Chapter One
Introduction

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?

- ***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***.

1.1. Generations of programming language.

- Programming languages are categorized into five generations: (1st, 2nd, 3rd, 4th and 5th generation languages)
- These programming languages can also be categorized into two broad categories: low level and high level languages.
  - Low level languages are machine specific or dependent.
  - High level languages like COBOL, BASIC are machine independent and can run on variety of computers.
- From the five categories of programming languages, first and second generation languages are low level languages and the rest are high level programming languages.
- The higher the level of a language, the easier it is to understand and use by programmers.
- Languages after the forth generation are refereed to as a very high level languages.

1.1.1. First Generation (Machine languages, 1940's):

- Difficult to write applications with.
- Dependent on machine languages of the specific computer being used.
- Machine languages allow the programmer to interact *directly with the hardware*, and it can be executed by the computer without the need for a translator.
- Is more powerful in utilizing resources of the computer.
- Gives power to the programmer.
- They execute very quickly and use memory very efficiently.

1.1.2. Second Generation (Assembly languages, early 1950's):

- Uses symbolic names for operations and storage locations.
- A system program called an *assembler* translates a program written in assembly language to machine language.
- Programs written in assembly language *are not portable*. i.e., different computer architectures have their own machine and assembly languages.
- They are *highly used in system software development*.

1.1.3. Third Generation (High level languages, 1950's to 1970's):

- Uses English like instructions and mathematicians were able to define variables with statements such as Z = A + B
- Such languages are much easier to use than assembly language.
- Programs written in high level languages need to be translated into machine language in order to be executed.

- The use of common words (reserved words) within instructions makes them easier to learn.
- All third generation programming languages are procedural languages.
- In procedural languages, the programmer is expected to specify what is required and how to perform it.

1.1.4. Fourth Generation (since late 1970's):

- Have a simple, English like syntax rules; commonly used *to access databases*.
- Forth generation languages are *non-procedural* languages.
- The non-procedural method is easier to write, but you have less control over how each task is actually performed.
- In non-procedural languages the programmer is not required to write traditional programming logic. Programmers concentrate on defining the input and output rather than the program steps required.
    - For example, a command, such as LIST, might display all the records in a file on screen, separating fields with a blank space. In a procedural language, all the logic for inputting each record, testing for end of file and formatting each column on screen has to be explicitly programmed.
- Forth generation languages have a minimum number of syntax rules. This saves time and free professional programmers for more complex tasks.
- Some examples of 4GL are structured query languages (SQL), report generators, application generators and graphics languages.

1.1.5. Fifth Generation (1990's):

- These are used in artificial intelligence (AI) and expert systems; also used for accessing databases.
- 5GLs are "natural" languages whose instruction *closely resembles* human speech. E.g. "get me Jone Brown's sales figure for the 1997 financial year".
- 5GLs require very powerful hardware and software because of the complexity involved in interpreting commands in human language.

1.2. Overview of Computers and Computer Organization.

- Regardless of differences in physical appearance, virtually every computer may be envisioned as being divided into six logical units or sections:
- *Input Unit*: it obtains information from various input devices and places this information at the disposal of the other units so that the information may be processed or stored.
- *Output Unit*: it takes information that has been processed by the computer and places it on various output devices to make the information available for use outside the computer.

- *Memory unit*: it retains information that has been entered through the input unit, so the information may be made immediately available for processing when it is needed. The memory unit also retains processed information until that information can be placed on output devices by the output unit.
- *Central Processing Unit (CPU)*: it is the computer's coordinator and is responsible for *supervising* the operations of the other sections. CPU tells the input unit when information should be read into memory, tells ALU when information from the memory should be used in calculation and tells output unit when to send information from the memory to certain output devices.
- *Arithmetic and Logic unit*: is a part found inside the Central Processing Unit and is responsible for performing calculations such as addition, subtraction, multiplication and division. It also performs comparisons.
- *Secondary Storage device*: Programs or data used by other units normally are placed on secondary storage devices (such as disks) until they are needed, possibly hours, days, months, or even years later.

1.3. The evolution of Operating Systems.

- Early computers were capable of performing only *one job at a time*. This form of computer operation is often called *single-user batch processing*. The computer runs a single program at a time while processing data in groups or batches.
- As computers became more powerful, it became evident that single-user batch processing rarely utilizes the computer's resources.
- Therefore, computer developers thought that many jobs or tasks from different programs or applications could be made to share the resources of the computer to achieve better utilization. This concept is called *multiprogramming*.
- Multiprogramming involving the "simultaneous" operation of many jobs on the computer.
- In the 1960's, many groups, pioneered *timesharing operating systems*.
- Timesharing is a special case of multiprogramming, in which users access the computers through terminals. Here, the computer does not run the users job paralleling, but shares the CPU's time.

