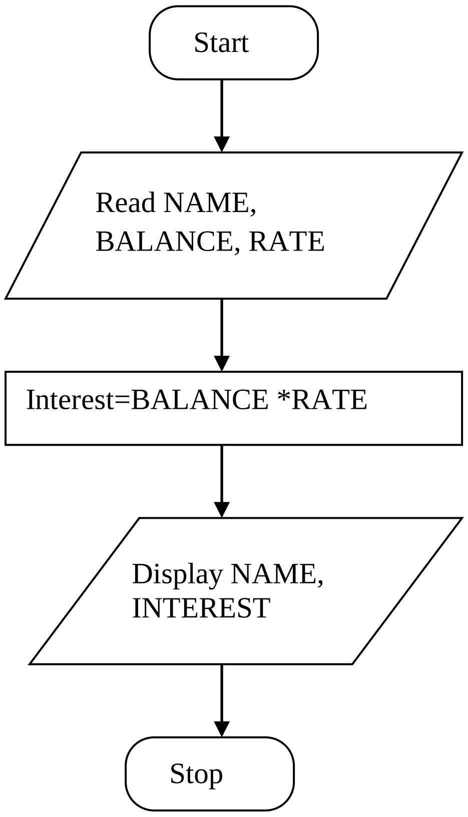
Basic flowcharting symbols are:



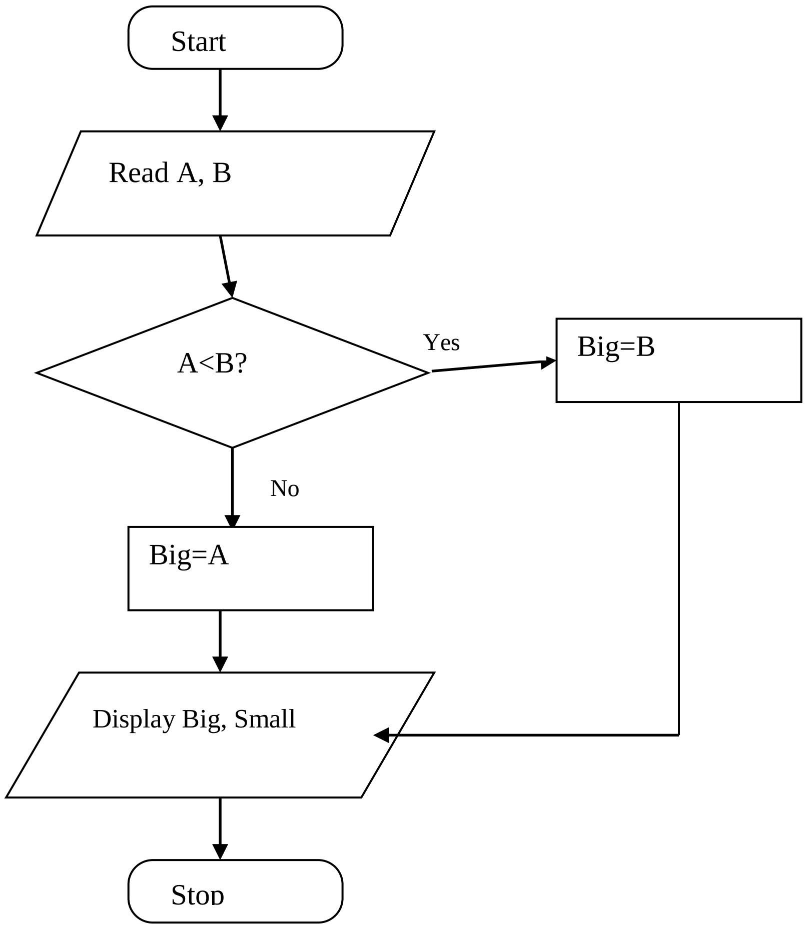
Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Example: Drawing flowchart

1) A flow chart for calculating interest amount

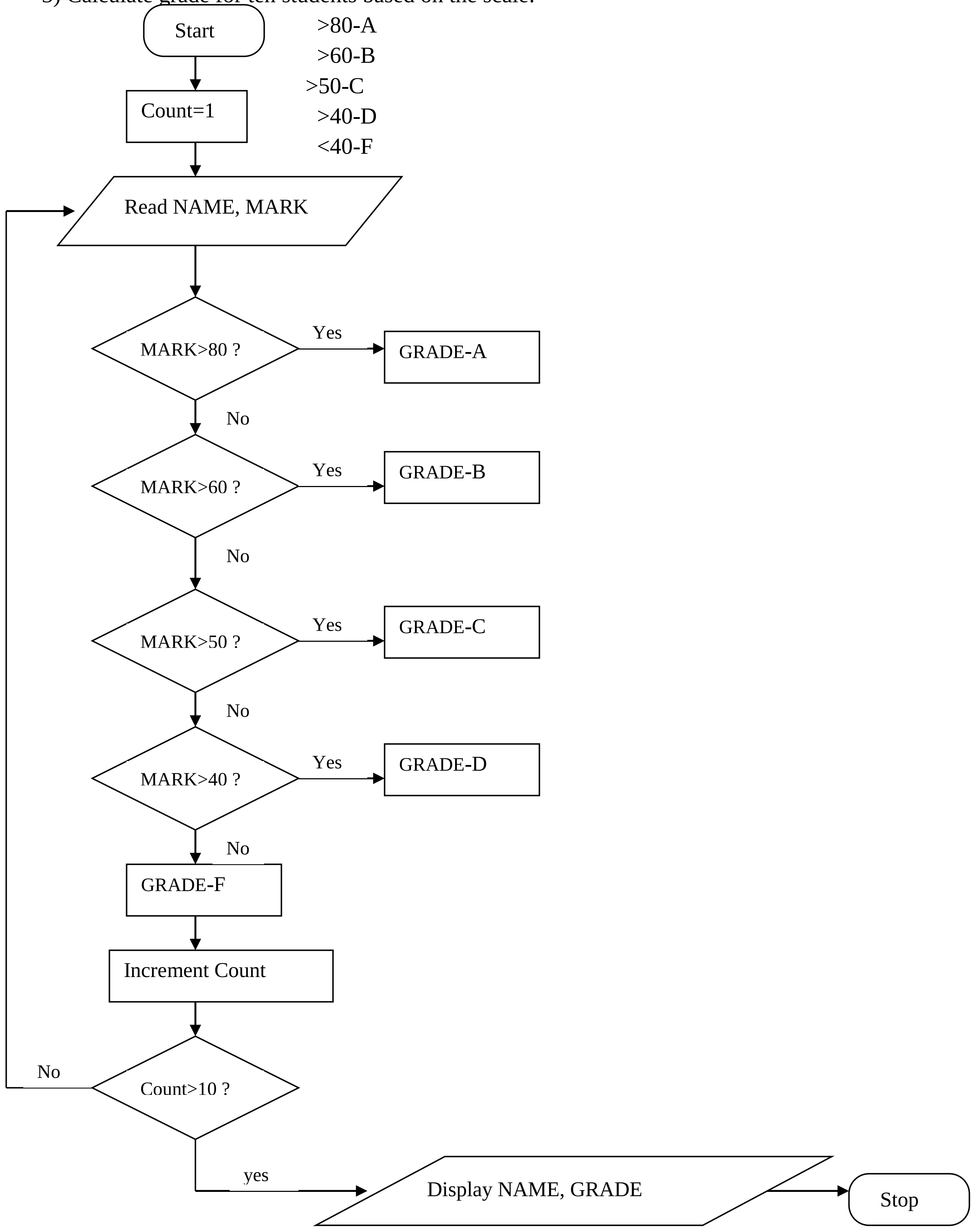


2) A program that identifies a larger and smaller number from two numbers



3) Calculate grade for ten students based on the scale:

>80-A
>60-B
>50-C
>40-D
<40-F



Pseudo-code:

It is much similar to real code. We use verbs to write pseudo-code. Capitalize important words that show actions.

Eg 1) A pseudo-code to calculate interest rate

```
ACCEPT Name, Principal, Rate
Interest=Principal X Rate
DISPLAY Name, Interest
```

2) A pseudo-code to calculate bonus

```
ACCEPT Name, Salary
Bonus=SalaryX0.1
ACCEPT Serviceyear
IF Serviceyear>10 Then
   Bonus+100
ENDIF
DISPLAY Bonus
```

3) A pseudo-code that calculates grade

```
ACCEPT Mark, Name
IF Mark>80 Then
   Grade←A
ELSE IF Mark>=70 Then
   Grade←B
ELSE IF Mark>=60 Then
   Grade←C
ELSE IF Mark>=50 Then
   Grade←D
ELSE
   Grade←F
ENDIF
DISPLAY Grade, Name
```

<u>Worksheet 1</u>

For each of the problems below, develop a flow chart

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) Find the average of two numbers given by the user.

Chapter Two

2. Basic Concepts of C++ Programming

2.1 Structure of C++ program

Structure of a C++ program includes:

```
✓ Documentation section
✓ Linking section
✓ Definition section
✓ Global declaration section
✓ Class declaration
✓ Function
✓ main() functions
  {
          Initializations;
          Executable parts;
  }
✓ Function1()
  {
          Initializations;
          Executable parts;
  }…n
```

- ***Documentation Section:*** include comment parts, which are ignored by compiler. The comments are used to describe the functionality of each statement in the program, its copyright, author, and date of compilation and purpose of the program to the user. C++ supports single line (// comment) and multiline (/* comments*/).

  Syntax: //This statement is a single line comment

  /*This is multiline comment*/
- ***Link section:*** here we can link the necessary files and libraries to current files. Using #include directive we can link the necessary libraries to current files.

  Syntax: #include<file or library name>

  Example: #include<iostream.h>
- ***Definition Section:*** It is possible to define symbolic constants. Symbolic constants are normal identifiers which value cannot be altered.

  Syntax: #define identifier constant
- ***Global declaration Section:*** variables declared under this section are available throughout the program.
- ***Class declaration Section:*** defines the class declaration.

- ***Functions:*** The function *main*, which has been written in the program, is an example of function. A function is defined as group of instructions, which are assigned a name and accessed by the name.

  Syntax:

  return type function name ()
  {
  Statement of function;
  }

  A function is identified by the compiler with the help of parenthesis ( ) after the function name. Without the parenthesis the compiler thinks it as a variable or complaints that it is undefined symbol. So to define a function, parenthesis is must. The *statement inside the brace* {} forms the body of the function and specifies the task of the function. *Return type specifies* the type of data the function has to send to the calling function after the execution. It may be the result of the task performed in the function.
- ***main ( ) function:*** the execution of the program starts from main function. Every program must have only one main function. All other functions are called either directly or indirectly from main. The initialization or other executable statements are included with in the main function.
- ***Initialization part:*** the variables that are used in the program should be initialized here. These variables are not available to all the functions. In C++ we can initialize the variables at any part in the function.
- ***Executable part:*** A statement declared under executable part performs some tasks. Every statement under global declaration, initialization and executable parts should be terminated with the *semi colon ';'. A semi colon acts as a statement terminator like full stop in English.*

  Syntax:

```
#include<iostream.h>
int main()
{
Statements;
Other functions;

}
```

2.2 The C++ Compilation Process

C++ is a 3rd-generation language. These types of languages must be translated into a machine language in order to be executed by a CPU. The process of translating high-level language into machine language is called the compilation process.

The compilation process consists of the following steps.

```
    edit source code ->  compile   ->   link    ->   execute
      (editor)         (compiler)    (linker)      (loader)
```

Program source code is entered into a file using a text editor. After the code has been entered, a compiler program is started that translates the source into an object code file. The object code file is linked with other object code files that come with the compiler and an executable file (or program) is created. In order to execute the program, a program called the loader copies the executable file into the memory of the computer and sends an execute command to the CPU. It should be noted that errors can occur during each step.

A ***source file*** ending with ".c" contains C source code; whereas, a file ending with ".cpp" is a C++ file. A file ending with ".h" can be both a C and C++ header file. Sometimes the suffix ".hpp" (or ".H") is used to indicate a C++ only header file.

The ***compiler*** is a program that usually consists of many phases. The first phase of compilation is called ***preprocessing***. The ***preprocessor*** does many things, but two features that must be learned immediately are file inclusion and macro (manifest constant) definitions. After preprocessing, the compiler executes two primary steps: lexical analysis and parsing. During lexical analysis, the source code is broken up into tokens and the tokens are passed to the parser. The parser does syntax and semantic analysis, which includes the generation of object code (i.e. machine language).

The ***linker*** "combines" all object code files into an executable file. Typically, the object files created by your source files are linked with object files that are packaged into libraries.

Most implementations allow each step of the compilation process to be executed as a stand-alone procedure. For example, compile a source file but do not invoke the linker; execute the preprocessor only; or, invoke the linker only.

Some older compilers translate C source code into assembly language, and then execute an assembler program to translate the assembly language into machine language.

Early C++ compilers (those prior to 1992) translated C++ code into C code and then executed the C compiler.

The ***loader*** reads a program (i.e. executable file) into memory. Once this is completed, it becomes a process and the CPU executes it.

2.3 Syntax and Semantics

The **Syntax** of a programming language consists of the rules for the correct use of the language. This involves the correct grammatical construction and arrangement of the language, correct spelling, hyphenation, inflection and so on.

The **semantics** of a programming language deal with the meanings given to syntactically correct constructs of the language. Usually, semantics is defined in terms of the program's run-time behavior: What happens when the program is executed with a certain set of inputs, what statements are executed, what values are assigned to the variables, and what output is produced.

Thus syntax has nothing to do with "meaning" or run-time behavior of a program. A program could be syntactically correct yet meaningless. The program below (code fragment) is syntactically correct but does not have any meaning at runtime (never terminates).

```
  sum=0;
while (sum!=-1)
sum=sum+10;
```

Yet syntax is a prerequisite to meaningful expression. Thus, a programming language must have a good syntactic definition before it can properly support the development of meaningful programs.

2.4 The parts of a simple C++ Program

- To understand the basic parts of a simple program in C++, lets have a look at the following code:

```
#include<iostream.h>
void main()
   {
        cout<<"\n Hello World!";
   }
```

- Any C++ program file should be saved with file name extension " ***.CPP*** "
- Type the program directly into the editor, and save the file as hello.cpp, compile it and then run it. It will print the words Hello World! on the computer screen.
- The first character is the #. This character is a signal to the preprocessor. Each time you start your compiler, the preprocessor runs through the program and looks for the hush or sharp (#) symbols and act on those lines before the compiler runs.
- 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.

- If the path of the file is not specified, the preprocessor looks for the file under ***c:\tc\include\*** folder or in ***include*** folder of the location where the editor is stored.
- The effects of line 1, i.e. include<iostream.h> is to include the file iostream.h into the program as if the programmer had actually typed it.
- When the program starts, main() is called automatically.
- Every C++ program has a main() function.
- The return value type for main() here is void, which means main function will not return a value to the caller (which is the operating system).
- The main function can be made to return a value to the operating system.
- The Left French brace "{"signals the beginning of the main function body and the corresponding Right French Brace "}" signals the end of the main function body. Every Left French Brace needs to have a corresponding Right French Brace.
- The lines we find between the braces are statements or said to be the body of the function.
- A statement is a computation step which may produce a value or interact with input and output streams.
- ***The end of a single statement ends with semicolon (;).***
- The statement in the above example causes the string "Hello World!" to be sent to the "cout" (VDU) stream which will display it on the computer screen.

