UNIT-I

Syllabus

Introduction to Object Oriented Programming: Object oriented paradigm-Differences between Object Oriented Programming and Procedure oriented programming, Basic concepts of Object Oriented Programming, Benefits of OOP, Structure of a C++ program, namespace, Data types, C++ tokens, Identifiers, Variables, Constants, Operators, Control structures &Loops.

Functions: Introduction to functions, Inline functions, Command Line arguments.

Introduction to C++

C++ is a general-purpose and multi-paradigm computer programming language. The C++ programming language supports both object-oriented programming and generic programming. C++ is an object-oriented programming language but it is not purely object-oriented programming language because of features like friend functions, virtual functions, and virtual base classes.

The C++ programming language is used to create applications that will run on a wide variety of hardware platforms such as personal computers running Windows, Linux, UNIX, and Mac OS. It also supports the applications for small form factor hardware such as IoT devices like the Raspberry PI and Arduino–based boards.

The C++ programming language was created by **Bjarne Stroustrup** in the year 1983, at Bell Laboratories, USA.

Bjarne Stroustrup
Creator of C++
Born in Denmark on 30th December 1950.
- www.btechsmartclass.com

The C++ programming language is said to be the superset of C programming language. The programs written in C programming language can run in the C++ compiler. The C++ programming language is an extension of the C programming language and it is case-sensitive language.

PROGRAMMING PARADIGMS

The programming paradigm is the way of writing computer programs. There are four programming paradigms and they are as follows.

- **Monolithic programming paradigm**
- **Structured-oriented programming paradigm**
- **Procedural-oriented programming paradigm**
- **Object-oriented programming paradigm**

Monolithic Programming Paradigm

The Monolithic programming paradigm is the oldest. It has the following characteristics. It is also known as the imperative programming paradigm.

- In this programming paradigm, the whole program is written in a single block.
- We use the **goto** statement to jump from one statement to another statement.

- It uses all data as global data which leads to data insecurity.
- There are no flow control statements like if, switch, for, and while statements in this paradigm.
- There is no concept of data types.

An **example** of a Monolithic programming paradigm is **Assembly language**.

Structure-oriented Programming Paradigm

The Structure-oriented programming paradigm is the advanced paradigm of the monolithic paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.
- In this paradigm, all the data is used as global data which leads to data insecurity.

Examples of a structured-oriented programming paradigm is **ALGOL, Pascal, PL/I and Ada**.

Procedure-oriented Programming Paradigm

The procedure-oriented programming paradigm is the advanced paradigm of a structure-oriented paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.
- It follows all the concepts of structure-oriented programming paradigm but the data is defined as global data, and also local data to the individual modules.
- In this paradigm, functions may transform data from one form to another.

Examples of procedure-oriented programming paradigm is **C, visual basic, FORTRAN**, etc.

Object-oriented Programming Paradigm

The object-oriented programming paradigm is the most popular. It has the following characteristics.

- In this paradigm, the whole program is created on the concept of objects.

- In this paradigm, objects may communicate with each other through function.
- This paradigm mainly focuses on data rather than functionality.
- In this paradigm, programs are divided into what are known as objects.
- It follows the bottom-up flow of execution.
- It introduces concepts like data abstraction, inheritance, and overloading of functions and operators overloading.
- In this paradigm, data is hidden and cannot be accessed by an external function.
- It has the concept of friend functions and virtual functions.
- In this paradigm, everything belongs to objects.

Examples of object-oriented programming paradigm is **C++, Java, C#, Python**, etc.

OOP Vs Procedural Paradigms(Differences b/w POP and OOP)

Both object-oriented and procedure-oriented paradigms are very popular and most commonly used programming paradigms. The following table gives the differences between those.

Procedure-oriented	Object-oriented
It is often known as POP (procedure-oriented programming).	It is often known as OOP (object-oriented programming).
It follows the top-bottom flow of execution.	It follows the bottom-top flow of execution.
Larger programs have divided into smaller modules called as functions.	The larger program has divided into objects.
The main focus is on solving the problem.	The main focus is on data security.

Procedure-oriented	Object-oriented
It doesn’t support data abstraction.	It supports data abstraction using access specifiers that are public, protected, and private.
It doesn’t support inheritance.	It supports the inheritance of four types.
Overloading is not supported.	It supports the overloading of function and also the operator.
There is no concept of friend function and virtual functions.	It has the concept of friend function and virtual functions.
Examples - C, FORTRAN	Examples - C++ , Java , VB.net, C#.net, Python, R Programming, etc.

Benefits of OOP

OOP offers several benefits to both the program designer and the user.

They are:

1. Through inheritance ,we can eliminate redundant code and extend the use of existing classes.
2. Encapsulation features that is packing of data & functions into a single component are to be allowed.
3. The principal of data hiding helps the programmer to built secure programs that can't be inveded by code in other parts of the program.

4. It is easy to partition the work in a project based on objects.
5. Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
6. Software complexity can be easily managed.
7. We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch which happens procedure oriented approach. This leads to saving of development time and higher productivity.
8. It is possible to have multiple instances of object to co-exist without any interference.
9. It is possible to map objects in the problem domain to those in the program.
10. Object oriented systems can be easily upgraded from small to large systems.
11. The data-centered design approach enables us to capture more details of a model in implementable from.

OOP Concepts

The main aim of the C++ programming language was to add the concepts of object-oriented programming paradigm to the procedure-oriented programming paradigm like C programming language. The following are the object-oriented programming concepts.

1. Object
2. Class
3. Encapsulation

4. Abstraction
5. Inheritance
6. Polymorphism
7. Data Binding
8. Message Passing

1. Object

An object is any real-time entity with some properties and behaviors. The object is the basic unit of the object-oriented programming paradigm.

The properties of an object are the data values, and the behaviors are the functionalities that are associated with it. In object-oriented programming, the objects are created through the concept of class.

Ex:

Student s;

s—>an object

2. Class

A class is a collection of data and code where data is defined as variables and code like methods.

(or)

A blueprint of an object is called ‘Class’.

Every class in OOPs defines a user-defined data type. The data and code defined in a class are said to be the members of that class (data members and member functions respectively).

