Unit - I

Basic Concepts of OOPs

Overview of C++:

- C++ is an extension of C programming language.
- C++ was first invented by 'Bjarne Stroustrup' in 1979 at Bell Laboratories, USA.
- Initial name of C++ was **"C with Classes"**, renamed as C++ in 1983 by 'Rick Mascitti'
- The invention of C++was necessitated by major programming factor, increasing complexity.
- C program is so complex that is difficult to grasp as a totality if the program exceeds from 25000 to 100000 lines of code. C++ overcomes this problem.
- C++ allows us to comprehend and manage larger, more complex programs.
- Most additions made to C, supports Object-Oriented Programming (OOP).
- Some features of OOP were inspired by another programming language Simula67. So C++ is blending of two powerful languages C and Simula67.
- First revision of C++ was made in the year 1985.
- Second revision of C++ was made in the year 1990.
- The first draft of the proposed standard C++ was jointly created by ANSI and ISO on January 25th 1994.
- The second draft of the proposed standard C++ was on November 11th 1997.
- Final standard C++ became reality in 1998
- C++ contains many advanced and new features along with the C features.

Characteristics of OOP

- Emphasis is on data rather than procedure
- Programs are divided into objects
- Data can be hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up programming approach.

Application of OOPs:

- Real-time systems and online application systems such as Air traffic control, Airline seat reservation, Railway ticket reservation, Bank transactions etc.,
- Simulation and modeling of vehicle design and performance in motor industry.
- Object-oriented database design and applications.
- Multimedia, Hypertext, hypermedia applications and other internet applications.
- Neural networks and parallel programming
- Office automation and data processing systems
- CIM / CAM / CAD systems.

Benefits of OOPs:

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.

- Through inheritance, we can increase reusability of code.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.
- Object-oriented system can be easily upgraded from small to large systems.

Limitations of OOPs:

- Data is not freely available for all functions
- Data cannot move around the program or system to satisfy the requirement of individual function.
- Defined action can only be implemented on associated data members. Instantaneously, it is not possible to define random actions.
- Sometimes one may not feel it convenient to write programs if data members and functions could not be separated out.
- For every new operation, user or programmer must define a member function and appropriate changes must reflect in class definition.
- OOP is thus complex and tedious if one does not know the design features of his problem to be solved interns of objects and its methods.

What is Object Oriented Programming (OOP)?

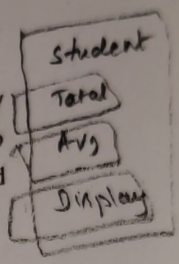
OOP is a powerful way to approach the job of programming. OOP was created to help programmers break through the several C barriers.

To support the principles of OOP, all OOP languages have three traits in common: **Encapsulation, Polymorphism, and Inheritance**. *(These will explain in basic concepts of OOPs in the next section).*

Basic Concepts of OOPs:

It is necessary to understand the basic concepts used extensively in object-oriented programming. These include:

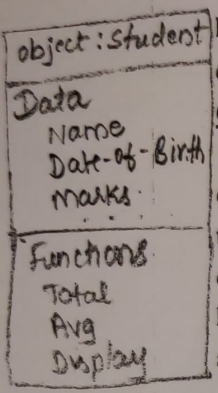
- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing



Objects:

Objects are the **basic run time entities** in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. Program objects should be chosen such that they match closely with the real-world objects.

Two ways of representing.



Classes:

A class is thus a collection of objects of similar type. A class is a specification describing a new data form and an object in a particular data structure constructed according to the requirement. The entire set of data and a code of an object can be made a user defined data type with the help of a *class*. In fact, objects are variables of the type *class*. Once a *class* has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created.

A **class** is a user defined data type consisting of private and public data members and member functions in a single unit. It is the fundamental building block of object oriented program.

A class definition contains two parts:

i) class head ii) class body

The class head is made up of the keyword **class** followed by the **name of the class.** The class body is the portion of the class definition enclosed within a pair of curly braces. The class body consists of both **data** and **functions.**

The data items used in a class are called **data members** and the functions defined are called as **member functions**. The class definition is terminated by a semicolon (;).

The general form of a class is:

```
class class_name
{
  private:                          // Not Mandatory, Default is private
     Variable declarations;
     Function declarations;         //Normally it is not be available in program

  public:
     Variable declarations;         //Normally it is not be available in program
     Function declarations;

};
```

Sample C++ Program:

```
#include<iostream.h>
class Sample
{
        public:
        void display( ) .
        {
                cout << " This is my first program in C++ \n ";
        }
};
void main( )
{
        Sample S;
        S.display( );
        getch( );
}
```

OUTPUT:
This is my first program in C++

Here, iostream.h - is an header file in C++ like stdio.h in C
class - is a keyword
Sample - is a name of the class (It is user defined)
public - is a keyword (which is a access specifier /visibility mode)
cout - is a keyword used for printing (Same as printf( ) in C)
<< - is an insertion (output) operator. Can take any datatype like C.
}; - Indicates end of the class.
s - is an object (User Defined) of the class Sample
s.display( ) - is the way of access the member function in C++.

