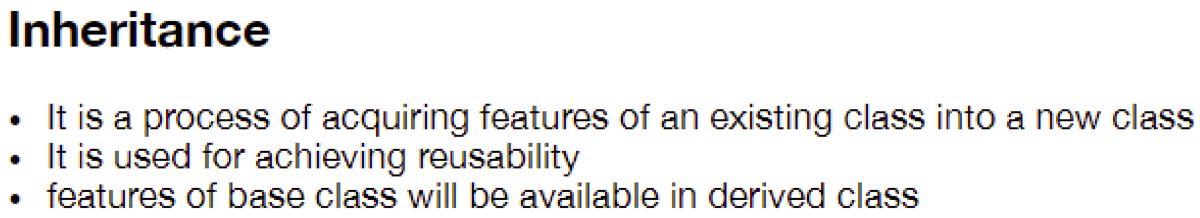
Inheritance

15 September 2024 16:20

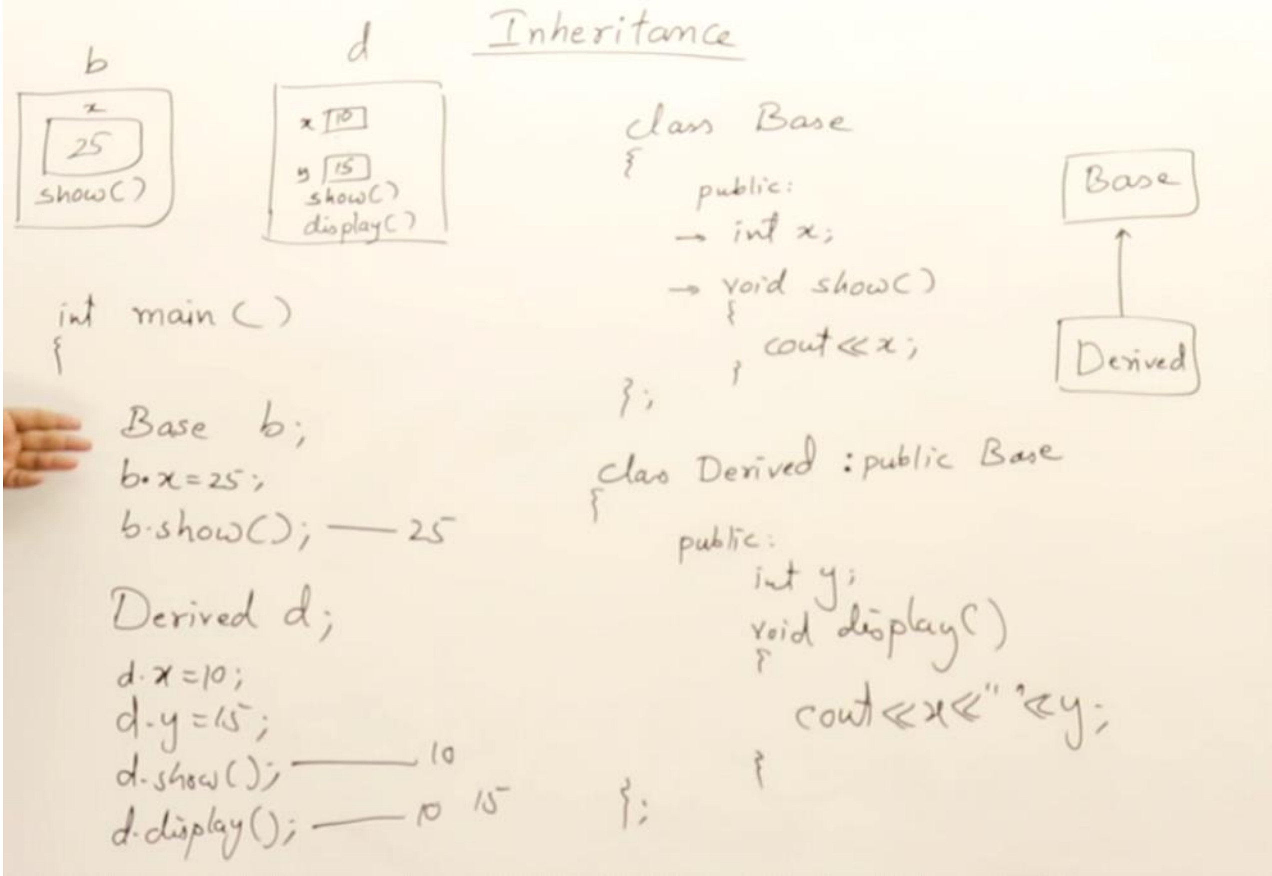
Inheritance is a way in which one class inherits the properties from another class.



