VISUAL C++ PROGRAMMING

ISIBOR O.O.
COMPUTER SCIENCE DEPARTMENT
LAGOS CITY POLYTECHNIC, IKEJA.
For: SOUTHWESTERN UNIVERSITY
OKUN-OWA, NIGERIA.
osesisibor@gmail.com, 08063546421.
JANUARY, 2019

CONTENT

1

INTRODUCTION TO PROGRAMMING

Programming is breaking a task down into small steps. It is a series of instructions to a computer to accomplish a task. Instructions must be written in a way the computer can understand and programming languages are used to write programs

Activities involved in programming includes:

- Write a program.
- Compile the program.
- Run the program.
- Debug the program.
- Repeat the whole process until the program is finished.

Common Terms

Computer Program: This is a list of instructions written in a special code, or language. It is a series of instructions that makes a computer work.

Computer program tells the computer which operations to perform, and in what sequence to perform them. It can be written in variety of programming languages. Software refers to programs that make the computer perform some task.

A program is a set of instructions that tells the computer what to do or written to perform a specific task by the computer.

Computer Language or Programming Language is a language that is acceptable to a computer system.

Programming or Coding: This is the process of creating a sequence of instructions in such a language defined above.

Software: This is a set of large program.

To develop software, one must have knowledge of a programming language and before moving on to any programming language, it is important to know about the various types of languages used by the computer.

COMPUTER PROGRAMMING LANGUAGES

Languages are a means of communication. Normally people interact with each other through a language. On the same pattern, communication with computers is carried out through a language. This language is understood both by the user and the machine. Just as every language like English, Hindi has its own grammatical rules; every computer language is also bounded by rules known as

syntax of that language. The user is bound by that syntax while communicating with the computer system.

Computer languages are broadly classified as:

A. Low Level Language: The term low level highlights the fact that it is closer to a language which the machine understands.

The low level languages are classified as:

- **Machine Language**: This is the language (in the form of 0's and 1's, called binary numbers) understood directly by the computer. It is machine dependent. It is difficult to learn and even more difficult to write programs.
- **Assembly Language**: This is the language where the machine codes comprising of 0'sand 1's are substituted by symbolic codes (called mnemonics) to improve their understanding. It is the first step to improve programming structure. Assembly language programming is simpler and less time consuming than machine level programming, it is easier to locate and correct errors in assembly language than in machine language programs. It is also machine dependent. Programmers must have knowledge of the machine on which the program will run.

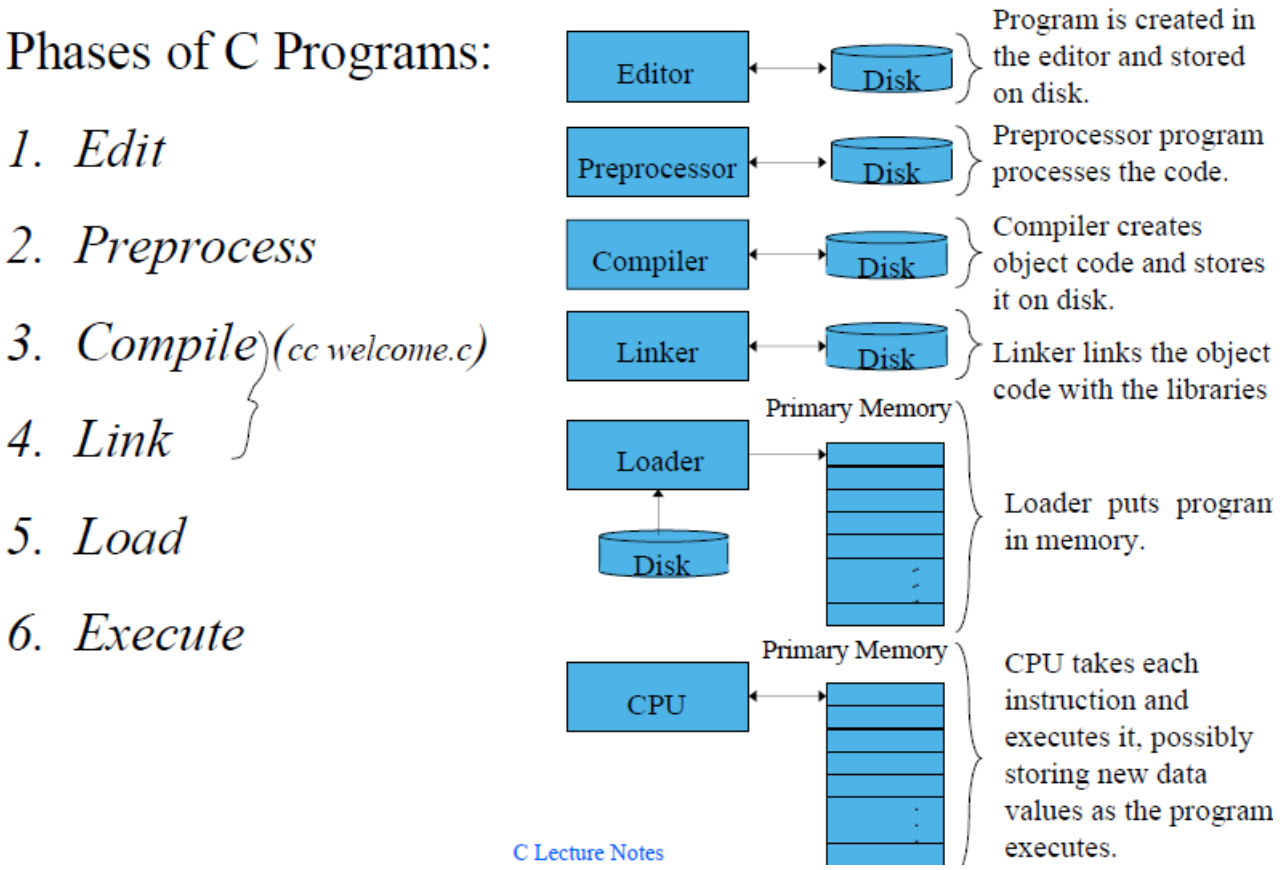
B. High Level Language*:* Low level language requires extensive knowledge of the hardware since it is machine dependent. To overcome this limitation, high level language has been evolved which uses normal English, which is easy to understand to solve any problem. High level languages are computer independent and programming becomes quite easy and simple. Various high level languages are given below:

- ✓ BASIC (Beginners All Purpose Symbolic Instruction Code): It is widely used, easy to learn general purpose language. Mainly used in microcomputers in earlier days.
- ✓ COBOL (Common Business Oriented language): A standardized language used for commercial applications.
- ✓ FORTRAN (Formula Translation): Developed for solving mathematical and scientific problems. One of the most popular languages among scientific community.
- ✓ C: Structured Programming Language used for all purpose such as scientific application, commercial application, developing games etc.
- ✓ C++: Popular object oriented programming language, used for general purpose. Etc.

PROGRAMMING LANGUAGE TRANSLATORS

As you know that high level language is machine independent and assembly language though it is machine dependent yet mnemonics that are being used to represent instructions are not directly understandable by the machine. Hence to make the machine understand the instructions provided by both the languages, programming language instructors are used. They transform the instruction prepared by programmers into a form which can be interpreted & executed by the computer. Flowing are the various tools to achieve this purpose:

- **Compiler**: The software that reads a program written in high level language and translates it into an equivalent program in machine language is called as compiler. The program written by the programmer in high level language is called source program and the program generated by the compiler after translation is called as object program. translate the source code to machine code to be executed
- **Interpreter**: it also executes instructions written in a high level language. Both complier & interpreter have the same goal i.e. to convert high level language into binary instructions, but their method of execution is different. The complier converts the entire source code into machine level program, while the interpreter takes 1 statement, translates it, executes it & then again takes the next statement. reads a little source code, translates it to machine code, and executes it, then reads a little more, etc.
- **Assembler**: The software that reads a program written in assembly language and translates it into an equivalent program in machine language is called as assembler.
- **Linker**: A linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file. connects code from libraries with your code to make one executable



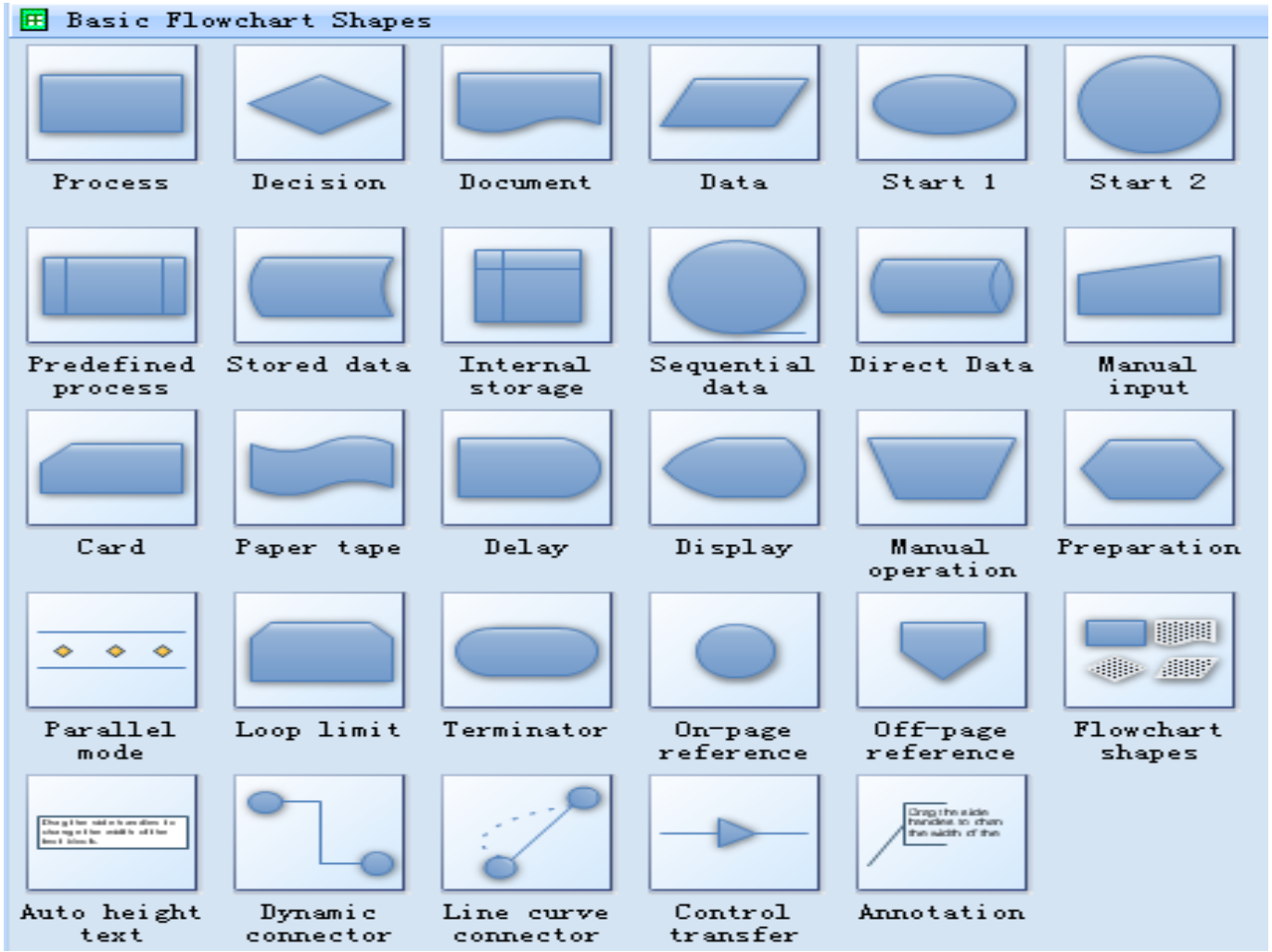
Few terms related to programming:

- **Programming paradigm**: a pattern that serves as a school of thoughts for programming of computers
- **Programming technique**: related to an algorithmic idea for solving a particular class of problems. For example," divide and conquer" or" program development by stepwise refinement"
- **Programming style**: the way we express ourselves in a computer program. Relates to elegance (or lack of elegance)
- **Programming culture**: The totality of programming behavior, which often is tightly related to a family of programming languages

ALGORITHMS: comes in **Pseudocodes and Flowcharts**

An algorithm is a finite sequence of steps, each step taking a finite length of time, that solves a problem or computes a result. A computer program is one example of an algorithm. In both of these examples, the order of the steps matter.

A flowchart is a graphical representation of an algorithm. Flowcharts are drawn using special-purpose symbols, such as ovals, rectangles, diamonds and small circles. These symbols are connected by arrows called ***flow lines***.

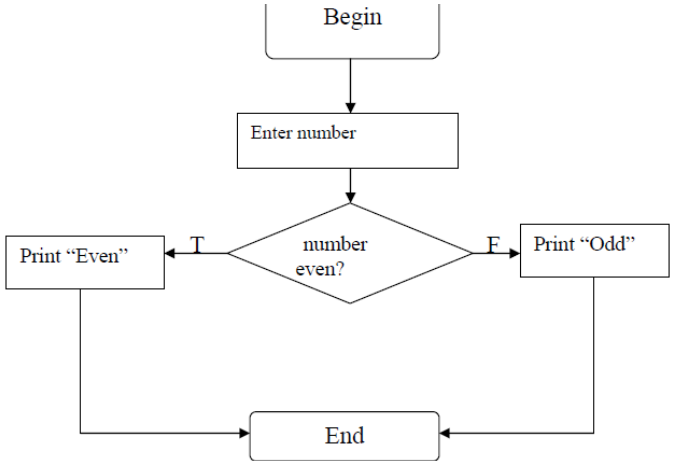


Example: -Write a program that reads a number and prints out "even" if the number is even or "odd" if the number is odd.

Pseudocode:

Input number
If number is even
Then Print "even"
Else print "odd"

Flowchart:



2

LEARNING WINDOWS PROGRAMMING

There are always two basic aspects to interactive applications executing under Windows: You need code to create the graphical user interface (GUI) with which the user interacts, and you need code to process these interactions to provide the functionality of the application. Visual C++ 2010 provides you with a great deal of assistance in both aspects of Windows application development.

Console Applications

As well as developing Windows applications, Visual C++ 2010 also enables you to write, compile, and test C++ programs that have none of the baggage required for Windows programs — that is, applications that are essentially character-based, command-line programs. These programs are called console applications in Visual C++ 2010 because you communicate with them through the keyboard and the screen in character mode.

When you write console applications it might seem as if you are being sidetracked from the main objective of Windows programming, but when it comes to learning C++ (which you do need to do before embarking on Windows-specific programming) it's the best way to proceed.

Windows Programming Concepts

The project creation facilities provided with Visual C++ 2010 can generate skeleton code for a wide variety of native C++ application programs automatically, including basic Windows programs. For Windows applications that you develop for the CLR you get even more automatic code generation. You can create complete applications using Windows Forms that require only a small amount of customized code to be written by you, and some that require no additional code at all. Creating a project is the starting point for all applications and components that you develop with Visual C++ 2010.

A Windows program, whether a native C++ program or a program written for the CLR, has a different structure from that of the typical console program you execute from the command line, and it's more complicated. In a console program you can get input from the keyboard and write output back to the command line directly, whereas a Windows program can access the input and output facilities of the computer only by way of functions supplied by the host environment; no direct access to the hardware resources is permitted. Because several programs can be active at one time under Windows, Windows has to determine which application a given raw input, such as a mouse click or the pressing of a key on the keyboard, is destined for, and signal the program concerned accordingly. Thus, the Windows operating system has primary control of all communications with the user.

What Is the Integrated Development Environment?

The integrated development environment (IDE) that comes with Visual C++ 2010 is a completely self-contained environment for creating, compiling, linking, and testing your C++ programs. It also happens to be a great environment in which to learn C++ (particularly when combined with a great book). Visual C++ 2010 incorporates a range of fully integrated tools designed to make the whole process of writing C++ programs easy.

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Examples of IDEs for C++ include Microsoft's Visual Studio 2013, the Eclipse Foundation's Eclipse CDT, and Apple's XCode.

The fundamental parts of Visual C++ 2010

The fundamental parts of Visual C++ 2010, provided as part of the IDE, are the **editor, the compiler, the linker, and the libraries**. These are the basic tools that are essential to writing and executing a C++ program. Their functions are as follows.

1. **The Editor:** The editor provides an interactive environment in which to create and edit C++ source code. As well as the usual facilities, such as cut and paste, which you are certainly already familiar with, the editor also provides color cues to differentiate between various language elements. The editor automatically recognizes fundamental words in the C++ language and assigns a color to them according to what they are. This not only helps to make your code more readable, but also provides a clear indicator of when you make errors in keying such words.
2. **The Compiler:** The compiler converts your source code into object code, and detects and reports errors in the compilation process. The compiler can detect a wide range of errors caused by invalid or unrecognized program code, as well as structural errors, such as parts of a program that can never be executed. The object code output from the compiler is stored in files called object files that have names with the extension .obj.
3. **The Linker:** The linker combines the various modules generated by the compiler from source code files, adds required code modules from program libraries supplied as part of C++, and welds everything into an executable whole. The linker can also detect and report errors — for example, if part of your program is missing, or a nonexistent library component is referenced.
4. **The Libraries:** A library is simply a collection of prewritten routines that supports and extends the C++ language by providing standard professionally-produced code units that you can incorporate into your programs to carry out common operations. The operations implemented by routines in the various libraries provided by Visual C++ 2010 greatly enhance productivity by saving you the effort of writing and testing the code for such operations yourself.
5. **Debuggers:** A debugger allows a programmer to more easily trace a program's execution in order to locate and correct errors in the program's implementation. With a debugger, a developer can simultaneously run a program and see which line in the source code is responsible for the program's current actions. The programmer can watch the values of variables and other program

elements to see if their values change as expected. Debuggers are valuable for locating errors (also called bugs) and repairing programs that contain errors.

Profilers: A profiler collects statistics about a program's execution allowing developers to tune appropriate parts of the program to improve its overall performance. A profiler indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Profilers also can be used for testing purposes to ensure all the code in a program is actually being used somewhere during testing. This is known as coverage. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing. The main purpose of profiling is to find the parts of a program that can be improved to make the program run faster.

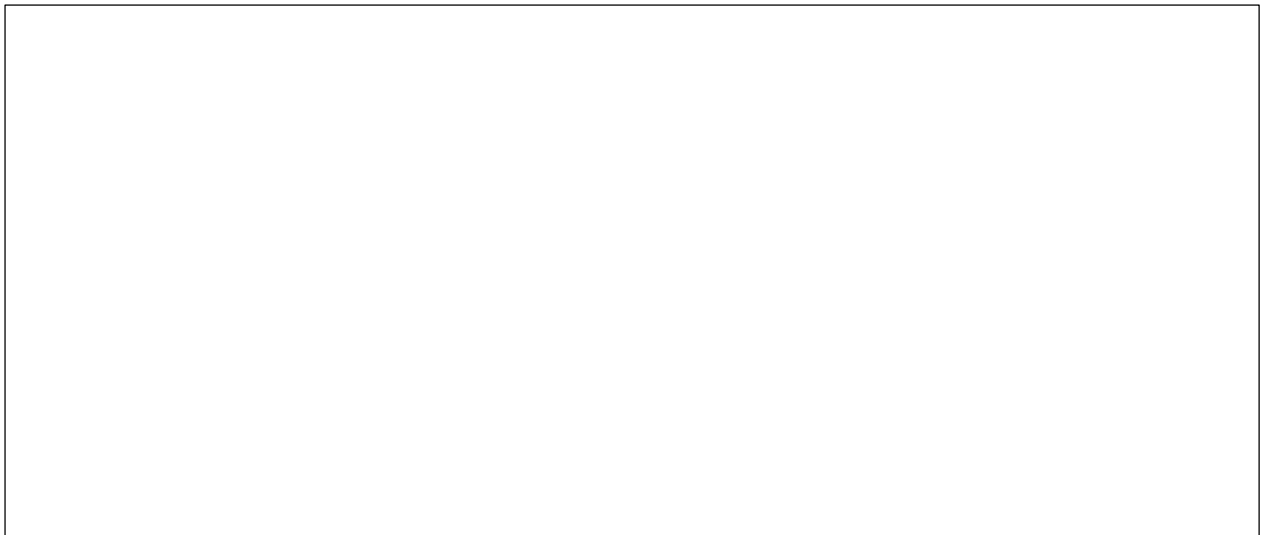
Computer languages have undergone dramatic evolution since the first electronic computers were built to assist in telemetry calculations during World War II. Early on, programmers worked with the most primitive computer instructions: machine language. These instructions were represented by long strings of ones and zeroes. Soon, assemblers were invented to map machine instructions to human readable and -manageable mnemonics, such as ADD and MOV.