A brief look at cout and cin

- ***cout*** is an object used for printing data to the screen.
- To print a value to the screen, write the word ***cout***, followed by the insertion operator also called output redirection operator (<<) and the object to be printed on the screen.
    - Syntax: ***cout***<<Object;
    - The object at the right hand side of **<<** operator can be variables, strings or expressions.
      E.g.
      cout<<"Hello There "; //prints Hello There on the screen
      cout<<x; // prints the content of variable x on the screen
      cout<<"x"; // prints the character x on the screen
      cout<<100; // prints number 100 on the screen
      cout<< x + y; // prints the result of the sum of the content of variables x and y
- **cin** is an object used for taking input from the keyboard.
- To take input from the keyboard, write the word ***cin***, followed by the input redirection operator (>>) also called extraction operator and the object name to hold the input value.
    - Syntax: cin>>Object
- **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.

- The user of the program should supply correct form of data when it is taken as input from the keyboard. If the input value is different from the variable's data type or capacity, then there is a possibility of erroneous result and erratic behavior of the program.
- Both << and >> return their right operand as their result, enabling multiple input or multiple output operations to be combined into one statement. The following example will illustrate how multiple input and output can be performed:

  E.g.:

  ❖ ***cin>>var1>>var2>>var3;***

  Here three different values will be entered for the three variables. The input should be separated by a space, tab or newline for each variable.

  It is equivalent to

  cin>>var1;
  cin>>var2;
  cin>>var3;

  ❖ ***cout<<var1<<", "<<var2<<" and "<<var3;***

  Here the values of the three variables will be printed where there is a "," (comma) between the first and the second variables and the "and" word between the second and the third.

Putting Comments on C++ programs

- A comment is a piece of descriptive text which explains some aspect of a program.
- Program comments are text totally ignored by the compiler and are only intended to inform the reader how the source code is working at any particular point in the program.
- C++ provides two types of comment delimiters:
  i. ***Single Line Comment***: Anything after // {double forward slash} (until the end of the line on which it appears) is considered a comment. ***Eg***:

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

  ii. ***Multiple Line Comment***: Anything enclosed by the pair /* and */ is considered a comment. ***Eg:***

  /*this is a kind of comment where
  Multiple lines can be enclosed in
  one C++ program */

- Comments should be used to enhance (not to hinder) the readability of a program. The following two points, in particular, should be noted:
  i. 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.
  ii. 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.

iii. 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.

2.5 Variables and Constants

2.5.1 Variables

- A variable can simply assumed as a name given to a certain portion of the computer's memory space by the programmer. Or we can say a variable is a name that can be given a variety of values. This memory space will be used to store values or the information which is stored at that location is known as value of the variable. And these values can change during the execution of a program. We can define variables (we name variables) anywhere in the program, but it is a must to do so before using them. A variable can hold only one value at a time.
- We could have called the variables any names we wanted to invent, as long as they were valid identifiers. A valid identifier is a sequence of one or more letters, digits or underline symbols (_). The length of an identifier is not limited, although for some compilers only the 32 first characters of an identifier are significant (the rest are not limited).
- Variables in C++ are given names by the programmer, in the way he/she thinks it is meaningful and that it reflects clearly what it represents in the program. Variable names may consist of alphabets (both upper and lower cases), the digits from 0 to 9, and the underscore (_).
- All variables have three important properties:
    - ***Data Type***: a type which is established when the variable is defined. (e.g. integer, real, character etc). Data type describes the property of the data and the size of the reserved memory. A variable can hold data in either of these types: a number or a character. The number may be an integer or a number with a decimal point (floating point), and the data size may be small or large. Data type is all about this.
    - ***Name***: a name which will be used to refer to the value in the variable. A unique identifier for the reserved memory location
    - ***Value***: a value which can be changed by assigning a new value to the variable.

Fundamental Variable types

- Several other variable types are built into C++. They can be conveniently classified as ***integer***, ***floating-point*** or ***character*** variables.
- Floating-point variable types can be expressed as fraction i.e. they are "real numbers".
- Character variables hold a single byte. They are used to hold 256 different characters and symbols of the ASCII and extended ASCII character sets.
- The types of variables used in C++ program are described in the next table, which lists the variable type, how much room.

Data Types

Type	*Length*	*Range*
unsigned char	8 bits	0 to 255
Char	8 bits	-128 to 127
Enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
Int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
Long	32 bits	-2,147,483,648 to 2,147,483,647
Float	32 bits	$-3.4\text{x}10^{-38}$ to $3.4\text{x}10^{+38}$
Double	64 bits	$-1.7\text{x}10^{-308}$ to $1.7\text{x}10^{+308}$
long double	80 bits	$-3.4\text{x}10^{-4932}$ to $1.1\text{x}10^{+4932}$
Bool	8 bits	true or false (top 7 bits are ignored)

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.

  - 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

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	→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

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

- In the above example, 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.

Declaring Variables

- Variables can be created in a process known as ***declaration***.
- Syntax: ***Datatype Variable_Name;***
- 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

- If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

  int a, b, c ;

This declares three variables (a, b, c), all of them of type int, and has exactly the same as:

```
int a;
int b;
int c;
```

Keywords are essential parts of a programming language, having a predefined meaning in the language and we cannot use them as variable names or any other purpose other than their predefined usage.

- **Reserved words** or **keywords** and are summarized in the following table.

Asm	continue	float	new	signed	try
Auto	Default	For	operator	sizeof	typedef
Break	Delete	friend	private	static	union
Case	Do	Goto	protected	struct	unsigned
Catch	Double	If	public	switch	virtual
Char	Else	inline	register	template	void

Class	Enum	Int	return	this	volatile
Const	Extern	Long	short	throw	while

➢ Identifiers

Identifiers: - name given to any programming element (like variables, functions, arrays, structures, arguments, classes, pointers etc) is known as identifiers. Identifiers are the fundamental requirement of any languages. They are named by the programmer (but by keeping the rules for giving names). A name should consist of one or more characters, each of which may be a letter (i.e., 'A'-'Z' and 'a' - 'z'), a digit i.e., '0'-'9'), or an underscore character ('_').

- A valid identifier is a sequence of one or more letters, digits or underscores symbols. The length of an identifier is not limited.
- Neither space nor marked letters can be part of an identifier.
- Only letters, digits and underscore characters are valid.
- The first character must be a letter or underscore (They can never begin with a digit).
- It should not be a **keyword**.
- It should not have **special characters** (except the underscore)

  Special characters: include:
  - ✓ Punctuation marks like **.,;,:,?,,** etc
  - ✓ Arithmetic operators like **+,-,*,/,%**
  - ✓ Others like **=,(,{,[,$,',",# ,&** etc

Note: the C $^{++}$ language is "***case sensitive*** ", that means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. E.g. the variable ***Age*** is not identical with variable ***age.***

Initializing Variables

- When declaring a variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable.

- 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.
- Initializing a variable is done by appending an equal sign followed by the value to which the variable will be initialized. The syntax is:

  DataType variable name = initial value;

```
e.g.  int a = 0;
or: int a;
   a=0;
```

Scope of Variables

- Scope of a variable is the boundary or block in a program where a variable can be accessed. The boundary or block is identified by the left and right French brackets.

- In C++, we can declare variables anywhere in the source code. But we should declare a variable before using it no matter where it is written.
- There are 2 types of variable visibilities. These are:-
  i. ***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.
  ii. ***Local Variables***: the scope of the local variable is limited to the code level or block within which they are declared.
- In the following example, the integer data type num1 is accessible everywhere whereas z and 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. E.g:

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

- 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.

Characters

- Characters variables (type char) are typically one byte in size, enough to hold 256 different values. A char can be represented as a small number (0 - 255).
- Char in C++ are represented as any value inside a ***single quote***.
  E.g.: ‘x’, ‘A’, ‘5’, ‘a’, etc.

- When the compiler finds such values (characters), it translates back the value to the ASCII values. E.g. ‘a’ has a value 97 in ASCII.

Special Printing characters

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

\n	new line
\t	tab
\b	backspace

\” double quote

\’ single quote

\? Question mark

\\ backslash

2.5.2 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 can not be changed after the declaration.
- Constants must be initialized when they are created by the program, and the programmer can’t assign a new value to a constant later.
- C++ provides two types of constants: literal and symbolic constants.

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

  E.g.: int num = 43;

  43 s a literal constant in this statement:

  ii. 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.
- The define directive can define only integer constants

  E.g.: #define studentPerClass 15
- In our example, each time the preprocessor sees the word studentPerClass, it inserts 15 into the text.

Defining constants with the const key word

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

  E.g.: const unsigned short int studentPerClass = 15;

Enumerated constants

- Used to declare multiple integer constants using a single line with different features.

- Enables programmers to define variables and restrict the value of that variable to a set of possible values which are integer.
- The enum type cannot take any other data type than integer
- enum types can be used to set up collections of named integer constants. (The keyword enum is short for "enumerated''.)
- The traditional way of doing this was something like this:

```
#define SPRING   0
#define SUMMER   1
#define FALL     2
```

- An alternate approach using enum would be
  enum SEASON{ SPRING, SUMMER, FALL, WINTER };

- You can declare COLOR to be an enumeration, and then you can define five possible values for COLOR: RED, BLUE, GREEN, WHITE and BLACK.
  E.g.: enum COLOR {RED,BLUE,GREEN,WHITE,BLACK};

- Every enumerated constant has an integer value. If the programmer does not specify otherwise, the first constant will have the value 0, and the values for the remaining constants will count up from the initial value by 1. thus in our previous example RED=0, BLUE=1, GREEN=3, WHITE=4 and BLACK=5
- But one can also assign different numbers for each.
  E.g.: enum COLOR{RED=100,BLUE,GREEN=500,WHITE,BLACK};

  Where RED will have 100 and BLUE will have 101 while GREEN will have 500, WHITE 501 and BLACK 502.

2.5 Expressions and Statements

- 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;
  The meaning is: assign the value of the sum of a and b to x.
- ***White spaces***: white spaces characters (spaces, tabs, new lines) can't be seen and generally ignored in statements. White spaces should be used to make programs more readable and easier to maintain.
- ***Blocks***: a block begins with an opening French brace ({) and ends with a closing French brace (}).
- ***Expressions***: an expression is a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value).
  E.g.: the statement 3+2; returns the value 5 and thus is an expression.

  - Some examples of an expression:
    E.g.1:

3.2 returns the value 3.2

PI float constant that returns the value 3.14 if the constant is defined.

secondsPerMinute integer constant that returns 60 if the constant is declared

E.g.2: complicated expressions:

```
x = a + b;
y = x = a + b;
```

The second line is evaluated in the following order:

1. add a to b.
2. assign the result of the expression a + b to x.
3. assign the result of the assignment expression x = a + b to y.

2.6 Operators

- An operator is a symbol that makes the machine to take an action.
- Different Operators act on one or more operands and can also have different kinds of operators.
- C++ provides several categories of operators, including the following:
  - Assignment operator
  - Arithmetic operator
  - Relational operator
  - Logical operator
  - Increment/decrement operator
  - Conditional operator
  - Comma operator
  - The size of operator
  - Explicit type casting operators, etc

Assignment operator (=)

- The assignment operator causes the operand on the left side of the assignment statement to have its value changed to the value on the right side of the statement.
- Syntax: Operand1=Operand2;
- Operand1 is always a variable
- Operand2 can be one or combination of:
  - A ***literal constan***t: 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.:

  value += *increase*; is equivalent to *value* = *value* + *increase*;

  a -= *5*; is equivalent to *a* = *a* – *5*;

  a /= *b*;is equivalent to *a* = *a* / *b*;

  price *= *units* + *1* is equivalent to *price* = *price* * (*units* + *1*);
- And the same is true for the rest.

Arithmetic operators (+, -, *, /, %)

- Except for remainder or modulo (%), all other arithmetic operators can accept a mix of integers and real operands. Generally, if both operands are integers then, the result will be an integer. However, if one or both operands are real then the result will be real.
- When both operands of the division operator (/) are integers, then the division is performed as an integer division and not the normal division we are used to.
- ***Integer division always results in an integer outcome.***
- ***Division of integer by integer will not round off to the next integer***

  E.g.:

  ```
  9/2   gives 4 not 4.5
  -9/2  gives -4 not -4.5
  ```

- To obtain a real division when both operands are integers, you should cast one of the operands to be real.

  E.g.:

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

- The ***module(%)*** is an operator that gives the remainder of a division of two integer values. For instance, 13 % 3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

  E.g.:

  ```
  a = 11 % 3
   a is 2
  ```

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

- In order to evaluate a comparison between two expressions, we can use the relational operator.
- The result of a relational operator is a bool value that can only be true or false according to the result of the comparison.

  E.g.:

  ```
  (7 = =  5)      would return false or returns 0
     (5 > 4)      would return true or returns 1
  ```

- The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For E.g.:

  ‘A’ < ‘F’   would return true or 1. it is like (65 < 70)

Logical Operators (!, &&, ||)

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

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

- The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable.

E.g.:

if a was 10 and if a++ is executed then a will automatically changed to 11.

Prefix and Postfix:

- The prefix type is written before the variable. Eg (++ myAge), whereas the postfix type appears after the variable name (myAge ++).
- Prefix and postfix operators can not be used at once on a single variable: Eg: ++age-- or --age++ or ++age++ or - - age - - is invalid
- In a simple statement, either type may be used. But in complex statements, there will be a difference.
- The prefix operator is evaluated before the assignment, and the postfix operator is evaluated after the assignment.

E.g.

```
int k = 5;
(auto increment prefix)     y= ++k + 10;  //gives 16 for y
(auto increment postfix)    y= k++ + 10;  //gives 15 for y
(auto decrement prefix)     y= --k + 10;   //gives 14 for y
(auto decrement postfix)    y= k-- + 10;   //gives 15 for y
```

Conditional Operator (?:)

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

Syntax:

operand1 ? operand2 : operand3

- First operand1 is a relational expression and will be evaluated. If the result of the evaluation is non zero (which means TRUE), then operand2 will be the final result. Otherwise, operand3 is the final result.

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.:

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

    The value stored in min is 1.
```

E.g.:

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

Comma Operator (,)

- Multiple expressions can be combined into one expression using the comma operator.
- The comma operator takes two operands. Operand1,Operand2
- The comma operator can be used during multiple declaration, for the condition operator and for function declaration, etc
- It first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.
  E.g.

```
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.

