OOPs 3

Object-Oriented Programming generally contains the following four concepts:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Let's study these in detail in other sections.

Abstraction & Encapsulation

You are travelling in a car and suddenly, you have to make it stop. For that, you will be applying the brake. This whole system is enclosed inside the car's body. This process is known as encapsulation. To stop the car, you just have to apply the brake without thinking of the internal functioning of the car's engine system.

Encapsulation refers to combining data along with the methods operating on that data into a single unit called class. It also refers to the mechanism of binding the direct access state values for all the variables of the particular object. The main reason for using encapsulation is to provide security to the data of the class.

Reasons for which encapsulation is considered are:

- Functionality is defined at a logical place and not at multiple locations.
- It prevents our data from being modified from any external influence.

Abstraction means providing only some part of the information to the user by hiding the internal implementation details of it. We just need to know about the methods of the objects that we need to call and the input parameters needed to trigger a specific operation excluding the details of implementation and type of action performed to get the result.

Reasons for using abstraction are:

- It allows us to group different related units(classes) as siblings.
- It helps in the reduction of design and implementation complexity.

Inheritance

Suppose we have three classes with names: car, bicycle, and truck. The properties for each are as follows:

Car	Bicycle	Truck
• Colour • MaxSpeed • Number of gears	• Colour • MaxSpeed • Is Foldable?	• Colour • MaxSpeed • Max weight

From above, we can see that two of the properties: Colour and MaxSpeed, are the same for every object. Hence, we can combine all these in one parent class and make the above three classes as its subclass. This property is called Inheritance.

Technically, inheritance is defined as the process of acquiring the features and behaviours of a class by another class. Here, the class that contains these members is called the **base class** and the class that inherits these members from the base class is called the **derived class** of that base class**.**

Access Modifiers

When creating a derived class from a base class, we need to use access modifiers to inherit data members of the base class. These are:

- Public
- Private
- Protected

The **public** data members can be accessed by any of its child class as well as the class objects.

Private data members are inaccessible outside the class. These can't be accessed even by the child classes.

Protected data members can only be accessed by the derived classes but are inaccessible using the class objects.

Inheritance: Syntax

The syntax for inheriting a base class by a derived class:

```
class derived_class_name : access_modifier base_class_name {
        ----
        ----
};
```

Kindly look at the following example for better understanding:

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

class Vehicle {                                    // Parent class
        private :
                int maxSpeed;

        protected :
                int numTyres;

        public :
                string color;
};

class Car : public Vehicle {                       // Child class with Public access modifier
        public :
                int numGears;
                void print() {
                        // protected data member accessible
                        cout << "NumTyres : " << numTyres << endl;
                        // public data member accessible
                        cout << "Color : " << color << endl;
                        cout << "Num gears : " << numGears << endl;
                }
};
```

```
int main() {
        Vehicle v;
        v.color = "Blue";
        // v.maxSpeed = 100;          // Can't be accessed being private
        // v.numTyres = 4;            // Can't be accessed being protected

