OOPs in C++

INTRODUCTION

Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOPs is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

C++ CLASSES AND OBJECTS

C++ is an object oriented programming language which is associated with classes and objects along with its attributes and methods.

Attributes and methods are basically variables and functions that belong to a class.

A class is a user defined data type which holds its own data members and member functions which can be accessed only by creating its instance/object and hence it acts like a blueprint for an object.

It is an identifiable entity with some characteristics and behaviour. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Creation of a class

To create a class, use the **class** keyword:

```
class className
{
  Access specifier:              // can be private,public or protected
     Data members;               // Variables to be used
     Member Functions() {}       // Methods to access data members
};                               // Class ends with a semicolon
```

Creation of an object

ClassName objectName;

ACCESS SPECIFIERS

It defines how the members(attributes and methods) of a class can be accessed outside the class.

There are 3 types of access modifiers:

- **PRIVATE:** Can be accessed inside the class only.
- **PROTECTED:** Can't be accessed outside the class except the inherited classes.
- **PUBLIC:** Can be accessed outside the class.

NOTE:

In C++, all the attributes and methods are private by default and public by default in java.

METHODS IN C++

Methods are functions that belong to a class which we can define in 2 ways:

- Inside class
- Outside class

For ex:

```
1. class MyClass {
2.  public:       // Access specifier
3.    void innerMethod() {
4.      cout << "This is the inner one";
5.    }
6.    void outerMethod();   // declaration
7. };
8.
9. void MyClass::outerMethod() {
10.     cout << "This is the outer one";
11.}
12.
13.   int main() {
14.     MyClass myObj;
15.     myObj.innerMethod();
16.     myObj.outerMethod();
17.     return 0;
18.   }
```

NOTE:

- Methods are accessed by creating an object of class and using dot syntax(.).

- In the outside class definition, declaration is done inside the class and then defined outside of the class which is done by specifying the name of the class, followed by the scope resolution :: operator, followed by the name of the function as shown in line 9. This operator resolves the scope of whatever is present after it and the scope is whatever present before it.
- Parameters can also be added.

ENCAPSULATION

Encapsulation is defined as wrapping up of data and information under a single unit.In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them. A class is the best example.

ABSTRACTION

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

INHERITANCE

The capability of a class to derive properties and characteristics from another class id called inheritance.

POLYMORPHISM

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

TYPES

1. Compile time Polymorphism
2. Runtime Polymorphism

Static Members in C++

Static variable in a function : When a variable is declared as static, space for it gets allocated for the lifetime of the program.(default initialized to 0) Even if the function is called multiple times, the space for it is allocated once.

Static variable in a class :-

-> Declared inside the class body.
-> Also Known as class members variable.
-> They must be defined outside the class.
-> Static variable doesn't belong to any object, but to the whole class.
-> There will be only one copy of the static member variable for the whole class.

Ex:-

```
1. Class A{
2.      static int a;
3. }
4.
5. // initialized outside the class
6. a=10;
7.
8. void main()
9. {
10.     A obj;
11.  }
```

Object can also be declared as static.

```
static A obj1;
```

Static function in a class:-

Static member function are allowed to access only the static data members or other static member function and any other function from outside the class.

Constructor

It is a special method of class that is automatically called when an object of class is created.

->It is used to initialise object attributes.

->Constructor has the same name as that of the class.

->These must be made public.

->Constructors don't have any return value.

Constructor are of 3 types:-

1. Default:-

```
Class_name()
{
a=10;    //used to initialise variables.
b=20;
}
```

2. Parameterized:-

```
Class_name(int x,int y)
{
  a=x;  //used to initialise variables with specific values.
  b=y;
}ere
```

3. Copy:-

```
Class_name(const Class_name &obj)
{
  a=obj.a;    //used to make another copy of an object.
  b=obj.b;
}
```

Note: If there is any parameterised constructor in the class, then the compiler doesn't create the default constructor.

Destructor

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

-> Destructors is a member function which is invoked when the objects are destroyed.
-> Destructors don't take any argument and don't have any return type.
-> Only one destructor is possible.
-> Destructors cannot be static.
-> A destructor should be declared in the public section of the class.
-> Destructors have same name as the class preceded by a tilde (~) .
-> You can't access the address of destructors.
-> Destructors don't take any argument and don't return anything

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

1. the function ends
2. the program ends
3. a block containing local variables ends
4. a delete operator is called

Syntax:-

```
~class_name();
```

Note: Destructors for non-static member objects are called in reverse order in which they appear in class declaration

Friend Class

A Friend Class can access private and protected members of other class in which it is declared as Friend.