In time, higher-level languages evolved, such as BASIC and COBOL. These languages let people work with something approximating words and sentences, such as Let I = 100. **These instructions** were translated back into machine language by **interpreters and compilers**. **An interpreter** translates a program as it reads it, turning the program instructions, or code, directly into actions. **A compiler translates** the code into an intermediary form. This step is called compiling, and produces an object file. **The compiler** then invokes a linker, which turns the object file into an executable program.

Because interpreters read the code as it is written and execute the code on the spot, interpreters are easy for the programmer to work with. Compilers, however, introduce the extra steps of compiling and linking the code, which is inconvenient. Compilers produce a program that is very fast each time it is run. However, the time-consuming task of translating the source code into machine language has already been accomplished.

Using the IDE

All program development and execution in this book is performed from within the IDE. When you start Visual C++ 2010 you'll notice an application window similar to that shown below:



Solution Explorer window enables you to navigate through your program files and display their contents in the Editor window, and to add new files to your program. **The Solution Explorer** pane presents a view of all the projects in the current solution and the files they contain — here, of course, there is just one project. You can display the contents of any file as an additional tab in the Editor pane just by double-clicking the name in the Solution Explorer tab. In the Editor pane, you can switch instantly to any of the files that have been displayed just by clicking on the appropriate tab.

The Class View tab displays the classes defined in your project and also shows the contents of each class. You don't have any classes in this application, so the view is empty. When I discuss classes you will see that you can use the Class View tab to move around the code relating to the definition and implementation of all your application classes quickly and easily.

The Property Manager tab shows the properties that have been set for the Debug and Release versions of your project. I'll explain these versions a little later in this chapter. You can change any of the properties shown by right-clicking a property and selecting Properties from the context menu; this displays a dialog box where you can set the project property. You can also press Alt+F7 to display the properties dialog box at any time. I'll discuss this in more detail when we go into the Debug and Release versions of a program.

The Editor window is where you enter and modify source code and other components of your application.

The Output window displays the output from build operations in which a project is compiled and linked.

Dockable Toolbars: A dockable toolbar is one that you can move around to position it at a convenient place in the window. You can arrange for any of the toolbars to be docked at any of the four sides of the application window. If you right click in the toolbar area and select Customize from the pop-up, the Customize dialog will be displayed. You can choose where a particular toolbar is docked by selecting it and clicking the Modify Selection button. You can then choose from the drop-down list to dock the toolbar where you want.

Documentation: There will be plenty of occasions when you'll want to find out more information about Visual C++ 2010 and its features and options. Press Ctrl+Alt+F1 to access the product

documentation. The Help menu also provides various routes into the documentation, as well as access to program samples and technical support.

PROJECTS AND SOLUTIONS

A project is a container for all the things that make up a program of some kind and possesses the following **characteristics**;

i. It might be a console program, a window-based program, or some other kind of program
ii. It usually consists of one or more source files containing your code,
iii. Contains possibly other files containing auxiliary data.
iv. All the files for a project are stored in the project folder;
v. Detailed information about the project is stored in an XML file with the extension .vcxproj**,** also in the project folder.
vi. The project folder also contains other folders that are used to store the output from compiling and linking your project.

The idea of a **Solution** is expressed by its name, in that it is a mechanism for bringing together all the programs and other resources that represent a solution to a particular data-processing problem. For example, a distributed order-entry system for a business operation might be composed of several different programs that could each be developed as a project within a single solution; therefore, **a solution** is a folder in which all the information relating to one or more projects is stored, and one or more project folders are subfolders of the solution folder. Its features include:

i. Information about the projects in a solution is stored in two files with the extensions **.sln** and **.suo**, respectively.
ii. When you create a project a new solution is created automatically unless you elect to add the project to an existing solution.
iii. When you create a project along with a solution, you can add further projects to the same solution. You can add any kind of project to an existing solution, but you will usually add only a project related in some way to the existing project, or projects, in the solution. Generally, unless you have a good reason to do otherwise, each of your projects should have its own solution.

DEFINING A PROJECT IN C++

The first step in writing a Visual C++ 2010 program is to create a project for it using the File | New | Project menu option from the main menu or by pressing Ctrl+Shift+N; you can also simply click New Project . . . on the Start page. As well as containing files that define all the code and any other data that makes up your program, the project XML file in the project folder also records the Visual C++ 2010 options you're using.

Steps to display line numbers on the visual C++ 2010 text editor.

If the line numbers are not displayed on your system, select Tools | Options from the main menu to display the Options dialog box. If you extend the C/C++ option in the Text Editor subtree in the left pane and select General from the extended tree, you can select Line Numbers in the right pane of the dialog box.

```
1. //BASIC STRUCTURE OF A C++ PROGRAM DISPLAYING HELLO WORLD
2. #include "stdafx.h"
3. #include <iostream>
4. Using namespace std;
5. int _tmain()
6. {
7. cout << "Hello world!\n";
8. return 0;
9. }
```

The line1 is just a comment. Anything following "//" in a line is ignored by the compiler. When you want to add descriptive comments in a line, precede your text with "//".

Line 2 is an #include directive that adds the contents of the file **stdafx.h** to this file in place of this #include directive. This is the standard way to add the contents of .h source files to a .cpp source file in a C++ program.

Examples of C++ Pre-processor commands. (ANY FIVE)

#include "stdafx.h"	#include <string>	#include <iostream>
#include <vector>	#include "myheader.h"	#include <iomanip>
#include <ctime>	#include <algorithm>	#include <cstdlib>
#include <complex>		

Line 4: **using namespace std**; The two items our program needs to display a message on the screen, cout and endl, have longer names: std::cout and std::endl. This using namespace std directive allows us to omit the std:: prefix and use their shorter names. This directive is optional, but if we omit it, we must use the longer names. The name **std** stands for "standard," and the using namespace std line indicates that some of the names we use in our program are part of the so-called "standard namespace."

Two examples of C++ namespace.

```
using std::vector;
using std::string;
using std::cout;
using std::endl;
using namespace std;
using std::uniform_int_distribution;
using std::random_device;
```

using std::mt19937;

Line 5 is the first line of the executable code in this file and the beginning of the function **_tmain().** A function is simply a named unit of executable code in a C++ program; every C++ program consists of at least one — and usually many more — functions.

Line 7: The body of our main function contains only one statement. This statement directs the executing program to print the message This is a simple C++ program! on the screen. A statement is the fundamental unit of execution in a C++ program. Functions contain statements that the compiler translates into executable machine language instructions. C++ has a variety of different kinds of statements, and the chapters that follow explore these various kinds of statements. All statements in C++ end with a semicolon (;).

Lines 6 and 9 contain left and right braces, respectively, that enclose all the executable code in the function_tmain(). The executable code is, therefore, just the single line 7, and all this does is end the program.

The **iostream** library defines facilities for basic I/O operations, and the one you are using in the second line that you added writes output to the command line. std::cout is the name of the standard output stream, and you write the string "Hello world!\n" to std::cout in the second addition statement. Whatever appears between the pair of double-quote characters is written to the command line.

Building the Solution

To build the solution, press F7 or select the Build | Build Solution menu item. Alternatively, you can click the toolbar button corresponding to this menu item. The toolbar buttons for the Build menu may not be displayed, but you can easily fix this by right-clicking in the toolbar area and selecting the Build toolbar from those in the list. The program should then compile successfully. If there are errors, it may be that you created them while entering the new code, so check the two new lines very carefully.

VARIABLES

```
#include <iostream>
using namespace std;
int main() {
int x;
x = 10;
cout << x << endl;
}
```

VARIABLE is a name given to a storage area that our programs can manipulate.

- **Signed variables:** These variables carry number values that can carry both positive and negative numbers meaning they do allow both a positive and a negative sign
- **Un-Signed variables:** These variables carry number values that exclude negative numbers meaning they do not allow a negative sign.

C++ IDENTIFIERS

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9). C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.

VARIABLE DECLARATION refers to the process of assigning a data type to a variable.

int x; above is a declaration statement. All variables in a C++ program must be declared. A declaration specifies the type of a variable. The word **int** indicates that the variable is an integer. The name of the integer variable is x. We say that variable x has type int. The compiler uses the declaration to reserve the proper amount of memory to store the variable's value. The declaration enables the compiler to verify the programmer is using the variable properly within the program; for example, we will see that integers can be added together just like in mathematics. For some other data types, however, addition is not possible and so is not allowed. The compiler can ensure that a variable involved in an addition operation is compatible with addition. It can report an error if it is not.

Multiple variables of the same type can be declared and, if desired, initialized in a single statement. The following statements declare three variables in one declaration statement:

```
int x, y, z;
```

The following statement declares three integer variables and initializes two of them:

```
int x = 0, y, z = 5;
```

Here y's value is undefined. The declarations may be split up into multiple declaration statements:

```
int x = 0;
int y;
int z = 5;
```

In the case of multiple declaration statements, the type name (here int) must appear in each statement.

Some programming languages do not require programmers to declare variables before they are used; the type of a variable is determined by how the variable is used. Some languages allow the same variable to assume different types as its use differs in different parts of a program. Such languages are known as **dynamically-typed languages**. C++ is a **statically-typed language**. In a statically-typed language, the type of a variable must be explicitly specified before it is used by statements in a program. While the requirement to declare all variables may initially seem like a minor annoyance, it offers several advantages:

• When variables must be declared, the compiler can catch typographical errors that dynamically-typed
languages cannot detect.

VARIABLE INITIALIZATION refers to the process of assigning a value to a declared variable.
x = 10;
This is an **assignment statement**. An assignment statement associates a value with a variable. The key to an assignment statement is the symbol = which is known as the assignment operator. Here the value 10 is being assigned to the variable x. This means the value 10 will be stored in the memory location the compiler has reserved for the variable named x. We need not be concerned about where the variable is stored in memory; the compiler takes care of that detail.
The meaning of the **assignment operator (=)** is different from equality in mathematics. In mathematics, = asserts that the expression on its left is equal to the expression on its right. In C++, = makes the variable on its left take on the value of the expression on its right. It is best to read x = 5 as "x is assigned the value 5," or "x gets the value 5."

Character Sets Variable names in C++ can be formed from:

1. letters A–z (upper- or lowercase),
2. The digits 0–9, and
3. The underscore character. No other characters are allowed,

Rules for forming valid variable names (Identifier) in C++.