<u>Creation of Class:</u>

<u>Syntax:</u>

```
 class  Class_Name
{
        AccessSpecifier:
                    datatype Var_name1;
                    datatype Var_Name2;
                     ……………………..
                    ………………………
                    Datatype Var_Namen;
        AccessSpecifier:
                    Returntype Function_Name1(Parameterslist);
                     Returntype Function_Name2(Parameterslist);
                    ……………………………………………….
                   ……………………………………………….
                   Returntype Function_Namen(Parameterslist);
   }:
```

Ex:

```
class Student
      {
         public:
                int SRno;
                char Sname[20];
                int M1,M2,M3;
                float Avg;
          public:
                 void getdata();
                 void putdata();
        }
```

3. Encapsulation

The Wrapping up of data into a single unit is called “Encapsulation”.

(or)

Encapsulation is the process of combining data with code into a single unit.

Ex: class

4. Abstraction

Abstraction is the process of hiding unnecessary details and showing only useful details to the end-user.

Using the abstraction concept, the actual business logic is hidden from the users and shown only required things to solve the problem.

Ex: ATM Machine

5. Inheritance

Inheritance is the process of acquiring properties from one object to another object.

(or)

Inheritance is the process of deriving a new class from an existing class where the members of the existing class are extended to the newly derived class.

Here, the existing class is called a ‘base class’ or a ‘superclass’ or a ‘parent class’, and the newly created class is called a ‘derived class’ or a ‘subclass’ or a ‘child class’.

Note: The inheritance concept enables a major feature called code reusability.

6. Polymorphism

The ability to take one form into many forms is called “Poilymorphism”.

Ex: 1) Function Overloading

2) Operator Overloading

7. Data Binding

The process of assigning the body of the function to its function call is called “Data Binding”.

Data Binding can be done in 2 ways:

1. Static Binding
2. Dynamic Binding

1. Static Binding:

The process of assigning the body of the function to its function call is done at compile time is called “Static Binding”.

Ex: User-defined functions

2.Dynamic Binding:

The process of assigning the body of the function to its function call is done at compile time is called “Static Binding”.

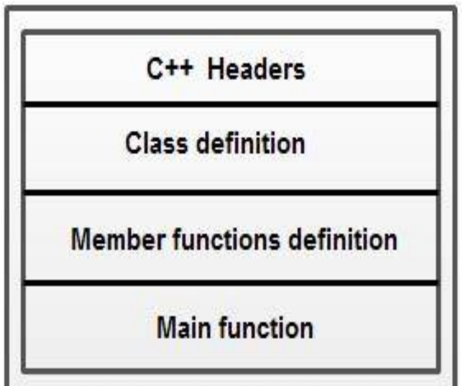
Ex: Pre-defined functions

8. Message Passing:

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Structure of a C+ + Program

Programs are a sequence of instructions or statements. These statements form the structure of a C++ program. C++ program structure is divided into various sections, namely, headers, class definition, member functions definitions and main function.

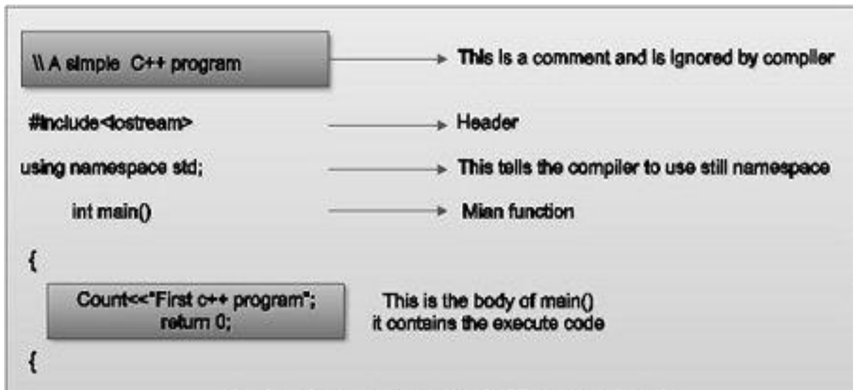


Structure of a C++ Program

```
/*documentation section*/

pre processing statements

class ClassName

{

   member variable declaration;

   ..member functions(){

      function body    }

};

void main ()

{   ClassName object;

   object.member;

object.memberfunction();
```

Note that C++ provides the flexibility of writing a program with or without a class and its member functions definitions. A simple C++ program (without a class) includes comments, headers, namespace, main() and input/output statements.

Comments are a vital element of a program that is used to increase the readability of a program and to describe its functioning. Comments are not executable statements and hence, do not increase the size of a file.



```
\\ A simple  C++ program
#include<iostream>
using namespace std;
      int main()
{
   Count<<"First c++ program";
   return 0;
{
```

This is a comment and is ignored by compiler
Header
This tells the compiler to use still namespace
Mian function
This is the body of main()
it contains the execute code

A simple c++ program(without a class)

C++ supports two comment styles: single line comment and multiline comment. Single line comments are used to define line-by-line descriptions. Double slash *(//)* is used to represent single line comments. To understand the concept of single line comment, consider this statement.

/ / An example to demonstrate single line comment It can also be written as

/ / An example to demonstrate

/ / single line comment

Multiline comments are used to define multiple lines descriptions and are represented as / * * /. For example, consider this statement.

/* An example to demonstrate
multiline comment */

Generally, multiline comments are not used in C++ as they require more space on the line. However, they are useful within the program statements where single line comments cannot be used. For example, consider this statement.

for(int i = 0; i<10; //loop runs 10 times i++)

Compiler ignores everything written after the single line comment and hence, an error occurs. Therefore, in this case multiline comments are used. For example, consider this statement.

for(int i = 0; i<10; /*loop runs 10 times */ i++)

Headers:

- Generally, a program includes various programming elements like built-in functions, classes, keywords, constants, operators, etc., that are already defined in the standard C++ library.
- In order to use such pre-defined elements in a program, an appropriate header must be included in the program.
- The standard headers contain the information like prototype, definition and return type of library functions, data type of constants, etc.
- As a result, programmers do not need to explicitly declare (or define) the predefined programming elements.

- Standard headers are specified in a program through the preprocessor directive" #include.
- In Figure, the iostream header is used. When the compiler processes the instruction #inc1ude<iostream>, it includes the contents of iostream in the program.
- This enables the programmer to use *standard input*, *output* and *error* facilities that are provided only through the standard streams defined in <iostream>.
- These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.

