# INHERITANCE

Inheritance is one of the fundamental concepts in Object oriented programming (OOP) that allows a class to derive properties and behaviors from another class. This mechanism promotes code reusability and establishes a hierarchical relationship between classes.

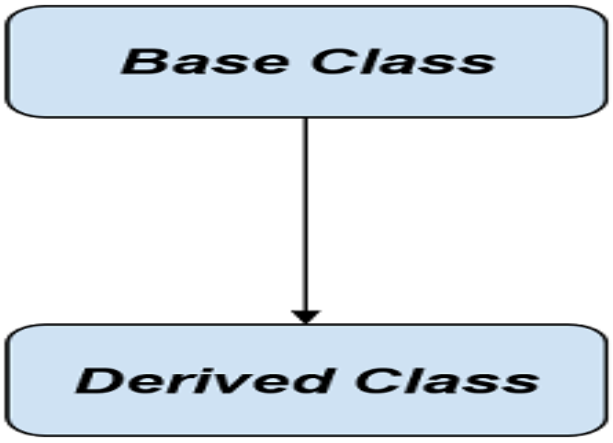
Inheritance is used only if there is some relationship or logical connection between two classes. Therefore, it works on **is-a relationship** that describes how one class (derived class) is a specialized version of another class (base class) in a real-world scenario. It signifies that the derived class **is a type of** the base class, for example, an orange is a fruit. Here, orange inherits the properties of a fruit.

## Types of Classes in Inheritance

There are two types of classes used in inheritance :

- **Base Class -** The class whose features and methods are inherited by a new class is known as base class. It is also termed as parent class or superclass. In the above mentioned example, fruit is a base class whose features are inherited by orange (new class).

- **Derived Class -** The class that inherits the features and methods of the existing class is known as a derived class. It is also termed as child class or subclass. In the above mentioned example, orange is a derived class that inherits the features of existing class i.e. fruit.




**Syntax : class derived_class : access_mode base_class{...};**

## Modes of Inheritance

There are three modes of inheritance, each determining the visibility and accessibility of base class members (attributes and methods) in the derived class.

- **Public Mode -** It allows the derived class to inherit the public and protected members of base class and the derived class. The public members of the base class become public in the derived class, while the protected members become protected. Private members of the base class are not inherited and cannot be accessed directly from the derived class. **‘**

  **Syntax :** class Derived : public Base {...};

- **Protected Mode -** In protected inheritance, all the public and protected members of the base class become protected members in the derived class. The private members of the base class are still not accessible from the derived class.

  **Syntax :** class Derived : protected Base {...};

- **Private Mode -** In private inheritance, all the public and protected members of the base class become private members in the derived class. Again, the private members of the base class cannot be inherited. This mode is used when the derived class should not expose the base class’s functionality as part of its own interface.

  **Syntax :** class Derived : private Base {...};

| | Derived Class | Derived Class | Derived Class |
|---|---|---|---|
| **Base Class** | **Private Mode** | **Protected Mode** | **Public Mode** |
| **Private** | Not Inherited | Not Inherited | Not Inherited |
| **Protected** | Private | Protected | Protected |
| **Public** | Private | Protected | Public |

## Advantages of Inheritance

- **Code Reusability -** Inheritance allows to reuse code by enabling a derived class (subclass) to inherit attributes and methods from existing class (base class).This reduces redundancy and makes code more maintainable.
- **Extensibility -** Through inheritance, new functionality can be added to existing classes without modifying their code.
- **Organised Code -** Inheritance provides a natural hierarchical structure for organizing classes.This makes the code cleaner and more understandable.
- **Polymorphism and Encapsulation -** Inheritance supports polymorphism, where a pointer or reference to a base class can be used to refer to objects of derived class. It allows dynamic method calls and makes code more flexible and reusable. It helps to implement encapsulation as well by using different access modifiers.
- **Increased Flexibility** - Inheritance supports flexible design patterns such as the template pattern or strategy pattern, where a common interface is defined in the base class and actual implementation is handled in the derived class.
- **Easy Refactoring** - Inheritance helps to update and improve systems much faster when the system evolves by restructuring base classes without changing every derived class, as long as the interface remains consistent.

## Constructor and Destructor in Inheritance

**Constructor :** The constructor of the base class is called first when an object of the derived class is created. If the base class constructor has [arametes, they must be explicitly called from the derived class constructor.