1.4. Major Programming Paradigms

- The major land marks in the programming world are the different kinds of features or properties observed in the development of programming languages. Among these the following are worth mentioning: Procedural, Structured and Object Oriented Programming Paradigms.

1.4.1. Procedural Programming.

- Procedural programming is a programming paradigm based upon the concept of *procedure call*. Procedural programming is often a better choice than simple sequential programming in many situations which involve moderate complexity or which require significant ease of maintainability.
- Possible benefits: the ability to re-use the same code (function or procedure) at different places, an easier way to keep track of program flow than a collection of "GO TO" or "JUMP" statements.

1.4.2. Structured Programming.

- Process of writing a program in small, independent parts. This makes it easier to control a program's development and to design and test its individual component parts.
- Structured programs are built up from units called ***modules***, which normally correspond to single procedures or functions.
- Can be seen as a *subset or sub discipline of procedural programming*. It is most famous for removing or reducing reliance on the GO TO statement.

1.4.3. Object-Oriented Programming.

- The idea behind OOP is that, a computer program is composed of a collection of *individual units, or objects* as opposed to traditional view in which a program is a list of instructions to the computer.
- Object-oriented programming is claimed to give more flexibility, easing changes to programs. The OOP approach is often simpler to develop and maintain.

1.5. Problem solving process and software engineering.

1.5.1. Software Engineering.

- Software engineering is the profession that creates and maintains software applications by applying technologies and practices from *computer science*, *project management*, *engineering*, *application domain* and other fields.
- The method used in solving problems in computer science and/or information systems is called the *software development life cycle*.
- The software development life cycle has the following components.
  - ***preliminary investigation***
  - ***analysis***
  - ***design***
  - ***implementation***
  - ***testing and maintenance***

1.5.2. Problem Solving

- Problem solving is the process of transforming the description of a problem into the solution by using our knowledge of the problem domain and by relying on our ability to select and use appropriate problem-solving strategies, techniques, and tools.
- A problem is an *undesirable situation* that prevents the organization from fully achieving its purpose, goals and objectives. Or problem can also be defined as the *gap between the existing and the desired situation* where problem solving will try to fill this gap.

➢ 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.

1.6. Basic Program development tips

➢ The program we design in any programming language need to be:
  - ***Reliable***: the program should always do what it is expected to do and handle all types of exception.
  - ***Maintainable***: the program should be in a way that it could be modified and upgraded when the need arises.
  - ***Portable***: It needs to be possible to adapt the software written for one type of computer to another with minimum modification.
  - ***Efficient***: the program should be designed to make optimal use of time, space and other resources of the computer.

1.7. Algorithm designing and modeling the logic (using flow chart).

➢ A digital computer is a useful tool for solving a great variety of **problems**. A solution to a problem is called an **algorith**m; it describes the sequence of steps to be performed for the problem to be solved.

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

➢ The algorithm should be:
  - ***Precise and unambiguous***
  - ***Simple***
  - ***Correct***
  - ***Efficient***