Private and Public: (Data Hiding)

A key feature of object-oriented programming is data hiding. This means, the data is concealed within a class. The primary mechanism for hiding data is to put it in a class and make it **private**. The private data and functions can only be accessed within the class. **Public** data or functions on the other hand are accessible from outside the class. The keyword **private** and **public** are known as visibility labels. These keywords are followed by a **colon.**

The use of the keyword **private** is optional. By default, the members of a class are **private**. If both the labels are missing, then, by default, all the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.

```
class student
{
        int regno;
        char name[10];
        public:  void getdata()
                 {
                        cout << "Enter the register no. and Name\n";
```

```
        cin >> regno>>"\n">>name;
    }
    void putdata()
    {
        cout << "Register No.: " << regno << "\n";
        cout << "Name : " << name << "\n";
    }
};
void main()
{
    student S;
    S.getdata();
    S.putdata();
}
```

OUTPUT:

Enter the register no. & Name

3456

~~Enter the name~~

Pavithra D R

Register No.: 3456

Name: Pavithra D R

We usually give a class, some meaningful name, such as **student**. This name now becomes a new type identifier that can be used to declare ***instances*** of that class type. The class item contains two data members and two function members. The function **getdata( )** can be used to assign values to the member variables number and cost, and **putdata( )** for displaying their values. The data members are usually declared as **private** and the member functions as **public**.

Creating Objects:

Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

```
student s;          //memory for the object s is created
```

creates a variable s of type **student**. In C++, the class variables are known as **objects**. Therefore, s is called an object of type **student**. We may also declare more than one object in one statement. For example:

```
student s1, s2, s3;
```

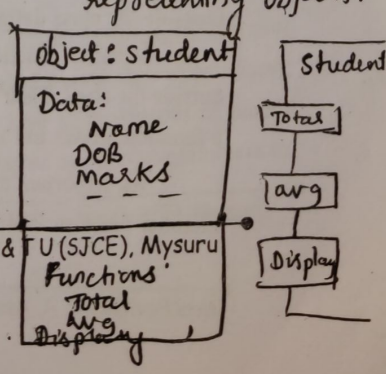
The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class student
{
    .....
} s1, s2, s3;
```

would create the object s1, s2 and s3 of type student.

Two ways of representing objects.



Accessing Members:

A member function of a class can be called by an object of that class followed by a dot operator followed by member function name.

For example,

```
s.getdata( );    // Here, s is an object, and getdata is a member function of the class
s.putdata( );    // Here, s is an object, and putdata is a member function of the class
```

Defining Member Functions:

Member functions can be defined in two places:

- Inside the class definition
- Outside the class definition

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. But there is a subtle difference in the way the function header is defined.

Inside the class definition:

In this method defining a member function is to replace the function declaration by the actual function definition inside the class. For example we could define the student class as follows:

```
class student
{
        int regno;
        float percentage;
        public:
        void getdata()
        {
                cout << "Enter the register no. and Percentage \n";
                cin >> regno >> percentage;
        }
        void putdata()
        {
                cout << "Reg No.: "<< regno << " \nPercetage : " << percentage;
        }
};
```

OUTPUT:

```
Enter the register no. and Percentage
142536987    88.25
Reg No.: 142536987
Percentage: 88.25
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an **inline** function are also applicable here.

Outside the class definition:

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells

the compiler which class the function belongs to. The general form of a member function definition is:

```
return-type class name :: function-name (argument declaration)
{
    function body
}
```

The membership label class-name with :: tells the compiler that the function function-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified in the header line. The symbol :: is called the scope resolution operator.

For instance, consider the member functions getdata( ) and putdata( ) of class student as discussed above. They can be coded as follows:

```
void student :: getdata( )
{
        cout << "Enter the register no.and Percentage \n";
        cin >> regno << percentage;

}
void student :: putdata( )
{
        cout << "Reg No.: "<< regno << " \nPercetage: " << percentage;
}
```

Data Abstraction and Encapsulation:

The wrapping up of data and functions into a single unit (called class) is known as ***encapsulation.*** Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called ***data hiding*** or ***information hiding.***

Abstraction refers to the act of representing essential features without including the background details of explanations. Classes use the concept of abstraction and are defined as a list of abstract *attributes* like size, weight, cost, and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called *methods* or *member functions*. Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

Inheritance:

Inheritance is a process of acquiring (copying) the properties from one class/object to another class/object. In OOP, the concept of inheritance provides the idea of ***reusability.*** This means that we can add additional features to an existing class without modifying it. The new class is called **child class (derived class).** The old class is called **parent class (base class).** The child class defines only the additional features and will have all the features of parent class. Once a class has been written and tested, it can be adapted by other programmers to

suit their requirements. This is basically done by creating new classes and reusing the properties of the existing ones. Reuse of code gives saving memory and time of writing code; reduce the length of the program.

Base class and Derived class:

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived_class_name : visibility-mode base-class-name
 {
     .....
 };
```

where,

class	-	is a keyword
derived_class_name	-	name of the derived class
:	-	shows the derivation from the base-class
visibility mode	-	specifies the types of derivation
base-class-name	-	name of the base class

Example:

```
class A
{
        int x, y;
        public:
            void setvalue(int a, int b)
            {
                x = a;
                y = b;
            }
            void display ( )
            {
                cout << x << "   " << y << "\n" ;
            }
};
class B ; public A
{
    int z;
    public:
        B (int k)
        {
            z = k;
        }

        void displayz ( )
```

```
        {
            cout <<    z << "\n";
        }
};
void main( )
{
        B dobj(5);
        dobj.setvalue(10, 20);
        dobj.display( );
        dobj.displayz( );
        getch( );
}
```

OUTPUT:
10 20 5

Types of Inheritance:

A derived class extends its features by inheriting some or all the properties from its base class and adding new features of its own. While inheriting, the derived class can share properties from only one class, more than one class, and more than one level.

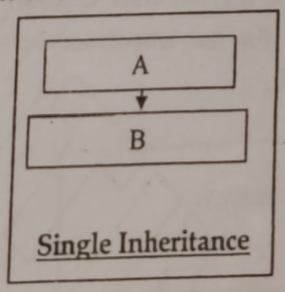
Based on the relationship, inheritance can be classified into **five** forms.

1. Simple or single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

1. Single Inheritance:

Deriving a single new class from a single base class is called simple or single inheritance. The single inheritance possesses one-to-one relation.