**Destructor :** The destructor of the derived class is called first, followed by the destructor of the base class when the derived object is destroyed. If the base destructor is virtual, it ensures proper cleanup when objects are deleted through base class pointers.

## Implementation of Inheritance

Here, take an example of fruit as base class and orange as derived class of fruit.

C/C++

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

// Base class: Fruit

class Fruit {
public:
    string name;

// Constructor to initialize the name of the fruit

    Fruit(string n) {
        name = n;
    }

  // function to display the name of the fruit

    void display() {
        cout << "This is " << name << "." << endl;
    }
};

// Derived class: Orange

class Orange : public Fruit {
```

```
public:

    // Constructor that calls the base class constructor

    Orange(string n) : Fruit(n) {}

    // Function to display a message specific to Orange

    void peel() {
        cout << "Peeling the orange..." << endl;
    }
};

int main() {

    // Creating an object of the base class (Fruit)

    Fruit apple("Apple");

    apple.display();        //Calls the base class function

    // Creating an object of the derived class (Orange)

    Orange orange("Orange");
    orange.display();  // Calls the inherited function from Fruit
    orange.peel();     // Calls the function specific to Orange

    return 0;
}
```

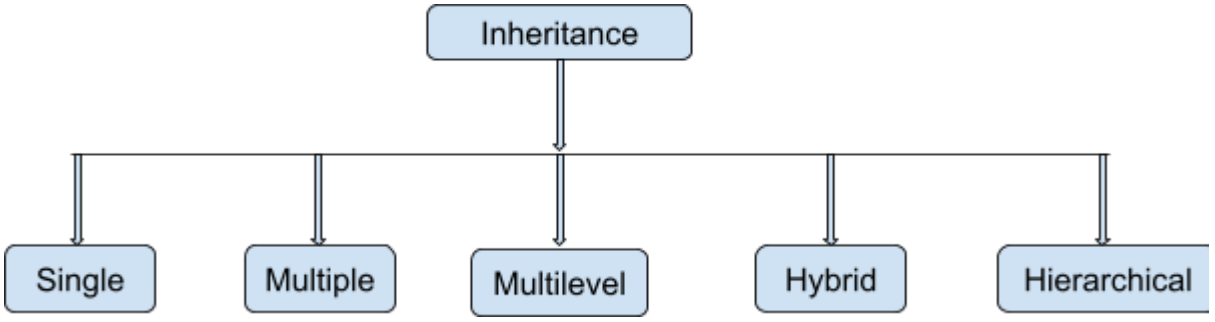
Unset

## Output

This is Apple.
This is Orange.
Peeling the orange...

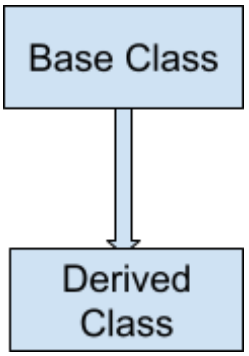
# Types of Inheritance

Above we have understood in details about Inheritance. Further, following are the types of inheritance :



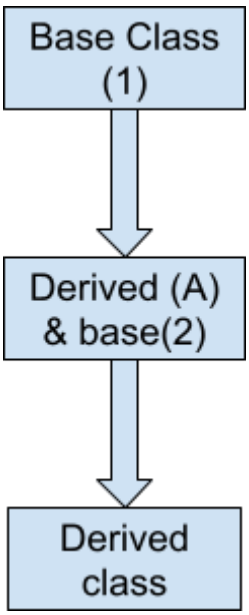