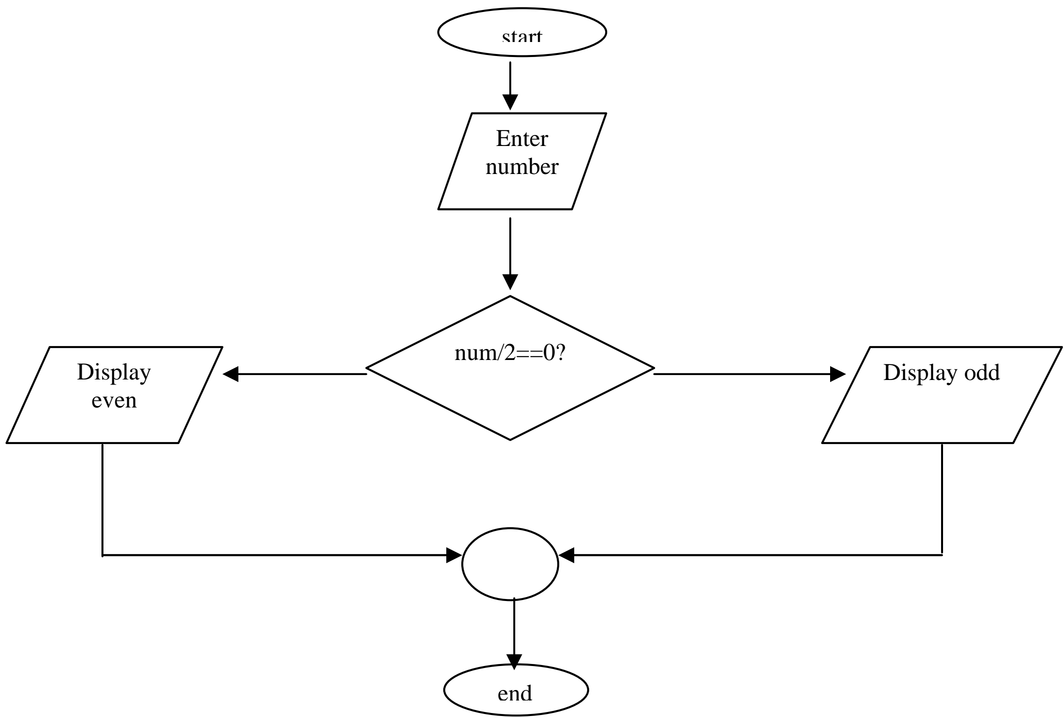
1.7.1. Modeling a programs logic using flow chart

- Algorithm could be designed using many techniques and tools. One tool of designing algorithm is by using flowcharts. Flowchart is a graphical way of expressing the steps needed to solve a problem.
- A flow chart is a schematic (*diagrammatic description*) representation of a process.

- Basic flowcharting symbols are:

Symbol	Meaning
(oval)	Terminal point
(diamond)	Decision
(rectangle)	Process
(parallelogram)	Input/Output
(arrow)	Flow line
(pentagon)	Inter-page connector
(circle)	On-page connector

e.g: to check weather a number is positive or negative.



1.8. Compilers and Interpreters.

- Any program written in a language other than *machine language* needs to be translated to machine language. The set of instructions that do this task are known as *translators*.
- There are different kinds of translator software, among which *compilers* and *interpreters* are of interest for most programmers.
- *Compilers*: a compiler is a computer program that translates a series of statements written in source code (a collection of statements in a specific programming language) into a resulting object code (translated instructions of the statements in a programming language). A compiler changes or translates *the whole* source code into executable machine code (also called *object code*) which is *output to a file* for latter execution. E.g. C++, Pascal, FORTRAN, etc.
- *Interpreters*: is a computer program that translates *a single* high level statement and executes it and then goes to the next high level language line etc. E.g. QBASIC, Lisp etc.

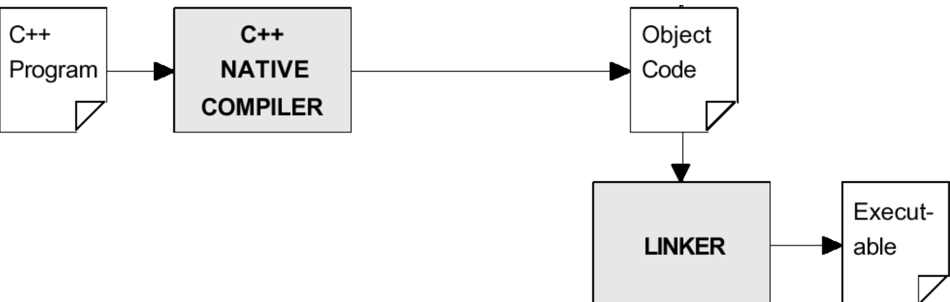
1.9. Mechanics of Creating a program (C++)

- C++ programs typically go through *five phases* to be executed: these are *edit*, *preprocess*, *compile*, *link*, *load*:
- *Edit*: this is accomplished with an editor program. The programmer types C++ statements with the editor and makes corrections if necessary.

The programs source file is then stored on *secondary storage device* such as a disk with a ".cpp" file name.