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
```

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

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	Order
Highest	sizeof()++ -- (pre fix)	Right to left
	* / %	Left to right
	+ -	Left to right
	< <= > >=	Left to right
	== !=	Left to right
	&&	Left to right
	||	Left to right
	? :	Left to right
	= ,+=, -=, *=, /=,^= ,%=, &= ,|= ,<<= ,>>=	Right to left
	++ -- (postfix)	Right to left
	,	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.

2.8 Debugging and programming errors

When we attempt to produce an efficient program we should take sufficient care to maintain clarity and readability of a program. The program errors are called ***bugs.*** The art of locating and eliminating bugs or errors is called ***debugging.***

There are actually three different kind of errors:

- compile-time errors: identified by the compiler, includes
  - missing brackets
  - undeclared variables
    (e.g. misspelling variable name in one place)
  - incorrect use of single or double quotes
  - invalid identifier names
- run-time errors: attempts by the program to do something illegal while executing, can include
  - trying to divide by zero (will crash)
  - using or printing a variable value before initializing it (usually will not crash, just give very weird results)
- logic errors: the implementation doesn't correctly solve the problem, includes any form of miscalculation or incorrect ordering of instructions

Worksheet 2

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.

6) Receive a number and determine whether it is odd or even.
7) Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
8) Receive 3 numbers and display them in ascending order from smallest to largest
9) Add the numbers from 1 to 100 and display the sum
10) Add the even numbers between 0 and any positive integer number given by the user.
11) Find the average of two numbers given by the user.
12) Find the average, maximum, minimum, and sum of three numbers given by the user.
13) Find the area of a circle where the radius is provided by the user.
14) Swap the contents of two variables using a third variable.
15) Swap the content of two variables without using a third variable.

16) Read an integer value from the keyboard and display a message indicating if this number is odd or even.
17) 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
18) Take an integer from the user and display the factorial of that number

Spotting errors

For each of the following programs, spot the flaws:

Program 1

```
/ this program will calculate the area of a square

#include <iostream.h>
 void main()
{
   int area;   / store calculated area of square
   int side;   / input length of side of square
    cout << "How wide is the square?" << endl;
   cin >> side;
    area = side * side;
    cout << "The square has area area" << endl;
}
```

Program 2

```
// this program prints information about the author
void main()
{
   cout << 'This program was written by Dave Wessels';
   cout << 'He didn't do a great job of it, ' << endl;
   cout << 'It won't even compile';
}
// this program calculates user age

#include <iostream.h>
int main()
  int currentyear;
  int birthyear;
   cout << "Enter the year you were born" << endl;
  cin >> birthyear;
   cout << "Enter the current year" << endl;
  cin >> currentyear;
   age = currentyear - birthyear;
   cout << "This year you will be" << age << endl;
}
```

Program 3

```
// given the radius of a circle,
// this program prints its circumference,
// and prints its area to three decimal places

#include <iostream.h>
#include <iomanip.h>

const int Pi = 3.1415

void mian()
(
  int radius;        // input circle radius
  int circumference; // calculated circumference
  float area;        // calculated area
   cout << "Enter the integer radius of the circle";
  cout << endl;
  cin >> radius;
   circumference = 2 * radius * Pi;
  area = Pi * raduis * radius;

  cout << "The circumference is " << circumference;
  cout << ", the area is ";
  cout << setprecision(2) << area << endl;
 )
```

Chapter Three
3. Flow of Control

- A running program spends all of its time executing instructions or statements in that program.
- The order in which statements in a program are executed is called ***flow*** of that program.
- Programmers can control which instruction to be executed in a program, which is called ***flow control***.
- This term reflects the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement.
- Flow control in a program is typically ***sequential***, from one statement to the next.
- But we can also have execution that might be divided to other paths by ***branching statements***. Or perform a block of statement repeatedly until a condition fails by ***Repetition*** or ***looping***.
- Depending upon requirement of the problem, it is often required to alter the normal sequence of execution in the program. This means that we may desire to selectively or repetitively execute a program statement.
- 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.

Sequential Statements

- Such kinds of statements are instruction in a program which will be executed one after the other in the order scripted in the program. In sequential statements, the order will be determined during program development and cannot be changed.
- A number of C^{++} structure known as control structures are available for the flow of processing.

3.1 Selection Statements/Conditional 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.
- They are mainly used for making decisions. Depending on the value of an expression decision can be made.
- There are two types of selection statements in C++, which are the "***if statement***" and the "***switch statement***"

3.1.1 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" statement are:
  - ✓ The simple if statement
  - ✓ The if else statement
  - ✓ The if else if statement

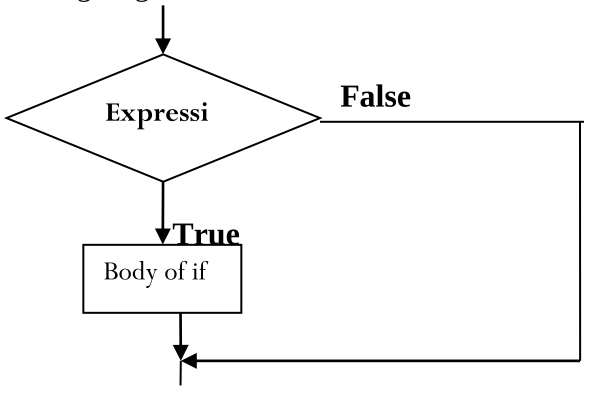
The simple if statement

- 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.

- 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.
- The following diagram shows the execution of if statement.



```
E.g.   if(age>18)
             cout<<"you are an adult";
E.g.:
             if(balance > 0)
                  {
                   interest = balance * creditRate;
                   balance += interest;
                  }
```

- 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.

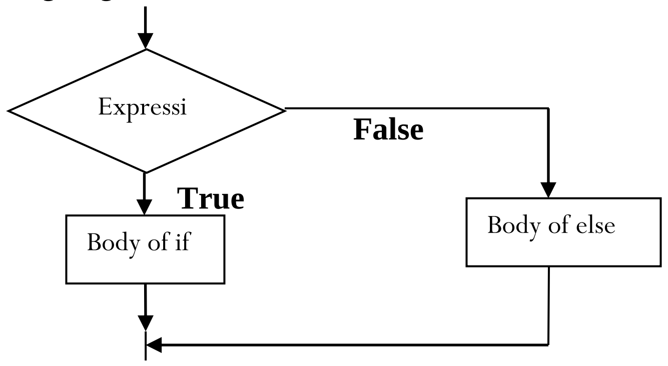
The if else statement

- Another form of the "if" is the "if …else" statement.
- The "if else" 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.
- The following diagram shows the execution of if else statement.



E.g.:
```
if(balance > 0)
        {
            interest = balance * creditRate;
            balance += interest;
          }
        else
          {
            interest = balance * debitRate;
            balance += interest;
          }
```

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 one 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.
- 'if' statements may be nested by having an if statement appear inside another if statement. For instance:

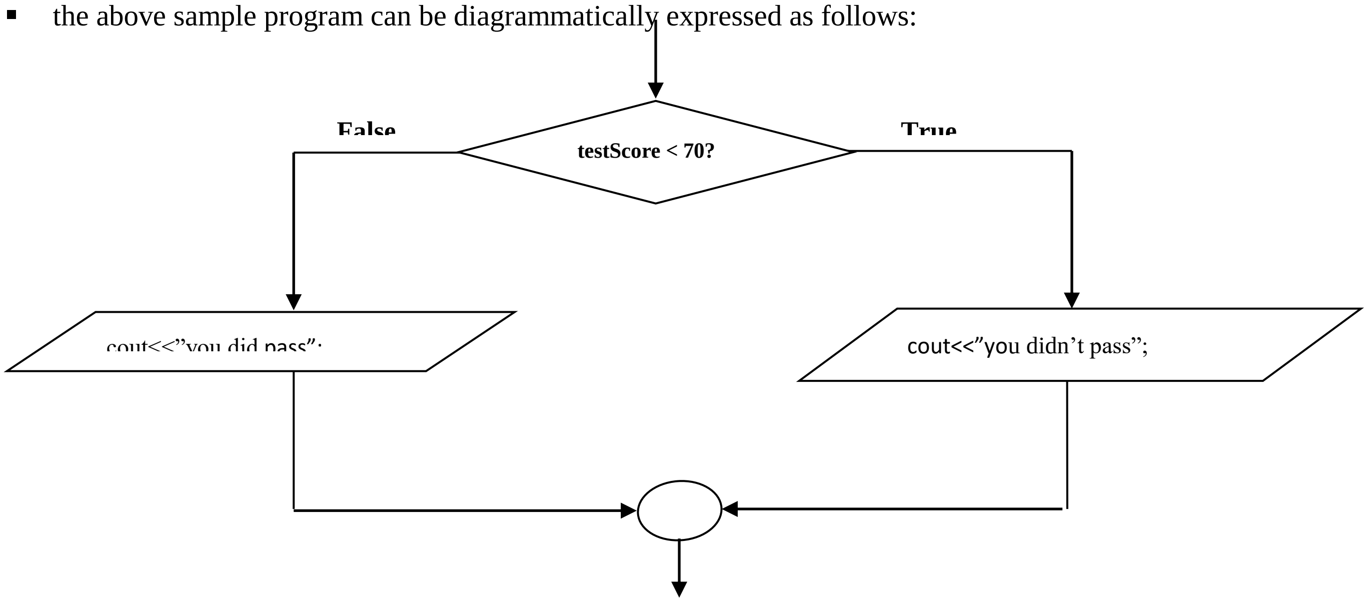
```
if(callHour > 3)
{
        if(callDuration <= 5)
                charge = callDuration * tarif1;
```

```
            else
                    charge = 5 * tarif1 + (callDuration - 5) * tarif2;
    }
    else
            charge = flatFee;
```

E.g:

```
#include<iostream.h>
#include<conio.h>
int main()
{
        int testScore;
        cout<<"\nEnter your test score:";
        cin>>testScore;

        if(testScore < 70)
                cout<<"\n You didn't pass:"; // then block
        else
                cout<<"\n You did pass"; //else block
        return 0;
}
```

- the above sample program can be diagrammatically expressed as follows:



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 two alternative 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)
    statementsN;
```

- 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 out put 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 False then "*statements"* under else will be executed.
- The "if…else" and "if…else if" statements are said to be exclusive selection 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 the above example, only one of the five cout statements will be executed 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 with in another if statement.
- The nesting will be used to test multiple conditions to perform a task.
- It is always recommended to indent nested if statements to enhance readability of a program
- The General Syntax might be:

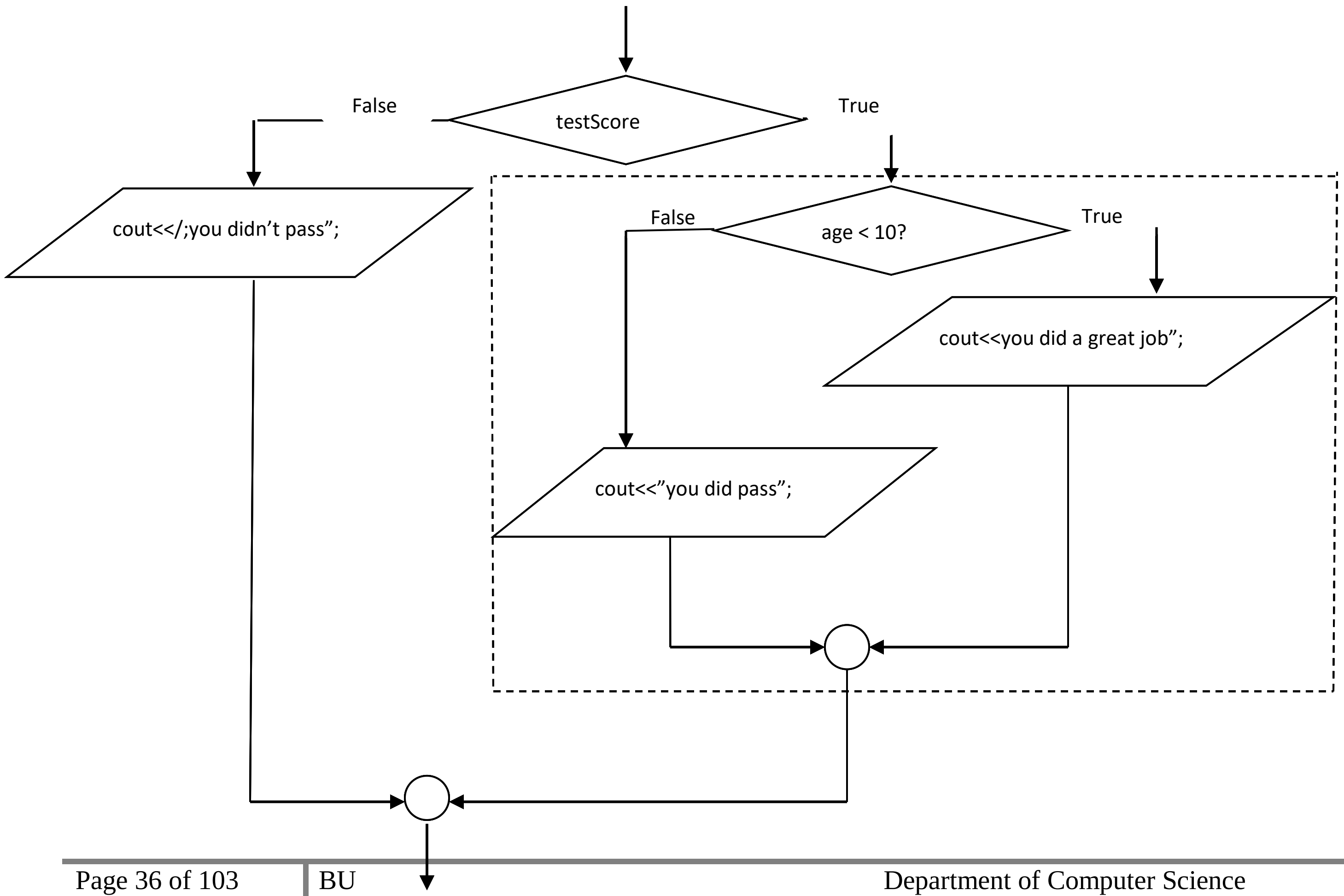
```
if (expression1)
   {
     if (expression2)
          statementsN;
    else
       statementsM;
   }
else
   {
     if (expression3)
         statementsR;
     else
        statementsT;
   }
```

- 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 TRUE and "*expression2*" is FALSE.
- StatementsR will be executed if and only if "*expression1*" is FALSE and "*expression3*" is TRUE.

- StatementsT will be executed if and only if "*expression1*" is FLASE and "*expression2*" is FALSE.
- Lets have an example on nested if statements

```
#...
int  main()
{
        int testScore, 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<<"\n You did pass";
        }
        else
                cout<<"\n You did not pass";
        getch();
        return 0;
}
```

- The above program can be expressed diagrammatically as follows.