Example:-

```
1. class A {
2. private:
3.    int a;
4.
5. public:
6.    A() { a = 0; }
7.    friend class B; // Friend Class
8. };
9.
10.    class B {
11. private:
12.       int b;
13.
14.    public:
```

```
15.     void showA(A& x)
16.     {
17.         // Since B is friend of A, it can access
18.         // private members of A
19.         std::cout << "A::a=" << x.a;
20.     }
21.  };
```

Inheritance

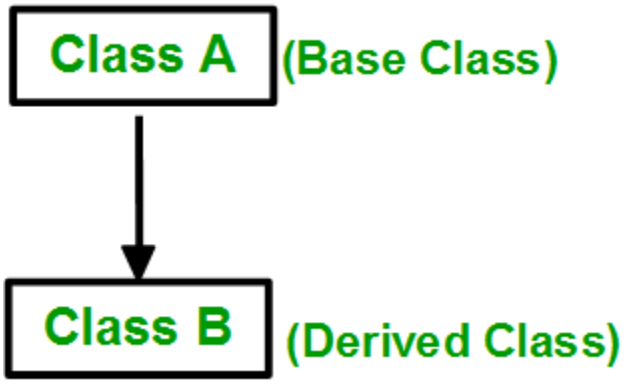
The capability of a class to derive properties and characteristics from another class is called **Inheritance**.
Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Syntax:

```
1.  class subclass_name : access_mode base_class_name
2.  {
3.   //body of subclass
4.  };
```

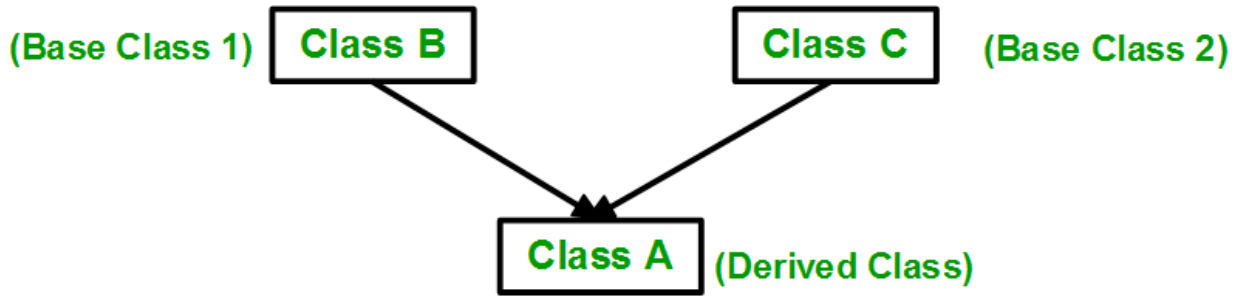
Single Inheritance:-

Single inheritance is one type of inheritance in which the derived class inherits only one base class.