1. Variable names can include the letters A-z (upper or lower case), the digits 0-9 and the underscore character
2. Variable names must also begin with either a letter or an underscore.
3. You can use variable names that begin with an underscore.
4. Blanks are not allowed in variable names
5. The use of keywords/ reserved words is not allowed.
6. C++ is a **case-sensitive language**. This means that capitalization matters. variable called **Name** is different from the variable called **name**.

Here are some examples of valid and invalid identifiers:
All of the following words are valid identifiers and so qualify as variable names:

x, **x2**, **total**, **port_22**, and **FLAG**.

None of the following words are valid identifiers:

sub-total (dash is not a legal symbol in an identifier),
first entry (space is not a legal symbol in an identifier),
4all (begins with a digit),
#2 (pound sign is not a legal symbol in an identifier), and
class (**class** is a reserved word).

VARIABLE SCOPE: This is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed.

Types of variable scope

1. **LOCAL VARIABLE SCOPE**: These are variables that are declared inside a subprogram or block. They can be used only by statements that are inside that subprogram or block of code.
2. **GLOBAL VARIABLE SCOPE**: These variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

RESERVED WORDS/KEYWORDS

These words are special and are used to define the structure of C++ programs and statements. These words provide commands to the compiler of the language. The keywords or reserved words also provide the tools needed to solve your programmable problems. C++ reserved words are listed below:

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

COMMENTS

Good programmers annotate their code by inserting remarks that explain the purpose of a section of code or why they chose to write a section of code the way they did. These notes are meant for human readers, not the compiler. It is common in industry for programs to be reviewed for correctness by other programmers or technical managers. These are important parts of any program, they're not executable code — they are there simply there to help the human reader. All comments are ignored by the compiler. A different programmer may be required to finish or fix a part of the program written by someone else. Well-written comments can help others understand new code quicker and increase their productivity modifying old or unfinished code. While it may seem difficult to believe, even the same programmer working on her own code months later can have a difficult time remembering what various parts do. Comments can help greatly.

Types of Comment

- SINGLE LINE COMMENT: The comment using // covers only the portion of the line following the two successive slashes. E.g.

// a simple C++ code to display the Firstname and Lastname receive from a user.

- MULTIPLE LINE/BLOCK COMMENT: **/*...*/** form defines whatever is enclosed between the /* and the */ as a comment, and this can span several lines.

DATA TYPE

This is a classification identifying one of various types of data that determines the possible values of that type and the operations that can be carried out on the values of that type.

Fundamental data types in C++

- **Integer:** they can have only values that are whole numbers. We can declare integer variables using the keyword int.

The type **short int**, which may be written as just **short**, represents integers that may occupy fewer bytes of memory than the **int** type. If the **short** type occupies less memory, it necessarily must represent a smaller range of integer values than the **int** type. The C++ standard does not require the **short** type to be smaller than the **int** type; in fact, they may represent the same set of integer values. The **long int** type, which may be written as just **long**, may occupy more storage than the **int** type and thus be able to represent a larger range of values. Again, the standard does not require the **long** type to be bigger then the **int** type. Finally, the **long long int** type, or just **long long**, may be larger than a **long**. The C++ standard guarantees the following relative ranges of values hold:

short int ≤ **int** ≤ **long int** ≤ **long long int**

Characteristics of Visual C++ Integer Types are shown below:

Type Name	Short Name	Storage	Smallest Magnitude	Largest Magnitude
short int	short	2 bytes	−32,768	32,767
int	int	4 bytes	−2,147,483,648	2,147,483,647
long int	long	4 bytes	−2,147,483,648	2,147,483,647
long long int	long long	8 bytes	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned short	unsigned short	2 bytes	0	65,535
unsigned int	unsigned	4 bytes	0	4,294,967,295
unsigned long int	unsigned long	4 bytes	0	4,294,967,295
unsigned long long int	unsigned long long	8 bytes	0	18,446,744,073,709,551,615

- **Floating-Point Types:** Floating-point numbers are an approximation of mathematical real numbers. Values that aren't integral are stored as floating-point numbers. A floating-point number can be expressed as a decimal value such as 112.5, or with an exponent such as 1.125E2 where the decimal part is multiplied by the power of 10 specified after the E (for Exponent). Our example is, therefore, 1.125×10^2, which is 112.5. The type double is used more often, since it stands for "double-precision floating-point," and it can represent a wider range of values with more digits of precision. The float type represents single-precision floating-point values that are less precise.

Characteristics of Floating-point Numbers on 32-bit Computer Systems are shown below

Type	Storage	Smallest Magnitude	Largest Magnitude	Minimum Precision
float	4 bytes	1.17549×10^{-38}	$3.40282 \times 10^{+38}$	6 digits
double	8 bytes	2.22507×10^{-308}	$1.79769 \times 10^{+308}$	15 digits
long double	8 bytes	2.22507×10^{-308}	$1.79769 \times 10^{+308}$	15 digits

- **Character Data Types:** The char data type is used to represent single characters: letters of the alphabet (both upper and lower case), digits, punctuation, and control characters (like newline and tab characters). Most systems support the American Standard Code for Information Interchange (ASCII) character set. Standard ASCII can represent 128 different characters. The char data type serves a dual purpose. It specifies a one-byte variable that you can use either to store integers within a given range, or to store the code for a single ASCII character, which is the American Standard Code for Information Interchange. You can declare a char variable with this statement:

  char letter = 'A';

  In C++ source code, characters are enclosed by single quotes ('), as in char ch = 'A';
  Standard (double) quotes (") are reserved for strings, which are composed of characters, but strings and chars are very different. Below is ASCII codes for characters

0	'\0'	16	—	32	' '	48	'0'	64	@@	80	'P'	96	'`'	112	'p'
1	—	17	—	33	'!'	49	'1'	65	'A'	81	'Q'	97	'a'	113	'q'
2	—	18	—	34	'"'	50	'2'	66	'B'	82	'R'	98	'b'	114	'r'
3	—	19	—	35	'#'	51	'3'	67	'C'	83	'S'	99	'c'	115	's'
4	—	20	—	36	'$'	52	'4'	68	'D'	84	'T'	100	'd'	116	't'
5	—	21	—	37	'%'	53	'5'	69	'E'	85	'U'	101	'e'	117	'u'
6	—	22	—	38	&	54	'6'	70	'F'	86	'V'	102	'f'	118	'v'
7	'\a'	23	—	39	'\''	55	'7'	71	'G'	87	'W'	103	'g'	119	'w'
8	'\b'	24	—	40	'('	56	'8'	72	'H'	88	'X'	104	'h'	120	'x'
9	'\t'	25	—	41	')'	57	'9'	73	'I'	89	'Y'	105	'i'	121	'y'
10	'\n'	26	—	42	'*'	58	':'	74	'J'	90	'Z'	106	'j'	122	'z'
11	—	27	—	43	'+'	59	';'	75	'K'	91	'['	107	'k'	123	'{'
12	'\f'	28	—	44	','	60	'<'	76	'L'	92	'\\'	108	'l'	124	'|'
13	'\r'	29	—	45	'-'	61	'='	77	'M'	93	']'	109	'm'	125	'}'
14	—	30	—	46	'.'	62	'>'	78	'N'	94	'^'	110	'n'	126	'~'
15	—	31	—	47	'/'	63	'?'	79	'O'	95	'_'	111	'o'	127	—

```
#include <iostream>
using namespace std;
int main() {
char ch1, ch2;
ch1 = 65;
ch2 = 'A';
cout << ch1 << ", " << ch2 << ", " << 'A' << endl;
}
```

Some characters are non-printable characters. The ASCII chart lists several common non-printable characters:

- **'\n'**—the newline character
- **'\r'**—the carriage return character
- **'\b'**—the backspace character
- **'\a'**—the "alert" character (causes a "beep" sound on many systems)
- **'\t'**—the tab character
- **'\f'**—the formfeed character
- **'\0'**—the null character (used in C strings, see Section D.7)

These special non-printable characters begin with a backslash (\) symbol. The backslash is called
an escape symbol, and it signifies that the symbol that follows has a special meaning and should not be interpreted literally. This means the literal backslash character must be represented as two backslashes:
'\\'.
These special non-printable character codes can be embedded within strings. To embed a backslash within a string, you must escape it; for example, the statement
cout << "C:\\Dev\\cppcode" << endl;
would print C:\Dev\cppcode

The following two statements behave identically on most computer systems:
cout << "End of line" << endl;
cout << "End of line\n";

- **The Boolean Type:** they can have only two values: a value called true and a value called false. The type for a logical variable is bool, named after George Boole, who developed Boolean algebra, and type bool is regarded as an integer type. Boolean variables are also referred to as logical variables.

Constants
Constants to the degree of precision to which they have been measured and/or calculated, they do not vary. C++ supports named constants. Constants are declared like variables with the addition of the **const** keyword:

const double PI = 3.14159;

Once declared and initialized, a constant can be used like a variable in all but one way—a constant may not be reassigned. It is illegal for a constant to appear on the left side of the assignment operator (=) outside its declaration statement.

Enumerated Types
C++ allows a programmer to create a new, very simple type and list all the possible values of that type. Such a type is called an enumerated type, or an enumeration type. The **enum** keyword introduces an enumerated type:
enum Color { Red, Orange, Yellow, Green, Blue, Violet };

Here, the new type is named **Color**, and a variable of type **Color** may assume one of the values that appears in the list of values within the curly braces. The semicolon following the close curly brace is required. Sometimes the enumerated type definition is formatted as

```
enum Color {
        Red,
        Orange,
        Yellow,
        Green,
        Blue,
        Violet
        };
```

but the formatting makes no different to the compiler. The values listed with the curly braces constitute all the values that a variable of the enumerated type can attain. The name of each value of an enumerated type must be a valid C++ identifier. Given the **Color** type defined as above, we can declare and use variables of the **enum** type as shown by the following code fragment:

```
Color myColor;
myColor = Orange;
```

Here the variable **myColor** has our custom type **Color**, and its value is **Orange**. It is illegal to reuse an enumerated value name within another enumerated type within the same program. In the following code, the enumerated value **Light** is used in both **Shade** and **Weight**:

```
enum Shade { Dark, Dim, Light, Bright };
enum Weight { Light, Medium, Heavy };
```

These two enumerated types are incompatible because they share the value **Light**, and so the compiler will issue an error.