## Single Inheritance

**Definition** - When one derived class inherits from another class which is the base class, is known as single inheritance.



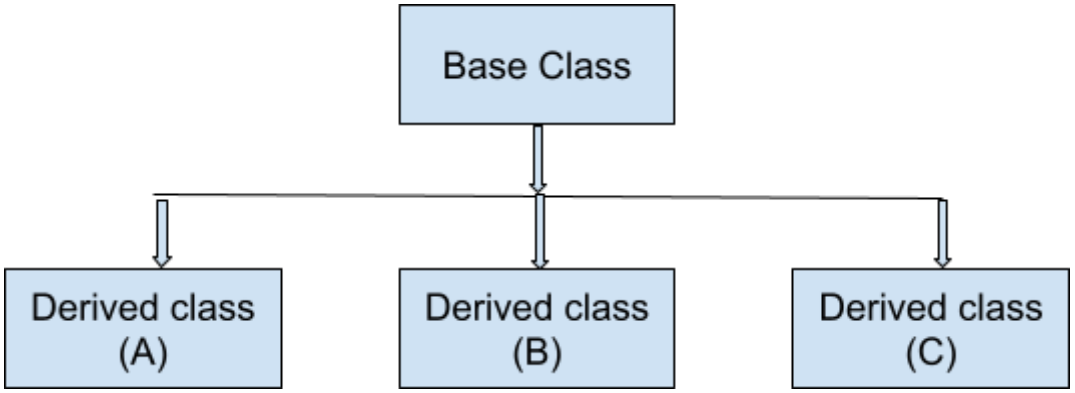

## Multilevel inheritance

**Definition** - In Multilevel inheritance, a class inherits from another derived class, forming a continuous chain. Each level builds upon the previous one, creating a hierarchy where the final class gets features from all preceding classes.



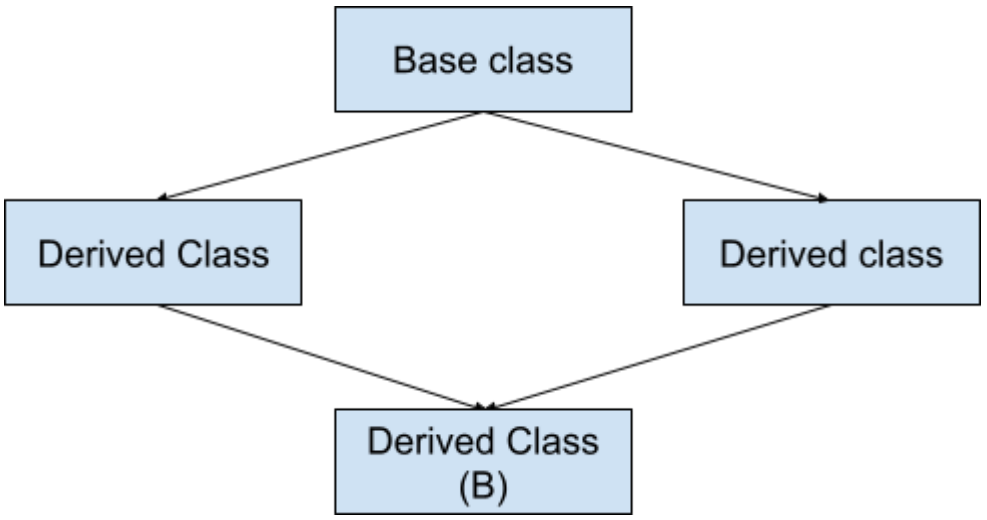

## Hierarchical Inheritance

**Definition -** In Hierarchical inheritance, multiple derived classes inherit from a single base class. This is ideal where multiple types share common functionality of a single base class.




## Hybrid inheritance

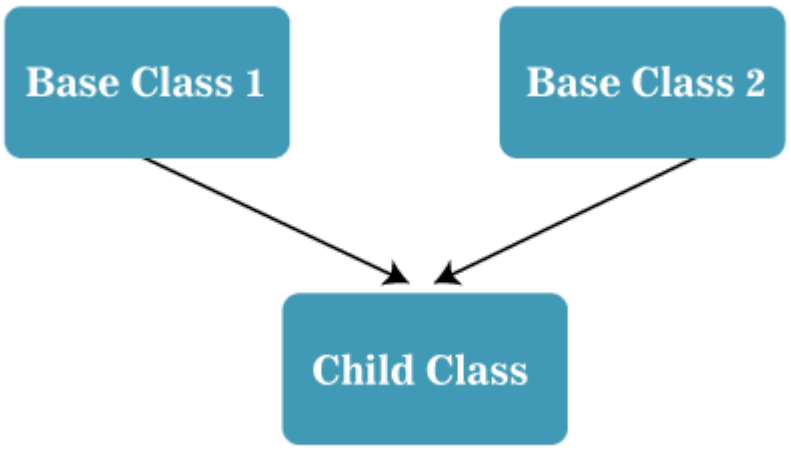
**Definition -** Hybrid inheritance is a combination of two or more types of inheritance, such as single and multiple inheritance. This often results in a complex inheritance hierarchy.




**NOTE:** When hybrid inheritance involves multiple inheritance, ambiguity can arise if the same base class is inherited more than once. This issue is resolved using virtual base classes.

## Multiple Inheritance

Multiple Inheritance is the concept of the Inheritance in C++ that allows a child class to inherit properties or behaviour from multiple base classes. Therefore, we can say it is the process that enables a derived class to acquire member functions, properties, characteristics from more than one base class.




In the above diagram, there are two-parent classes: Base Class 1 and Base Class 2, whereas there is only one Child Class. The Child Class acquires all features from both Base class 1 and Base class 2.

## Syntax:

```
class DerivedClass : accessSpecifier BaseClass1, accessSpecifier
BaseClass2 {

};
```

## Advantages of Multiple Inheritance:

1. **Code Reusability**
    - Allows a derived class to reuse code from multiple base classes, reducing redundancy and improving modularity.

2. **Enhanced Functionality**
    - Combines features and functionalities of multiple classes into one derived class, enabling more versatile and complex designs.

3. **Better Representation of Real-World Models**
    - Useful for modeling real-world scenarios where an entity needs to inherit characteristics from multiple sources (e.g., a **Teacher** inheriting from **Person** and **Employee**).

4. **Promotes Modularity**
    - Breaks down functionality into smaller, independent classes that can be reused in different combinations through multiple inheritance.

5. **Flexibility in Design**
    - Offers the flexibility to create a hierarchy where a single class can inherit and combine behaviors from more than one source class.

## Implementation of Multiple Inheritance:

Taking an example of Person and Employee as Base class and Manager as Derived class.

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

// Base class 1
class Person {
public:
    void displayPerson() {
        cout << "This is a Person." << endl;
    }
};

// Base class 2
class Employee {
public:
    void displayEmployee() {
        cout << "This is an Employee." << endl;
    }
};

// Derived class inheriting from both Person and Employee
class Manager : public Person, public Employee {
public:
    void displayManager() {
        cout << "This is a Manager." << endl;
    }
};

int main() {
    Manager m;

    // Accessing methods from both base classes
    m.displayPerson();      // From Person
    m.displayEmployee();    // From Employee
    m.displayManager();     // From Manager

    return 0;
}
```

**OUTPUT :**

```
This is a Person.
This is an Employee.
This is a Manager.
```

# Virtual Base Class

## *Introduction*

- Virtual base classes in C++ are used to prevent multiple instances of a given class from appearing in an inheritance hierarchy when using multiple inheritances.
- Base classes are the classes from which other classes are derived. The derived(child) classes have access to the variables and methods/functions of a base(parent) class. The entire structure is known as the inheritance hierarchy.
- Virtual Class is defined by writing a keyword “virtual” in the derived classes, allowing only one copy of data to be copied to Class B and Class C (referring to the above example). It prevents multiple instances of a class appearing as a parent class in the inheritance hierarchy when multiple inheritances are used.

## *Need for Virtual Base Class*

- To prevent the error and let the compiler work efficiently, we've to use a virtual base class when multiple inheritances occur. It saves space and avoids ambiguity.
- When a class is specified as a virtual base class, it prevents duplication of its data members. Only one copy of its data members is shared by all the base classes that use the virtual base class.
- If a virtual base class is not used, all the derived classes will get duplicated data members. In this case, the compiler cannot decide which one to execute.

## *Syntax*

```
class B: virtual public A
{  // statement 1};
class C: public virtual A
 {  // statement 2};
```

## *Example 1 (without virtual base class)*

```
#include <iostream>
using namespace std;
class A
 {
  public:    A()
 {
        cout << "Constructor A\n";
   }
void display()
{
   cout << "Hello form Class A \n";
    }
};
class B: public A {};
class C: public A {};
class D: public B, public C {};
int main()
{
  D object;
```

```
  object.display();
}
```

***Output:***

```
Main.cpp: In function ‘int main()’:
Main.cpp:20:10: error: request for member ‘display’ is ambiguous
   20 |   object.display();
      |          ^~~~~~~
Main.cpp:9:6: note: candidates are: ‘void A::display()’
    9 | void display()
      |      ^~~~~~~
Main.cpp:9:6: note:                 ‘void A::display()’
```

- **To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as :**

## *Example*

```
#include <iostream>
using namespace std;
class A
 {
  public:    A() // Constructor
   {
       cout << "Constructor A\n";
    }
};
class B: public virtual A
 {
};
class C: public virtual A
 {
};
class D: public B, public C
 {
};
int main()
 {
  D object; // Object creation of class D.
  return 0;
}
```

***Output:***

Constructor A

# Function Overriding

- It is a feature of object-oriented programming that allows a derived class to provide a specific implementation for a function that is already defined in its base class.