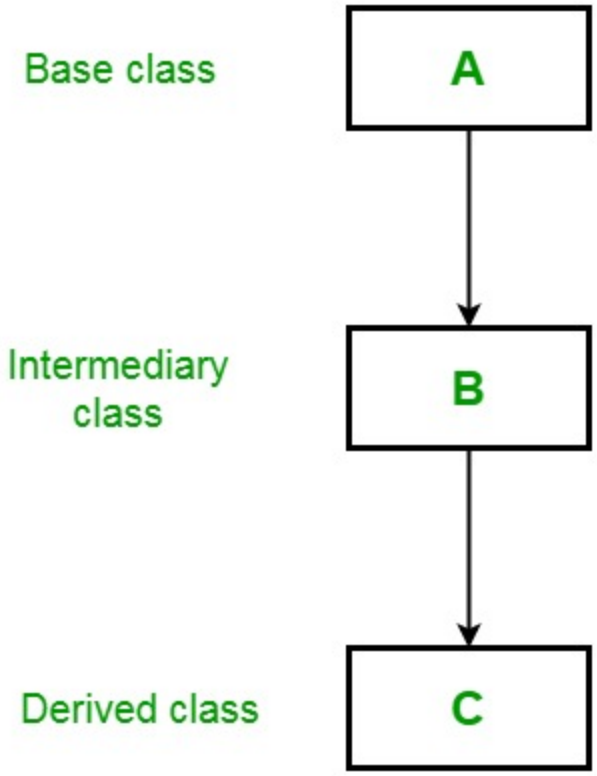
Multiple Inheritance:-

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes



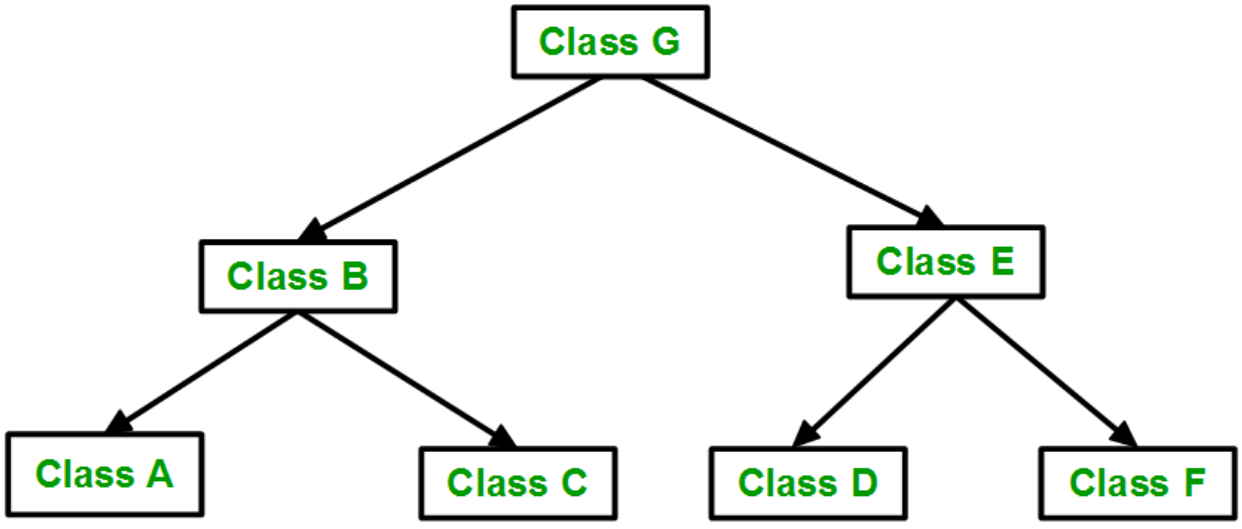
Multilevel Inheritance:-

Multilevel Inheritance is a property wherein an object of one class possesses the properties of another class and can further inherit the properties to other classes.