The value names within an **enum** type must be unique. The convention in C++ is to captialize the first letter of an **enum** type and its associated values, although the language does not enforce this convention.

Type Inference with auto

C++ requires that a variable be declared before it is used. Ordinarily this means specifying the variable's type, as in

```
int count;
char ch;
double limit;
```

A variable may be initialized when it is declared:

```
int count = 0;
char ch = 'Z';
double limit = 100.0;
```

Each of the values has a type: **0** is an **int**, **'Z'** is a **char**, and **0.0** is a **double**. The **auto** keyword allows the compiler to automatically deduce the type of a variable if it is initialized when it is declared:

```
auto count = 0;
auto ch = 'Z';
auto limit = 100.0;
```

The **auto** keyword may not be used without an accompanying initialization; for example, the following declaration is illegal:

```
auto x;
```

because the compiler cannot deduce **x**'s type.

RECEIVING INPUTS FOM A USER

```
#include <iostream>
using namespace std;
int main() {
        int value1, value2, sum;
        cout << "Please enter two integer values: ";
        cin >> value1 >> value2;
        sum = value1 + value2;
        cout << value1 << " + " << value2 << " = " << sum << endl;
}
```

• **int value1, value2, sum;**
This statement declares three integer variables, but it does not initialize them. As we examine the rest of the program we will see that it would be superfluous to assign values to the variables here.
• **cout << "Please enter two integer values: ";**
This statement prompts the user to enter some information. This statement is our usual print statement, but it is not terminated with the end-of-line marker **endl**. This is because we want the cursor to remain at the end of the printed line so when the user types in values they appear on the same line as the message prompting for the values. When the user presses the enter key to complete the input, the cursor will automatically move down to the next line.
• **cin >> value1 >> value2;**
This statement causes the program's execution to stop until the user types two numbers on the keyboard and then presses enter. The first number entered will be assigned to **value1**, and the second number entered will be assigned to **value2**. Once the user presses the enter key, the value entered is assigned to the variable. The user may choose to type one number, press enter, type the second number, and press enter again. Instead, the user may enter both numbers separated by one of more spaces and then press enter only once. The program will not proceed until the user enters two numbers.

The **cin** input stream object can assign values to multiple variables in one statement, as shown here:

```
int num1, num2, num3;
cin >> num1 >> num2 >> num3;
```

cin is a object that can be used to read input from the user. The >> operator—as used here in the context of the **cin** object—is known as the extraction operator. Notice that it is "backwards" from the << operator used with the **cout** object. The **cin** object represents the input stream—information flowing into the program from user input from the keyboard. The >> operator extracts the data from the input stream **cin** and assigns the pieces of the data, in order, to the various variables on its right.

OPERATOR AND OPERATOR PRECEDENCE

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

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators

The following are arithmetic operators supported by C++ language:

Operator	Description	Example
+	Adds two operands	A + B
-	Subtracts second operand from the first	A - B
*	Multiplies both operands	A * B
/	Divides numerator by denumerator	B / A
%	Modulus Operator and remainder of after an integer division	B % A
++	Increment operator, increases integer value by one	A++
--	Decrement operator, decreases integer value by one	A--

Relational Operators

The following are relational operators supported by C++ language:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B)
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B)
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B)
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B)
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true	(A >= B)
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B)

Logical Operators

The following are logical operators supported by C++ language.

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

Assignment Operators

The following are assignment operators supported by C++ language:

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

Operator Precedence and Associativity

When different operators are used in the same expression, the normal rules of arithmetic apply. All C++ operators have a precedence and associativity:

• **Precedence**—when an expression contains two different kinds of operators, which should be applied first?

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:
For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.
• **Associativity**—when an expression contains two operators with the same precedence, which should be applied first?

Arity	Operators	Associativity
Unary	+, –	
Binary	*, /, %	Left
Binary	+, –	Left
Binary	=	Right

Standard Math Functions
The **cmath** library provides much of the functionality of a scientific calculator. Table 8.1 lists only a few of the available functions.

mathfunctions Module
`double sqrt(double x)` Computes the square root of a number: `sqrt(x)` = $\sqrt{x}$
`double exp(double x)` Computes e raised a power: `exp(x)` = e^x
`double log(double x)` Computes the natural logarithm of a number: `log(x)` = $\log_e x = \ln x$
`double log10(double x)` Computes the common logarithm of a number: `log(x)` = $\log_{10} x$
`double cos(double)` Computes the cosine of a value specified in radians: `cos(x)` = $\cos x$; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent
`double pow(double x, double y)` Raises one number to a power of another: `pow(x,y)` = x^y
`double fabs(double x)` Computes the absolute value of a number: `fabs(x)` = $\|x\|$

// a simple C++ code to display the Firstname and Lastname receive from a user.

```
#include <iostream>
using namespace std;
string firstname, lastname ;
```

```
int main()
{
        cout << "Enter your first name" << endl;
        cin >> firstname;
        cout << endl;

        cout << "Enter your last name" << endl;
        cin >> lastname;
        cout << endl << "Your full name is", firstname, lastname << endl;
  return 0;
}
```

// C++ console code to calculate the average of 3 numbers expected to be impute by the user.

```
#include <iostream>
using namespace std;
float a,b,c, sum,average ;
int main()
{
        cout << "Enter first number" << endl;
        cin >> a;
        cout << endl;
        cout << "Enter second number" << endl;
        cin >> b;
        cout << endl;
        cout << "Enter third number" << endl;
        cin >> c;
        cout << endl;
  sum = a + b + c ;
average = sum / 3.0;
        cout << "The average of the three numbers is", average << endl;
 return 0;
}
```

// a simple C++ code calculate the Area of a triangle ($^1/_2$ BASE × HEIGHT)

```
#include <iostream>
using namespace std;
float Area, Base, Height;
int main()
{
        cout << "Enter the value for the BASE of the triangle" << endl;
        cin >> Base;
```

```
                cout << endl;

                cout << "Enter the value for the HEIGHT of the triangle" << endl;
                cin >> Height;
                cout << endl;
                Area = 0.8 * (Base * Height);
                cout << "the Area of the triangle is: ",  Area  << endl;
                 return 0;
        }
```

// program OHMSLAW

```
#include <iostream>
using namespace std;
real resistance, current, volts;
int main()
{
        cout << "'Please enter resistance value'" << endl;
cin >> resistance;
        cout << endl;

cout << "'Please enter current value'" << endl;
cin >> current;
volts := current * resistance ;
        cout << endl << 'The voltage is ', volts << endl;
  return 0;
}
```

// File temperature converter.cpp

```
#include <iostream>
using namespace std;
int main() {
                double degreesF, degreesC;
// Prompt user for temperature to convert
                cout << "Enter the temperature in degrees F: ";
// Read in the user's input
                cin >> degreesF;
// Perform the conversion
                degreesC = 5.0/9*(degreesF - 32);
// Report the result
```

```
        cout << degreesC << endl;
}
```

// File timeconverter.cpp

```
#include <iostream>
using namespace std;
int main() {
int hours, minutes, seconds;
        cout << "Please enter the number of seconds:";
        cin >> seconds;
// First, compute the number of hours in the given number
// of seconds
        hours = seconds / 3600; // 3600 seconds = 1 hours
// Compute the remaining seconds after the hours are
// accounted for
        seconds = seconds % 3600;
// Next, compute the number of minutes in the remaining
// number of seconds
        minutes = seconds / 60; // 60 seconds = 1 minute
// Compute the remaining seconds after the minutes are
// accounted for
        seconds = seconds % 60;
// Report the results
        cout << hours << " hr, " << minutes << " min, "
        << seconds << " sec" << endl;
}
```

// File enhancedtimeconverter.cpp

```
#include <iostream>
using namespace std;
int main() {
int hours, minutes, seconds;
    cout << "Please enter the number of seconds:";
    cin >> seconds;
// First, compute the number of hours in the given number
// of seconds
    hours = seconds / 3600; // 3600 seconds = 1 hours
// Compute the remaining seconds after the hours are
```

```
        // accounted for
                seconds = seconds % 3600;
        // Next, compute the number of minutes in the remaining
        // number of seconds
                minutes = seconds / 60; // 60 seconds = 1 minute
        // Compute the remaining seconds after the minutes are
        // accounted for
                seconds = seconds % 60;
        // Report the results
                cout << hours << ":";
        // Compute tens digit of minutes
                int tens = minutes / 10;
                cout << tens;
        // Compute ones digit of minutes
                int ones = minutes % 10;
                cout << ones << ":";
        // Compute tens digit of seconds
                tens = seconds / 10;
                cout << tens;
        // Compute ones digit of seconds
                ones = seconds % 10;
                cout << ones << endl;
        }

//Iteration codes
        #include <iostream>
                using namespace std;
                int main() {
                int power = 1;
                while (power <= 1000000000) {
                cout << power << endl;
                power *= 10;
                }
                }
```

timestable-3rd-try.cpp

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

```
int main() {
		int size; // The number of rows and columns in the table
		cout << "Please enter the table size: ";
		cin >> size;
		// Print a size x size multiplication table
		int row = 1;
		while (row <= size) { // Table has size rows.
		int column = 1; // Reset column for each row.
		while (column <= size) { // Table has size columns.
		int product = row*column; // Compute product
		cout << setw(4) << product; // Display product
		column++; // Next element
		}
		cout << endl; // Move cursor to next row
		row++; // Next row
		}
	}
```

printprimes.cpp

```
#include <iostream>
using namespace std;
int main() {
		int max_value;
		cout << "Display primes up to what value? ";
		cin >> max_value;
		int value = 2; // Smallest prime number
		while (value <= max_value) {
		// See if value is prime
		bool is_prime = true; // Provisionally, value is prime
		// Try all possible factors from 2 to value - 1
		int trial_factor = 2;
		while (trial_factor < value) {
		if (value % trial_factor == 0) {
		is_prime = false; // Found a factor
		break; // No need to continue; it is NOT prime
		}
		trial_factor++;
		}
		if (is_prime)
		cout << value << " "; // Display the prime number
```

```
        value++; // Try the next potential prime number
        }
        cout << endl; // Move cursor down to next line
}
```

PROGRAMMING ERRORS (Errors and Warnings)

Programming Errors are errors within our codes that causes our codes or program not to run, compile, execute or build when any of these operations are initiated. Beginning programmers make mistakes writing programs because of inexperience in programming in general or because of unfamiliarity with a programming language. Seasoned programmers make mistakes due to carelessness or because the proposed solution to a problem is faulty and the correct implementation of an incorrect solution will not produce a correct program. Regardless of the reason, a programming error falls under one of three categories:

- compile-time error
- run-time error
- logic error

Compile-time Errors

A compile-time error results from the programmer's misuse of the language. A syntax error is a common compile-time error. For example, in English one can say

The boy walks quickly.