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

- *Preprocess*: In a C++ system, a *preprocessor* program executes automatically before the compiler's translation phase begins. The C++ preprocessor obeys command called *preprocessor directives*, which indicate that certain manipulations are to be performed on the program before compilation. The preprocessor is invoked by the compiler before the program is converted to machine language. The C++ **preprocessor** goes over the program text and carries out the instructions specified by the preprocessor directives (e.g., #include). The result is a modified program text which no longer contains any directives.
- *Compile*: Then, the C++ **compiler** translates the program code. The compiler may be a true C++ compiler which generates native (assembly or machine) code. The outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program. For example, the << operator which is actually defined in a separate IO library.
- *Linking*: C++ programs typically contain references to functions and data defined else where, such as in the standard libraries. The object code produced by the C++ compiler typically contains "*holes*" due to these missing parts. A linker links the *object code* with the code for the missing function to produce an *executable image* (with no missing pieces). Generally, the **linker** completes the object code by linking it with the object code of any library modules that the program may have referred to. The final result is an executable file
- *Loading*: the loader takes the executable file from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded. Finally, the computer, under the control of its CPU, executes the program.



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

Chapter Two
Basics of C++

2.1. The parts of a C++ Program.

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

```
#include<iostream>

void main()
{
        cout<<"\n Hello World!";
        return 0;
}
```

- Any C++ program file should be saved with file name extension " ***.CPP*** "
- Type the program directly into the editor, and save the file as hello.cpp, compile it and then run it. It will print the words Hello World! to 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 and looks for the pound (#) 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.
- The effects of line 1, i.e. include<iostream> 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 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.
- A statement is a computation step which may produce a value.
- The end of a single statement ends with semicolon (;).

- The statement in the above example causes the sting “Hello World!” to be sent to the “cout” stream which will display it on the computer screen.

2.2. 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.
- 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 (>>) and the object name to hold the input value.
- 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
  - ***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.

2.3. Putting Comments on C++ programs

- but that may.
- 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:
  - ***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
  - ***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:
  - A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.
  - Over-use of comments can lead to even less readability. A program which contains so much comment that you can hardly see the code can by no means be considered readable.
  - Use of descriptive names for variables and other entities in a program, and proper indentation of the code can reduce the need for using comments.

2.4. A brief look at functions

- In general, a function is a block of code that performs one or more actions.
- Most functions are called, or invoked, during the course of the program, which is after the program starts execution.
- Program commands are executed line by line, in the order in which they appear in the source code, until a function is reached. The program then branches off to execute the function. When the function finishes, it returns control to the line of code immediately following the call to the function.
- Functions consist of a ***header*** and a ***body***. The header, in turn is made up of the ***return type***, the ***function name***, and the ***parameters*** to that function.
- The parameters to a function allow values to be passed into the function. A parameter is a declaration of the type of value that will be passed to the function; the actual value passed by the calling function is called ***arguments***.

- Let us have an example:

```
#include<iostream>

void demoFunction()
{
	cout<<"\n demoFunction";
}

void main()
{
```

```
        cout<<"\n In main function";
        demoFunction();
        cout<<"\n Back in main function";
}
```

- Functions return either a value or void (i.e. they return nothing).
- A function that adds two integers might return the sum, and thus would be defined to return an integer value. A function that just prints a message has nothing to return and would be declared to return void.
- A function may return a value using a return statement; a return statement also causes the function to exit.
- If there is no return statement at the end of a function, the function will automatically return void.

2.5. Data types, Variables and Constants.

2.5.1. Variables.

- A variable is a place to store information in.
- A variable will have its own reserved memory location based on the declaration.
- Variables are used for holding data values so that they can be used in various computations in a program.
- All variables have three important properties:
    - ***Data Type***: a type which is established when the variable is defined (e.g. integer, real, character etc).
    - ***Name***: a name which will be used to refer to the value in the variable.
    - ***Value***: a value which can be changed by assigning a new value to the variable.

2.5.1.1. 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 the 256 characters and symbols of the ASCII and extended ASCII character sets.
- The type of variables used in C++ program are described in the next table, which lists the variable type, how much room

- You can assume they can take in memory, and the kinds of values that can be stored in the variable.

Data Types

Type	*Length*	*Range*
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
enum	6 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	32 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	$-3.4x10^{-38}$ to $3.4x10^{+38}$
double	64 bits	$-1.7x10^{-308}$ to $1.7x10^{+308}$
long double	96 bits	$-3.4x10^{-4932}$ to $1.1x10^{+4932}$
bool	8 bits	true or false (top 7 bits are ignored)

2.5.1.2. 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 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
1	0	1	0	→10
1	0	1	1	→11
1	1	0	0	→12
1	1	0	1	→13
1	1	1	0	→14
1	1	1	1	→15

Signed

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

- 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.