• **cin** (pronounced **"see in"**) : It is the standard input stream that is associated with the standard input device (keyboard) and is used to take the input from users.

• **cout** (pronounced **"see out"**) : It is the standard output stream that is associated with the standard output device (monitor) and is used to display the output to users.

• **cerr** (pronounced **"see err"**) : It is the standard error stream that is associated with the standard error device (monitor) and is used to report errors to the users. The cerr object does not have a buffer (temporary storage area) and hence, immediately reports errors to users. '

• **clog** (pronounced **"see log"**): It is the buffered error stream that is associated with the standard error device (computer screen) and is used to report errors to users. Unlike cerr, clog reports errors to users only when the buffer is full

For many years, C++ applied C-style headers, that is, .h extension in the headers. However, the standard C++ library introduced new-style headers that include only header name. Hence, the most modem compilers do not require any extension, though they support the older .h extension. Some of C-style headers and their equivalent C++ style headers are listed in Table.

Namespace: Since its creation, C++ has gone through many changes by the C++ Standards Committee.

- One of the new features added to this language is namespace.
- A namespace permits grouping of various entities like classes, objects, functions and various C++ tokens, etc., under a single name.

- ➢ Different users can create separate namespaces and thus can use similar names of the entities.
- ➢ This avoids compile-time error that may exist due to identical-name conflicts.
- ➢ The C++ Standards Committee has rearranged the entities of the standard library under a namespace called std.
- ➢ In Figure, the statement using namespace std informs the compiler to include all the entities present in the namespace std.
- ➢ The entities of a namespace can be accessed in different ways which are listed here.

• By specifying the using directive

using namespace std;

cout<<"Hello World";

• By specifying the full member name

std: :cout<<"Hello World";

• By specifying the using declaration

using std:: cout;

cout<<"Hello World";

As soon as the new-style header is included, its contents are included in the std namespace. Thus, all the modern C++ compilers support these statements.

#include<iostream>

using namespace std;

However, some old compilers may not support these statements. In that case, the statements are replaced by this single statement.

#include<iostream.h>

Main Function:

- The main () is a startup function that starts the execution of a c++ program. All C++ statements that need to be executed are written within main ( ).
- The compiler executes all the instructions written within the opening and closing curly braces' {}' that enclose the body of main ( ).
- Once all the instructions in main () are executed, the control passes out of main ( ), terminating the entire program and returning a value to the operating system.
- By default, main () in C++ returns an int value to the operating system.
- Therefore, main () should end with the return 0 statement. A return value zero indicates success and a non-zero value indicates failure or error.

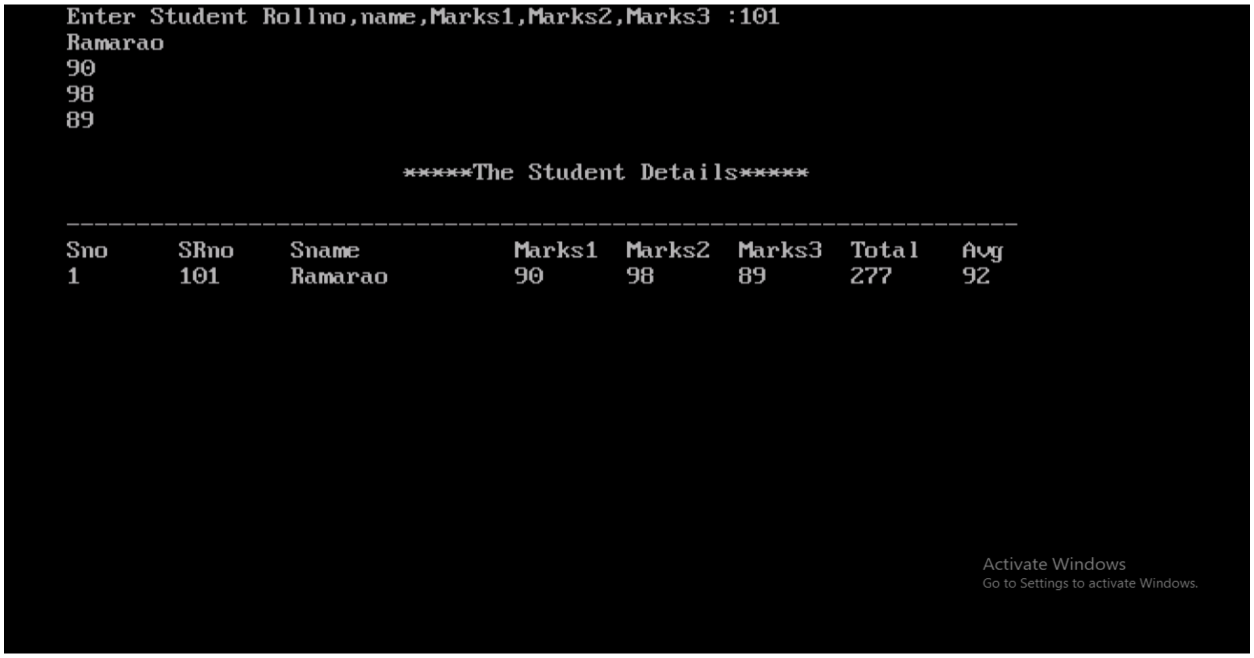
Example:

```
#include<iostream.h>

#include<conio.h>

class Student

{

  public:

     int Srno;

     char Sname[20];

     int Marks1,Marks2,Marks3;

     int Total;

     float Avg;

  public:

     void getdata();
```

```
    void putdata();
};
void Student::getdata()
{
  cout<<"Enter Student Rollno,name,Marks1,Marks2,Marks3 :";
  cin>>Srno>>Sname>>Marks1>>Marks2>>Marks3;
}
void Student::putdata()
{
  int i=0;
  Total=Marks1+Marks2+Marks3;
  Avg=Total/3;
  cout<<"\n*****The Student Details*****\n";

cout<<"Sno\tSRno\tSname\t\tMarks1\tMarks2\tMarks3\tTotal\tAvg\n
";

cout<<i+1<<"\t"<<Srno<<"\t"<<Sname<<"\t\t"<<Marks1<<"\t"<<Ma
rks2<<"\t"<<Marks3<<"\t"<<Total<<"\t"<<Avg;
}
void main()
{
  Student s;
```

```
        clrscr();
        s.getdata();
        s.putdata();
    }
```

Output:



```
Enter Student Rollno,name,Marks1,Marks2,Marks3 :101
Ramarao
90
98
89
                        *****The Student Details*****

--------------------------------------------------------------------
Sno     SRno    Sname           Marks1  Marks2  Marks3  Total   Avg
1       101     Ramarao         90      98      89      277     92
```

Activate Windows
Go to Settings to activate Windows.