This sentence uses correct syntax. However, the sentence

The boy walk quickly.

is not correct syntactically: the number of the subject (singular form) disagrees with the number of the verb (plural form). It contains a syntax error. It violates a grammatical rule of the English language. Similarly, the C++ statement

x = y + 2;

is syntactically correct because it obeys the rules for the structure of an assignment statement. However, consider replacing this assignment statement with a slightly modified version:

y + 2 = x;

If a statement like this one appears in a program and the variables **x** and **y** have been properly declared, the compiler will issue an error message; for example, the Visual C++ compiler reports (among other things):

error C2106: '=' : left operand must be l-value

The syntax of C++ does not allow an expression like **y + 2** to appear on the left side of the assignment operator.

(The term l-value in the error message refers to the left side of the assignment operator; the **l** is an "elle," not a "one.".) The compiler may generate an error for a syntactically correct statement like

x = y + 2;

if either of the variables **x** or **y** has not been declared; for example, if **y** has not been declared, Visual C++ reports:

error C2065: 'y' : undeclared identifier

Other common compile-time errors include missing semicolons at the end of statements, mismatched curly braces and parentheses, and simple typographical errors.

Compile-time errors usually are the easiest to repair. The compiler pinpoints the exact location of the problem, and the error does not depend on the circumstances under which the program executes. The exact error can be reproduced by simply recompiling the same source code. Compilers have the reputation for generating cryptic error messages. They seem to provide little help as far as novice programmers are concerned. Sometimes a combination of errors can lead to messages that indicate errors on lines that follow the line that contains the actual error. Once you encounter the same error several times and the compiler messages become more familiar, you become better able to deduce the actual problem from the reported message.

Run-time Errors

The compiler ensures that the structural rules of the C++ language are not violated. It can detect, for example, the malformed assignment statement and the use of a variable before its declaration. Some violations of the language cannot be detected at compile time, however. A program may not run to completion but instead terminate with an error. We commonly say the program "crashed."

```
// File dividedanger.cpp
#include <iostream>
using namespace std;
int main() {
        int dividend, divisor;
// Get two integers from the user
        cout << "Please enter two integers to divide:";
        cin >> dividend >> divisor;
// Divide them and report the result
        cout << dividend << "/" << divisor << " = "
        << dividend/divisor << endl;
    }
```

The expression **dividend/divisor**

is potentially dangerous. If the user enters, for example, 32 and 4, the program works nicely

Please enter two integers to divide: 32 and 4

32/4 = 8

and displays the answer of 8. If the user instead types the numbers 32 and 0, the program reports an error and terminates. Division by zero is undefined in mathematics, and integer division by zero in C++ is illegal. When the program attempts the division at run time, the system detects the attempt and terminates the program.
This particular program can fail in other ways as well; for example, outside of the C++ world, 32.0 looks like a respectable integer. If the user types in 32.0 and 8, however, the program crashes because 32.0 is not a valid way to represent an integer in C++. When the compiler compiles the source line

cin >> dividend >> divisor;

given that **dividend** has been declared to be an **int**, it generates slightly different machine language code than it would if **dividend** has been declared to be a **double** instead. The compiled code expects the text entered by the user to be digits with no extra decoration. Any deviation from this expectation results in a run-time error.

Logic Errors

Consider the effects of replacing the expression

dividend/divisor;

with the expression:

divisor/dividend;

The program compiles with no errors. It runs, and unless a value of zero is entered for the dividend, no run-time errors arise. However, the answer it computes is not correct in general. The only time the correct answer is printed is when **dividend** = **divisor**. The program contains an error, but neither the compiler nor the run-time system is able detect the problem. An error of this type is known as a logic error.

Beginning programmers tend to struggle early on with compile-time errors due to their unfamiliarity with the language. The compiler and its error messages are actually the programmer's best friend. As the programmer gains experience with the language and the programs written become more complicated, the number of compile-time errors decrease or are trivially fixed and the number of logic errors increase. Unfortunately, both the compiler and run-time environment are powerless to provide any insight into the nature and sometimes location of logic errors. Logic errors, therefore, tend to be the most difficult to find and repair. Tools such as debuggers are frequently used to help locate and fix logic errors, but these tools are far from automatic in their operation.
Errors that escape compiler detection (run-time errors and logic errors) are commonly called **bugs**. Since the compiler is unable to detect these problems, such bugs are the major source of frustration for developers.

Objected Oriented Programming (OOP).

It's a programming paradigm based on data types. An object stores a data type value; variable name refers to object. "Everything" in Java is an object. OOP enables: Data abstraction, Modular programming, Encapsulation, Inheritance etc.

Advantages of Object-Oriented Design Techniques.

Object-oriented techniques have gained wide acceptance because of its:

1. Simplicity (due to abstraction)
2. Code and design reuse
3. Improved productivity
4. Better understandability
5. Better problem decomposition
6. Easy maintenance

Object-Oriented Programming (OOP) Concepts:

1. **Object**: An Object is a special kind of record that contains fields like a record; however, unlike records, objects contain procedures and functions as part of the object. These procedures and functions are held as pointers to the methods associated with the object's type.
2. **Class**: A Class is defined in almost the same way as an Object, but there is a difference in way they are created.
3. **Instantiation of a class**: Instantiation means creating a variable of that class type. Since a class is just a pointer, when a variable of a class type is declared, there is memory allocated only for the pointer, not for the entire object. Only when it is instantiated using one of its constructors, memory is allocated for the object. Instances of a class are also called **'objects'**.
4. **Member Variables**: These are the variables defined inside a Class or an Object.
5. **Member Functions**: These are the functions or procedures defined inside a Class or an Object and are used to access object data.
6. **Visibility of Members**: The members of an Object or Class are also called the fields. These fields have different visibilities. Visibility refers to accessibility of the members, i.e., exactly where these members will be accessible. Objects have three visibility levels: public, private and protected. Classes have five visibility types: public, private, strictly private, protected and published. We will discuss visibility in details.
7. **Inheritance**: When a Class is defined by inheriting existing functionalities of a parent Class, then it is said to be inherited. Here, child class will inherit all or few member functions and variables of a parent class. Objects can also be inherited.

8. **Parent Class**: A Class that is inherited by another Class. This is also called a **base class** or **super class**.
9. **Child Class**: A class that inherits from another class. This is also called a **subclass** or d**erived class.**
10. **Polymorphism**: This is an object-oriented concept where same function can be used for different purposes. For example, function name will remain same but it may take different number of arguments and can do different tasks. Pascal classes implement polymorphism. Objects do not implement polymorphism.
11. **Overloading**: It is a type of polymorphism in which some or all of operators have different implementations depending on the types of their arguments. Similarly, functions can also be overloaded with different implementation.
12. **Data Abstraction**: Any representation of data in which the implementation details are hidden (abstracted).
13. **Encapsulation**: Refers to a concept where we encapsulate all the data and member functions together to form an object.
14. **Constructor**: Refers to a special type of function, which will be called automatically whenever there is an object formation from a class or an Object.
15. **Destructor**: Refers to a special type of function, which will be called automatically whenever an Object or Class is deleted or goes out of scope.

CONTROL STRUCTURE:

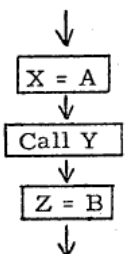
Most popular languages are imperative and use structured programming techniques. Structured programming techniques involve giving the code you write structures, these often involve writing code in blocks such as **sequence** (code executed line by line), **selection** (branching statements such as if-then-else, or case) and **repetition** (iterative statements such as for, while, repeat, loop).

CONTROL STRUCTURE is any mechanism that departs from the default of straight-line execution.

1. Sequential Structure

The simplest construct to prove correct is a group of **sequential statements** or blocks of code. These are nothing more than an ordered list in which all possibilities are clearly visible by simply following the code. Sequence statements in a program are executed one after another in the order in which they are written.

"**Sequence**"; ordered statements or subroutines executed in sequence, it specifies a linear ordering on statements.



2. Selection Structure:

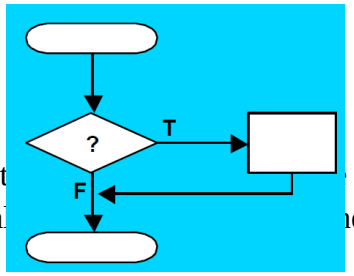
"Selection"; one or a number of statements is executed depending on the state of the program. A selection structure chooses among alternative courses of action. This is usually expressed with keywords such as if, then, else, endif.

A selection in a program allows us to choose between two or more actions depending on whether a condition is true or false. This condition is based on a comparison of two items, and is usually expressed with one of the following **relational operators**:

<	less than
>	greater than
==	equal
<=	less than or equal to
>=	greater than or equal to
!=	not equal to

a. The If Statement (if Then Selection Structure)

In the if statement, a choice is made between two alternative paths, based on a decision about whether the "condition" is true or false.



if (condition)
statement;

Example, we want to write a program that det for a bank account based on the account ba hen there is a fee of $5 on the balance.

Pseudocode:

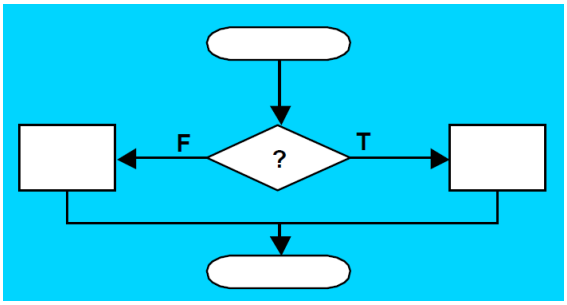
```
if (accountBalance <1000)
Fee= 5;
```

Such an expression is called a **condition** because it establishes a criterion for either executing or skipping a group of statements.

b. The if/else Statement (if Then Else Selection Structure)

This is of the form:

if (condition)
statementT ;
else
statementF ;



The **if/else** selection structure allows the programmer to specify the actions to be performed when the condition is ***true*** and the actions to be performed when the condition is ***false***. When the **if** condition evaluates to true, the statement *statementT* is executed. When the **if** condition evaluated

to false, the statement *statementF* that is the target of **else** will be executed. Only the code associated with **if** condition **OR** the code associated with **else** executes, never both.