        Car c;
        c.color = "Black";
        // c.numTyres = 4;            // Can't be accessed being protected
        c.numGears = 5;
}
```

Order of Constructor/Destructor

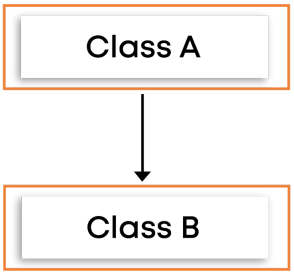
Constructors and Destructors follow a specific order of invocation.

- In the case of constructors, base class constructors are called first, and the derived class constructors are called next. Moreover, the order of constructor invocation depends on the order of how the base class is inherited.
- Destructors are called in reverse order of the constructor invocation, i.e., the destructor of the derived class is called first followed by the destructor of the base class sequentially.

Inheritance: Types

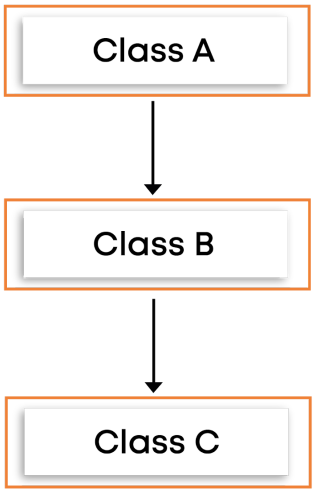
There are six types of inheritance:

- **Single Inheritance:** Here, a child class is created from a single parent class.



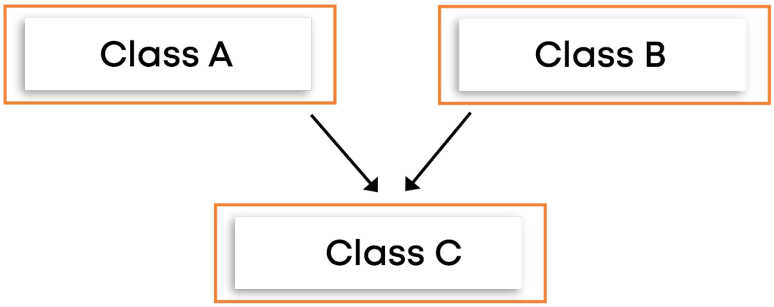
Single Inheritance

- **Multi-level Inheritance:** Here, there is a series of derived classes, which means a derived class is created from another derived class.



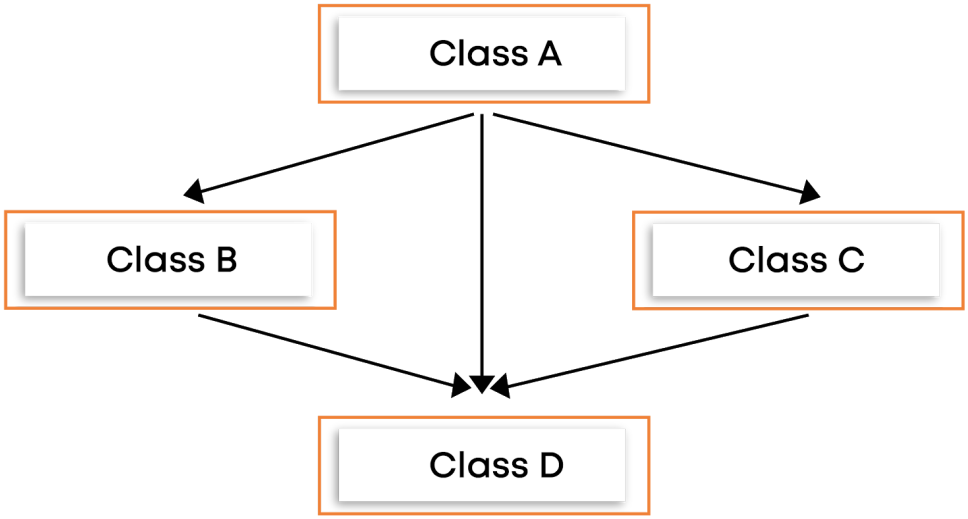
Multi-Level Inheritance

- **Multiple Inheritance:** Here, a child class is created from more than one parent/base class.



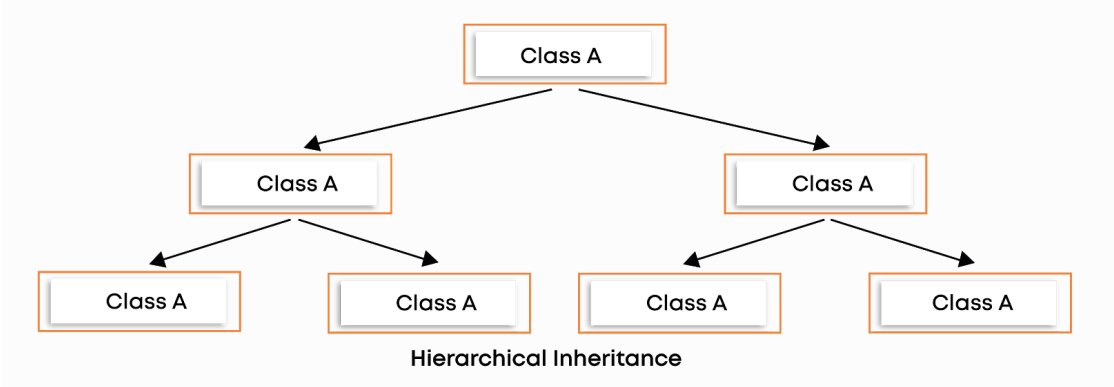
Multiple Inheritance

- **Multipath Inheritance:** In this type of inheritance, a derived class is created from other derived classes and the same base class of other derived classes.



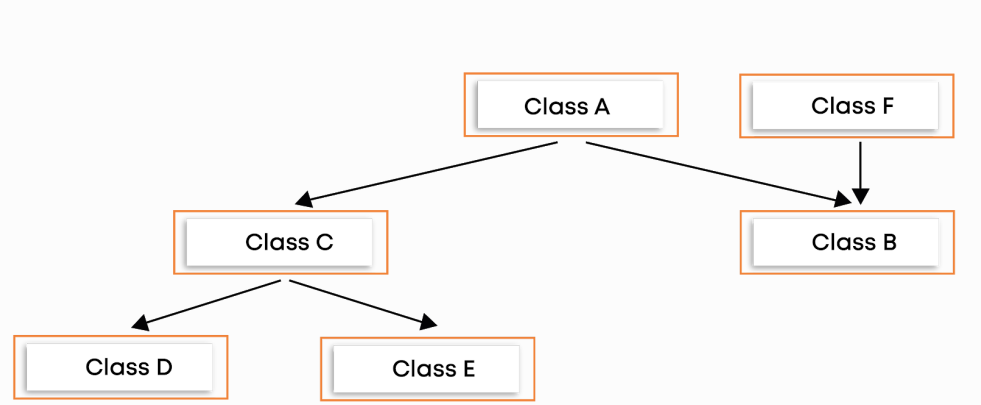
Multipath Inheritance

- **Hierarchical Inheritance:** Here, more than one derived classes are created from a single base class, and further, child classes act as parent classes for more than one child class.



Hierarchical Inheritance

- **Hybrid Inheritance:** It is the combination of more than one inheritances. It is also known as **Diamond inheritance**.



Hybrid Inheritance- (a combination of Hierarchical and multiple)

Polymorphism

The word polymorphism means **different forms**. It occurs when multiple classes are related to each other by inheritance. In C++, polymorphism means, a call to a member function will cause a different function to be executed depending on the type of object that invokes it. For example, (+) sign is used as an addition operator as well as the concatenation operator.

Polymorphism is generally of two types:

- Compile-time polymorphism
- Run-time polymorphism

Compile-time polymorphism:

It demonstrates the properties of static/early binding and occurs in the following cases:

- Function Overloading and Operator Overloading
- Function Overriding

We already have studied operator overloading. So, let's discuss function overloading.

When two or more functions have the same name but differ in any of the number of arguments or the return-type, then this process is known as **Function Overloading.**

Note: In static/early binding, the compiler (or linker) directly associates an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

For example:

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

int test(int a, int b) {                // function with 2 arguments

}
int test(int a) {                       // function with 1 argument

}
int test() {                            // function with no argument

}
int main() {
    // All the above functions have same name but differ in number of arguments
}
```