NameSpaces in C++

- Let's take a situation where there are two students with the same name in an institution.
- Then we have to differentiate them in a different manner and more likely we have to add some more information along with their name, like roll number or parents name or email address.

- The same situation may arise in C++ programming also where you might write some code having function name i.e. *fun()* and there is already existing another library having same function name.
- This makes the compiler halt down and left it with no way to know which of these two function to use within the C++ program. **Namespaces** are used to solve this situation.
- Namespaces provide a scope for identifiers (variables, functions etc) within own declarative region.
- Namespaces are used to systematize code in logical groups which prevents naming conflict, which can occur especially if there are multiple libraries with single names in your code base.
- Simply we can say that the namespace defines a scope.

Defining the NameSpace:

The keyword '*namespace*' is used to define a namespace followed by the name of the namespace.

Syntax:

```
namespace namespace_name
{
        // code declarations
}
```

Example:

```
namespace Salary
{
        int make_money();
        int check_money();
         //so forth....

}
```

The using Directive:

- The using directive permits all the names in a namespace to be applied without the namespace-name as an explicit qualifier.
- Programmers can also avoid pre-awaiting of namespaces with the using namespace directive.
- ‘*using’ keyword* tells the compiler that subsequent code is making use of names in an identified namespace.

Program showing the use of Namespace in C++:

Example:

```
#include<iostream>
using namespace std;
// first name space
namespace firstone {
void fun()
{
   cout << "This is the first NS" << endl;
}
}
// second name space
namespace secondone {
void fun()
{
   cout << "This is the second NS" << endl;
}
}
using namespace firstone;
int main()
{
   // calls the function from first namespace.
   fun();

}
```

Output:

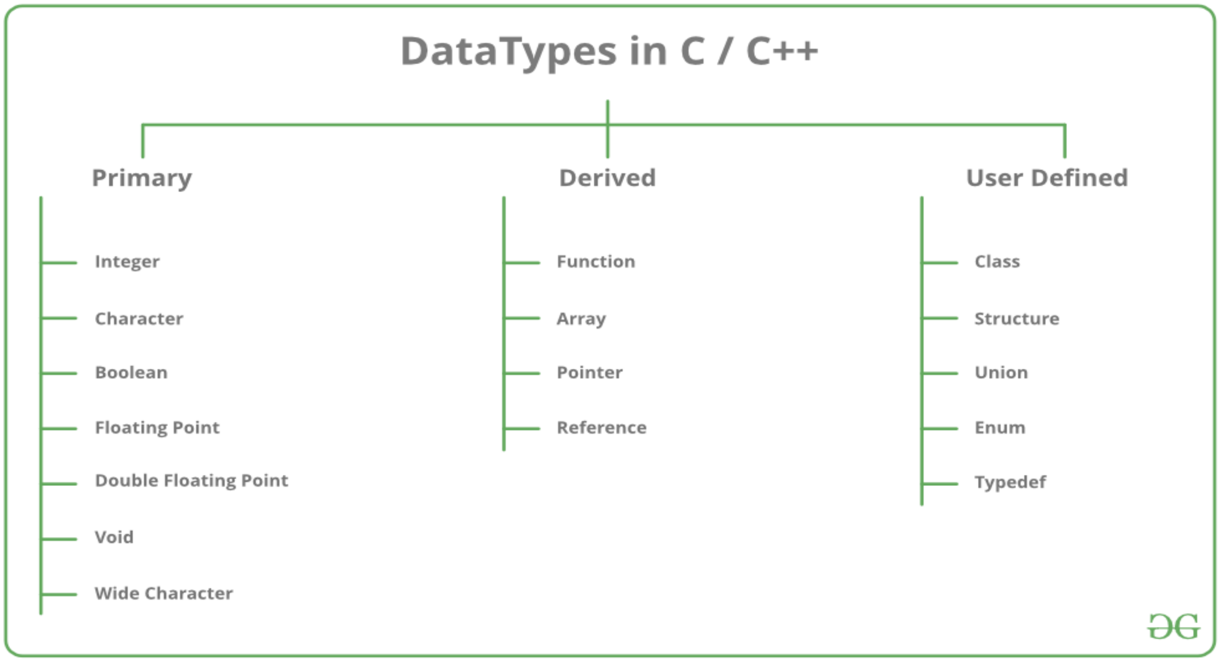
```
This is the first NS
```

C++ DATA TYPES

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

Data types in C++ is mainly divided into three types:

1. Basic(or)Primary Datatypes
2. Derived Datatypes
3. User-defined Datatypes



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

i. Integer
ii. Character
iii. Boolean
iv. Floating Point
v. Double Floating Point
vi. Valueless or Void
vii. Wide Character

Integer:

- ➢ Keyword used for integer data types is **int**.
- ➢ Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.

Character:

- ➢ Character data type is used for storing characters.
- ➢ Keyword used for character data type is **char**.
- ➢ Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.

Boolean:

- ➢ Boolean data type is used for storing boolean or logical values.
- ➢ A boolean variable can store either *true* or *false*.
- ➢ Keyword used for boolean data type is '**bool'**.

Floating Point:

- ➢ Floating Point data type is used for storing single precision floating point values or decimal values.
- ➢ Keyword used for floating point data type is **float**.

- Float variables typically requires 4 byte of memory space.

Double Floating Point:

- Double Floating Point data type is used for storing double precision floating point values or decimal values.
- Keyword used for double floating point data type is **double**.
- Double variables typically requires 8 byte of memory space.

void:

- Void means without any value.
- void datatype represents a valueless entity.
- Void data type is used for those function which does not returns a value.

Wide Character:

- Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype.
- Represented by **wchar_t**. It is generally 2 or 4 bytes long.