Example 1:

Suppose the passing grade on an exam is 60. Write a program that prints "Passed" if a grade entered is greater than 60 and "Failed" otherwise.

Begin
Read student_grade
IF student_ grade ≥ 60 THEN
    print "Passed"
ELSE
    print "Failed"

NB: Indentation is very important in making a program more readable and understanding the logic.

Example 2:

We want to write a program that determines the service charge for a bank account based on the account balance. When the account balance is less than $1000, then there is a fee of $5 on the balance and a message is printed informing us about it. If the balance is above the amount, another message is sent.

Algorithm:
Begin
Read account balance
Set fee to 0
IF account balance <= 1000 THEN
    print "Fee is due"
    fee =5
ELSE
    print "No fee is due"
End
account balance= account balance – fee

The program in C++

```
#include <iostream>
using namspace std;
int main(void) {
double account_balance;
int fee=0;
cout << "Please enter the account_balance: \n";
cin>> account_balance;
if (account_balance <=1000.00)
{cout << "Fee is due";
fee=5;}
else
cout << "No fee is due";
account_balance= account_balance - fee;
cout << "The account balance is "<< account_balance;
return 0;}
```

c. Nested ifs Selection Structure

Nested **if/else** structures test for multiple cases by placing **if/else** structures inside **if/else** structures. In C++, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and not already associated with an **if.**

if (expri) {

```
if (exprj)
        statement1;
if (exprk)
        statement2;/*this if is */
else
        statement3;/*associated with this else*/
}/*end if
else
        statement4;
```

d. The if-else-if Selection Structure

This is a common programming construct, also called the if-else-if staircase because of its appearance:

```
if (expri)
        statement1;
else
if (exprj)
        statement2;
else
if (exprk)
        statement3;
else
        statement4;
```

The conditions are evaluated in a top down fashion. When a condition returns true, the rest of the ladder is bypassed. If none of the conditions are true, the final else statement is executed.

Example: The following algorithm will print:

A for exam grades greater than or equal to 90.
B for grades greater than or equal 80.
C for grades greater than or equal to 70.
D for grades greater than or equal to 60 and
F for all other grades.

```
Begin
        Read grade
        if grade ≥ 90
        print "A"
else
if grade ≥ 80
        print "B"
else
if grade ≥ 70
        print "C"
else
if grade ≥ 60
        print "D"
else
        print "F"
```

The C++ code snippet:

```
if (grade >= 90)
        cout<< "A\n";
else if (grade >= 80)
        cout << "B\n";
else
if (grade >= 70)
        cout << "C\n";
else
        if (grade >= 60)
        cout << "D\n";
else    cout << "F\n";
```

e. The Switch Multiple-Selection Structure

Sometimes an algorithm contains a series of decisions in which a variable or expression is tested separately. We used an if-else-if ladder to solve it.

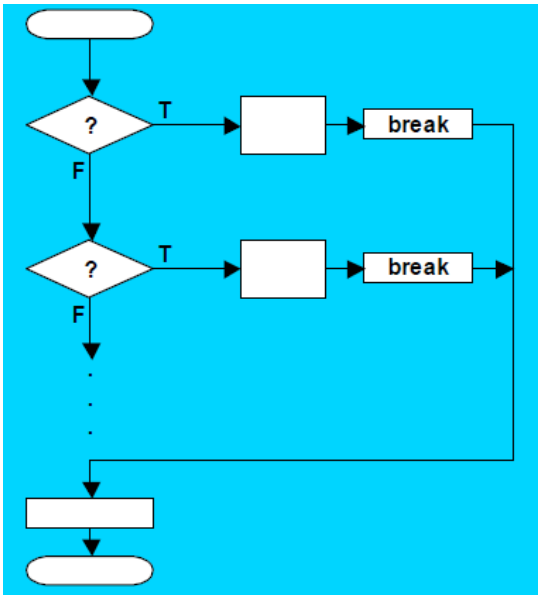
The if-else-if Ladder

```
if (expression==value1) statement
else
```

```
if (expression==value2) statement
else
        if (expression== value3) statement
```

C++ also provides the switch multiple selection structure to handle this kind of tasks.
The *switch case* statement is a better way of writing a program when a series of *if elses* occurs.
The general format for the switch statement is:

```
switch (variable) {
        case value1:
                statement;
                statement;
                ......
        break;
        case value2:
                statement;
                .......
        break;
default:
                .......
        break;
        }
```



The **keyword *break*** must be included at the end of each case statement.
The **default** clause is optional, and is executed if the cases are not met.
The **right brace** at the end signifies the end of the case selections.

The **break** statement is one of C's jump statements. When the **break** statement is encountered in a **switch** statement, program execution "jumps" to the line following the switch statement. The **switch** structure consists of a series of **case** labels and an optional **default** case.

Rules for SWITCH Statements

i. Values for 'case' must be integer, Boolean or character constants.
ii. The order of the 'case' statements is unimportant
iii. The default clause may occur first (convention places it last)
iv. You cannot use expressions or ranges for the variable.

- The value of expression is tested, in order, against that value specified in the case.
- When a match is found the sequence associated with that case is executed until the break statement or the end of the switch statement is reached.
- The default statement is executed if no matches are found.
- If the default statement is not present, no action takes place.

Things to know about the SWITCH statement:

i. The ***switch*** statement differs from the ***if*** statement in that ***switch*** can only test for equality, whereas ***if*** can evaluate relational expressions.
ii. No two case constants in the same ***switch*** can have identical values.
iii. The statements can be empty, in which case next case statement is executed.
iv. No braces are required for multiple statements in a ***switch*** structure.

Example, we want to implement the following decision table, that classifies a ship depending on its class ID

Class ID	Ship Class
B or b	Battleship
C or c	Cruiser
D or d	Destroyer
F or f	Frigate

```
#include <iostream>
using namespace std;
int main(void) {
char Class_id;
cout<< “Enter ship class: ";
cin>> Class_id;
switch (Class_id) {
        case 'B': cout<<”Battleship\n”; break;
        case 'b': cout <<”Battleship\n"; break;
        case 'C':
        case 'c': cout<<”Cruiser\n"; break;
        case 'D':
        case 'd' : cout<<” Destroyer \n"; break;
        case 'F':
        case 'f' : cout<<”Fregate\n";
break;
default : cout<<”Unknown ship class<< Class_id<<endl;
}
return 0;
}
```

3. Repetition Structure

"Iteration"; a statement or block is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as while, repeat, for or do…until. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).

A **repetition structure** also called iteration or loop allows an action or a set of instructions to be performed while some condition remains true. There are 3 types of loop or repetition structures each one is suitable for a specific type of problems.

It is also called a Repetitive control structure. Sometimes we require a set of statements to be executed a number of times by changing the value of one or more variables each time to obtain a different result. This type of program execution is called looping.

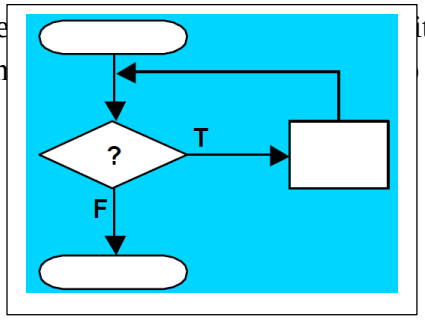
The Essentials of Repetition

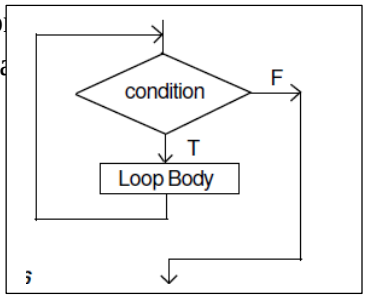
A loop is a group of instruction the computer executes repeatedly while some condition remains true, we have seen **two types of repetition**:

1. **Counter-Controlled Repetition**:
   - The number of iterations or how many times the loop is going to be executed are known.
   - A control variable is used to count the number of iterations and is incremented (usually by 1).
   - The loop exits when the control variable value has reached the final count. E.g. the For loop
2. **Sentinel-Controlled Repetition**:
   - The precise number of iterations is not known in advance.
   - The sentinel value indicates "end of data"
   - Sentinel value must be distinct from regular, valid data.

a. The While Loop Repetition Structure

The flow diagrams below indicate that a condition is first evaluated. If the condition is true, the loop body is executed and the condition is re-evaluated. Hence, the loop body is executed repeatedly as long as the condition re tio
out of the loop and goes to the staten ma
the following example.





Syntax of while loop

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

Normally, the three operations listed below must be performed on the loop control variable.

1. Initialize the loop control variable
2. Test the loop control variable
3. Update the loop control variable

Operation (i) must be performed before the loop is entered.

Operation (ii) must be performed before each execution of the loop body; depending on the result of this test, the loop will either be repeated or make an exit.

Operation (iii) must be included as part of the loop body. Unless the loop control variable is updated in the loop body, its value cannot change and loop exit will never occur.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever.

Example: To find the sum of first ten natural numbers i.e. 1, 2, 3, ...10.

```
# include <iostream.h>
void main ( )
{ int n, total = 0 ;
n = 1 ;
while (n < = 10)
{
total + = n;
n + +;
}
cout << “sum of first ten natural number” << total:
}
```

NB: The variable n is called a loop control variable since its value is used to control loop repetition.

Example: Create a program to countdown using a while-loop:

```
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{ int n;
cout << "Enter the starting number > ";
cin >> n; while (n>0) {
cout << n << ", ";
--n;
}
cout << "FIRE!\n"; return 0;
}
```

OUTPUT: Enter the starting number > 8

8, 7, 6, 5, 4, 3, 2, 1, FIRE!

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

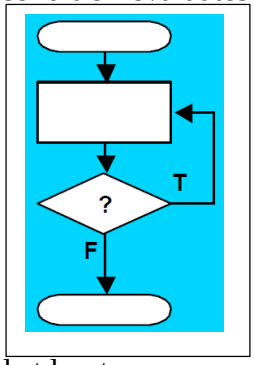
b. The Do-While Loop Repetition Structure

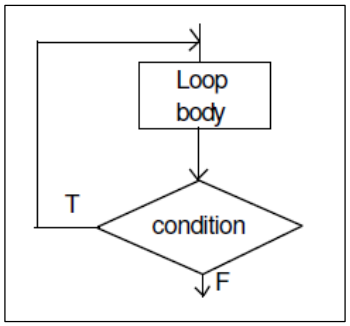
This structure is similar to the **while** loop, except for one thing:

- In the **while** loop, the continuation condition is tested at *the beginning* of the loop.