```
class A
{
    public: int x,y;
    .....
};
class B : public A
{
    public: int a,b;
    .....
};
```



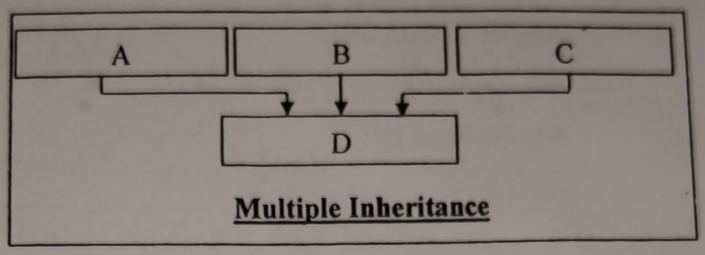
Single Inheritance

Here, public members of baseclas are become public members of derivedclas.

2. Multiple Inheritance:

Multiple inheritance is a mechanism in which a new class is derived from more than one class. It follows many-to-one relation.

```
class A
 {
  public: int x,y;
  ......
 };
class B
 {
  public: int a,b;
  ......
 };
class C
 {
  public: int m,n;
  ......
 };
class D : public A, public B, public C
 {
  ......
 };
```



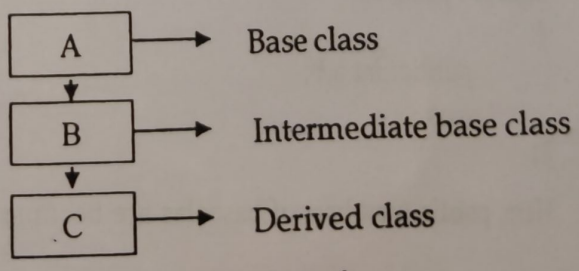
Multiple Inheritance

Here, public members of all the 3 baseclasses are become public members of D.

3. Multilevel inheritance:

Multilevel inheritance is a mechanism in which we derive a class from another derived class. It's just like **grandfather-father-son** relationship for exhibiting multilevel inheritance. Class **A** serves as base class for the derived class **B**, which in turn serves as a base class for the derived class **C**. Class **B** is known as **intermediate** base class since it provides a link for the inheritance between **A** and **C**. The chain ABC is known as **inheritance path**.

```
class A
{
  public: int x,y;
  ......
};
class B : public A
{
  public: int p, q;
};
class C : public B
{
  ......
};
```



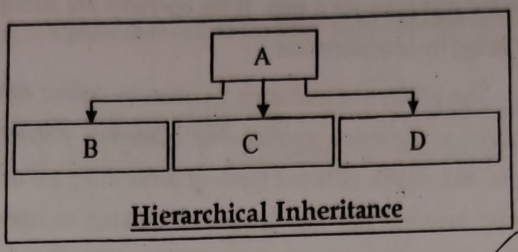
Multilevel Inheritance

Here, public members of calss A (i.e., x, y) become public member class B and finally public members class B (i.e., p, q, x, and y) become public members of derivedclas.

4. Hierarchical Inheritance:

Hierarchical inheritance is a mechanism in which the features of one class may be inherited by more than one class. These classes, in turn, may be inherited. In the hierarchical inheritance, there will be only one base class from which several other classes are created. This form of inheritance follows one-to-many relationship. A derived class can be created by deriving the properties of base class and adding its own properties.

```
class A
{
   public: int x, y;
   ......
};
class B : public A
{
   public: int p, q;
   ......
};
class C : public A
{
   public: int m, n;
   ......      };
class D : public A
{
   public: int i, j;
   ......
};      //Here, public members of A (i.e., x, y) become public member B, C, D
```

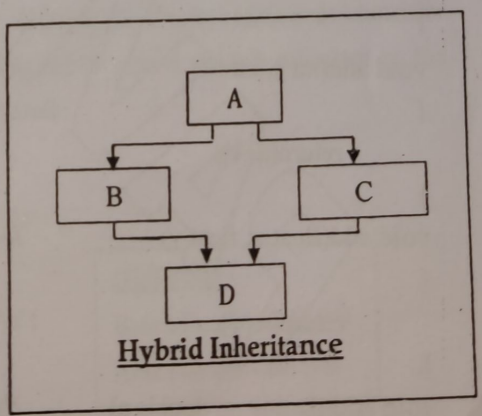


Hierarchical Inheritance

5. Hybrid inheritance:

Hybrid inheritance is a mechanism in which the program design would require two or more forms of inheritance. Here, the derived class may be created from the multilevel and multiple classes. Or, it could be created from the hierarchical and multiple classes.

```
class A
{
   public: int x, y;
   ......
};
class B : public A
{
   public: int a, b;
   ......
};
class C : public A
{
   public: int p, q;
   ......
};
class D : public B, public C
{
   public: int m, n;
   ......  };
```



Hybrid Inheritance

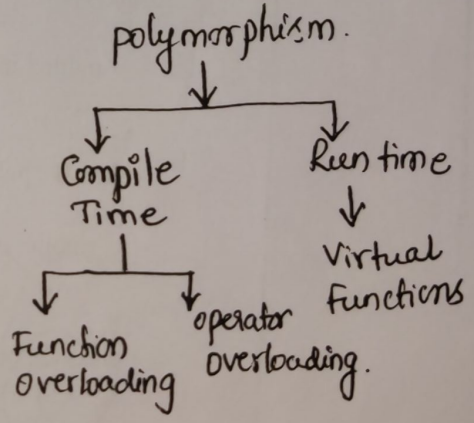
Polymorphism:

Polymorphism is an ability to take more than one form. An operation may exhibit different behaviors in different instances. The behaviors depend upon the types of data used in operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

The process of making a function to exhibit different behaviors in different instances is known as ***function overloading (function polymorphism)***. In other words, the same function name with different types of parameters are used to perform various task is called as ***function overloading***. The process of making an operator to exhibit different behaviors in different instances is known as ***operator overloading (operator polymorphism)***.

For Example:

```
void main ( )
{
        int a, b, sum;
        float x, y, total;
        double m, n, gross;
        cout<<"Enter the values for a, b, x, y, m, and n\");
        cin>>a>>b>>x>>y>>m>>n;
        sum = add(a, b);                    // Function name is add( )
        total = add(x, y);                  // Function name is add( )
        gross = add(m, n);                  // Function name is add( )
        cout<<"Sum of a and b is: "<< sum <<"\n";
        cout<<"Sum of x and y is: "<< total <<"\n";
        cout<<"Sum of m and n is: "<< gross <<"\n";
}
void add(int a, int b)                      // Function name is add( )
{
        return(a+b);
}
void add(float x, float y)                  // Function name is add( )
{
        return(x+y);
}
void add(double m, double n)                // Function name is add( )
{
        return(m+n);
}
```



In the above program, contains three user defined functions in the name **add( )** but in all the three functions, type of parameters used are dissimilar (i.e., a and b are integer, x and y are float, m and n are double datatype).

A same function name used for multiple purposes with dissimilar type of parameter is called as function overloading.

Dynamic Binding:

Binding refers to the linking of a procedure call to the code to be executed in response to the call.

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance.

Message Passing:

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

a) Creating classes that define objects and their behavior.
b) Creating objects from class definitions, and
c) Establisning communication among objects.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. *Message passing* involves specifying the name of the function (message) and the information to be sent.

Declaration of variables:

There are two types of variables declared in functions.

i) Local variable ii) Global variable

i) Local variable:

Variables declared within functions are called local variables. The scope of a local variable is confined to the function; in particular its value and meaning are well defined within the body of that function. These local variables cannot be accessed outside that function. But they can be redefined in some other function can locally redefine an identifier or variable defined outside the function. Thus the scope of a local variable is limited to the extent of the body of the function in which it is declared.

Example for Local variables:

```
void main()
{
    int i=3;
    cout << "Before Function Call: i=" << i << "\n";
    modify ();
    cout << "After Function Call: i=" << i << "\n";
}
```

OUT PUT:

Before Function Call: i=3
After Function Call: i=5

```
void modify()
{
    int i=5;
}
```

ii) Global variables:

A global variable is declared outside of any functions. Hence they do not belong to any particular function. Scope of a global variable specify that the extent of the meaning of that variables is throughout the program and its meaning is not at all local to any particular function.

Therefore any function in a C program can access the value and meaning of the global variable and these functions can change the value of global variable.

Example for Global variables:

```
a=30, b=20;
void main()
{
    cout << "The value of a and b in Main function \n" ;
    cout << "a = " << a << "\n" << "b = " << b << "\n";
    global( );
}
void global()
{
    cout << "The value of a and b in function global( ) \n";
    cout << "a = " << a << "\n" << "b = " << b << "\n";
}
```

OUT PUT:

The value of a and b in function Main( )
a=30
b=20
The value of a and b in function global( )
a=30
b=20

Dynamic Initialization of Variables

The process of initializing variable at the time of its declaration at run time is known as dynamic initialization of variable.

Thus in dynamic initialization of variable a variable is assigned value at run time at the time of its declaration.

Example for dynamic initialization of variables:

```
void main( )
{
    int a;
    cout << "Enter Value of a";
    cin >> a;
    int cube = a * a * a;
}
```

In above example variable cube is initialized at run time using expression a * a * a at the time of its declaration

Reference Variables:

Reference Variables (or references in short), is an *alias*, or an *alternate name* to an existing variable, that we can use to read or modify the original data stored in that variable. When we declare a reference and assign it a variable, it will allow us to treat the reference

exactly as though it were the original variable for the purpose of accessing and modifying the value of the original variable even if the second name (the reference) is located within a different scope.

Declaring a variable as a reference rather than a normal variable simply entails appending an *ampersand* to the type name.

Syntax:

Datatype& reference_variable_name = original_variable_name;

Example:

```
int X;
int& foo = X; // Here, X is an Original Variable and foo is a reference variable to X
foo = 56;
```

In a way, this is similar to having a pointer that always points to the same thing. One key difference is that references do not require dereferencing like pointers do; we just treat them as normal variables. A second difference is that when we create a reference to a variable, we need not do anything special to get the memory address. The compiler figures this out for us.

Operators in C++:

C++ supports all the operators (Such as arithmatic, logical, relational, conditional, assignment, increment/decrement, bitwise operators, and special operators like comma operator, dot operator, arrow operator, * operator, & operator, and sizeof( ) operator) available in C Programming. Along with these operators C++ provides following operator.

- << operator
- >> operator
- cast operator
- Memory Management operator

Input Operator (>>):

The symbol >> is called an **extraction operator**. The >> operator for accepting standard data types viz., int, float, char, double, long etc., The input operator >> sends the data on its right . For example,

cin >> radius;

which causes the program execution to wait for the user or programmer to type in a number or numeric value as radius of a circle. Consider another example,

cin >> a>>b>>c;

The input statement cin prompts the user to type-in 3 integer numbers. If we enter the numbers 3 2 4 the first operator >> takes value 3 from its stream object **cin** and places it in the variable **a** that follows on its right. A second operator >> reads a value **2** and stores it in the variable **b**. Similarly third operator >> extracts a value **4** from its input stream on its left and copies it into third variable **c**.

The multiple use of >> in statement is known as cascading.

Output Operator (<<):

The output operator << is called the **insertion or put to** operator. It inserts the contents of the variable on its right to the object on its left.

We can also include variables with single cout whose contents are displayed simultaneously. For example,

cout << "SUM ="<<sum << "\n";

first sends the string "SUM =" to cout and then sends the value of sum. Finally it sends the new line character. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank space between different items.

The above cout statement can be written as two cout statements to give same output effect i.e.,

cout << "SUM=";

cout << sum;

Using the cascading technique, the last two statements can be combined as follows:

cout << "SUM = " << sum << "\n" << "AVERAGE = " << average << "\n" ;

Memory Management Operator:

The two operators **new** and **delete** are the C++ mechanism that perform dynamic memory allocation and deallocation respectively. An advantage of using these operators involve the existence of a data object or value created by **new**, until it is explicitly destroyed by **delete** operator. Thus user has an explicit control over the allocation and deallocation of memory for variables of fundamental types, arrays, structures etc., Since these operators manipulate memory on the free store, they are also known as ***free store operators.***

The **new** operator with the pointer to an object allocates memory for that object and assigns the address of that memory to the pointer. But the **delete** operator does the reverse. i.e., it returns the memory occupied by the object back to the **heap**.

The **new** operator can be used to create objects of any type. Syntax:

pointer-variable = **new** data-type;

Here, *pointer-variable* is a pointer of type *data-type*..

The **new** operator allocates sufficient memory to hold a data object of type *data-type* and return the address of the object. The *data-type* may be any valid data type. The pointer-variable holds the address of the memory space allocated. For example:

int *p = **new** int;

float *q = **new** float;

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

delete pointer-variable;

Since **heap** is finite, it can become exhausted. If there is insufficient memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception (generated by the header **new**) will be generated.

Example: **delete** p;

Finally it frees the dynamically allocated memory by the delete operator **only with the valid pointer previously allocated** by using the new.

If sufficient memory is not available for allocation the new returns a null pointer.

The new operator offers the following advantages over the function **malloc( )**.

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialise the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.

Program to illustrate memory management operators (//new_del.cpp)

```
#include<iostream.h>
#include<new.h>
void main()
{
  int *a = new int;
  int *b = new int;
  int *sum = new int;
  cout<<"Enter the values of a,b\n";
  cin>>*a>>*b;
  *sum = *a + *b;
  cout<<"Sum = "<<*sum<<endl;
  delete a;
  delete b;
  getch();
}
```

OUTPUT:

Enter the values of a,b

4

6

Sum = 10

For array Allocation:

```
ptr_var = new array_type[size];
delete [] ptr_var;
```

Example:

```
Ptr = new int[5];
delete [] ptr;
```

The cast operator (Type Casting):

Converting one type of data item to another type using an operator named **cast** is called **casting** or **type casting.** Type casting in C++ is same as the type casting in C Programming. (Refer C programming for more about type casting).

Syntax:

```
(type) expression;
```

Or `type (expression);`

Example 1:

```
(float) x/2;          // Here x/2 evaluates to type float
```

Example 2:

```
float ( x/2 );          // Here also x/2 evaluates to type float
```

Putting bracket either to datatype or to expression is called casting.

Functions in C++:

A function is a self-contained block or a sub-program of one or more statements that perform a special task when called.

Every program starts with user defined function **main( )**. The main( ) calls another function to share the work. C++ language supports two types of functions,

i) Library functions ii) User defined functions

i) C++ Library Functions:

A library function is a built-in, pre-defined subprogram. It defines a series of such ready-made functions, which are supposed to do the assigned work. Their task is limited. The user can only use the function but cannot change or modify them.

For example, sqrt(x), pow(x, y), and strcmp(a, b) etc.,

ii) User Defined Functions:

In addition to the standard library functions supplied by the C++ system, the user can define a function according to his requirement are called as user-defined functions. Programmer can write his own function to perform a specific sub-task. Just as you define a main( ) function. Function definition begins with a name and a set of statements enclosed in braces. The user can modify the function according to the requirement.

Actual arguments and Formal arguments:

These are used as a means for communication between user defined functions and the main( ) function. The variables to be passed to the functions are called **actual arguments**, they are enclosed in a parentheses following the functions name. They may have to either convey input values necessary for computations within the user-defined function or they may be used to return the computed results from user-defined functions to main( ) function.

The **formal arguments**, which receives the value of the actual arguments or address of the actual arguments from the main program and send to the body of the user defined function for specified operation.

Return Statement:

A function can display the results after computation by including a cout statement within it. Sometimes user may wish after that the results need no to be printed by the function itself, instead let the function return the result to its main function (calling routine). The general format of return statement is **return (value);**

The return statement when executed from within a user-defined function returns the value. This value may be a constant, variable or expression that represents useful results of computation. This statement specifies that the function has to return a value to its calling function. Therefore if the function has to return a value to its calling routine, then the last statement executed in the function definition must be a return statement.

Function Declaration:

Function declaration means specifying the function as a variable depending on the return value. It is declared as the part of the main program. It helps the compiler to treat functions differently from other program elements. A function declaration has two principal components, one the name of the function and the pair of parenthesis with or without arguments. The arguments are called formal arguments, because they represent the names of the data items that are transferred into the function from the calling portion of the program. The identifiers used as formal arguments are 'local' in the sense that they are not recognized outside the function. Hence, the names of the arguments need not be the same as the names of the actual arguments in the calling portion of the program, however it must be the same data type.

Write a program to add two numbers by user-defined function

```
int add(int x, int y);          // Function Prototype
void main()
{
  int a, b, sum;
  cout << "Enter the values of a and b:\n";
  cin >> a >> b;
  sum=add(a,b);                 // Function Declaration, a and b are actual arguments
  cout << "The sum of a and b = " << sum;
  getch();
}
int add(int x, int y)           // Function Definition, x and y are Formal Arguments
{
  return(x + y);                //The return statement, returns sum of x+y to main function
}
```

OUTPUT:

```
Enter the values of a and b:
6
4
The sum of a and b = 10
```

Explanation of the above Program:

a, b	- Actual arguments or Arguments
x, y	- Formal Arguments or Formal Parameters or Local Parameters
void main( )	- The calling function (It is a Built in Function)
int add( )	- The **int** is a return type of the return value. The add( ) is user defined Function name
int add(int x,int y)	- The called function. (Function Definition)
return	- It is a keyword to send the output of the function back to the Calling function.
{	- beginning of the main function or user defined function
}	- end of the main function or user defined function
body of the function	- all the statements placed between { and }

Note: The number of the actual arguments should be equal to number of formal arguments.

There must be one-to-one mapping between arguments i.e., they should be in the same order and should be in same data type.

A user defined function can be defined in a C++ program, which may / may not have class and objects concepts in it. If a C++ program does not have class concepts in it, then we can define a user defined function exactly similar to defining user defined function in C programming (See example in page no.17). If a C++ program contains a class concepts in it, then, we can define user defined functions either in inside the class definition and in outside the class definition (Refer Defining Member Functions and examples in page no. 6 and 7).

Argument Passing:

There are two ways in which we can pass arguments to the function.

i) Call by value ii) Call by reference

i) Call by Value:

Here, the value of actual arguments are passed to the formal arguments and the operation is done on the formal arguments. Any change made in the formal argument does not affect the actual arguments because formal arguments are photocopy of actual arguments. Changes made in the formal arguments are local to the block of the called function. Once control returns back to the calling function the changes made vanish.

Write a program to send values using call by value procedure

```
void main( )
{
        int x, y;
        cout << "Enter the values of x and y:\n";
        cin >> x >> y;
        change(x,y);
        cout <<"In main()   x = " << x << " y = " << y;
        return 0;
}
void change(int a, int b)
{
        int temp;
        temp = a;
        a = b;
        b= temp;
        cout <<"In change() x =" << a << "y = "<< b;
}
```

OUTPUT:

Enter the values of x and y:

6

9

In change( ) x=9 y=6

In main( ) x=6 y=9

Note: In this program when the **change( )** is called, the value of **x** and **y** are passed to **a** and **b** respectively. In such cases, the changes made inside the function cannot affect the main program.

ii) Call by Reference:

The variables are stored somewhere in memory. Instead of passing the value of the variables, we can pass the address of the variable. Here, the formal arguments are pointers to the actual arguments. In this type formal arguments point to the actual argument. Hence changes made in the arguments are permanent.

Write a program to send values using call by reference procedure

```
void main()
 {
      int x, y;
      cout << "Enter the values of x and y:\n";
      cin >> x >> y;
      change(&x, &y);
      cout << "In main( )  x = " << "y = " << y;
      getch();
 }
void change(int *a, int *b)
{
      int *temp;
      *temp = *a;
      *a = *b;
      *b= *temp;
      cout << "In change() x = " << *a << " y = " << *b;
}
```

OUTPUT:

```
Enter the values of x and y:
4
8
In change() x=8  y=4
In main()  x=8  y=4
```

Note: In this program when the **change( )** is called, the value of **x** and **y** are passed to **a** and **b** respectively. In such cases, the changes made inside the function affect the main program. This is called as the function call by reference.

Recursive Functions:

In C++, it is possible for the functions to call themselves. ***Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.*** Recursive functions are commonly used in such applications where the solution to the problem can be found in terms of successively applying the same solution (intermediate solution) to the same subsets (sub-tasks) of the problem.

For example to compute the factorial of a given number n we write as

fact = fact * i; where i = 1 to n

In the recursive method to compute the factorial of a given number **n** as follows:

n! = n * (n-1)!;

or

fact(n) = n * fact(n-1)!;

Write a C++ Program to calculate factorial of entered numbers using recursive function

```
void main( )
{
        int x, f;
        cout << "Enter a number \n";
        cin >> x ;
        f = fact(x);
        cout  << "Factorial of" << x << "is" << f ;
}
int fact (int n)
{
        int f = 1;
        if(n= =1)
                return(1);
        else
                f = n * fact (n-1);
                return (f);
}
```

OUTPUT:

Enter a number

6

Factorial of 6 is 720

Inline functions:

One of the objectives of using functions in a program is to save memory space, when a function is likely to be called many times. However, every time a function is called, it takes a lot of time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

An ***inline*** *function is a function that is expanded in line when it is invoked.* That is, the compiler replaces the function call with the corresponding function code. Each time the actual code from the function is inserted into the program in place of function call instead of taking jump into the function.

Syntax:

```
inline function-header
{
        function body
}
```

Example:

```
inline double cube(double a)
{
        return(a*a*a);
}
```

It is easy to make a function inline. All we need to do is to prefix the keyword **inline** to the function definition. All inline functions must be defined before they are called.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch, or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

This makes a program run faster as overhead of a function call and return is eliminated.

- A friend function cannot access the class members directly. An object name followed by dot operator, followed by the individual data member is specified for valid access. Example, if we want to access the member **x** of an object **obj**, then it specified as **obj.x**.
- Member functions of one class can be **friend** functions of another class.
- A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access right to private member of the class.

Write a C++ Program to illustrate friend function