DATA TYPE	SIZE (IN BYTES)	RANGE
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

2.Derived Data Types: The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

i. Function
ii. Array
iii. Pointer
iv. Reference

3.Abstract or User-Defined Data Types: These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

i. Class
ii. Structure
iii. Union

iv. Enumeration
v. Typedef defined DataType

C++ TOKENS

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators

1.Keyword:

- Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program.
- Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. You cannot redefine keywords. However, you can specify text to be substituted for keywords before compilation by using C/C++ preprocessor directives.**C** language supports **32** keywords which are given below:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**

char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

While in **C++** there are **31** additional keywords other than **C** Keywords they are:

asm	**bool**	**catch**	**class**
const_cast	**delete**	**dynamic_cast**	**explicit**
export	**false**	**friend**	**inline**
mutable	**namespace**	**new**	**operator**
private	**protected**	**public**	**reinterpret_cast**
static_cast	**template**	**this**	**throw**
true	**try**	**typeid**	**typename**
using	**virtual**	**wchar_t**	

2.Identifiers: Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character. Identifier names must differ in spelling and case from any keywords. You cannot use keywords as identifiers; they are reserved for special use. Once declared, you can use the identifier in later program statements to refer to the associated value. A special kind of identifier, called a statement label, can be used in goto statements.

There are certain rules that should be followed while naming c identifiers:

- They must begin with a letter or underscore(_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only first 31 characters are significant.

Some examples of c identifiers:

NAME	**REMARK**
_A9	Valid
Temp.var	Invalid as it contains special character other than the underscore
void	Invalid as it is a keyword

3.Constants: Constants are also like normal variables. But, only difference is, their values can not be modified by the program once they are defined.
Constants refer to fixed values. They are also called as literals.
Constants may belong to any of the data type.**Syntax:**
const data_type variable_name; (or) **const data_type *variable_name;**
Types of Constants:

1. Integer constants – Example: 0, 1, 1218, 12482

2. Real or Floating point constants – Example: 0.0, 1203.03, 30486.184
3. Octal & Hexadecimal constants –

   Example: octal: $(013)_8 = (11)_{10}$, Hexadecimal: $(013)_{16} = (19)_{10}$
4. Character constants -Example: ‘a’, ‘A’, ‘z’
5. String constants -Example: “GeeksforGeeks”

4. **Strings:** Strings are nothing but an array of characters ended with a null character (‘\0’).This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas, a character is enclosed in single quotes in C and C++.**Declarations for String:**

   i. char string[20] = {‘g’, ’e’, ‘e’, ‘k’, ‘s’, ‘f’, ‘o’, ‘r’, ‘g’, ’e’, ‘e’, ‘k’, ‘s’, ‘\0’};
   ii. char string[20] = “geeksforgeeks”;
   iii. char string [] = “geeksforgeeks”;

Difference between above declarations are:

- when we declare char as “string[20]”, 20 bytes of memory space is allocated for holding the string value.
- When we declare char as “string[]”, memory space will be allocated as per the requirement during execution of the program.

5.Special Symbols: The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.[] () {}, ; * = #

Brackets[]: Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

Parentheses(): These special symbols are used to indicate function calls and function parameters.

Braces{}: These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.

comma (,): It is used to separate more than one statements like for separating parameters in function calls.

semi colon : It is an operator that essentially invokes something called an initialization list.

asterick (*): It is used to create pointer variable.

assignment operator: It is used to assign values.

pre processor(#): The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

6.Operators: Operators are symbols that triggers an action when applied to C variables and other objects. The data items on which operators act upon are called operands.

Depending on the number of operands that an operator can act upon, operators can be classified as follows:

Unary Operators: Those operators that require only single operand to act upon are known as unary operators.For Example increment and decrement operators

Binary Operators: Those operators that require two operands to act upon are called binary operators.

Binary operators are classified into :

I. Arithmetic operators
II. Relational Operators(<,<=,>,>=,++,!=)
III. Logical Operators(&&,||,!)(or)Compound Operators
IV. Assignment Operators(=,+=,-=,*=,/=,%=)
V. Bitwise Operators
VI. **Ternary Operators(or)** Conditional Operator

: These operators requires three operands to act upon. For Example Conditional operator(?:).

C++ Variables

Variables are containers for storing data values.

In C++, there are different **types** of variables (defined with different keywords), for example:

- `int` - stores integers (whole numbers), without decimals, such as 123 or -123
- `double` - stores floating point numbers, with decimals, such as 19.99 or -19.99
- `char` - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- `string` - stores text, such as "Hello World". String values are surrounded by double quotes
- `bool` - stores values with two states: true or false

Declaring (Creating) Variables:

Syntax:

```
datatype  var_name;
```

Example:

```
int   Num1;
```

Initialization of Variables:

Like C programming, in C++ also we can initialize the variables in 2 ways.

1.While declaration of the variables

2. Through keyboard

1. While declaration of the variables

Syntax:

```
datatype  var_name=value;
```

Example:

```
int   Num1=10;
```

2. Through keyboard

In this we can use cin,cout objects to read the values through keyboard.

Example:

```
int main()
{
      int a,b;
     cout<<"Enter the values of a,b: ";
    cin>>a>>b;
}
```

Reading values through keyboard

Control Statements

The Statements which controls the execution flow of the program is called "Control Statements"

In C++ we have different types of control statements. They are

1. Conditional (or) Decision making Statements

2. Loop Statements(or)Iterative Statements

3. Jump Statements

1. Conditional (or) Decision making Statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

C++ programming language provides following types of decision making statements.They are

1.Simple if Statement

2.if-else Statement

3.Nested if-else Statement

1. Simple if Statement

An **if** statement consists of a boolean expression followed by one or more statements.