3.1.2 The Switch Statement

- Another C++ statement that implements a selection control flow is the switch statement (*multiple-choice statement*).
- The switch statement provides a way of choosing between a set of alternatives based on the value of an expression. The general form of the switch statement is:
- 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;
                  .
                  .
          case constant n:
                  statements;
          default:
                  statements;
    }
```

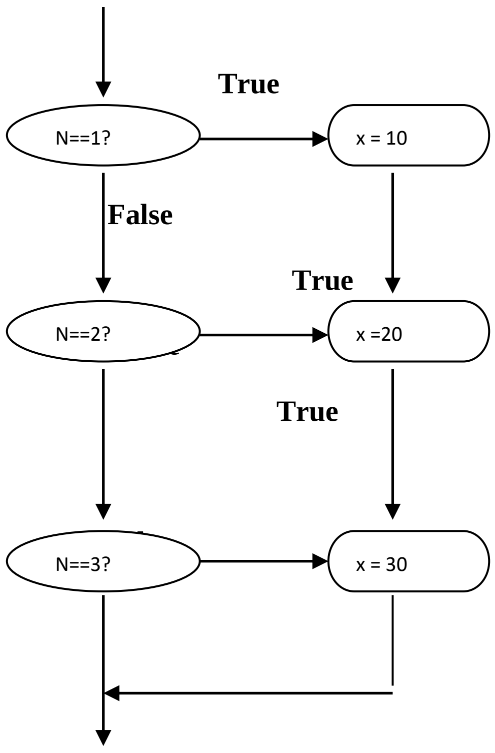
- Expression is called the ***switch tag*** and the constants preceding each case are called the ***case labels***.
- The output of "*expression*" should always be a constant value.
- First expression is evaluated, and the outcome, which is a constant value, will compared to each of the numeric constants in the case labels, in the order they appear, until a match is found.
- Note, however, that the evaluation of the ***switch tag*** with the ***case labels*** is only for ***equality.***
- 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. This means that, if the value of the "*expression*" is not equal to any of the case labels, then the statements under ***default*** will be executed.
- Now let us see the effect of including a break statement in the switch statement.

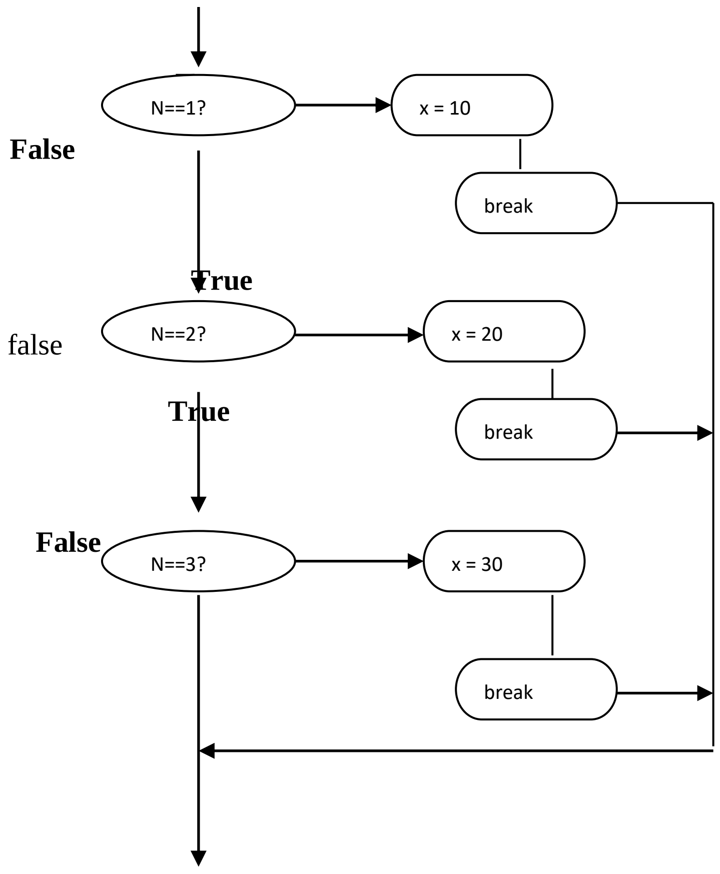
Scenario One

```
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 10, 20 or 30 based on the value of N
```





- The break terminates the switch statement by jumping to the very 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;
                                  break;
                    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.

3.2 Repetition/Loop Statements

- Repetition statements control a block of code to be executed repeatedly for a fixed number of times or until a certain condition fails.
- Loop causes a section of our program to be repeated a certain number of times. The repetition continues up to when a condition is true.
- There are three C++ repetition statements:
  1) The ***For*** Statement or loop
  2) The ***While*** statement or loop
  3) The ***do…while*** statement or loop

3.2.1 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:
- ***expression1***: is one or more statements 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 non zero (to proceed with the loop) or zero (to stop the loop) to represent true and false output respectively.
- ***Expression3***: is one or more statements that will be executed after each iteration.
- Thus, first ***expression1*** is evaluated and then each time the loop is executed, ***expression2*** is evaluated. If the outcome of ***expression2*** is non zero then ***statements*** is executed and ***expression3*** is evaluated. Otherwise, the loop is terminated.
- 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)
statement;

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

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

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

```
                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 colons 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++;
             }
       }
```

- 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.

3.2.2 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.
- 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;
  }
```

3.2.3 Do…while loop

- The do statement (also called the do while loop) is similar to the while statement, except that its body is executed first and then the loop condition is examined.
- In do…while loop, we are sure that the body of the loop will be executed at least once. Then the condition will be tested.
- The general form is:

```
do
  {
       statement;
  }

while(expression);
```

- First statement is executed and then expression is evaluated. If the outcome of the expression is nonzero, then the whole process is repeated. Otherwise the loop is terminated.
  E.g.:
  //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.g. 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);
```

<u>Pitfalls in writing repetition statements</u>

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

❖ ***<u>Infinite loop</u>***: 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++;
}
```

3.3 The continue and break statements

3.3.1 The 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 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.

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++
      }
  }
```

3.3.2 The break statement

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

E.g.:

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

Worksheet 3

Make sure to use looping whenever applicable.

1. Write for, do-while, and while statements to compute the following sums and products.
   a. 1+2+3+…+100
   b. 5+10+15+…+50
   c. 1+1/2+1/3+1/4+…1/15
   d. 1*2*3*…*20
2. write an application to print out the numbers 10 through 49 in the following manner

```
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
```

```
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
```

3. 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.
4. Write a C++ program that counts the number of digits in an integer number. For example; 23,498 has five digits.
5. 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.
6. 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.
7. Write a C++ code that computes the sum of the following series.

   Sum = 1! + 2! + 3! + 4! + …n!

   The program should accept the number from the user.

8. 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
```

9. Write a C++ program that displays the following output using their ASCII values.

```
a
bc
def
gehi
jklmn
opqrst
```

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

```
A.           B.           C.              D.
*            *****            *             *
**           ****            ***           ***
***          ***            *****         *****
****         **            *******         ***
*****        *            *********         *
```

11. Write a weather-calculator program that asks for a list of the previous 10 days' temperatures, computes the average, and prints the results. You have to compute the total as the input occurs, then divide that total by 10 to find the average. Use a while loop for the 10 repetitions.
12. Write a C++ program that accepts marks of five students and then displays their average. The program should not accept mark which is less than 0 and mark greater than 100.
13. Develop a calculator program that computes and displays the result of a single requested operation.

    E.g. if the input is

    15 * 20, then the program should display 15 * 20 equals 300

    If the operator is not legal, as in the following example

    24 ~ 25 then the program displays ~ is unrecognized operator

    As a final example, if the denominator for a division is 0, as in the following input: 23 / 0 then the program should display the following:

    23 / 0 can't be computed: denominator is 0.
14. 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.
15. Write a C++ code to display only even numbers found between 0 and 20.
16. Write a C++ application that extracts a day, month and year and determine whether the date is valid. If the program is given a valid date, an appropriate message is displayed. If instead the program is given an invalid date, an explanatory message is given. Note: to recognize whether the date is valid, we must be able to determine whether the year is a leap year or not.

    An example of the expected input/output behavior for a valid date follows

    ```
    Please enter a date (dd mm yyyy) : 30 4 2006
    30/4/2006 is a valid date
    Please enter a date (dd mm yyyy) : 1 13 2006
    Invalid month: 13
    Please enter a date (dd mm yyyy) : 29 2 1899
    Invalid day of month 29
    ```

If the year is a leap year, then February will have total of 29 days. Otherwise, it will have 28 days. If the year is not a century year and is evenly divisible by 4, then the year is a leap year. If the year is a century year (years whose last digits are 00) and is evenly divisible by 400, then the year is a leap year.

Chapter Four
4. Arrays and Strings

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.
- ✓ Set of values of the same type, which have a single name followed by an index. In C++, square brackets appear around the index right after the name.
- ✓ 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.

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.

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:
  1. The type of the array (i.e. integer, floating point, char etc.)
  2. Name of the array,
  3. 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:

  DataTypename 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.

  <u>Note</u>: array size cannot be a variable whose value is set while the program is running.
- ✓ Thus to declare an integer with size of 10 having a name of num is:

  int num [10];

  This means: ten consecutive two byte memory location will be reserved with the name num.
- ✓ That means, 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.

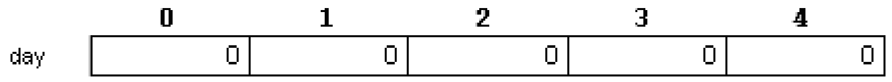
Initializing Arrays

- ✓ When declaring an array of local scope (within a function), if we do not specify the array variable will not be initialized, 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];

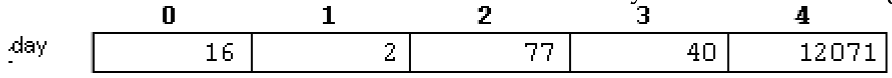
  - ✓ every element of day will be set initially to **0:**



  - ✓ 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.

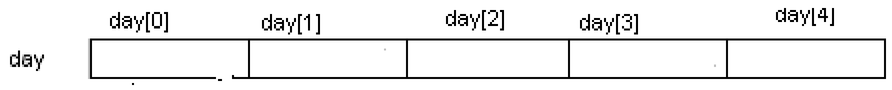
Accessing and processing array elements

- ✓ 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 the two 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:

Eg: 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;
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(); }
```

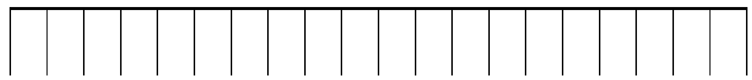
Strings of Characters:

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.
- ✓ In C++ there is no specific elementary variable type to store string of characters. In order to fulfill this feature we can use arrays of type **char**, which are successions of char elements. Remember that this data type (**char***)* is the one used to store a single character, for that reason arrays of them are generally used to make strings of single characters.
- ✓ For example, the following array (or string of characters) can store a string up to 20 characters long. You may imagine it thus:

  char name [20];

  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	0	\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.

Initialization of Strings

- ✓ Because 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'.**
- ✓ Nevertheless, string of characters have an additional way to initialize its values: using constant strings.
- ✓ In the expressions we have used in examples of previous chapters there have already appeared several times constants that represented entire strings of characters. These are specified enclosed between double quotes (" "), for example:

  Eg: "the result is: "
  is a constant string that we have probably used in some occasion.

- ✓ 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.

Assigning Values to Strings

- ✓ Just like any other variables, array of character can store values using assignment operators. But the following is not allowed.
  mystring="Hello";
- ✓ This is allowed only during initialization. Therefore, since the *lvalue* of an assignation can only be an element of an array and not the entire array, what would be valid is to assign a string of characters to an array of **char** using a method like this:
  mystring[0] = 'H';
  mystring[1] = 'e';
  mystring[2] = 'l';
  mystring[3] = 'l';
  mystring[4] = 'o';
  mystring[5] = '\0';
- ✓ But as you may think, this does not seem to be a very practical method. Generally for assigning values to an array, and more specifically to a string of characters, a series of functions like **strcpy** are used. **strcpy** (**str** ing **c** o **py**) is defined in the (string.h) library and can be called the following way:
  strcpy (*string1* **,** *string2* **);**
- ✓ This does copy the content of *string2* into *string1* . *string2* can be either an array, a pointer, or a constant string , so the following line would be a valid way to assign the constant string **"Hello"** to **mystring** :
  strcpy (mystring, "Hello");

  For example:

```
#include <iostream.h>
#include <string.h>
int main ()
{
char szMyName [20];
strcpy (szMyName,"Abebe");
cout << szMyName;
return 0;
}
```

- ✓ Look how we have needed to include **<string.h>** header in order to be able to use function **strcpy.**
- ✓ Although we can always write a simple function like the following **setstring** with the same operating than cstring's **strcpy** :

```
// setting value to string
#include <iostream.h>
#include<conio.h>
void namecopy(char dest[], char source[])
{
   int c = 0;
   while(source[c] != '\0')

    {
          dest[c] = source[c];
          c++;
    }
     dest[c] = '\0';
```

```
        cout<< “\n your name after copying : ”<<dest;
    }

    void main()
    {     clrscr();
        char name[10],dest[10];
        cout<< “\n enter your name : ”;
        cin>>name;
        namecopy(dest,name);
        getch();
    }
```

- ✓ Another frequently used method to assign values to an array is by using directly the input stream (**cin)**. In this case the value of the string is assigned by the user during program execution.
- ✓ When **cin** is used with strings of characters it is usually used with its **getline** method, that can be called following this prototype:
- ✓ **cin.getline (char** *buffer* **[], int** *length* **, char** *delimiter* **= ' \n');**
- ✓ where ***buffer*** is the address where to store the input (like an array, for example), ***length*** is the maximum length of the buffer (the size of the array) and ***delimiter*** 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 following example repeats whatever you type on your keyboard. It is quite simple but serves as example on how you can use **cin.getline** with strings:

```
    // cin with strings
    #include <iostream.h>
    #include<conio.h>
    int main ()
    {    char mybuffer [100];
        cout << "What's your name? ";
        cin.getline (mybuffer,100);
        cout << "Hello " << mybuffer << ".\n";
        cout << "Which is your favourite team? ";
        cin.getline (mybuffer,100);
        cout << "I like " << mybuffer << " too.\n";
        getch();
        return 0;
     }
```

- ✓ Notice how in both calls to **cin.getline** we used the same string identifier (mybuffer**)**. What the program does in the second call is simply step on the previous content of **buffer** by the new one that is introduced.
- ✓ If you remember the section about communication through console, you will remember that we used the extraction operator (**>>**) to receive data directly from the standard input. This method can also be used instead of **cin.getline** with strings of characters. For example, in our program, when we requested an input from the user we could have written:

```
    cin >> mybuffer;
```

- ✓ this would work, but this method has the following limitations that **cin.getline** has not:
  - It can only receive single words (no complete sentences) since this method uses as delimiter any occurrence of a blank character, including spaces, tabulators, newlines and carriage returns.
  - It is not allowed to specify a size for the buffer. What makes your program unstable in case that the user input is longer than the array that will host it.
- ✓ For these reasons it is recommendable that whenever you require strings of characters coming from **cin** you use **cin.getline** instead of **cin >>** .

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:

```
// 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;
}
```

Functions to manipulate strings

- ✓ The **cstring** library (string.h) defines many functions to perform some manipulation operations with C-like strings (like already explained strcpy). 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* **);**
  - **b) 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.
      - o Appending the whole content of the source

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

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

        Where size is the number characters to be appended
  - **c) String Copy:**
    - ➢ Overwrites the content of the *dest* string by the *src* string. Returns *dest*.
    - ➢ 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.
      - o Copy the whole content of the source

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

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

        Where size is the number characters to be copied
  - **d) String Compare:**
    - ➢ 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.
      - o Compare the whole content of *string1* and *string2*

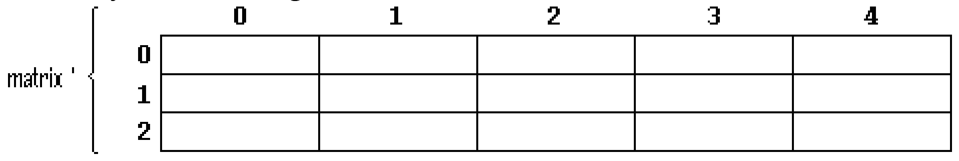
        strcmp (const char* *string1* **, const char*** *string2* **);**
      - o Compare part of *string1* and *string2*

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

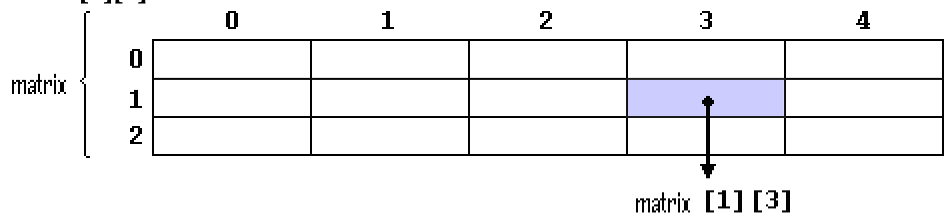
        Where size is the number of characters to be compared
    - ➢ Both string compare functions returns three different values:
      - o Returns ***0*** is the strings are equal
      - o Returns ***negative*** value if the first is less than the second string
      - o Returns ***positive*** value if the first is greater than the second string

<u>*Multidimensional Arrays*</u>

- ✓ Multidimensional arrays can be described as arrays of arrays. For example, a bi-dimensional array can be imagined as a bi-dimensional table of a uniform concrete data *type.*



- ✓ Matrix represents a bi-dimensional array of 3 per 5 values of type **int** . The way to declare this array would be:
  int matrix[3][5];
- ✓ For example, the way to reference the second element vertically and fourth horizontally in an expression would be:
  matrix[1][3]

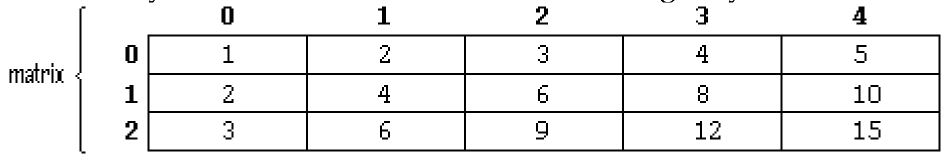


(remember that array indices always begin by **0**)

- ✓ *Multidimensional arrays* are not limited to two indices (two dimensions). They can contain so many indices as needed, although it is rare to have to represent more than 3 dimensions. Just consider the amount of memory that an array with many indices may need. For example:
  char century [100][365][24][60][60];
- ✓ Assigns a **char** for each second contained in a century, that is more than 3 billion **chars!** What would consume about 3000 *megabytes* of RAM memory if we could declare it?
- ✓ 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 the compiler remembers for us the depth of each imaginary dimension. Serve as example these two pieces of code, with exactly the same result, one using bi-dimensional arrays and the other using only simple arrays:

```
// 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;
}
```

- ✓ 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
  by the following code

#define HEIGHT 4

Worksheet No 4:

1. Define strlen function (i.e write the function body of strlen)
2. Define the strcmp function and the strncmp function
3. Define strcpy function and the strncpy function
4. Define strcat function and the strncat function
5. 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
6. Write a C++ program that accepts 10 integers from the user and finally displays the smallest value and the largest value.
7. Write a program that accepts ten different integers from the user and display these numbers after sorting them in increasing order.
8. 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
9. Modify the program on Q8 to print the ages in descending order.
10. 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.
11. 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.
12. Develop a C++ program that accepts a word from the user and then checks whether the word is palindrome or not. (NB a word is palindrome if it is readable from left to right as well as right to left).
13. Write a C++ program that accepts a word from the user and then displays the word after reversing it.
14. Develop a C++ program that accepts the name of a person and then counts how many vowels the person's name have.
15. Modify the question in Q14 in such a way that it should replace vowel characters with * in the person name.
16. 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 – 123, 213, 312, 132, 231, 321
17. Write a program which read a set of lines until you enter #.
18. Write a program which read two matrixes and then print a matrix which is addition of these two matrixes.
19. Write a program which reads two matrix and multiply them if possible
20. 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.

Chapter Five
Functions

5.1 Introduction

A function is defined as a group of instruction used to achieve a specific task. These functions can be called from other function. Hence if any groups of statements are to be frequently used then the statements are grouped together and named which is accessed by that new function when needed.

Functions decrease the complexity of a program. A program having multiple functions can be easily debugged than a program that is large and runs without a single function. So a function reduces the size of a program and increases the program modularity.

C++ adopts functions as its fundamental programming approach .Basics of C++ deals more with functions and even classes also use the functions as their member functions.

Advantages of functions

- ✓ Increase program modularity
- ✓ Reduction in the amount of work & development time
- ✓ Program and function debugging is easier
- ✓ Division of work is simplified
- ✓ Reduction in the size of the program due to code reusability
- ✓ They can be accessed repeatedly without redevelopment, which in turn promotes reuse of code

5.2 Types of function

Functions are of two types:

- Library/built-in function
- User defined function

5.2.1 Library/Built-in function

Library functions are pre-written functions or built in functions. Many activities in C++ use library functions. These functions include graphical functions, text formatting functions, memory management and data conversion.

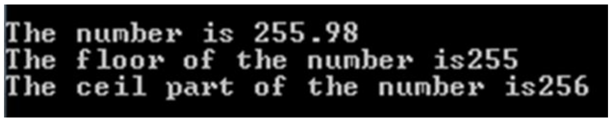
Some of the mathematical functions available in the C++ mathematics library are listed below.

acos(x)	inverse cosine, -1 <= x <= +1, returns value in radians in range 0 to PI
asin(x)	inverse sine, -1 <= x <= +1, returns value in radians in range 0 to PI
atan(x)	inverse tangent, returns value in radians in range -PI/2 to PI/2
cos(x)	returns cosine of x, x in radians
sin(x)	returns sine of x, x in radians
exp(x)	exponential function, e to power x
log(x)	natural log of x (base e), x > 0
sqrt(x)	square root of x, x >= 0
fabs(x)	absolute value of x
pow(x,y)	x the power of y
floor(x)	largest integer not greater than x
ceil(x)	smallest integer not less than x

Consider the following example which uses ceil and floor library functions from math.h. The floor function rounds up the float value to the nearest lower integer where as ceil rounds up the float value to the nearest upper integer.

Example:

Output



```
The number is 255.98
The floor of the number is255
The ceil part of the number is256
```

```
#include<math.h>
#include<iostream.h>
int main()
{
float number=255.98;
float down,up;
down=floor(number);
up=ceil(number);
cout<<"The number is "<<number<<endl;
cout<<"The floor of the number is"<<down<<endl;
cout<<"The ceil part of the number is"<<up<<endl;
return 0;
}
```

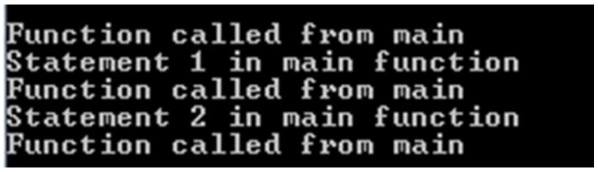
5.2.2 User defined functions

Besides the library functions, we can also define our own functions which are called as user defined functions. C++ allows declaring user-defined functions, which reduces program size and increase modularity.

The following program shows how to use user-defined functions. Example:

```
#include<iostream.h>
void func();//Function declaration
void main()
{
func();//Function calling
cout<<"Statement 1 in main function"<<endl;
func();//Function calling
cout<<"Statement 2 in main function"<<endl;
func();//function can be called from main any number of times
}
void func()
{
cout<"Function called from main"<<endl;//function definition
}
```

Output



```
Function called from main
Statement 1 in main function
Function called from main
Statement 2 in main function
Function called from main
```

Just like a variable has to be declared before using, a function also has to be declared.

The statement just above the main is the ***function declaration***. A ***function declaration*** tells the compiler that at some later point we are going to use a function by that name. It is called as prototype declaration. It should be terminated because it acts like a single statement.

Summarized function basics.

- C++ functions generally adhere to the following rules.
  1. Every function must have a ***name***.
  2. Function names are made up and assigned by the programmer following the same rules that apply to naming variables. They can contain up to 32 characters depending on the compiler, they must begin with a letter, and they can consist of letters, numbers, and the underscore (_) character.
  3. All function names have ***one set of parenthesis*** immediately following them. This helps you (and C++ compiler) differentiate them from variables.
  4. The body of each function, starting immediately after parenthesis of the function name, must be enclosed by braces.

<u>Declaring function:</u>

- 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.

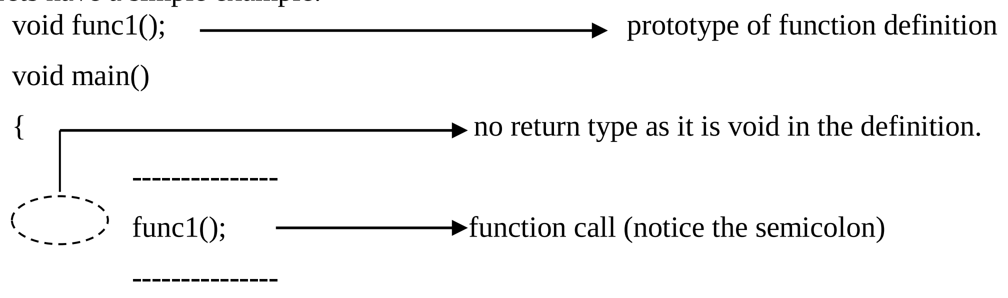
<u>Defining a function:</u>

- A function definition consists of two parts: interface (***prototype***) & ***body***. The brace of a function contains the computational steps (statements) that computerize the function. The definition consists of a line called the decelerator.
- 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.

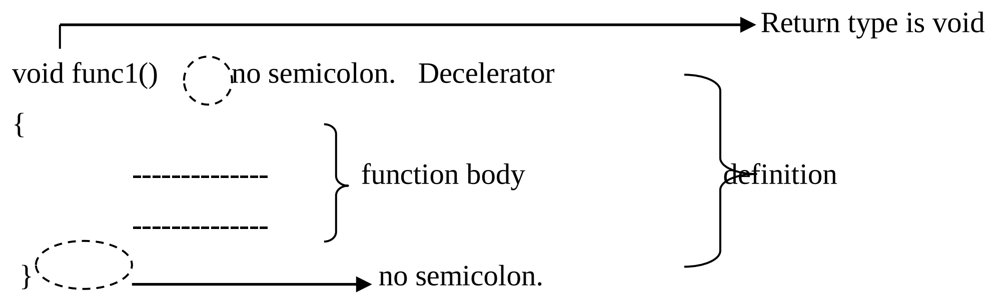
<u>Calling the function:</u>

- Using a function involves 'calling' it.
- 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.

- 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.
- Lets have a simple example:



```
void func1();          → prototype of function definition
void main()
{                      → no return type as it is void in the definition.
        ---------------
        func1();       → function call (notice the semicolon)
        ---------------
}
```



➢ Lets have another very simple function example:

```
#include<iostream.h>
#include<conio.h>
void starLine(); //→prototype of the function with a name starLine
int main()
 {
   starLine(); //→Calling the function with a name starLine
  cout<< "Data type Range" << endl;
 starLine();
 cout<< "char        -128 to 127" <<endl
      << "short      -32,768  to 32,767" <<endl
      << " int    system dependent" <<endl
      << "long    -2,147,483,648  to 2,147,483,647" << endl;
starLine();
return 0;
 }
     void starLine()
     {
                    for(int j=0;j<45;j++)
                           cout<< "*";
                    cout<<endl;
     }
```

definition of the function with a name starLine

➢ Given the next program, which function is the calling function and which is the called function?

```
#include<iostream.h>
#include<conio.h>
void nextMsg()
int main()
{
  cout<< "Hello!\n";
  nextMsg();
  return 0;
}
```

```
void nextMsg()
{
  cout<< "GoodBye!\n";
  return;
}
```

<u>Function Return</u>

- Functions can be grouped into two categories: function that do not have a return value (void function and functions that have a return value)
- The return statement in a function need not at the end of the function. It can occur anywhere in the function body and as soon as it as countered, execution control will be returned to the caller.

```
Caller                  called

void main ( )         int f(x,y)
{                         {
---------                 -----------
---------                 -----------
---------                 -----------
f(a,b);                   return y;
}                         }
```

- in void functions the use of return state is optional.

5.3. Function Parameters and arguments

- ➢ The parameters of a function are list of variables used by the function to perform its task and the arguments passed to the function during calling of a function are values sent to the function.
- ➢ The arguments of function calling can be using either of the two supported styles in C++: ***calling/passing by value*** or ***calling/passing by reference***.

5.3.1. Function call by value

A function can also have arguments. An argument is a piece of data passed from a program to a function. The arguments are the variables that are passed in the parenthesis of a function. These arguments are used to process the data sent by the program.

Syntax:

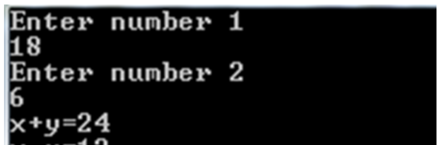
void myfunc(int,int);

This specifies the function called **myfunc** that takes two integer variables as arguments. A compiler identifies a function with the number of arguments and type of arguments other-wise which raises a compiler error. When we are declaring functions, the variables in function declaration are known as **parameters**. During function execution they are known as ***function arguments.***

Example:

```
#include<iostream.h>

void disp(int,int) ;//declares prototype of the function disp

void main()

{

int x,y;//declares two integer variables
```

<u>Output</u>



```
Enter number 1
18
Enter number 2
6
x+y=24
```

```
cout<<"Enter number 1"<<endl;

cin>>x;

cout<<"Enter number 2"<<endl;

cin>>y;

disp(x,y);//function calling by passing value for a and b

}

void disp(int a,int b)
{
cout<<"x+y="<<a+b<<endl;
cout<<"x-y="<<a-b<<endl;
cout<<"x*y="<<a*b<<endl;
cout<<"x/y="<<a/b<<endl;
}
```

The program accepts two integer variables and passes them as **arguments** to the function **disp.** The disp function performs all the arithmetic operations on them and displays the result.

Sometimes it becomes necessary that a program passing arguments expecting a return from the function. This is known as ***arguments with return value.*** Actually when you declare a function as **void** it implies that the function need not return anything.

This means that ***if you don't specify void before the function name, the calling function is expecting a return from the called function.***

- Default return type is integer.

The return may be an int variable or a char variable or any other variable of any type. In this case a function that returns a value should be specified before the function at the time of declaration and definition.

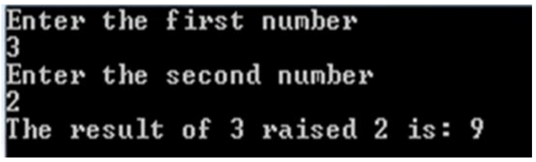
Consider the following program

Example:

The program defines a function called power, which takes two integer variables and prints the power of the first number raised to second number. Finally after calculating power the value is returned to the calling function. You will observe that in place of void before the function name it is given as int so that it will return a value of int.