2.5.1.3. 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;
- The name of a variable sometimes is called an identifier
- Certain words are reserved by C++ for specific purposes and may not be used as identifiers. These are called **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**
  - A valid identifier is a sequence of one or more letters, digits or underlined symbols. The length of an identifier is not limited.
  - Neither space nor marked letters can be part of an identifier.
  - Only letters, digits and underlined characters are valid.
  - Variable identifiers should always begin with a letter or an underscore. By any means they should begin with a digit.

- Key words should not be used as names for identifiers.

2.5.1.4. Initializing Variables.

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

  DataType variable name = initial value;

  e.g. int a = 0;

2.5.1.5. Scope of Variables.

- In C++, we can declare variables any where in the source code. But we should declare a variable before using it no matter where it is written.
- ***Global variables***: can be referred any where in the code, within any function, as long as it is declared first.
- ***Local Variables***: the scope of the local variable is limited to the code level in which they are declared.

  e.g:

```
#include<iostream>
int num1;
char character;

void 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 (a block is a group of instructions grouped together within curly brackets {} signs). 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.

2.5.1.6. Defining a data type (User defined data type)

- The "typedef" Keyword is used to define a data type by the programmer.
- It is very difficult to write unsigned short int many times in your program. C++ enables you to substitute an alias for such

phrase by using the keyword typedef, which stands for type definition.

E.g.:

```
#include<iostream>
typedef unsigned short int USHORT;
void main()
{
  USHORT  width = 9;
  …
}
```

2.5.1.7. Wrapping around in signed integers.

- A signed integer differs from an unsigned integer in that half of its value is negative. When the program runs out of positive numbers, the program moves into the largest negative numbers and then counts back to zero.

  E.g.: 32767 + 1 → -32769-8 → 0

2.5.1.8. Wrapping around in unsigned integers

- When an unsigned integer reaches its maximum value, it wraps around and starts over.

  E.g.: 65535 + 1 → 0 → 65535

2.5.1.9. 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.

2.5.1.10. 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. But, constants, unlike variables do not change.
- Constants must be initialized when they are created by the program, and the program can't assign a new value to a constant later.
- C++ provides two types of constants: literal and symbolic constants.
- ***Literal constant***: is a value typed directly into the program wherever it is needed.

  E.g.: int num = 43;

  43 is a literal constant in this statement:
- ***Symbolic constant***: is a constant that is represented by a name, similar to that of a variable. But unlike a variable, its value can't be changed after initialization.

  E.g.:
  ```
  Int studentPerClass =15;
  students = classes * studentPerClass;
  ```
  studentPerClass is a symbolic constant having a value of 15.
- In C++, we have two ways to declare a symbolic constant. These are the #define and the const method.

2.5.2.1.defining constants with #define:

- The ***#define*** directive makes a simple text substitution.

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

2.5.2.2.defining constants with the key word const:

- 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;

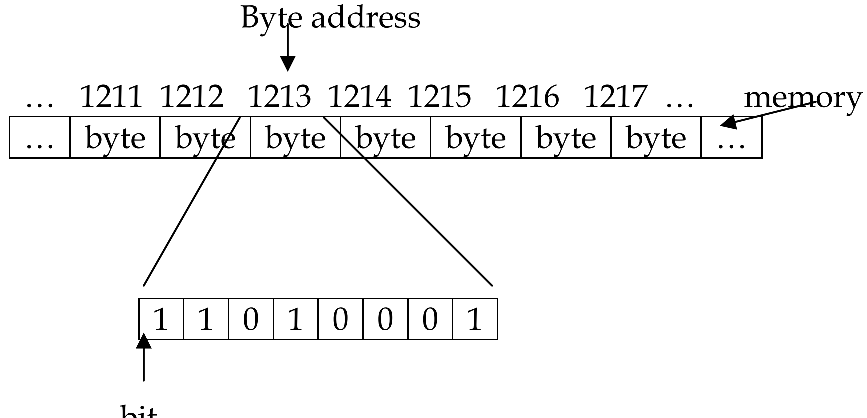
2.5.3. Enumerated constants.

- Enables programmers to define variables and restrict the value of that variable to a set of possible values which are integer.
- The enum type can not take any other datatype than int
- 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
#define WINTER   3
```
- An alternate approach using enum would be
```
enum { 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=700};

2.6. Setting aside memory/memory concept.

- A computer provides RAM for storing executable program code as well as the data the program manipulates.
- Memory can be thought of as a contiguous sequence of bits, each of which is capable of storing a binary digit. Typically, the memory is divided into groups of 8 consecutive bits called bytes.
- The bytes are sequentially addressed. Therefore, each byte can be uniquely identified by its address.



2.7. 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.
  secondsPerMinuteint constant that returns 60

  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.

2.8. Operators.

- An operator is a symbol that makes the machine to take an action.
- Operators act on operands.
- 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

2.8.1. 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.
  E.g.: x = 35; assigns the integer value 5 to variable x.
- All left values are right values, but not the other way round. Example for this is a literal.
  E.g.: x = 5; but not 5 = x;

2.8.1.1. Compound assignment 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.

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

- Except for remainder or modulo (%), all other arithmetic operators can accept a mix of integers and real operands. Generally, if both operands are integers then, the result will be an integer. However, if one or both operands are real then the result will be real.
- When both operands of the division operator (/) are integers, then the division is performed as an integer division and not the normal division we are used to.
- ***Integer division always results in an integer outcome.***
- ***Division of integer by integer will not round off to the next integer***
  E.g.:
  9/2 gives 4 not 4.5
  -9/2 gives -4 not -4.5
- To obtain a real division when both operands are integers, you should cast one of the operands to be real.
  E.g.:
  int cost = 100;
  Int volume = 80;
  Double unitPrice = cost/(double)volume;
- The ***modulo(%)*** 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

2.8.3. 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)

2.8.4. Logical Operators (!, &&, | |):

- Logical negation (!) is a unary operator, which negates the logical value of its operand. If its operand is non zero, it produce 0, and if it is 0 it produce 1.
- Logical AND (&&) produces 0 if one or both of its operands evaluate to 0 otherwise it produces 1.
- Logical OR (| |) produces 0 if both of its operands evaluate to 0 otherwise, it produces 1.
  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.
- Think about the following code:
  x / y > z | | y = = 0
- What will be the result if y is equal to 0? Easy, the result is true, many of you might say. Actually a run time error called *arithmetic exception* will result, because the expression x / y causes a problem known as *division-by-zero*.
- However, if we reverse the order to:
  y = = 0 | | x / y > z then no arithmetic exception will occur. Why? Because the test x / y > z will *not be evaluated*.

- For OR operator (||), if the left operand is evaluated to true, then the right operand *will not be evaluated*, because the whole expression is *true*, whether the value of the right operand is true or false. We call such evaluation method a *short-circuit evaluation*.
- For AND operator (&&), the right operand need not be evaluated if the left operand is evaluated to *false*, because the result will then be false whether the value of the right operand is true or false.

2.8.5. 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.

2.8.5.1. Prefix and Postfix:

- The prefix type is written before the variable. Eg (++ myAge), whereas the postfix type appears after the variable name (myAge ++).
- 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
```