Inheritance

- It is a process of acquiring features of an existing class into a new class
- It is used for achieving reusability
- features of base class will be available in derived class

```
#include<iostream>
using namespace std;
class Base{
public:
    int x;
    void print() {
        cout<<x<<endl;
    }
};
class Derived : public Base {
public:
    int y;
    void print(){
        cout<<x + y<<endl;
    }
};
int main() {
    Derived d;
    d.y = 5;
    d.x = 22;
    d.print();
    return 0;
}
```



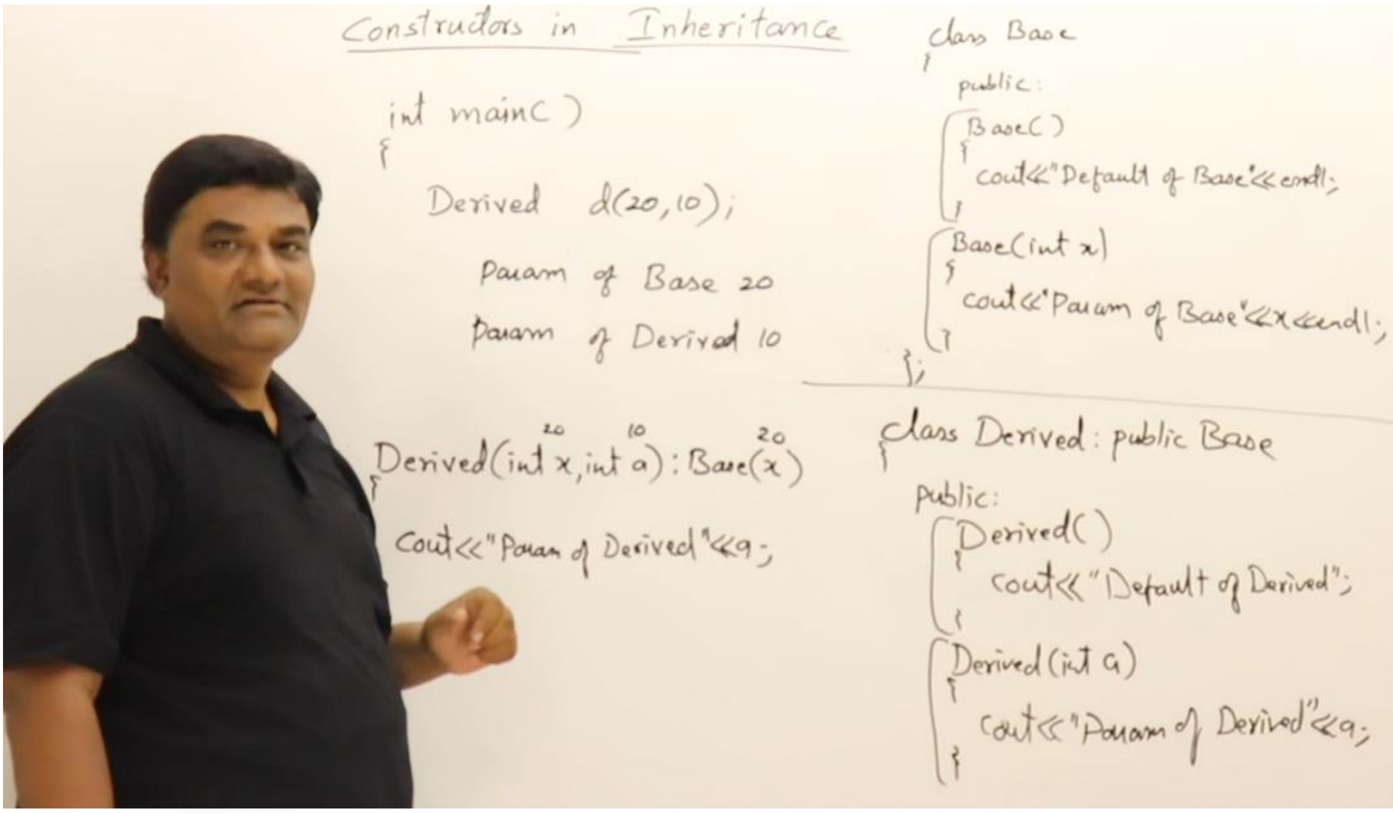
```
int main ( )
{
    Base b;
    b.x=25;
    b.show(); —— 25
    Derived d;
    d.x=10;
    d.y=15;
    d.show(); —— 10
    d.display(); —— 10 15
```

```
class Base
{
    public:
    → int x;
    → void show()
      {
        cout<<x;
      }
};
class Derived :public Base
{
    public:
      int y;
      void display()
      {
        cout<<x<<" "<<y;
      }
};
```