Now, suppose we have two classes, A and B where A is the base/parent class, and B is the child/derived class. If the base class has a function named *print()* and a child

class of it also contains such a function, then this is a case of function overriding. Here, we are deciding at the compile time about the function to be called. Look at the code below:

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

class Vehicle {
        public :
                string color;
                void print() {
                        cout << "Vehicle" << endl;
                }
};

class Car : public Vehicle {
        public :
                int numGears;
                void print() {
                        cout << "Car" << endl;
                }
};

int main() {
        Vehicle v;
        Car c;

        v.print();
        c.print();

        Vehicle *v1 = new Vehicle;
        Vehicle *v2;

        v2 = &c;
        v1 -> print();          // This will print Vehicle class' print()
        v2 -> print();          // This will also print Vehicle class' print() due to overriding
}
```

Run-time polymorphism:

Run-time polymorphism demonstrates the property of dynamic resolution or late binding and is achieved by using function overriding.

In the above code, we saw that by using the car class' object, when we were calling the print() function, we were redirected to Vehicle class's print() function. Suppose, in the aforementioned example, if we want that instead of the base class'(vehicle) print function, the derived class'(car) print function gets called, we can use **Virtual Functions** to achieve the same..

When the base class's function is overridden in the child class, then that function is known as a virtual function. Keyword **virtual** is used for the same. By doing so, we are redirecting the compiler to take the decision at runtime only.

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

class Vehicle {
        public :
                string color;
                void print() {
                        cout << "Vehicle" << endl;
                }
};

class Car : public Vehicle {
        public :
                int numGears;
                virtual void print() {                  // function made virtual
                        cout << "Car" << endl;
                }
};

int main() {
        Vehicle v;
        Car c;

        v.print();
        c.print();

        Vehicle *v1 = new Vehicle;
        Vehicle *v2;
        v2 = &c;
        v1 -> print();      // This will print Vehicle class' print()
        v2 -> print();      // This will now print car class' print() due to virtual function
}
```

Pure Virtual functions and Abstract classes

A virtual function whose declaration ends with =0 is called a **pure virtual function**. These functions do not have any definition. An Abstract class has at least one pure virtual function. Abstract classes are those that can't be instantiated, i.e., we cannot create an object of this class. However, we can derive a class from it and instantiate the object of the derived class.

Properties of the abstract classes:

- It can have normal functions and variables along with the pure virtual functions.
- Prominently used for upcasting(converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as a base type), so its derived classes can use its interface.
- If an abstract class has derived class, they must implement all pure virtual functions, or else they will become abstract too.

Kindly check the following code for better understanding.

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

class Vehicle {  // This class becomes abstract as it contains a pure virtual function
        public :
                string color;
                // Pure virtual function
                virtual void print() = 0;
};
```

Friend function and classes

Data hiding is an essential part of OOPs which means a non-member function is restricted from accessing an object's private or protected data. This restriction

sometimes forces us to write long and complicated codes; Instead we can use friend function/class to avoid the same.

Friend Function:

A friend function can access the private and protected data members of a class. To declare it, we use the keyword **friend.** For accessing the data, the friend function should be declared inside the body of the class.

Friend Class:

It is similar to the friend function. A class can be made a friend of another class by using the same keyword **friend.** Creating a class as a friend class, all the data functions of the class becomes friend functions.

Kindly follow the code below for better understanding.

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

class Bus {
        public :
                void print();
};

void test();

class Truck {
        private :
                int x;

        protected :
                int y;

        public :
                int z;
                friend class Bus; // Bus class declared as friend class to the Truck class
        /*
                friend void Bus :: print();      // Invoking the print() using friend class

                friend void test();              // In short, making a friend function itself.
        */
```

```
};

void Bus :: print() {
        Truck t;
        t.x = 10;
        t.y = 20;
        cout << t.x << " " << t.y << endl;
}

void test() {
// Access truck private (Not Possible, hence gives the error of inaccessible private variables)
        Truck t;
        t.x = 10;
        t.y = 20;
        cout << t.x << " " << t.y << endl;
}

int main() {
        Bus b;
        b.print();
        test();
}
```