```
#include<iostream.h>
int power(int,int);//function declaration with return value
void main()
{
int a,b,c;
cout<<"Enter the first number"<<endl;
cin>>a;
cout<<"Enter the second number"<<endl;
cin>>b;
c=power(a,b);//assigning the returned value of the  function power
cout<<"The result of  "<<a<<"  raised  "<<b<<"  is: "<<c;
}//definition of function power
int power(int i,int j)
{
int p=1;
while(j>0)
{
p=p*i;
j--;
}
return p;//p is returned to initialize to c
}
```

Output



```
Enter the first number
3
Enter the second number
2
The result of 3 raised 2 is: 9
```

From the above program it is understood that, to make a function to return a value, the type has to be specified before the function name.

5.3.2. Function call by reference

Call by value can return the result of a calculation. But it cannot affect the original variables. That is, in call by value the variables that are passed with as arguments from the caller function gets copied into the variables that are defined with the function definition. So the copies of the original variables are created at another memory location. Thus the changes to these variables are not affected to original variables.

Sometimes it is necessary to change the values of original variables that have been passed in the caller function. We can achieve this using ***call by reference.***

In ***call by reference***, the address of the variables passes as arguments to the function. Thus the variables which are defined along with the function definition can actually access the values of the variables that are being passed to the function to the caller function. And hence the alteration for the original variables is possible.

Example:

The program follows call by reference method to swap two variables.

```
#include<iostream.h>
int swap(int &,int &);//& indicates you are passing the addresses of the values not the values
void main()
{
int a=10;

int b=90;

swap(a,b);//values of a and b are passed to the function

cout<<"The values of a is:"<<a<<endl;

cout<<"The value of b is: <<b<<endl;

}
int swap(int &p,int &q)
{

int temp;
temp=p;
p=q;
q=temp;
return 0;
}
```

Output

```
The value of a is: 90

The value of b is: 10
```

In the example shown above you passed the address of the variables as argument. You will find that the declaration of swap included an **&** specified in place of arguments. It means that the function is expecting addresses of the variables as arguments.

& is called "address of" operator. It is used to pass the address of a variable to the called function or variable. So the variable declared in definition of function also gets created at the same address. It implies that the variables can access the value of original variables. Thus they can be altered. So call by reference works fine when it is needed to access the original variables not created a copy of it.

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

5.4. Global versus local variables

- ➢ Everything defined at the program scope level (outside functions) is said to have a *global scope*, meaning that the entire program knows each variable and has the capability to change any of them.

  Eg.

```
int year = 1994;//global variable
 int max(int,int);//gloabal funcrion
 int main(void)
 {
        //...
 }
```

- ➢ Global variables are visible ("known") from their point of definition down to the end of the program.
- ➢ Each block in a program defines a local scope. Thus the body of a function represents a local scope. The parameters of a function have the same scope as the function body.
- ➢ Variables defined within a local scope are visible to that scope only. Hence, a variable need only be unique within its own scope. Local scopes may be nested, in which case the inner scope overrides the outer scopes. Eg:

```
int xyz;//xyz is global
```

```
void Foo(int xyz)//xyz is local to the body of Foo
{
        if(xyz > 0)
        {
                Double xyz;//xyz is local to this block
                …
        }
}
```

Scope Operator

- Because a local scope overrides the global scope, having a local variable with the same name as a global variable makes the latter inaccessible to the local scope.
- Eg

```
int num1;
void fun1(int num1)
{
    //…
}
```

- The global num1 is inaccessible inside fun1(), because it is overridden by the local num1 parameter.
- This problem is overcome using the scope operator '::' which takes a global entity as argument.

```
int num1 = 2;
void fun1(int num1)
{
    //…
    num1=33;
    cout<<num1; // the output will be 33
    cout<<::num1;  //the output will be 2 which is the global
    if(::num1 != 0)//refers to global num1
        //…

}
```

Automatic versus static variables

- The terms automatic and static describe what happens to local variables when a function returns to the calling procedure. By default, all local variables are automatic, meaning that they are erased when their function ends. You can designate a variable as automatic by prefixing its definition with the term auto.
- Eg. The two statements after main()'s opening brace declared automatic local variables:

```
main()
{       int i;
        auto float x;
        …
}
```

- The opposite of an automatic is a static variable. All global variables are static and, as mentioned, all static variables retain their values. Therefore, if a local variable is static, it too retains its value when its function ends-in case this function is called a second time.
- To declare a variable as static, place the static keyword in front of the variable when you define it. The following code section defines three variables i, j, k. the variable i is automatic, but j and k are static.
- Static variables can be declared and initialized within the function, but the initialization will be executed only once during the first call.
- If static variables are not declared explicitly, they will be declared to 0 automatically.

  Eg. *void my_fun()*

```
{       static int num;
        static int  count = 2;
       count=count*5;
      num=num+4;
}
```

- In the above example:
  - *During the first call of the function my_fun(), the static variable count will be initialized to 2 and will be multiplied by 5 at line three to have the value 10. During the second call of the same function count will have 10 and will be multiplied by 5.*
  - *During the first call of the function my_fun(), the static variable num will be initialized to 0 (as it is not explicitly initialized) and 4 will be added to it at line four to have the value 4. During the second call of the same function num will have 4 and 4 will be add to it again.*
- **N.B**. if local variables are static, their values remain in case the function is called again.

5.5. <u>Overloaded Functions</u>

- Unlike C, C++ lets you have more than one function with the same name. In other words, you can have three functions called abs() in the same program.
- Functions with the same name are called overloaded functions. C++ requires that each overloaded functions differ in its argument list. Overloaded functions enable you to have similar functions that work on different types of data.
- Suppose that you wrote a function that returned the absolute value of whatever number you passed to it:

```
int iabs(int i)
{         if(i<0)
               return (i*-1);
          else
               return (i);
}

float fabs(float x)
{          if(x<0.0)
               return (x * -1.0);
           else
                return (x);
}
```

  - Without using overloading, you have to call the function as:

```
int ans = iabs(weight);//for int arguments
float ans = fabs(weight);//for float arguments
```

  - But with overloading, the above code can be used as:

```
int abs(int i);

float abs(float x);

int main()

{

int y;

float p;

 …

ians = abs(y); //calling abs with int arguments
fans = abs(p); //calling abs with float arguments

 …

}
```

```
int abs(int i)
{
    if(i<0)
            return i*-1;
    else
            return i;
}

float abs(flaot x)
{
    if(x<0.0)
            return x*-1.0;
    else
            return x;
}
```

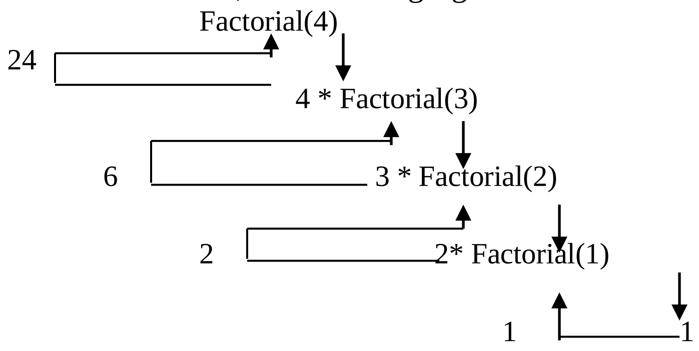
⇨ **N.B**: if two or more functions differ only in their return types, C++ can't overload them. Two or more functions that differ only in their return types must have different names and can't be overloaded

5.6. Recursion

- A function which calls itself is said to be recursive. Recursion is a general programming technique applicable to problems which can be defined in terms of them selves. 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 * factrial(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
- let us implement two more mathematical functions using recursion
- e.g the following function computes the sum of the first N positive integers 1,2,…,N. Notice how the function includes the three necessary components of a recursive method.

```
int sum(int N)
{
        if(N==1)
                return 1;
        else
                return N+sum(N-1);
}
```

- The last method computes the exponentiation A^n where A is a real number and N is a positive integer. This time, we have to pass two arguments. A and N. the value of A will not change in the calls, but the value of N is decremented after each recursive call.

```
float expo(float A, int N)
{
        if(N==1)
                return A;
        else
                return A * expo(A,N-1);
}
```

Recursion versus iteration

- Both iteration and recursion are based on control structure. Iteration uses a repetition structure (such as for, while, do…while) and recursive uses a selection structure (if, if else or switch).
- Both iteration and recursive can execute infinitely-an infinite loop occurs with iteration if the loop continuation test become false and infinite recursion occurs id the recursion step doesn't reduce the problem in a manner that coverage on a base case.
- Recursion has disadvantage as well. It repeatedly invokes the mechanism, and consequently the overhead of method calls. This can be costly in both processor time and memory space. Each recursive call creates another copy of the method (actually, only the function's variables); this consumes considerable memory.
- **N.B**: 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!!

Worksheet 4.

1. Write an int function cube () that returns the cube of its single int formal parameter.
2. 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.
3. 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.
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. Write a function called isPrime() that accepts a number and determine whether the number is prime or not.
6. Write a function called isEven() that uses the remainder operator(%) to determine whether an integer is even or not.

Chapter Six
6. Pointers

- 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

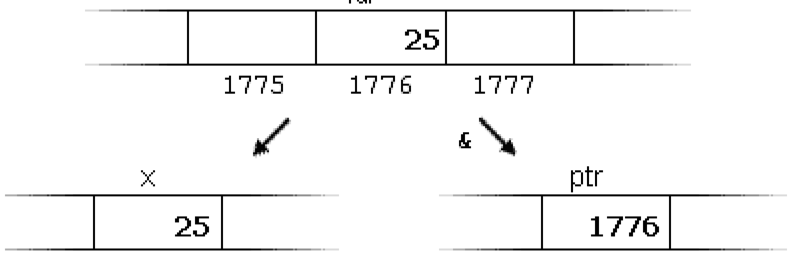
  * the dereference operator

- 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.**

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.

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.

<u>N.B:</u> 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.

```
#...
#...
int 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 = &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

}
```

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 can not be stored in a float or character pointer.

```
flaot 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 can not be de referenced. They are aimed only to store address and the dereference operator is not allowed for void pointers.

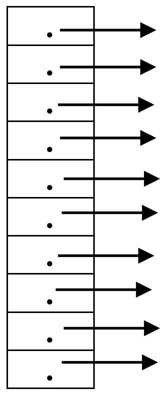
Arrays of Pointers

- 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 10 integer pointer variables:

  *int *iptr[10]; //reserves an array of 10 integer pointers*

- The above statement will create the following structure in RAM



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

Pointer and arrays

- The concept of array goes very bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing. For example, supposing these two declarations:

  int numbers [20];
  *int * p;*

- the following allocation would be valid:

  p = numbers;

- At this point **p** and **numbers** are equivalent and they have the same properties, with the only difference that we could assign another value to the pointer **p** whereas **numbers** will always point to the first of the 20 integer numbers of type int with which it was defined. So, unlike **p,** that is an ordinary *variable pointer*, **numbers** is a *constant pointer* (indeed that is an Array: a constant pointer). Therefore, although the previous expression was valid, the following allocation is not:

  numbers = p;

- Because **numbers** is an array (constant pointer), and no values can be assigned to constant identifiers.
- **N.B:** An array name is just a pointer, nothing more. The array name always points to the first element stored in the array. Therefore , we can have the following valid C++ code:

  int ara[5] = {10,20,30,40,50};
  *cout<< *(ara + 2); //prints ara[2];*
- The expression *(ara+2) is not vague at all if you remember that an array name is just a pointer that always points to the array's first element. *(ara+2) takes the address stored in ara, adds 2 to the address, and dereferences that location.
- 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____
```

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:

```
#...
#...
int main()
{
        float v1 = 679.54;
        float v2 = 900.18;
        float * p_v;

        p_v = &v1;
        cout<< "\n the first value is "<<*p_v;//print the value of v1;
        p_v = &v2;
        cout<< "\n the second value is "<<*p_v; m//print the value of v2;

}
```

- 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.

```
int main()
{
  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";

 }
```

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

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

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

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

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

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.

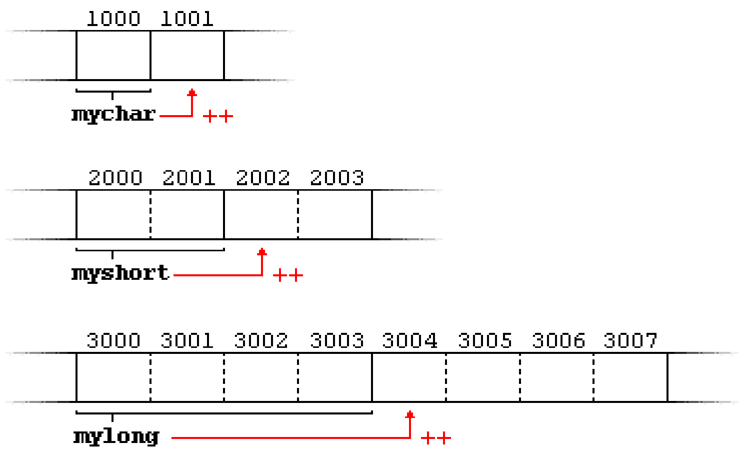
- Let's suppose that we have 3 pointers:

```
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:

  ```
  int 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;
  }
  ```

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	m	\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.

Pointer to pointer:

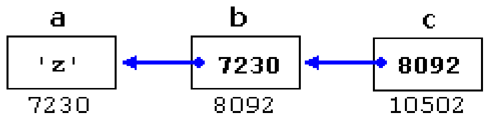
- As the memory address where integer, float or character is stored in can be stored into a pointer variable, the address of a pointer can also be stored in another pointer. This pointer is said to be pointer to a pointer.
- An array of pointer is conceptually same as pointer to pointer type. The pointer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



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

- Have a look at the following code:

  #...

```
int main()
 {      int data;
        int *p1;
        int **p2;
        data = 15;
        cout<< “data = ”<<data<<endl;
        p1 = &data;
        p2 = &p1;
        cout<< “data through p1 = ”<<*p1<<endl;//print the value of data;
        cout<< “data through p2 = ”<< **p2<<endl; //print the value of data;
 }
```

Dynamic memory:

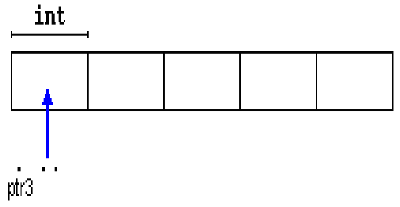
- Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them to be fixed before the execution of the program. But, What if we need a variable amount of memory that can only be determined during the program execution (runtime)? For example, in case that we need a user input to determine the necessary amount of space. The answer is *dynamic memory,* for which C++ integrates the operators *new* and *delete.*
- Pointers are useful for creating **dynamic** objects during program execution. Unlike normal (global and local) objects which are allocated storage on the runtime stack, a dynamic object is allocated memory from a different storage area called the **heap**. Dynamic objects do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.