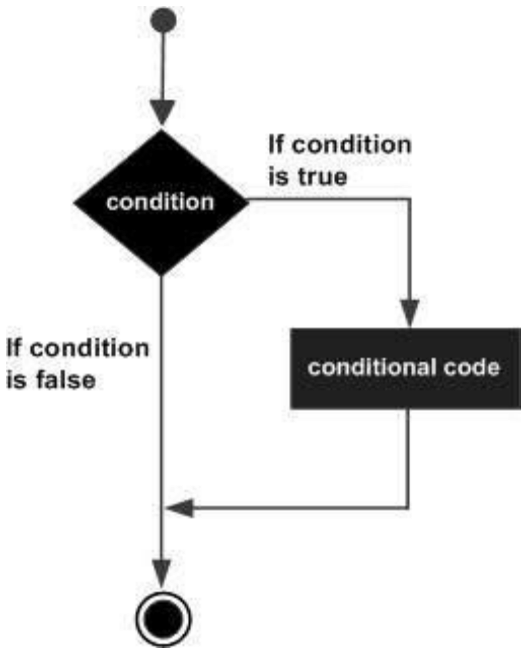
Syntax:

The syntax of an if statement in C++ is −

```
if(boolean_expression)
{
  // statement(s) will execute if the
boolean expression is true
}
```

If the boolean expression evaluates to **true**, then the block of code inside the if statement will be executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram:



Example:

```
#include <iostream>
```

```
using namespace std;

int main ()

{

   // local variable declaration:

   int a = 10;

    // check the boolean condition

   if( a < 20 ) {

      // if condition is true then print the following

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

   }

   cout << "value of a is : " << a << endl;

    return 0;

}
```

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

```
a is less than 20;

value of a is : 10
```

2. if-else Statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

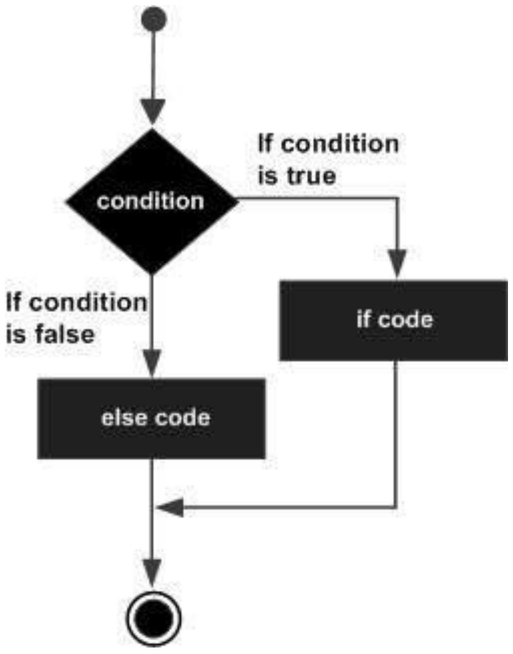
Syntax:

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

```
if(boolean_expression)
{
   // statement(s) will execute if the
boolean expression is true
} else {
  // statement(s) will execute if the
boolean expression is false
}
```

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

FLOW DIAGRAM



Example:

```
#include <iostream>

using namespace std;
```

```
int main () {
  // local variable declaration:
  int a = 100;
   // check the boolean condition
  if( a < 20 ) {
     // if condition is true then print the following
     cout << "a is less than 20;" << endl;
  } else {
     // if condition is false then print the following
     cout << "a is not less than 20;" << endl;
  }
  cout << "value of a is : " << a << endl;
   return 0;
}
```

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

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

3.Nested If Statements

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

Syntax:

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

```
if( boolean_expression 1)
{
  // Executes when the boolean expression 1 is true
  if(boolean_expression 2)
{
    // Executes when the boolean expression 2 is true
  }
}
```

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

Example:

```
#include <iostream>
using namespace std;
int main () {
  // local variable declaration:
  int a = 100;
  int b = 200;
   // check the boolean condition
  if( a == 100 ) {
    // if condition is true then check the following
```

```
      if( b == 200 ) {
         // if condition is true then print the following
         cout << "Value of a is 100 and b is 200" << endl;
      }
   }
   cout << "Exact value of a is : " << a << endl;
   cout << "Exact value of b is : " << b << endl;
    return 0;
}
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

<u>4.else-if ladder</u>

An **if** statement can be followed by an optional **else if...else** statement, which is very usefull to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

<u>Syntax:</u>

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

```
if(boolean_expression 1) {
   // Executes when the boolean expression 1 is true
} else if( boolean_expression 2) {
   // Executes when the boolean expression 2 is true
} else if( boolean_expression 3) {
   // Executes when the boolean expression 3 is true
} else {
   // executes when the none of the above condition is
true.
}
```

Example:

```
#include <iostream>
using namespace std;
 int main () {
   // local variable declaration:
   int a = 100;
    // check the boolean condition
   if( a == 10 ) {
      // if condition is true then print the following
      cout << "Value of a is 10" << endl;
   } else if( a == 20 ) {
      // if else if condition is true
      cout << "Value of a is 20" << endl;
   } else if( a == 30 ) {
      // if else if condition is true
      cout << "Value of a is 30" << endl;
   } else {
      // if none of the conditions is true
      cout << "Value of a is not matching" << endl;
   }
   cout << "Exact value of a is : " << a << endl;

