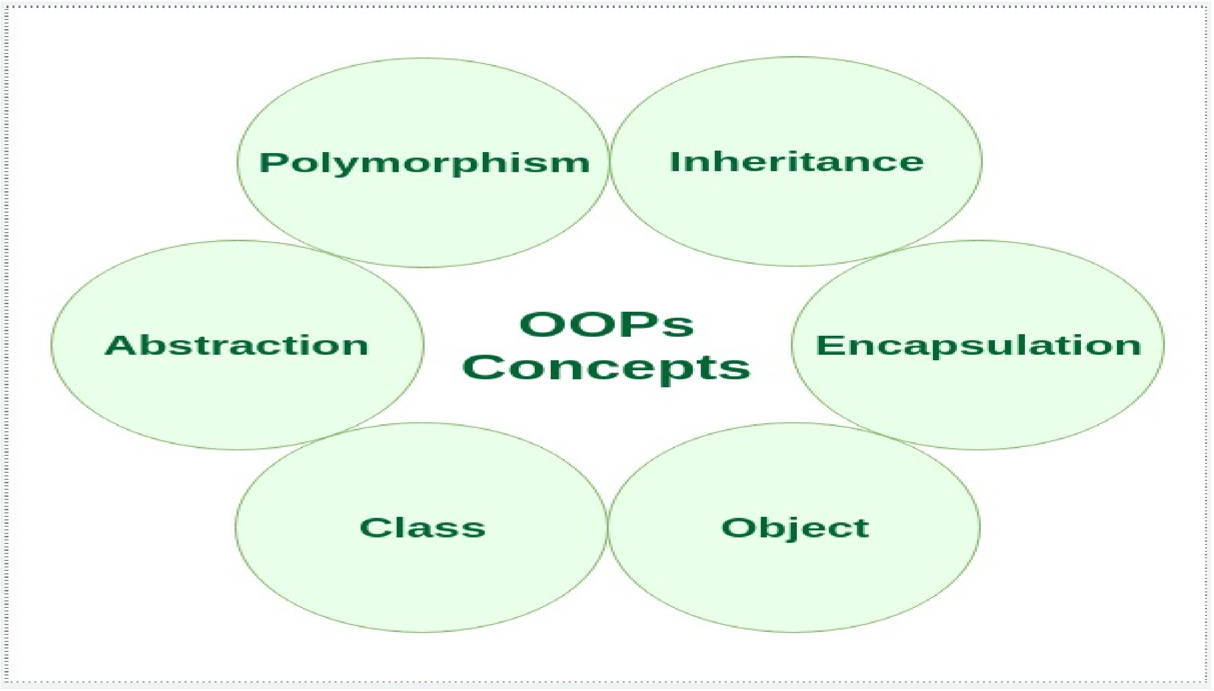
The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an object oriented programming language



CLASS :

It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

```
class person {
    char name[50];
    int id;
     public:
     void getDetails() {}
}
```

The only difference between a structure and a class is that structure members have public access by default and class members have private access by default

Static member function in cpp :

A function is made static by using a static keyword with function name. These functions work for the class as whole rather than for a particular object of a class.

```
class X
{
    public:
    static void f()
    {
        // statement
    }
};

int main()
{
    X::f();   // calling member function directly with class name
}
```

Const member function in cpp :

When used with member function, such member functions can never modify the object or its related data members.

```
void fun() const
{
    // statement
}
```

OBJECT :

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

```
person p1;
```

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code.

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 manipulate them.

Encapsulation also leads to *data abstraction or hiding*.

ABSTRACTION :

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.

- **Abstraction using classes**
- **Abstraction in header files**

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.**

- **Reusability : Inheritance supports the concept of "re-usability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.**

```
class Subclass_name : access_mode Superclass_name
```

Public Inheritance :

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

```
class Subclass : public Superclass
```

Private Inheritance :

In private mode, the protected and public members of super class become private members of derived class.

```
class Subclass : Superclass   // By default its private inheritance
```

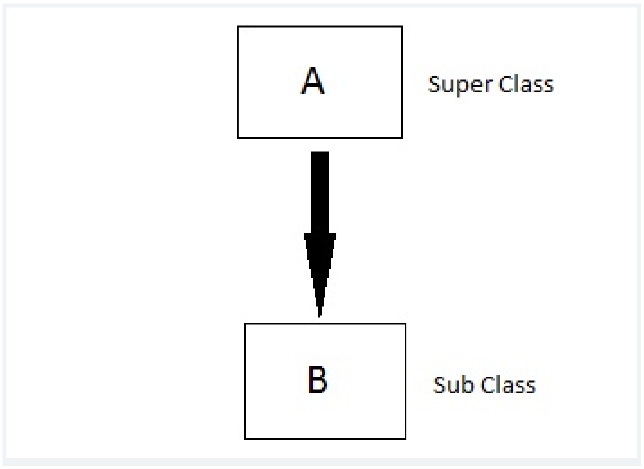
Protected Inheritance :