  a) The New Operator

  - In C++ *new* operator can create space dynamically i.e at run time, and similarly *delete* operator is also available which releases the memory taken by a variable and return memory to the operating system.
  - 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
    ```
  - Lets 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>
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!";
        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;
}
```

- **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:

Chapter Seven

Structures

What is a Structure?

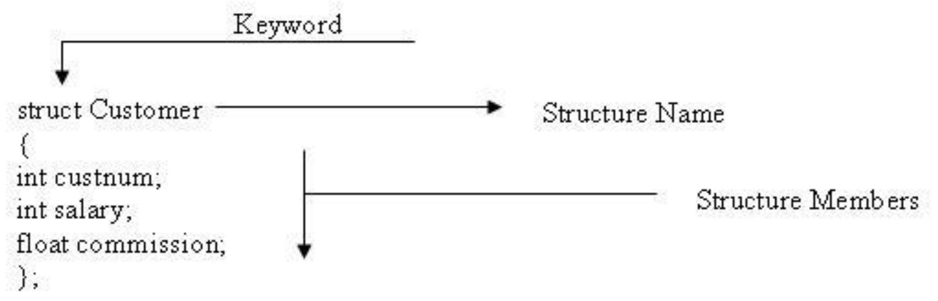
Structure is a collection of variables under a single name. Variables can be of any type: int, float, char etc. The main difference between structure and array is that arrays are collections of the same data type and structure is a collection of variables under a single name.

Declaring a Structure*:*

The structure is declared by using the keyword struct followed by structure name, also called a tag. Then the structure members (variables) are defined with their type and variable names inside the open and close braces { and }. Finally, the closed braces end with a semicolon denoted as ; following the statement. The above structure declaration is also called a Structure Specifier.

Example:

Three variables: *custnum* of type int, *salary* of type int, *commission* of type float are structure members and the structure name is Customer. This structure is declared as follows:



```
struct Customer
{
int custnum;
int salary;
float commission;
};
```

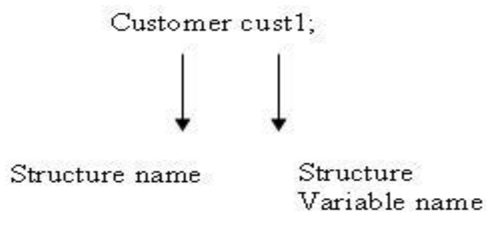
In the above example, it is seen that variables of different types such as int and float are grouped in a single structure name Customer.

Arrays behave in the same way, declaring structures does not mean that memory is allocated. Structure declaration gives a skeleton or template for the structure.
After declaring the structure, the next step is to define a structure variable.

How to declare Structure Variable?

This is similar to variable declaration. For variable declaration, data type is defined followed by variable name. For structure variable declaration, the data type is the name of the structure followed by the structure variable name.

In the above example, structure variable cust1 is defined as:



```
Customer cust1;
```

Here are examples of declaring structure variable

#include<iostream.h>
struct customer
{
int custnum;
char name[10];
int phonenum;
}**cust1,cust2,cust3;**//structure variable declaration
int main()
{
….
}
	#include<iostream.h>
	struct customer
	{
	int custnum;
	char name[10];
	int phonenum;
	}**;**
	int main()
	{
	struct customer cust1,cust2,cust3; //structure variable declaration
	}

What happens when this is defined? When structure is defined, it allocates or reserves space in memory. The memory space allocated will be cumulative of all defined structure members. In the above example, there are 3 structure members: custnum, name and phonenum. Of these, two are of type in and one is of type char. If integer space allocated by a system is 2 bytes and char one bytes the above would allocate 2bytes for custnum, 2 bytes for phonenum and 1byte for name.

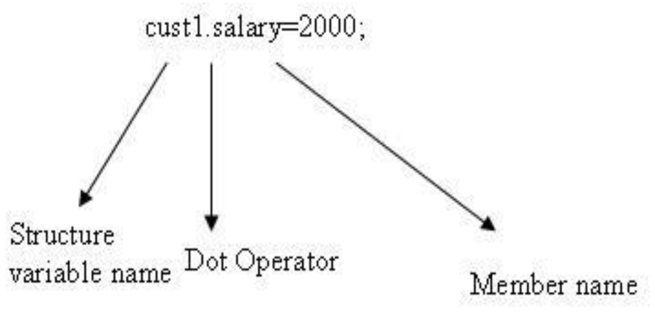
How to access structure members in C++?

- ✓ Use the dot operator to access or initialize members of structures.
- ✓ More importantly, you usually do not even know the contents of the structure variables. Generally, the user enters data to be stored in structures, or you read them from a disk file.

- ✓ A better approach to initializing structures is to use the dot operator (.). The dot operator is one way to initialize individual members of a structure variable in the body of your program. With the dot operator, you can treat each structure member almost as if it were a regular non structure variable.
- ✓ The General syntax to access members of a structure variable would be: structurevariablename.membername
- ✓ A structure variable name must always precede the dot opera- tor, and a member name must always appear after the dot operator. Using the dot operator is easy, as the following examples show.

For example:

A programmer wants to assign 2000 for the structure member *salary* in the above example of structure *Customer* with structure variable *cust1* this is written as:



Here are some examples

```
#include<iostream.h>
struct  customer
{
int custnum;
char name[10];
int phonenum;
}cust1;
int main()
{
cout<<"Enter customer number"<<endl;
cin>>cust1.custnum;
cout<<"Enter customer name"<<endl;
cin>>cust1.name;
cout<<"Enter customer phone number"<<endl;
cin>>cust1.phonenum;
cout<<"Customer Number:"<<cust1.custnum;
cout<<"\nCustomer Name:"<<cust1.name;
cout<<"\nCustomer Phone:"<<cust1.phonenum;
}
```

```
#include<iostream.h>
struct  customer
{
int custnum;
char name[10];
int phonenum;
};
int main()
{
struct customer cust1;
cout<<"Enter customer number"<<endl;
cin>>cust1.custnum;
cout<<"Enter customer name"<<endl;
cin>>cust1.name;
cout<<"Enter customer phone number"<<endl;
cin>>cust1.phonenum;
cout<<"Customer Number:"<<cust1.custnum;
cout<<"\nCustomer Name:"<<cust1.name;
cout<<"\nCustomer Phone:"<<cust1.phonenum;
}
```

Initializing structure members

You cannot initialize individual members because they are not variables. You can assign only values to variables. The only structure variable in the above structure is cust1. The braces must enclose the data you initialize in the structure variables, just as they enclose data when you initialize arrays.
This method of initializing structure variables becomes tedious when there are several structure variables (as there usually are). Putting the data in several variables, each set of data enclosed in braces, becomes messy and takes too much space in your code.
You can initialize members when you declare a structure, or you can initialize a structure in the body of the program.
For example

Initializing members at declaration

```
#include<iostream.h>
struct employee
{
char Emp_id[7];
char Name[20];
float Salary;
}emp1={"DMU/001","Solomon",2808};
int main()
{
cout<<"Your id:"<<emp1.Emp_id<<endl;
cout<<"Your name:"<<emp1.Name<<endl;
cout<<"Your salay:"<<emp1.Salary<<endl;
}
```

Initializing members at the body of the program

```
#include<iostream.h>
#include<string.h>
struct employee
{ char Emp_id[7];
  char Name[20];
  float Salary;
}emp1;
int main()
{
strcpy(emp1.Emp_id,"DMU/001/");
strcpy(emp1.Name,"Solomon");
emp1.Salary=2808;
cout<<"Your id:"<<emp1.Emp_id<<endl;
```

	cout<<"Your name:"<<emp1.Name<<endl;
	cout<<"Your salay:"<<emp1.Salary<<endl;
	}

Arrays of Structures

- ✓ Arrays of structures are good for storing a complete employee file, inventory file, or any other set of data that fits in the structure format.
- ✓ Consider the following structure declaration:

```
struct Company
   {
        int employees;
        int registers;
        double sales;
   }store[1000];
```

- ✓ In one quick declaration, this code creates 1,000 ***store*** structures with the definition of the ***Company*** structure, each one containing three members.
- ✓ NB. Be sure that your computer does not run out of memory when you create a large number of structures. Arrays of structures quickly consume valuable information.
- ✓ You can also define the array of structures after the declaration of the structure.

```
struct Company

   {

        int employees;

        int registers;

        double sales;

   }; // no structure variables defined yet

#include<iostream.h>

…

void main()

{

struct Company store[1000]; //the variable store is array of the structure Company

        …
}
```

Referencing the array structure

- ✓ The ***dot operator (.)*** works the same way for structure array element as it does for regular variables.
- ✓ Look the following example

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

```
struct student
{
int id;
float mark;
char name[15];
}stud[2];
int main()
{
for(int i=0;i<=1;i++)
{
cout<<"Enter the id, name and mark of student"<<i+1<<endl;
cin>>stud[i].id>>stud[i].name>>stud[i].mark;
}
for(int i=0;i<=1;i++)
{
cout<<"Id of Student "<<i+1<<"="<<stud[i].id<<endl;
cout<<"Name of Student "<<i+1<<"="<<stud[i].name<<endl;
cout<<"Mark of student "<<i+1<<"="<<stud[i].mark<<endl;
}
}
```

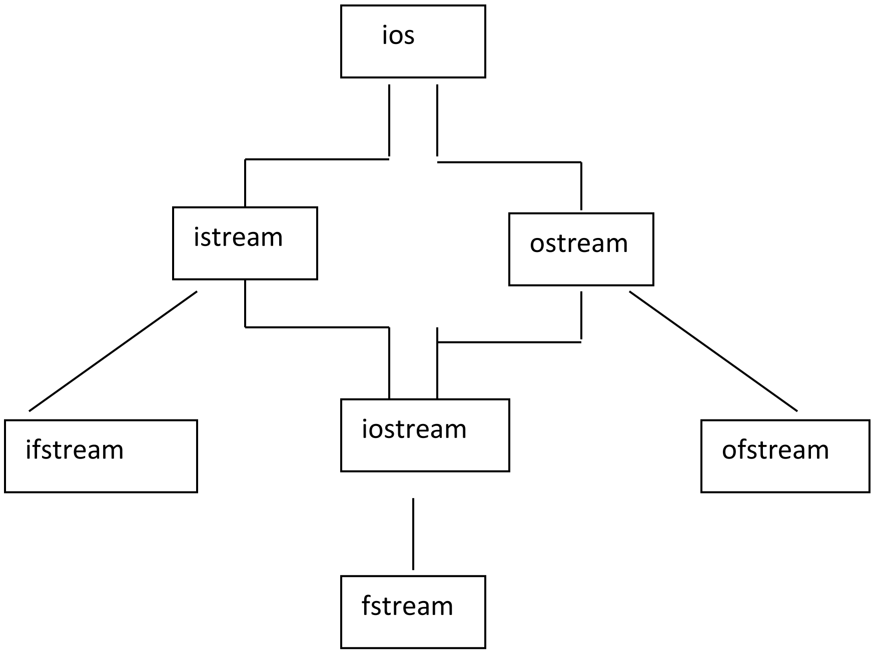
Chapter Eight
8. Streams and External Files

Introduction

- The data created by the user and assigned to variables with an assignment statement is sufficient for some applications. With large volume of data most real-world applications use a better way of storing that data. For this, disk files offer the solution.
- When working with disk files, C++ does not have to access much RAM because C++ reads data from your disk drive and processes the data only parts at a time.

8.2 Stream

- Stream is a general name given to *flow of data*. In C++, there are different types of streams. Each stream is associated with a particular class, which contains member function and definition for dealing with file. Lets have a look at the figure:



- According to the above hierarchy, the class iostream is derived from the two classes' istream and ostream and both istream and ostream are derived from ios. Similarly the class fstream is derived from iostream. Generally two main header files are used iostream.h and fstream.h. The classes used for input and output to the video display and key board are declared in the header file iostream.h and the classes used for disk file input output are declared in fstream.h.
- Note that when we include the header file fstream.h in our program then there is no need to include iostream.h header file. Because all the classes which are in fstream.h they are derived from classes which are in iostream.h therefore, we can use all the functions of iostream class.

8.3: Operation With File

- First we will see how files are opened and closed. A file can be defined by following class ifstream, ofstream, fstream, all these are defined in fstream.h header file.
    - if a file object is declared by *ifstream* class, then that object can be used for *reading* from a file.
    - if a file object is declared by *ofstream* class, then that object can be used for *writing* onto a file.
    - If a file object is declared by *fstream* class then, that object can be used for both *reading from* and *writing to* a file

8.4: Types of Disk File Access

- Your program can access files either in *sequential* manner or *random* manner. The access mode of a file determines how one can read, write, change, add and delete data from a file.
- A sequential file has to be accessed in the same order as the file was written. This is analogues to cassette tapes: you play music in the same order as it was recorded.
- Unlike the sequential files, you can have random-access to files in any order you want. Think of data in a random-access file as being similar to songs on compact disc (CD): you can go directly to any song you want to play without having to play or fast-forward through the other songs.

8.4.1: Sequential File Concepts

- You can perform three operations on sequential disk files. You can *create* disk files, *add* to disk files, and *read* from disk files.

8.4.1.1: Opening and Closing Sequential Files

- When you open a disk file, you only have to inform C++, the file name and what you want to do with it. C++ and your operating system work together to make sure that the disk is ready, and they create an entry in your file directory for the filename (if you are creating a file). When you close a file, C++ writes any remaining data to the file, releases the file from the program, and updates the file directory to reflect the file's new size.
- You can use either of the two methods to open a file in C++:
    - using a ***Constructor*** or
    - using the ***open function***
- The following C++ statement will create an object with the name ***fout*** of ofstream class and this object will be associated with file name "hello.txt".

```
ofstream fout ("hello.txt");
```

- This statement uses the constructor method.
- The following C++ statement will create an object with the name ***fout*** of ofstream class and this object will be associated with file name "hello.txt".

```
ofstream fout;

fout.open("hello.txt");
```

- If you open a file for writing (out access mode), C++ creates the file. If a file by that name already exists, C++ *overwrite* the old file with no warning. You must be careful when opening files not to overwrite existing data that you want.
- If an error occurs during opening of a file, C++ does not create a valid file pointer (file object). Instead, C++ creates a file pointer (object) equal to zero. For example if you open a file for output, but use an invalid disk name, C++ can't open the file and therefore makes the file object equal to zero.
- You can also determine the file access mode when creating a file in C++. If you want to use the open function to open a file then the syntax is:

  fileobject.open(filename,accessmode);

  - File name is a string containing a valid file name for your computer.
  - Accessmode is the sought operation to be taken on the file and must be one of the values in the following table.

Mode	**Description**
app	Opens file for appending
ate	Seeks to the end of file while opening the file
in	Opens the file for reading
out	Opens the file for writing
binary	Opens the file in binary mode

- You should always check for the successful opening of a file before starting file manipulation on it. You use the fail() function to do the task:
- Lets have an example here:

```
ifstream indata;

indata.open("c:\\myfile.txt",ios::in);

if(indata.fail())

{

    //error description here

}
```

- In this case, the open operation will fail (i.e the fail function will return true), if there is no file named myfile.txt in the directory C:\
- After you are done with your file manipulations, you should use the close function to release any resources that were consumed by the file operation. Here is an example

```
indata.close();
```

- The above close() statement will terminate the relationship between the ifstream object indata and the file name "c:\myfile.txt", hence releasing any resource needed by the system.

8.4.1.2: Writing to a sequential File

- The most common file I/O functions are
  - get() and put()
- You can also use the output redirection operator (<<) to write to a file.
- The following program creates a file called names.txt in C:\ and saves the name of five persons in it:

```
#include<fstream.h>
#include<stdlib.h>
ofstream fp;
void main()
{
        fp.open("c:\\names.txt" ,ios::out);

        if(fp.fail())

        {       cerr<< "\nError opening file";

                getch();

                exit(1);

        }

        fp<< "Abebe Alemu"<<endl;

        fp<< "Lemelem Berhanu"<<endl;

        fp<< "Tesfaye Mulugeta"<<endl;

        fp<< "Mahlet Kebede"<<endl;

        fp<< "Assefa Bogale"<<endl;

        fp.close();

}//end main
```

Writing characters to sequential files:

- A character can be written onto a file using the *put()* function. See the following code:

```
#include<fstream.h>

#include<stdlib.h>// for exit() function

…

void main()
```

```
{
        char c;
        ofstream outfile;
        outfile.open(“c:\\test.txt”,ios::out);
        if(outfile.fail())
        {
                cerr<< “\nError opening test.txt”;
                getch();
                exit(1);
        }
        for(int i=1;i<=15;i++)
        {
                cout<< “\nEnter a character : ”;
                cin>>c;
                outfile.put(c);
        }
        output.close();
}//end main
```

- The above program reads 15 characters and stores in file test.txt.
- You can easily add data to an existing file, or create new files, by opening the file in *append access mode*.
- Files you open for append access mode (using ios::app) do *not have to exist*. If the file exists, C++ appends data to the end of the file (as is done when you open a file for write access).
- The following program adds three more names to the *names.txt* file created in the earlier program.