   return 0;
}
```

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

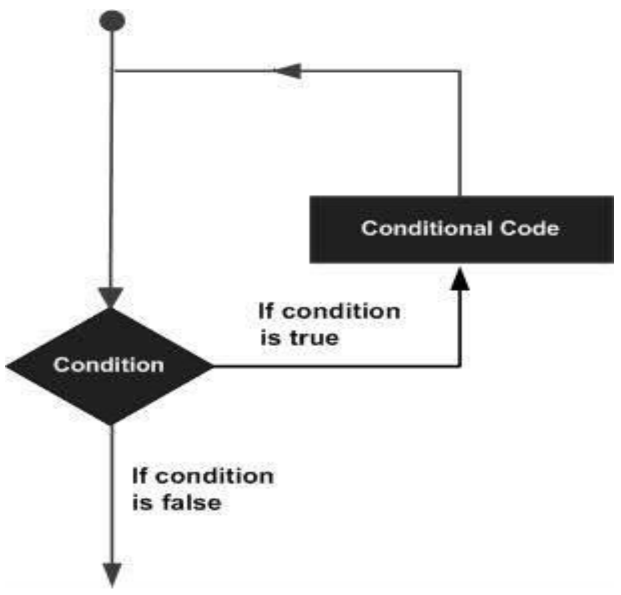
```
Value of a is not matching
Exact value of a is : 100
```

Loop Statements:

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C++ programming language provides the following type of loops to handle looping requirements.

1.while loop

2.do-while loop

3.for loop

4.Nested for loop

1.while loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax:

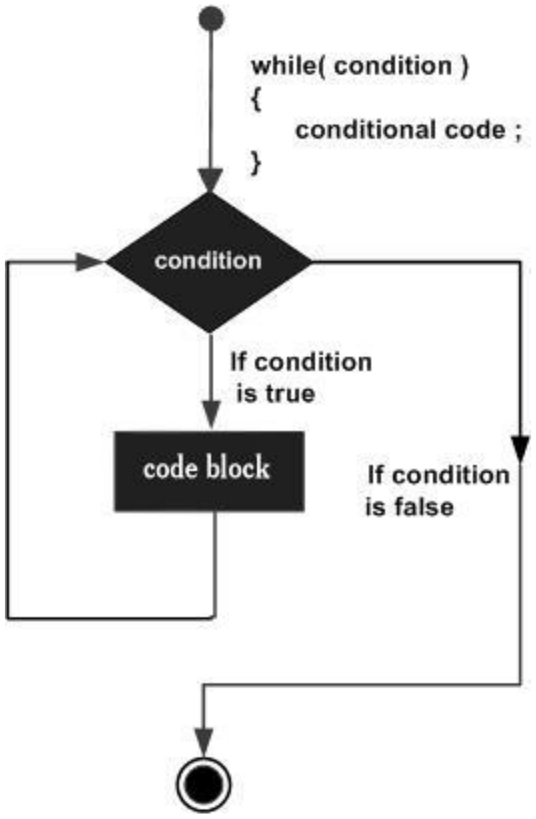
The syntax of a while loop in C++ is −

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

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram:



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
#include <iostream>
using namespace std;
 int main () {
   // Local variable declaration:
   int a = 10;
   // while loop execution
   while( a < 20 ) {
      cout << "value of a: " << a << endl;
```

```
                a++;
              }
             return 0;
           }
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

2.do-while loop:

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

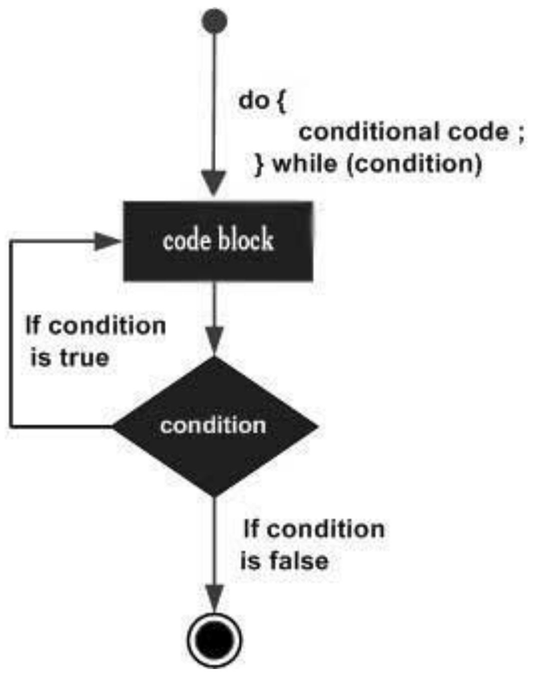
Syntax:

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram:



Example:

```
#include <iostream>

using namespace std;
```

```
int main () {
   // Local variable declaration:
   int a = 10;
   // do loop execution
   do {
      cout << "value of a: " << a << endl;
      a = a + 1;
   } while( a < 20 );
    return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

3.for loop:

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

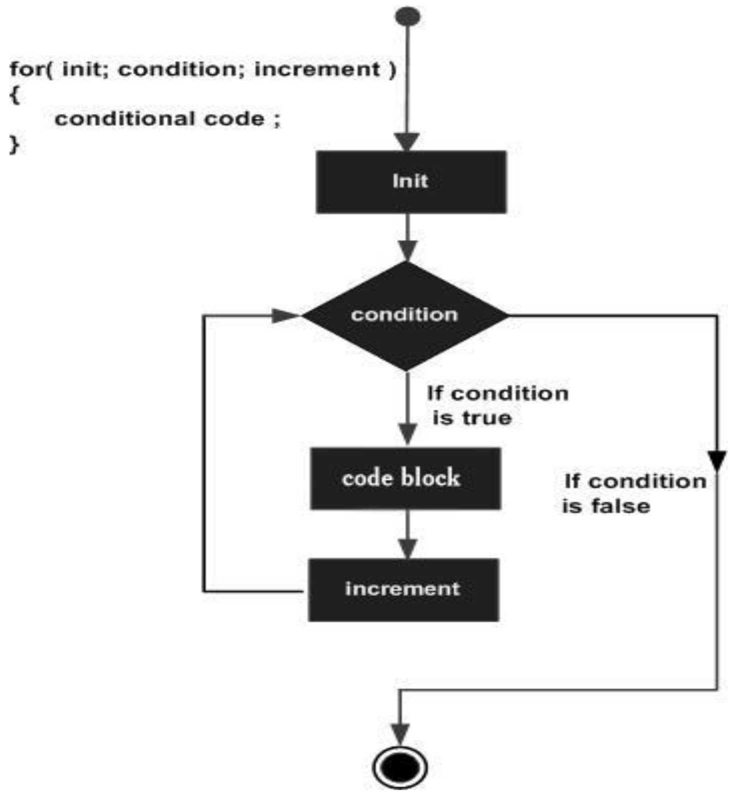
Syntax:

```
for ( init; condition; increment )
 {
   statement(s);
}
```

Here is the flow of control in a for loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example:

```
#include <iostream>

using namespace std;

 int main () {

   // for loop execution

   for( int a = 10; a < 20; a = a + 1 ) {

      cout << "value of a: " << a << endl;

   }

    return 0;

}
```

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

```
value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19
```

4.Nested for loop:

A loop can be nested inside of another loop. C++ allows at least 256 levels of nesting.