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

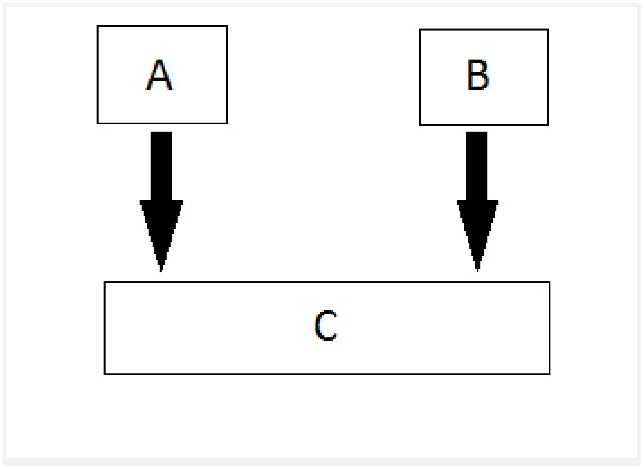
```
class subclass : protected Superclass
```

Types of Inheritance :

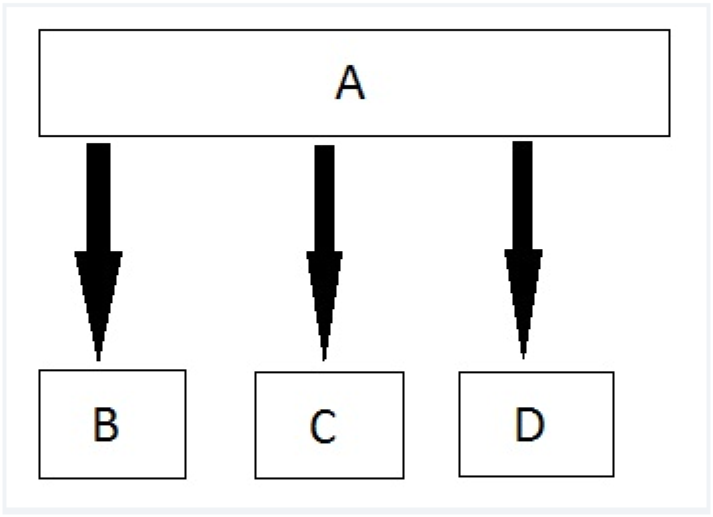
1. Single Inheritance :



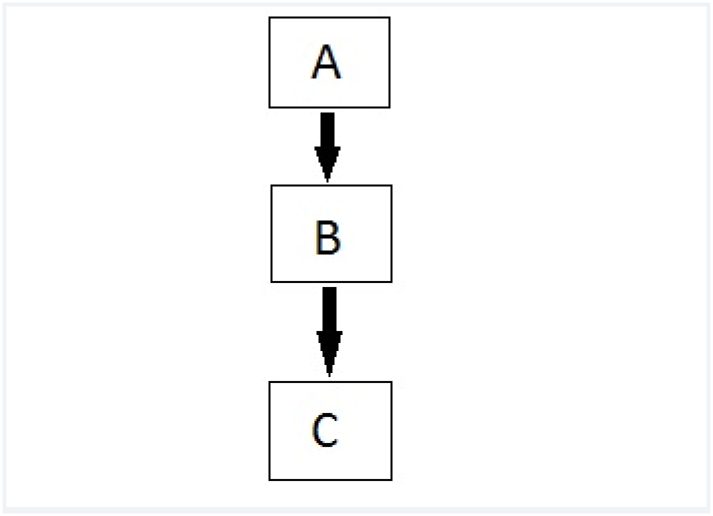
2. Multiple Inheritance :



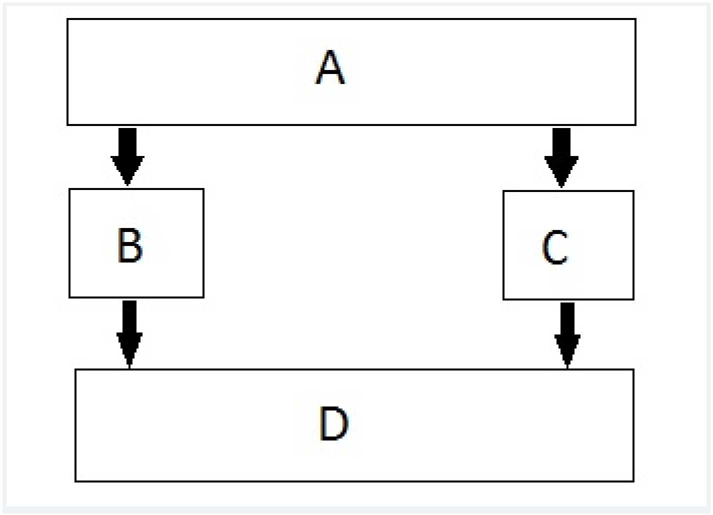
3. Hierarchical Inheritance :