- This is an important concept in inheritance and supports runtime polymorphism.

## *Example of Function Overriding:*

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

class Animal {
public:
    virtual void sound() {  // Virtual function
        cout << "Animal makes a sound" << endl;
    }

    void eat() {  // Non-virtual function
        cout << "Animal eats food" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {  // Overriding function
        cout << "Dog barks: Woof Woof!" << endl;
    }

    void eat() {  // Hides the base class function (not overriding)
        cout << "Dog eats bones" << endl;
    }
};
```

```
int main() {
    Animal* animalPtr;        // Base class pointer
    Dog dogObj;

    animalPtr = &dogObj;

    // Function overriding
    animalPtr->sound();       // Calls Dog's sound() due to runtime polymorphism

    // No overriding for non-virtual function
    animalPtr->eat();         // Calls Animal's eat() (not Dog's)

    return 0;
}
```

## Output:

```
Dog barks: Woof Woof!
Animal eats food
```

### Key Advantages of Function Overriding:

- Promotes code reusability by allowing derived classes to modify base class functionality.
- Enables runtime polymorphism, making code more dynamic and modular.
- Enhances flexibility by allowing base class pointers or references to interact with different derived class objects.

# Abstract Classes in C++

An abstract class is a class that cannot be instantiated.It is primarily used as a blueprint to enforce the implementation of certain methods in derived classes. An abstract class is created by declaring at least one pure virtual function.

## *Pure Virtual Function:*

A function declared in a base class that has no definition relative to the base class. It must be overridden in derived classes.

## *Syntax for a Pure Virtual Function:*

virtual void functionName() = 0;

## *Purpose of Abstract Classes:*

To enforce that derived classes implement specific functions.
To provide a common interface for all derived classes.

## Example:

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

// Abstract class
class Shape {
public:
    // Pure virtual function
    virtual void draw() = 0;

    // Non-pure virtual function
    void description() {
        cout << "This is a shape." << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Circle." << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Rectangle." << endl;
    }
};

int main() {
```

```
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw();
    shape2->draw();

    delete shape1;
    delete shape2;

    return 0;
}
```

## Explanation of Example:

Shape is an abstract class because it contains a pure virtual function draw().
Derived classes Circle and Rectangle override the draw() function.
Abstract classes can have non-pure virtual or normal functions (e.g., description() in Shape).

## *Benefits of Abstract Classes:*

**1.Enforcing Method Implementation:**
Abstract classes ensure derived classes implement specific methods. For example, in the above code, every animal must define how it makes a sound.

**2.Polymorphism Support:**
Abstract classes are widely used with pointers or references to achieve runtime polymorphism.

**3.Code Reusability:**
Abstract classes can include common member functions or variables for derived classes, avoiding redundant code.

**4.Modularity and Design Flexibility:**
They provide a structured way to define the contract or interface that multiple derived classes must adhere to.

# Pure virtual function

For understanding pure virtual function , Firstly we have to learn about what is virtual function.

- Virtual funcion is a member function of a base class that can be re defined class in a derived class.
- Visual function Support runtime polymorphism and dynamic dispatch, which can improve code efficiency and Streamline development.
- They also allow objects of multiple derived classes to be treated consistently through a single interface.

## *Pure Virtual function*

- A pure virtual function in c++ is a virtual function that is declared in a base dass but does not have a defination.

- It is also known as a do-nothing function, and are assigned value of zero when they are declared.

**Example:-**
(Syntax)

```
Class Test{
  Public:
                virtual void show()=0;
};
```

## *Advantages of Pure Virtual Function:-*

- Pure virtual function ensure that derived classes implement a common interface, which promotes consistency.

- It enasies dynamic binding which allows for polymorphic behaviors and runtime flexibility.

- It allow each derived class to provide its own unique implementation.

- It prevent abstract class from being instantiated directly, which can reduce errors.

- Functions catch missing implementations at compile time, which can reduce run time errors.

## *Disadvantages:*

- Pure virtual functions cannot be instantiated on their own.
- Pure virtual functions ensure that all derived classes provide an implementation for those functions.