2.8.6. Conditional Operator (?:)

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

  Syntax:

  operand1 ? operand2 : operand3
- First operand1 is evaluated. If the result of the evaluation is non zero, 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.1:

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

the value stored in min is 1.

E.g.2:

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

2.8.7. Comma Operator (,).

- Multiple expressions can be combined into one expression using the comma operator. The comma operator takes two operands.
- It the 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.

2.8.8. The sizeof() Operator.

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

E.g.:

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

2.8.9. Explicit type casting operators.

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

E.g.

```
int i;
float f = 3.14;
i = (int)f;  → equivalent to i = int(f);
```

then I will have a value of 3 ignoring the decimal point

2.9. 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:

<table>
<tr><td>Level</td><td>Operator</td><td>Order</td></tr>
<tr><td>Highest</td><td>++ -- (post fix)</td><td>Right to left</td></tr>
<tr><td></td><td>sizeof() ++ -- (prefix)</td><td>Right to left</td></tr>
<tr><td></td><td>* / %</td><td>Left to right</td></tr>
<tr><td></td><td>+ -</td><td>Left to right</td></tr>
<tr><td></td><td>< <= > >=</td><td>Left to right</td></tr>
<tr><td></td><td>== !=</td><td>Left to right</td></tr>
<tr><td></td><td>&&</td><td>Left to right</td></tr>
<tr><td></td><td>||</td><td>Left to right</td></tr>
<tr><td></td><td>? :</td><td>Left to right</td></tr>
<tr><td></td><td>= ,+=, -=, *=, /=,^= ,%=, &= ,|= ,<<= ,>>=</td><td>Right to left</td></tr>
<tr><td></td><td>,</td><td>Left to right</td></tr>
</table>

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.