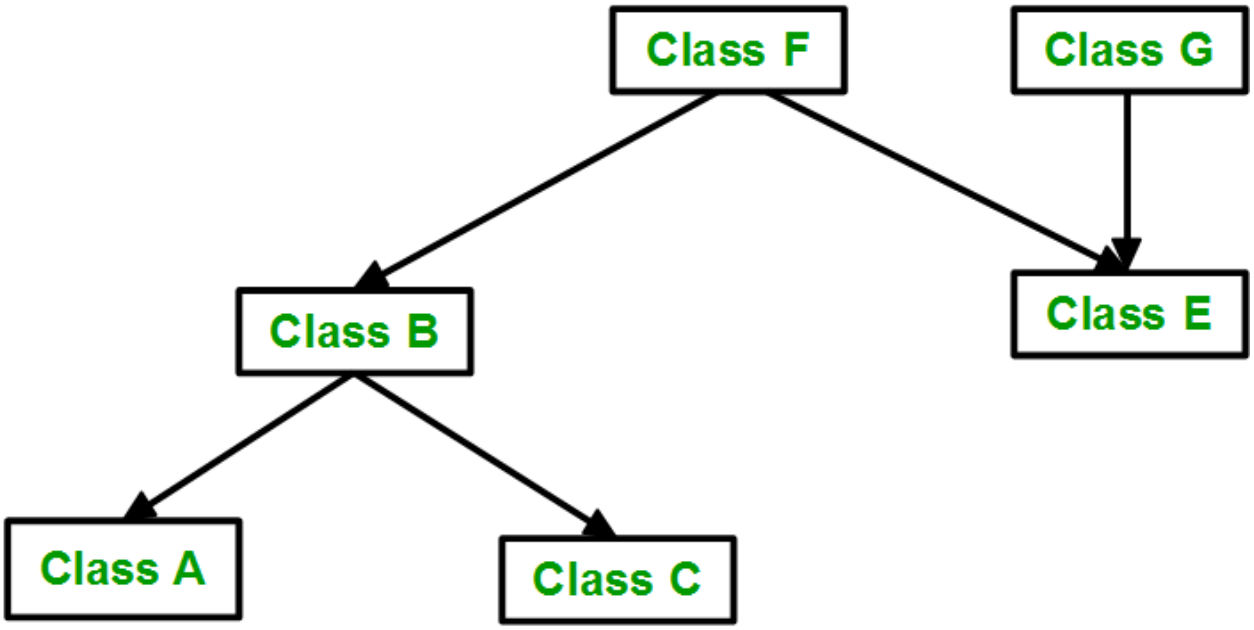
Hierarchical Inheritance:-

If more than one class is inherited from the base class, it's known as hierarchical inheritance.



Hybrid Inheritance:-

Hybrid inheritance in C++ is the inheritance where a class is derived from more than one form or combinations of any inheritance.



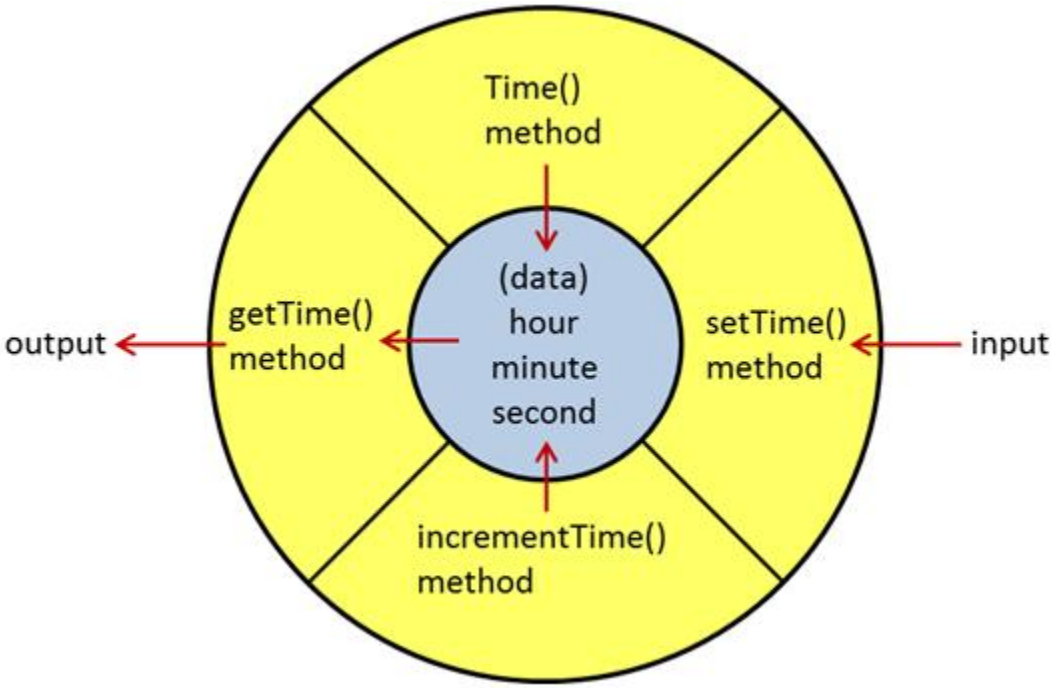
Constructor and Destructor in Inheritance:-

First child class constructor will run during creation of object of child class,but as soon as object is created child constructo run and it will call constructor of its parent class and after the execution of parent class constructor it will resume its constructor execution.

While in case of destructor, first child destructor executes then parent destructor get executed.

Encapsulation

Encapsulation is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.



Ex:-

```
1. class MyAccount
2. {
3.      private:
4.      float balance;
5.      public:
6.      MyAccount(float amount)
7.      {
8.           balance=amount;
9.      }
10.     void withdraw(float amount)
```

```
    {
        if(amount>balance)
        {
            cout<<" Balance is less than
  Amount"<<endl;
        }
        else
        {
            balance-=amount;
        }
    }
    float get_balance()
    {
        cout<<"Balance: Rs."<<balance<<endl;
    }
    void Add_Money(float amount)
    {
        balance+=amount;
        cout<<"Current Balance : "<<balance<<endl;
    }
}

```

Polymorphism

Compile time Polymorphism:-

1.Function Overloading is a feature in C++ where two or more functions can have same name but different parameters.