Syntax:

```
for ( init; condition; increment ) {
   for ( init; condition; increment ) {
      statement(s);
   }
   statement(s); // you can put more statements.
}
```

Example:

```
#include <iostream>
using namespace std;
 int main () {
   int i, j;
      for(i = 2; i<100; i++) {
      for(j = 2; j <= (i/j); j++)
         if(!(i%j)) break; // if factor found, not prime
         if(j > (i/j)) cout << i << " is prime\n";
   }
      return 0;
}
```

Output:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
```

```
31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime
```

Functions

Introduction to Functions:

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantages of Functions

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Example:

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

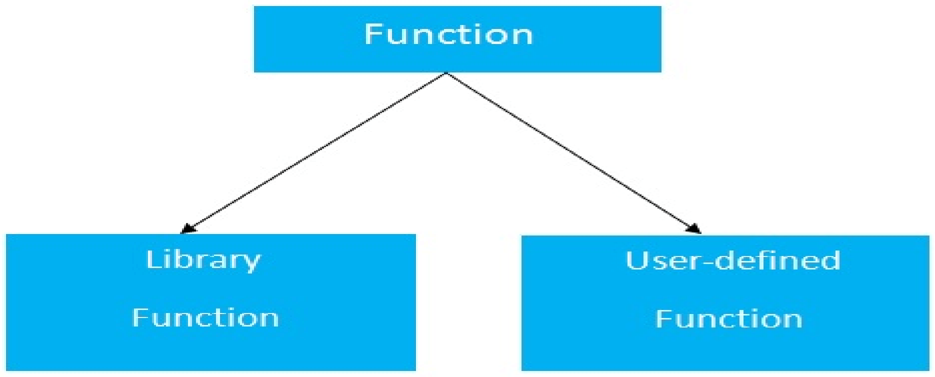
But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions:

There are two types of functions in C++ programming:

1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: These are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Declaration of Functions

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...)
{
        //code to be executed
}
```

Example:

```
#include <iostream>
using namespace std;
void func() {
   static int i=0; //static variable
   int j=0; //local variable
```

```
  i++;
  j++;
  cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
}
```

Inline Functions in C++

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To make a function as inline function, place the keyword '**inline'** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

Syntax:

```
inline return-type function-name(parameters)
{
   // function code
}
```

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers −

```
#include <iostream>
 using namespace std;
inline int Max(int x, int y)
 {
   return (x > y)? x : y;
}
// Main function for the program
int main()
 {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
      return 0;
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while)
2) If a function contains static variables.
3) If a function is recursive.
4) If a function return type is other than void, and the return statement doesn't exist in function body.
5) If a function contains switch or goto statement.

CommandLine Arguments in C++

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main()
{ /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.

To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[])
{ /* ... */ }
```

(or)

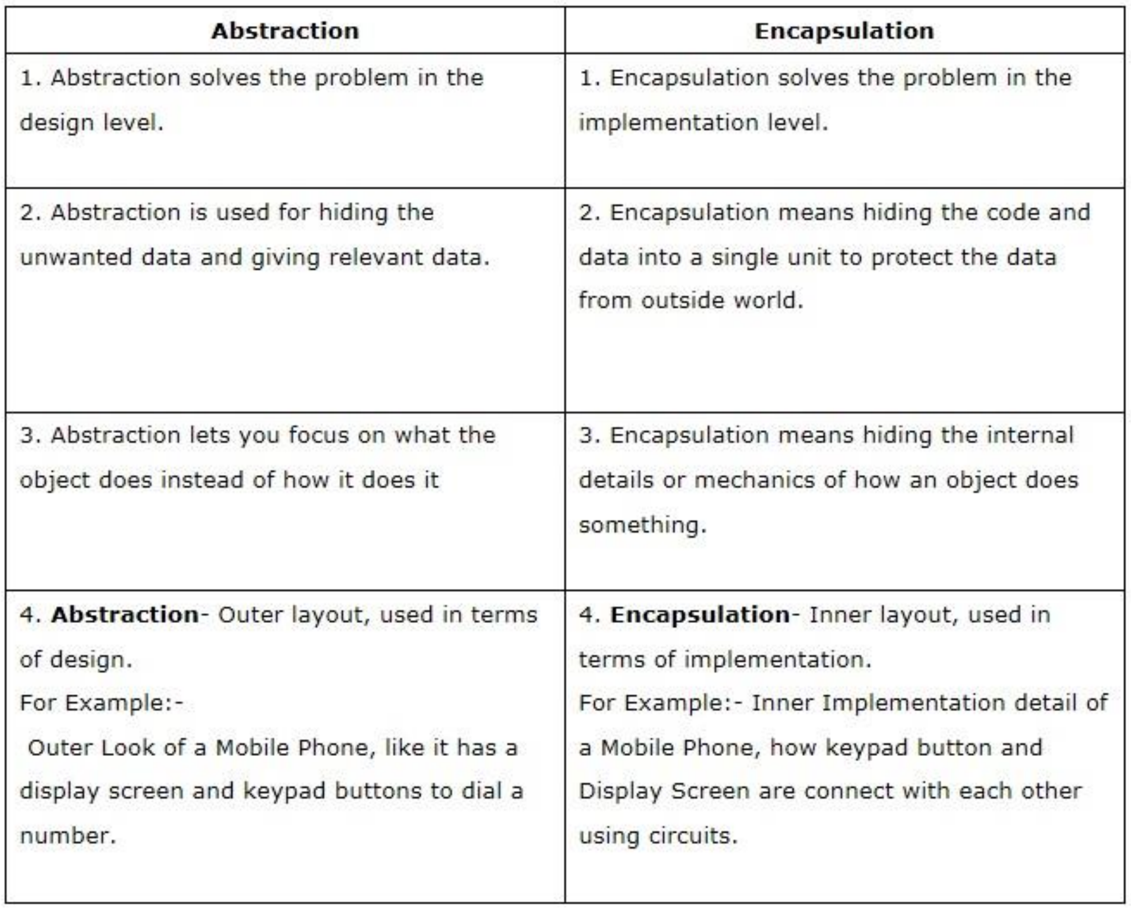
```
int main(int argc, char **argv)
{ /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

Example:

```
// Mainreturn.cpp
#include <iostream>
using namespace std;
 int main(int argc, char** argv)
{
    cout << "You have entered " << argc
         << " arguments:" << "\n";
    for (int i = 0; i < argc; ++i)
        cout << argv[i] << "\n";
      return 0;
}
```

Differences Between Abstraction and Encapsulation



Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. **Abstraction**- Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. **Encapsulation**- Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.