```
#include<fstream.h>
#include<stdlib.h>
…
void main()
```

```
{
        ofstream outdata;
        outdata.open(“c:\\names.txt”,ios::app);
        if(outdata.fail())
        {
                cerr<< “\nError opening names.txt”;
                getch();
                exit(1);
        }
        outdata<< “Berhanu Teka”<<endl;
        outdata<< “Zelalem Assefa”<<endl;
        outdata<< “Dagim Sheferaw”<<endl;
        outdata.close();
}//end main
```

- If the file names.txt *does not exist*, C++ creates it and stores the three names to the file.
- Basically, you have to change only the open() function’s access mode to turn a file-creation program into a file-appending program.

8.4.1.3: Reading from a File

- Files you open for read access (using ios::in) *must exist already*, or C++ gives you an error message. You can’t read a file that does not exist. Open() returns zero if the file does not exist when you open it for read access.
- Another event happens when you read files. Eventually, you read all the data. Subsequently reading produces error because there is *no more* data to read. C++ provides a solution to the end-of-file occurrence.
- If you attempt to read a file that you have completely read the data from, C++ returns the value zero. To find the end-of-file condition, be sure to check for zero when reading information from files.
- The following code asks the user for a file name and displays the content of the file to the screen.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
```

```
{
  clrscr();
  char name[20],filename[15];
  ifstream indata;
  cout<<"\nEnter the file name : ";
  cin.getline(filename,15);
  indata.open(filename,ios::in);
  if(indata.fail())
  {

    cerr<<"\nError opening file : "<<filename;

    getch();

    exit(1);
  }

  while(!indata.eof())// checks for the end-of-file
  {
    indata>>name;
    cout<<name<<endl;
  }
  indata.close();

  getch();

}
```

Reading characters from a sequential file

- You can read a characters from a file using *get()* function. The following program asks for a file name and displays *each character* of the file to the screen. NB. A space among characters is considered as a character and hence, the exact replica of the file will be shown in the screen.

```
#include<fstream.h>

#include<conio.h>

#include<stdlib.h>

void main()

{

  char c,filename[15];

  ifstream indata;
```

```
    cout<<"\nEnter the file name : ";
    cin.getline(filename,15);
    indata.open(filename,ios::in);
    if(indata.fail())// check id open succeeded
    {
        cerr<<"\nError opening file : "<<filename;
        getch();
        exit(1);
    }
    while(!indata.eof())// check for eof
    {
        indata.get(c);
        cout<<c;
    }
    indata.close();
    getch();
}
```

8.4.1.4: File Pointer and their Manipulators

- Each file has two pointers one is called *input pointer* and second is *output pointer*. The input pointer is called *get pointer* and the output pointer is called *put pointer*.
- When input and output operation take places, the appropriate pointer is automatically set according to the access mode.
- For example when we open a file in reading mode, file pointer is automatically set to the start of the file.
- When we open a file in append mode, the file pointer is automatically set to the end of file.
- In C++ there are some manipulators by which we can control the movement of the pointer. The available manipulators are:
    1. seekg()
    2. seekp()
    3. tellg()
    4. tellp()

1. seekg(): this moves get pointer i.e input pointer to a specified location.

    For eg. infile.seekg(5); move the file pointer to the byte number 5 from starting point.

2. seekp(): this move put pointer (output pointer) to a specified location for example: outfile.seekp(5);
3. tellg(): this gives the current position of get pointer (input pointer)
4. tellp(): this gives the current position of put pointer (output pointer)

```
eg.     ofstream fileout;

        fileout.open("c:\\test.txt",ios::app);

        int length = fileout.tellp();
```

- By the above statement in length, the total number bytes of the file are assigned to the integer variable length. Because the file is opened in append mode that means, the file pointer is the last part of the file.
- Now lets see the seekg() function in action

```
#include<fstream.h>

#include<conio.h>

#include<stdlib.h>

void main()

{

  clrscr();

  fstream fileobj;

  char ch; //holds A through Z

  //open the file in both output and input mode

  fileobj.open("c:\\alph.txt",ios::out|ios::in);

  if(fileobj.fail())

  {

     cerr<<"\nError opening alph.txt";

     getch();

     exit(1);

  }

  //now write the characters to the file

  for(ch = 'A'; ch <= 'Z'; ch++)

  {
     fileobj<<ch;
  }
  fileobj.seekg(8L,ios::beg);//skips eight letters, points to I
  fileobj>>ch;
  cout<<"\nThe 8th character is : "<<ch;
  fileobj.seekg(16L,ios::beg);//skips 16 letters, points to Q
  fileobj>>ch;
  cout<<"\nThe 16th letter is : "<<ch;
  fileobj.close();
  getch();
}
```

- To point to the end of a data file, you can use the seekg() function to position the file pointer at the last byte. This statement positions the file pointer to the last byte in the file. ***Fileobj.seekg(0L,ios::end);***
- This seekg() function literally reads "move the file pointer 0 bytes from the end of the file." The file pointer now points to the end-of-file marker, but you can seekg() backwards to find other data in the file.
- The following program is supposed to read "c:\alph.txt" file backwards, printing each character as it skips back in the file.
- Be sure that the seekg() in the program seeks two bytes backwards from the *current* position, not from the beginning or the end as the previous programs. The for loop towards the end of the program needs to perform a "skip-two-bytes-back", read-one-byte-forward" method to skip through the file backwards.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
  clrscr();
  ifstream indata;
  int ctr=0;
```

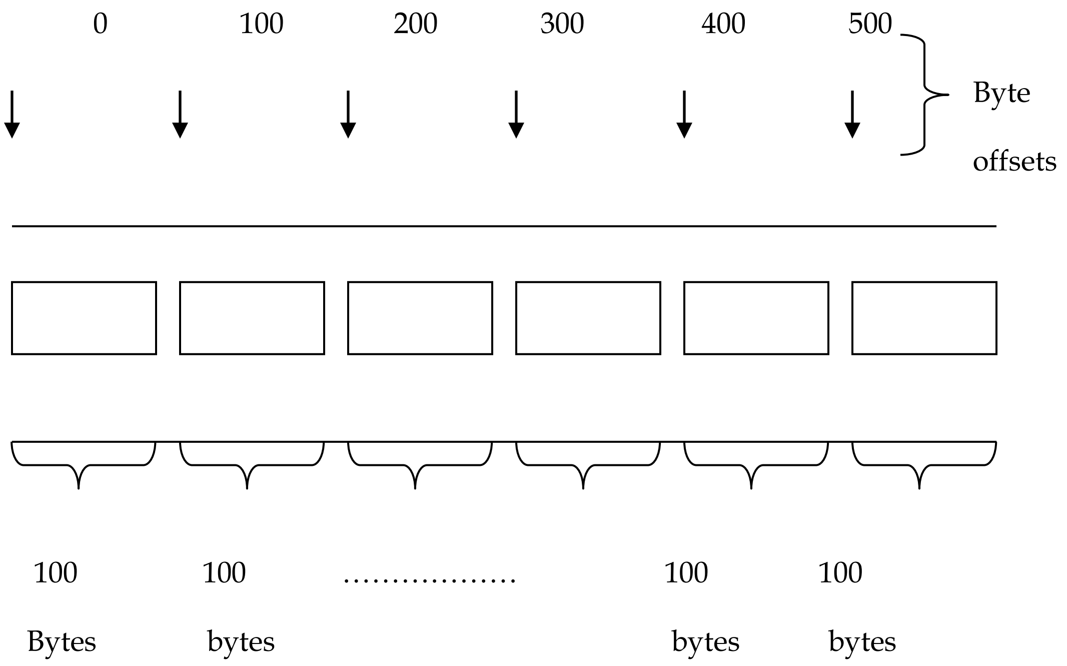
```
    char inchar;

    indata.open("c:\\alph.txt",ios::in);

    if(indata.fail())

    {

       cerr<<"\nError opening alph.txt";

       getch();

       exit(1);

    }

    indata.seekg(-1L,ios::end);//points to the last byte in the file

    for(ctr=0;ctr<26;ctr++)

    {

       indata>>inchar;

       indata.seekg(-2L,ios::cur);

       cout<<inchar;

    }

    indata.close();

    getch();

  }
```

Text Files And Binary Files (Comparison)

- The default access mode for file access is *text mode*. A text file is an ASCII file, compatible with most other programming languages and applications. Programs that read ASCII files can read data you create as C++ text files.
- If you specify binary access, C++ creates or reads the file in binary format. Binary data files are *"squeezed"*- that is, they take less space than text files. The disadvantage of using binary files is that other programs can't always read the data files. Only C++ programs written to access binary files can read and write them. The advantage of binary files is that you save disk space because your data files are more compact.
- The binary format is a *system-specific* file format. In other words, not all computers can read a binary file created on another computer.

8.4.2: Random Access File Concepts

- Random access enables you to read or write any data in your disk file with out having to read and write every piece of data that precedes it.
- Generally you read and write file records. A record to a file is analogues to a C++ structure. A record is a collection of one or more data values (called fields) that you read and write to disk. Generally you store data in the structures and write structures to disk.
- When you read a record from disk, you generally read that record into a structure variable and process it with your program.
- Most random access files are fixed-length records. Each record (a row in the file) takes the same amount of disk space.
- With fixed length records, your computer can better calculate where on the disk the desired record is located.



8.2.2.1 Opening Random-Access Files

- There is really no difference between sequential files and random files in C++. The difference between the files is not physical, but lies in the method that you use to access them and update them.
- The ostream member function write outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream.
- When the stream is associated with a file, function write writes the data at the location in specified by the "put" file position pointer.
- The istream member function read inputs a fixed number of bytes from the specified stream into an area in memory beginning at the specified address.
- When the stream is associated with a file, function read inputs bytes at the location in the file specified by the "get" file poison pointer.
- Syntax of write: *fileobject.write((char*) & NameOfObject, sizeof(name of object))*
- Function write expects data type const char* as its first argument. The second argument of write is an integer of type size specifying the number of bytes to be written.

Writing randomly to a random access file

- Here is an example that shows how to write a record to a random access file.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
struct stud_info{
    int id;
    char name[20];
    char fname[20];
    float CGPA;

}student;



void main()

{

  clrscr();

  char filename[15];

  ofstream outdata;

  cout<<"\nenter file name : ";

  cin>>filename;

  outdata.open(filename,ios::out);

  if(outdata.fail())

  {

     cerr<<"\nError opening "<<filename;

     getch();

     exit(1);

  }

  //stud_info student;

  //accept data here

  cout<<"\nEnter student id : ";
```

```
    cin>>student.id;

    cout<<"\nEnter student name : ";

    cin>>student.name;

    cout<<"\nEnter student father name : ";

    cin>>student.fname;

    cout<<"\nEnter student CGPA : ";

    cin>>student.CGPA;



    //now write to the file

    outdata.seekp(((student.id)-1) * sizeof(student));

    outdata.write((char*) &student, sizeof(student));

    outdata.close();

    cout<<"\nData has been saved";

    getch();

  }
```

- The above code uses the combination of ostream function seekp and write to store data at exact locations in the file.
- Function seekp sets the put file-position pointer to a specific position in the file, then the write outputs the data.
- 1 is subtracted from the student id when calculating the byte location of the record. Thus, for record 1, the file position pointer is set to the byte 0 of the file.
- The istream function read inputs a specified number of bytes from the current position in the specified stream into an object.
- The syntax of read : read((char*)&name of object, sizeof(name of object));
- Function read requires a first argument of type char *. The second argument of write is an integer of type size specifying the number of bytes to be read.
- Here is a code that shows how to read a random access record from a file.

```
#include<fstream.h>

#include<conio.h>

#include<stdlib.h>

struct stud_info{
    int studid;

    char name[20];

    char fname[20];

    float CGPA;

};

void main()

{

     clrscr();

     ifstream indata;

     char filename[15];

     cout<<"\nEnter the file name : ";

     cin>>filename;

     indata.open(filename,ios::in);

     if(indata.fail())

     {

            cerr<<"\nError opening "<<filename;

            getch();

            exit(1);

     }

     stud_info student;

     cout<<"\nEnter the id no of the student : ";

     int sid;

     cin>>sid;

     indata.seekg((sid-1) * sizeof(student));

     indata.read((char*) &student, sizeof(student));

     cout<<"\nhere is the information";
```

```
        cout<<"\nstudent id : "<<student.studid;

        cout<<"\nstudent name : "<<student.name;

        cout<<"\nstudent fname : "<<student.fname;

        cout<<"\nstudent CGPA : "<<student.CGPA;

        indata.close();

        getch();

    }
```

8.5: Command Line Argument

- Command line argument means facility by which you can supply arguments to the main() function. These arguments are supplied to the program when the main function is called from the command line. Eg. c:\> file-name arg1 arg2
- Where file-name is the name of file(the program) and arg1, arg2 are arguments passed to the program.
- If we want to pass arguments to the main function, then main function should be written as follow.

  Return type main(int argc, char * argv[])

- The first argument argc represents the number of arguments in a command line. The second argument argv is an array of character type pointers that points to the command line arguments. Argc is known as argument counter and argv is called argument vector.
- C:\> student A B. the value of argc is 3 (student, A & B) and the argv would be an array of three pointers to string as follows:

  Argv[0] – points to student

  Argv[1] – points to A

  Argv[2] – points to B

- note that argv[0] always reperesents the command name that invokes the program.
- Here is a sample command line argument code

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main(int argc, char *argv[])
{
   clrscr();
   char ch;
   if(argc < 3)
   {
     cerr<<"\ntoo few parameters";
     getch();
     exit(1);
   }
   else if(argc > 3)
   {
     cerr<<"\ntoo many parametes";
     getch();
     exit(1);
   }
   else//number of parameters okay
   {
          ofstream outdata;
          ifstream indata;
          outdata.open(argv[2],ios::out);
          if(outdata.fail())
          {
             cerr<<"\nlow disk space to create file : "<<argv[2];
             getch();
             exit(1);
          }
          indata.open(argv[1],ios::in);
          if(indata.fail())
          {
             cerr<<"\nfile : "<<argv[1]<<" does not exist";
             getch();
             exit(1);
          }
          //now start copying the file
          while(!indata.eof())
          {
            indata.get(ch);
            outdata.put(ch);
          }
         indata.close();
         outdata.close();
         cout<<"\nfinished copying";
   }
}
```