```
class sample
{
        int a, b;
        public:
        void setvalue()
        {
                a=25;
                b=40;
        }
        friend float mean(sample s);
};
float mean(sample s)
{
        return float(s.a + s.b)/2.0;
}
void main()
{
        sample x;                       //object x
        x.setvalue();
        cout<<"Mean value = " << mean(x) << "\n";
        getch();
}
```

OUTPUT:

Mean value = 32.5

The friend function accesses the class variables **a** and **b** by using the dot operator and the object passed to it (Example: **x.a, x.b**). The function call **mean(x)** passes the object **x** by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator, which is shown as follows:

```
class X
{
        int fun1( );                    //member function of X
        .....
};
class Y
{
        friend int X :: fun1( );        //fun1 of X is friend of Y
        .....
};
```

Here, the function **fun1( )** is a member of class **X** and a **friend** of class **Y**

Write a C++ Program to find the largest of three numbers using inline function

```
void main()
{
   int a, b, c;
   cout<<"Enter 3 numbers \n" ;
   cin>> a >> b >> c;
   large(a, b, c);
   getch();
}
inline void large(int x, int y, int z)
{
   int big;
   if(x > y)
        big = x;
   else
        big = y;
   if(z > big)
   big = z;
   cout<< "The largest number is "<<big;
}
```

OUTPUT:
Enter 3 numbers
7
2
6
The largest number is 7

Friend functions:

The *private members of a class* cannot be accessed by non- member functions. But there may be some situations where it is needed to access the private members of a class by the non-member functions. This can be achieved by changing the access specifier private members to public. The attempt to change the access specifier of private members to public violates the concept of data hiding and encapsulation. So, *we can overcome this problem by declaring the non-member functions as* ***friend functions*** *to the class.*

To make a non-member function "friendly" to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
class ABC
{
      public:
      .....
      friend void xyz(void);          // friend function declaration
};
```

Important points to be observed while using 'friend functions':

- The function declaration should be preceded by the keyword **friend**.
- A friend function is not in the scope of the class to which it has been declared as friend.
- A friend function is just like a normal C++ function.
- A friend function can be declared either in the private or the public section of a class.
- Usually, a friend function has the objects as its arguments.

We can also declare all the member functions of one class as the friend function of another class. *In such cases, the class is called a* ***friend class***. This can be specified as follows:

```
class Z
{
    .....
    friend class X;          // All the member functions of X are friend to class Z
};
```

Constructors and Destructors

A **constructor** is a special member function which is used to initialize the data members automatically the moment the objects are created. This is known as ***automatic initialization*** of objects.

A constructor is declared and defined as follows:

```
class sample
{
    int m, n;
    public:
    sample(void);                    //constructor declared
};
sample :: sample(void)               //constructor defined outside class
{
    m = n = 0;
}
```

OR

```
class sample
{
    int m, n;
    public:
    sample(void);                    //constructor defined inside class
    {
        m = 0;
        n = 0;
    }
};
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
sample obj1;                         //Object obj1 is created
```

Characteristics of a Constructor:

- A constructor has the same name as that of class.
- A constructor is declared in the public section of a class definition.
- A constructor is invoked automatically as soon as the class object is created.

- A constructor does not return any value, so return type is not associated with its definition (even void is not used).
- The programmer has no direct control over constructor functions.
- A constructor are not inherited
- The **const** or **volatile** class objects cannot be initialized with a constructor
- A constructor cannot be declared as virtual (Virtual functions).
- A constructor can be overloaded.
- Like other C++ functions, they can have default arguments.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Types of Constructors:

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Dynamic Constructor
- Overloaded Constructor

Default Constructors:

A constructor that accepts no parameters is called the ***default constructor***. The default constructor for **class Sample** is **sample::sample( ).** If no such constructor is

sample a; // Invokes the default constructor of the compiler to create the object **a.**

Program to illustrate default constructor

```
class box
{
        int length, breadth, height;
        public:
        box(void)
        {
                length = breadth = height = 0;
        }
void print(void)
        {
                cout << "length = " << length << endl;
                cout << "breadth = " << breadth << endl;
                cout << "height = " << height << endl;
        }
};
void main()
{
        box obj1;
        obj1.print();
        getch();  }
```

OUTPUT:

length = 0

breadth = 0

height = 0

Parameterized Constructors:

In practice it may be necessary to initialise the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called **parameterised constructors.**

The constructor **sample( )** may be modified to take arguments as shown below:

```
class sample
{
        int m, n;
        public:
        sample(int x, int y);          // Parameterised Constructor
        m = x;
        n = y;
};
```

When a constructor has been parameterized, the object declaration statement such as

sample obj1;

may not work. We must pass the initial values as arguments to the constructor function to which an object is declared. This can be done in two ways:

- By calling the constructor explicitly
- By calling the constructor implicitly

The following declaration illustrates the first method:

```
sample obj1 = sample(0, 100);          //explicit call
```

This statement creates a sample object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
sample obj1(0, 100);                   //implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. The following program demonstrates the passing of arguments to the constructor functions.

Program to illustrate parameterized constructor

```
class box
{
        int length, breadth, height;
        public:
        box(int l, int b, int h)
        {
                length = l;
                breadth = b;
                height = h;
        }
```

```
    void print(void)
    {
        cout << "length = " << length << endl;
        cout << "breadth = " << breadth << endl;
        cout << "height = " << height << endl;
    }
};
void main()
{
    box obj1(2,4,3);
    obj1.print();
    getch();
}
```

OUTPUT:
length = 2
breadth = 4
height = 3

Copy Constructor:

A constructor can accept a *reference* to its own class as a parameter. For example,

sample(sample &i);

is valid. In such cases, the constructor is called the **copy constructor.**

A **copy constructor** is one which constructs another object by copying the member of a given object. It is used to declare and initialize an object from another object. For example, the statement

sample X2(X1);

would define the object X2 and at the same time initialize it to the values of X1. Another form of this statement is

sample X2 = X1;

The process of initializing through a copy constructor is known as ***copy initialization***.