- In the **do-while**, the continuation condition is tested after the loop body is performed.
- When a **do-while** terminates, execution continues with the statement after the while clause.

In other words, its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled.
The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end.
The flow diagram below indicates that after each execution of the loop body, the condition is true, the loop body is executed again. If the condition evaluates to false, loop exit occurs and the next program statement is executed.





Syntax of do-while loop
do
{statement
} while (condition);

Note: The loop body is always executed at least once.
One important difference between the while loop and the do-while loop is the relative ordering of the conditional test and loop body execution. In the while loop, the loop repetition test is performed before each execution of the loop body; the loop body is not executed at all if the initial test fails. In the do-while loop, the loop termination test is Performed after each execution of the loop body; hence, the loop body is always executed at least once.

Example: To find the sum of the first N natural number.

```
# include < iostream.h>
void main ( )
{
int N, number, sum;
cin >> N;
sum = 0;
number = 1;
do
{
sum + = number;
number + + ;
}
while (number < = N) ;
```

```
cout << sum;
}
```

Example: The following example program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream>
using namespace std;
int main ()
{ unsigned long n; do {
cout << "Enter number (0 to end): ";
cin >> n;
cout << "You entered: "<< n << "\n";
} while (n != 0); return 0;
}
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

Major differences between while and do while loop

	While	do-while
1.	It is an entry controlled loop.	It is exit controlled loop.
2.	It first check the condition & then execute the loop.	It execute loop first and then check the condition.
3.	If the condition is false, while loop does not get executed.	Even if the condition is false, do-while loop get execute once.

c. The For Loop Repetition Structure

The for structure handles all the details of a counter-controlled repetition.

General form of for loop structure is:

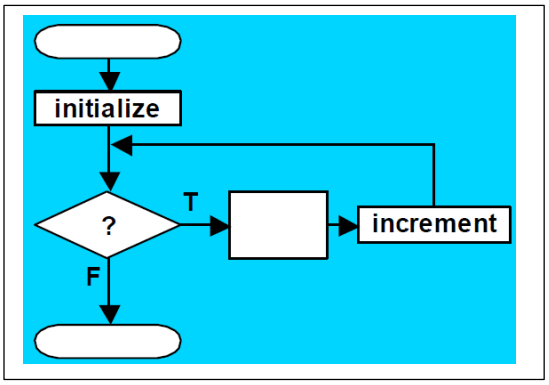
for (*expression1; expression2; expression3)*

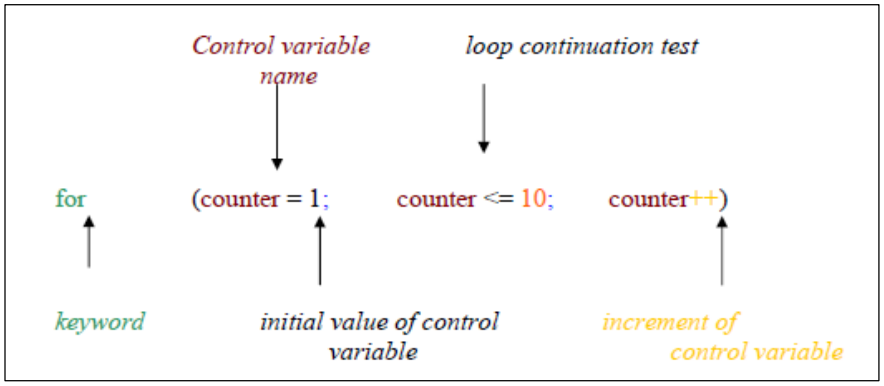
where:

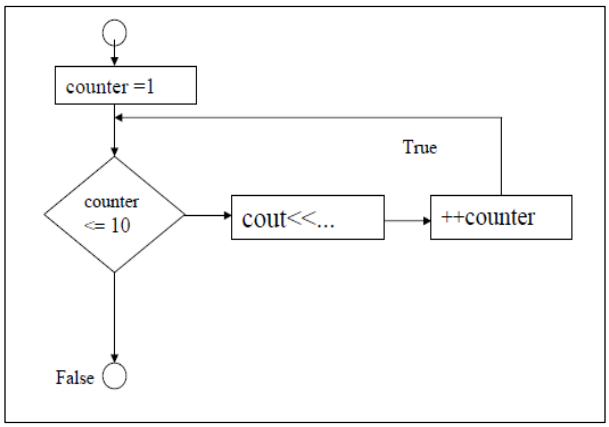
expression1 initializes the loop's control variable,
expression2 is the loop continuation condition,
and *expression3* increments the control variable.

This is equivalent to:

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







its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

The flow diagrams above indicate that in for loop **three operations take place**:

1. Initialization of loop control variable
2. Testing of loop control variable
3. Update the loop control variable either by incrementing or decrementing.

Operation (i) is used to initialize the value. On the other hand,

Operation (ii) is used to test whether the condition is true or false. If the condition is true, the program executes the body of the loop and then the value of loop control variable is updated. Again it checks the condition and so on. If the condition is true, it gets out of the loop.

It works in the following way:

i. Initialization is executed. Generally, it is an initial value setting for a counter variable. This is executed only once.

ii. Condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).

iii. Statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.

iv. Finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{ for (int n=10; n>0; n--) {
cout << n << ", ";
}
cout << "FIRE!\n"; return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).
Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example.
The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{ // whatever here...
}
```

Examples

```
for (int i = 0 ; i < 5 ; i + + )
        cout << i ;
```

The output of the above program is 0 1 2 3 4

```
for (char i = 'A'; i < 'E' ; i + + )
        cout < < i ;
```

The output of the above program is A B C D

Vary the control variable from 1 to 100 in increments of 1**: for (i=1; i<= 100; i++)**
Vary the control variable from 100 to 1 in increments of –1: **for (i=100; i>=1; i--)**
Vary the control variable from 20 to 2 in increments of –2: **for (i=20; i >= 2; i -=2)**

Write a program that will sum all the even numbers from 2 to 100 using a for loop
Input: Numbers from 2 to 100.
Output: Sum of even numbers

```
#include <iostream>
using namespace std;
int main(void) {
int sum = 0, number;
for (number = 2; number <= 100; number += 2)
sum+= number;
cout<< "The sum of even numbers from 2 to 100 is " << sum <<endl;
return 0;
}
```

```
/* example of counter controlled
repetition  with while structure*/
#include <iostream>
using namespace std;
int main(void)
{
int counter =1;
while (counter <= 10) {
cout << counter;
++counter;
}
cout<<endl;
```

```
/* counter-controlled loop with for
structure */
#include <iostream>
using namespace std;
int main(void)
{
int counter;
for (counter = 1; counter <= 10;
counter++)
cout<< counter;
cout<<"\n";
return 0;
```

JUMP STATEMENTS: THE BREAK, CONTINUE AND GOTO STATEMENTS:

break and **continue** are used to alter the flow of control.

A. The **BREAK** statement:

When used in a **for, while, do-while** or **switch** structures

i. It causes immediate exit from the structure,
ii. Program execution continues with the first statement after the structure.
iii. Commonly used to escape early from a loop, or skip the remainder of a **switch** structure.

Using **break** we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.

```
#include <iostream>
using namespace std;
int main(void) {
int x;
for (x=1; x<=10; x++) {
if (x==5)
break;
cout<< x;
}//end for
Condition
statement
True
cout << "\nBroke out of loop at “<< x <<endl;
return 0;
}//end main
```

Program Output:

1 2 3 4

Broke out of loop at x == 5

Example: stop the count down before its natural end

```
// break loop example
#include <iostream>
using namespace std;
int main ()
{ int n; for (n=10; n>0; n--)
{
cout << n << ", "; if (n==3)
{
cout << "countdown aborted!"; break;
}
} return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

B. The **CONTINUE** statement:

i. The **continue** statement when executed in a **while, for** or **do-while** structure, skips the remaining statements in the body of the loop and performs an iteration of the loop.

ii. In **while** and **do-while** loop, the loop continuation test is evaluated *after* the **continue** is executed.
iii. In the **for** structure, the increment expression is executed, then the continuation test is evaluated.

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.

```
#include<stdio.h>
int main(void) {
int x;
for (x=1; x<=10; x++) {
if (x==5)
continue;
cout<< x;
}
cout<<"\n Used continue to skip printing the value 5"<<endl;
return 0;
}//end main
```

Program Output:
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5

Example: Lets skip the number 5 in our countdown

```
// continue loop example
#include <iostream>
using namespace std;
int main ()
{ for (int n=10; n>0; n--) { if
(n==5) continue;
cout << n << ", ";
}
cout << "FIRE!\n"; return 0;
}
```

10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

C. GOTO Statement

GOTO allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations. The destination point is identified by a label, which is then used as an argument for the GOTO statement. A label is made of a valid identifier followed by a colon (:).
Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it.

syntax of GOTO statement

```
GOTO pgr;
|
|
|
pgr :
```

pgr is known as label. It is a user defined identifier. After the execution of GOTO statement, the control transfers to the line after label pgr.
Note: It is not a good programming to use GOTO statement in a program.

Example: Countdown loop using GOTO:

```
// goto loop example
#include <iostream>
using namespace std;
int main ()
{ int n=10;
loop:
cout << n << ", ";
n--; if (n>0) goto loop;
cout << "FIRE!\n"; return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

OTHERS:

exit ( ) function

The execution of a program can be stopped at any point with **exit ( )** and a status code can be informed to the calling program. The general format is:

exit (code);

where code is an integer value. The code has a value 0 for correct execution. The value of the code varies depending upon the operating system. It requires a process.h header file

The purpose of exit is to terminate the current program with a specific exit code. The exit code is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

"**Recursion**"; a statement is executed by repeatedly calling itself until termination conditions are met. While similar in practice to iterative loops, recursive loops may be more computationally efficient, and are implemented differently as a cascading stack.

Subroutines; callable units such as procedures, functions, methods, or subprograms are used to allow a sequence to be referred to by a single statement.

Blocks: Blocks are used to enable groups of statements to be treated as if they were one statement.

Block structured languages have a syntax for enclosing structures in some formal way, such as an if-statement bracketed by if as in ALGOL 68, or a code section bracketed by BEGIN..END , as in PL/I and Pascal, whitespace indentation as in Python - or the curly braces {...} of C and many later languages.