It provides multiple definitions of the function by changing signature i.e changing number of parameters, change datatype of parameters, return type doesn't play anyrole.

```
1. void print(int i)
2. {
3.      cout<<"Here is int "<<i<<endl;
4. }
5.
6. void print(float i)
7. {
8.      cout<<"Here is Float "<<i<<endl;
9. }
10.
11.  int main() {
12.     print(10);
13.     print(10.1);
14.     return 0;
15.  }
```

Functions that cannot be overloaded:-

1. Function declarations that differ only in the return type.

2. Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.
3. Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration.
4. Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.
5. Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent.
6. Two parameter declarations that differ only in their default arguments are equivalent.

2.Operator overloading in C++ have the ability to provide special meaning to the operator.

```
1. class Complex {
2. private:
3.    int real, imag;
4. public:
5.    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
6.
7.    Complex operator + (Complex const &obj) {
8.         Complex res;
9.         res.real = real + obj.real;
10.          res.imag = imag + obj.imag;
11.          return res;
12.     }
13.     void print() { cout << real << " + i" << imag <<
   endl; }
14.  };
15.
```

```
16.  int main()
17.  {
18.      Complex c1(10, 5), c2(2, 4);
19.      Complex c3 = c1 + c2;
20.      c3.print();
21.  }
```

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

```
1.)   . (dot)
2.)   ::
3.)   ?:
4.)   sizeof
```

The precedence and associativity of operators do not change.

Run time Polymorphism:-

Method Overriding:- It is the redefinition of base class function in its derived class, with same return type and same parameters.

```
1. class Animal {
2.    public:
3. void eat(){
4. cout<<"Eating...";
5.    }
6. };
7. class Dog: public Animal
8. {
9. public:
10.  void eat()
11.     {
12.        cout<<"Eating bread...";
13.     }
14.  };
```

```
int main(void) {
    Dog d = Dog();
    d.eat();
    return 0;
}
```

Virtual Function

A virtual Function is a member function which is declared with a 'virtual' keyword in the base class and redeclared (overridden) in a derived class. When you refer to an object of derived class using a pointer to a base class , you can call a virtual function of that object and execute the derived class's version of the function.

->They are used to achieve Runtime Polymorphism.
->Virtual Function cannot be static and also cannot be a friend function of another class.

Compile -time (Early Binding) Vs Run-time (Late Binding)

```
class base {
public:
   virtual void print()
   {
       cout << "print base class" << endl;
   }

   void show()
   {
        cout << "show base class" << endl;
    }
};

```

```
14.  class derived : public base {
15.  public:
16.      void print()
17.      {
18.          cout << "print derived class" << endl;
19.      }
20.
21.      void show()
22.      {
23.          cout << "show derived class" << endl;
24.      }
25.  };
26.
27.  int main()
28.  {
29.      base* bptr;
30.      derived d;
31.      bptr = &d;
32.
33.      // virtual function, binded at runtime
34.      bptr->print();
35.
36.      // Non-virtual function, binded at compile time
37.      bptr->show();
```

Output:-

Derived print //Late Binding
Base show fun //Early Binding

As during compiler time bptr behaviour is judged on the base's of which class it belongs, so bptr represents base class.

If the function is not virtual then it will allow binding at compile time and print fun of base class will get binded with bptr representing base class.

But at run time bptr points to the object at class derived, so it will bind the function of derived at run time.

Working of virtual function(Vtable & Vptr)

If a class contains virtual function then compiler itself does two things:

1. A virtual Pointer (Vptr) is created every time obj is created for the class which contains virtual function.
2. Irrespective of whether an object is created or not, a static array of pointer called Vtable where each cell points to each virtual Function is created, in base class and derived class.

Pure virtual Function and Abstract Class

Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an abstract class.

For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw().

```
//Abstract Class
Class test
{
      Public:
      Virtual Void fun()=0;          //pure virtual function.
}
```

1. A class is Abstract if it has at least one pure virtual function.
   we cannot declare objects of abstract class. It will show errors.
2. We can have pointers or references of abstract classes.

3. We can access the other function except virtual by object of its derived class.
4. If we don't override the pure virtual function in the derived class then it becomes abstract.
5. An abstract class can have constructor.