Program to illustrate copy constructor

```
class point
{
    int x,y;
    public:
    point()                    //default constructor
    {
        x = y = 0;
    }
    point(int xp,int yp)       //parameterized constructor
    {
        x = xp;
        y = yp;
    }
    point(point &p)            //copy constructor
    {
        x = p.x;
        y = p.y;
    }
```

```
    void display()
    {
        cout << "x = " << x << "\t" << "y = " << y << "\n";
    }
};
void main()
{
    point p1(20,30);      //object p1 is created and initialised
    point p2(p1);         //copy constructor is called
    point p3;
    p1.display();
    p2.display();
    p3.display();
    getch();
}
```

OUTPUT:

x = 20	y = 30
x = 20	y = 30
x = 0	y = 0

Destructor:

A **destructor** is a special member function which is also having the same name as that of its class but prefixed with tilde(~) symbol. For example, if **sample** is the name of the class then

```
~sample ( );        // is a destructor for the constructor sample( ).
```

Characteristics of a Destructor:

- A destructor is invoked automatically by the compiler upon exit from the program
- A destructor does not return any value
- A destructor cannot be declared as **static, const** or **volatile.**
- A destructor does not accept any arguments and therefore cannot be overloaded.
- A destructor is declared in the public section of a class,
- If the constructor uses the new expression to allocate memory, then the destructor should use the delete expression to free that memory.

Program to illustrate the destructors

```
class sample
{
    int x;
    public:
    sample(void)
    {
        x = 0;
        cout<< "In constructor, x = "<<x<<"\n";
    }
    void printx(void)
    {
        x = 25;
        cout<< "In printx, x = "<<x<<"\n";
    }
```

```
    ~sample()
    {
        cout<< "In destructor, object destroyed";
    }
};
void main()
{
    sample s1;
    s1.printx( );
    getch();
}
```

OUTPUT:

In constructor, x = 0

In printx, x = 25

In destructor, object destroyed

Virtual Functions:

As we know, Polymorphism is one of the crucial features of OOP. It simply means 'having many forms'. Polymorphism can be achieved in two levels, one at compile time and another at run-time. In C++, compile time polymorphism is achieved using overloaded functions and operators, and the run time polymorphism is achieved using the **virtual functions**. The implementation of virtual functions requires the use of pointers to objects.

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword **virtual** preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual functions.

Rules for Virtual Functions:

Following are the some of the basic rules that satisfy the compiler requirements:

1. The virtual functions must be member of some class.
2. They cannot be static member function.
3. They are accessed by using object pointers.
4. A virtual functions can be a friend of another class.
5. They should be declared in public section of the class.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. (If two functions with the same name have different prototypes, C++ considers them as overloaded functions.)
7. While a base pointer can point to any type of the derived object, the reverse is not true.
8. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Write a C++ Program to illustrate Virtual Functions

```
class baseclas
{
    public:
    virtual void virtfunc( )
```

```
    {
        cout<<"This is baseclas's virtfunc ( )\n";
    }
};
class derivedclas1 : public baseclas
{
    public:
    void virtfunc( )
    {
        cout<<"This is derivedclas1's virtfunc ( )\n";
    }
};
class derivedclas2 : public baseclas
{
    public:
    void virtfunc( )
    {
        cout<<"This is derivedclas2's virtfunc ( )\n";
    }
};
void main( )
{
    baseclas *ptr, b;
    derivedclas d1;
    derivedclas d2;
    ptr = &b;                  // Point to baseclas
    ptr -> virtfunc( );        // Access baseclas's virtfunc( )
    ptr = &d1;                 // Point to derivedclas1
    ptr -> virtfunc( );        // Access derivedclas1's virtfunc( )
    ptr = &d2;                 // Point to derivedclas2
    ptr -> virtfunc( );        // Access derivedclas2 virtfunc( )
    getch( );
}
```

OUTPUT:

This is baseclas's virtfunc ( )
This is derivedclas1's virtfunc ( )
This is derivedclas2's virtfunc ( )

Here, the address of the object b is stored in ptr (is a pointer variable). Then we are accessing the member function virtfunc( ) through the ptr. Next the address of the object d1 is stored in ptr and accessing the member function virtfunc( ) through the ptr. Finally the address of the object d2 is stored in ptr and accessing the member function virtfunc( ) through the ptr. It is also possible to access virtfunc( ) through the objects directly. (i.e., b.virtfunc( ))

Differentiate between Procedure Oriented Programming and Object Oriented Programming:

Procedure Oriented Programming	Object Oriented Programming
1) The primary focus is on doing things via algorithm to perform the required computations.	1) The primary focus is on representing data objects
2) Large programs are divided into smaller programs known as functions.	2) Programs are divided into objects. Data structures are designed such that they characterized the objects.
3) It is not possible to hide data for allowing only restricted procedures to access them.	3) Data can be hidden and can only be accessed outside via its associated member function.
4) All data should be declared at the beginning of the program itself.	4) New data and functions can be easily added whenever necessary.
5) It's difficult to design & deliver a solution that resembles the problem. A functions & data structure does not cooperate to model the real world situations.	5) It is powerful tool that enable the programmers to solve the real world problems.
6) Follows *top-down* approach in program design.	6) Follows *bottom-up* approach in program design.
7) It does not provides defaults arguments	7) It provides defaults arguments
8) Memory management operators are not available	8) Memory management operators are available (new and delete)
9) The key feature is structures and unions	9) The key feature is class
10) It supports multi-line comment statement (i.e.,/* */)	10) It supports both single-line (//) and multi-line comment statement.
11) Example: C, Fortran, Cobol, Pascal	11)Example: C++, Visual Basic, Java, J2EE