4. Multilevel Inheritance :



5. Hybrid Inheritance (also known as virtual Inheritance)



POINTS TO REMEMBER :

1. Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.

2. To call base class's parameterised constructor inside derived class's parameterised constructor, we must mention it explicitly while declaring derived class's parameterised constructor.

```
class Base {
    int x;
    public:
    // parameterized constructor
    Base(int i) {
        x = i;
        cout << "Base Parameterized Constructor\n";
    }
};

class Derived : public Base {
 int y;
    public:
    // parameterized constructor
    Derived(int j):Base(j) {
        y = j;
        cout << "Derived Parameterized Constructor\n";
    }
};
```

3. All the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

```
class A : public B, public C ;
```

Ambiguity Resolution in Inheritance :

```
class C : public A, public B {
    void view() {
        A :: display();  // Calling the display() function of class A.
        B :: display();  // Calling the display() function of class B.
    }
};

//or
int main() {
   B b;
   b.display();               // Calling the display() function of B class.
   b.B :: display();       // Calling the display() function defined in B class.
}
```

Hybrid Inheritance and virtual Class :

In Multiple Inheritance, the derived class inherits from more than one base class. Hence, in Multiple Inheritance there are a lot chances of ambiguity.

```
class A
{
    void show();
};

class B:public A
{
    // class definition
};

class C:public A
{
    // class defintion
};

class D:public B, public C
{
    // class definition
};

int main()
{
    D obj;
    obj.show();
}
```

In this case both class B and C inherits function `show()` from class A. Hence class D has two inherited copies of function `show()`. In main() function when we call function `show()`, then ambiguity arises, because compiler doesn't know which `show()` function to call. Hence we use Virtual keyword while inheriting class.

```
class B : virtual public A
{
    // class definition
};

class C : virtual public A
{
    // class definition
};

class D : public B, public C
{
    // class definition
};
```

Now by adding virtual keyword, we tell compiler to call any one out of the two show() functions. Also see https://www.javatpoint.com/cpp-virtual-function

POLYMORPHISM :

The word polymorphism means having many forms.

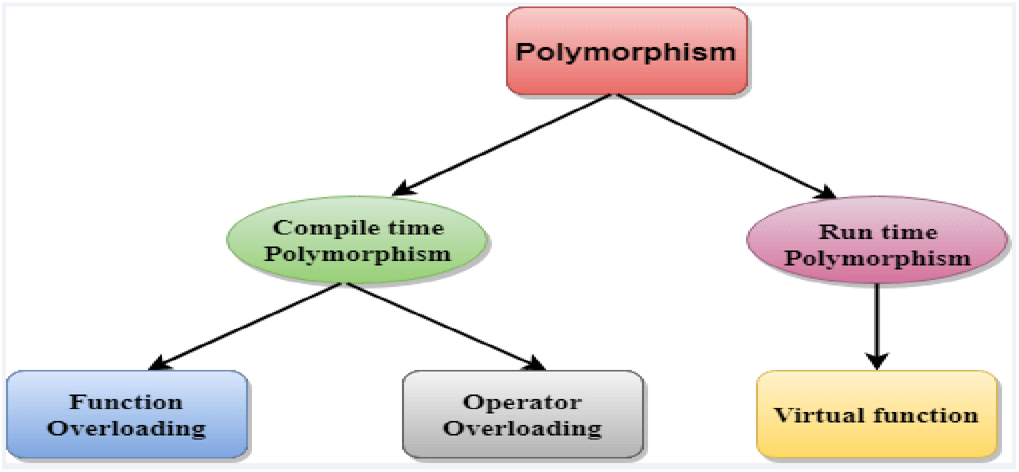
There are two types of polymorphism in C++ :

1. Compile Time Polymorphism :

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding.

2. Run Time Polymorphism :

Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.



C++ supports operator overloading and function overloading :

1. Operator overloading : The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.

2. Function overloading : Function overloading is using a single function name to perform different types of tasks.

Polymorphism is extensively used in implementing inheritance.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.