Q. Write a inherited Cuboid class from Base Rectangle Class.

```
class Rectangle{
protected:
    int length, width; // Protected to allow access in derived class
public:
    Rectangle(int l, int w){
        this->length = l;
        this->width = w;
    }
    void perimeter() {
        cout<<"Perimeter : "<< 2 *(length + width)<<endl;
    }
    void area() {
        cout<<"Area : "<< (length * width)<<endl;
    }
    void setLength(int l){
        this->length = l;
    }
    void setWidth(int w){
        this->width = w;
    }
};
class Cuboid : public Rectangle{
private:
int height;
public:
    Cuboid(int l, int w, int h) : Rectangle(l, w) { // Call Rectangle
constructor
        this->height = h;
    }
    void volume() {
        cout << "Volume: " << (length * width * height) << endl;
    }
    void setHeight(int h){
        this->height = h;
    }
};
```

Whenever we are calling a class object first the constructor from the base class is called and then the constructor from derived class is executed.

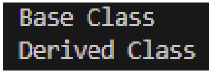


Constructors in Inheritance

```
int main()
{
    Derived d(20,10);

        Param of Base 20
        Param of Derived 10

Derived(int x,int a):Base(x)
{
    cout<<"Param of Derived"<<a;
```

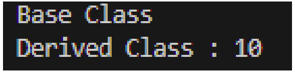
```
class Base
{
  public:
    Base()
    {
      cout<<"Default of Base"<<endl;
    }
    Base(int x)
    {
      cout<<"Param of Base"<<x<<endl;
    }
};

class Derived: public Base
{
  public:
    Derived()
    {
      cout<<"Default of Derived";
    }
    Derived(int a)
    {
      cout<<"Param of Derived"<<a;
    }
```

```
#include<iostream>
using namespace std;
class Base{
public:
Base(){
    cout<<"Base Class"<<endl;
}
Base(int a){
    cout<<"Base Class : "<<a<<endl;
}
};
class Derived : public Base {
public:
Derived(){
    cout<<"Derived Class"<<endl;
}
Derived(int b){
    cout<<"Derived Class : "<<b<<endl;
}
Derived(int a, int b) : Base (a) {
    cout<<"Derived Class : "<<b<<endl;
}
};
int main() {
    Derived d;
```



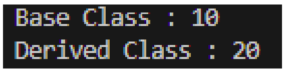
```
Base Class
Derived Class
```

```
    Derived d(10);
```



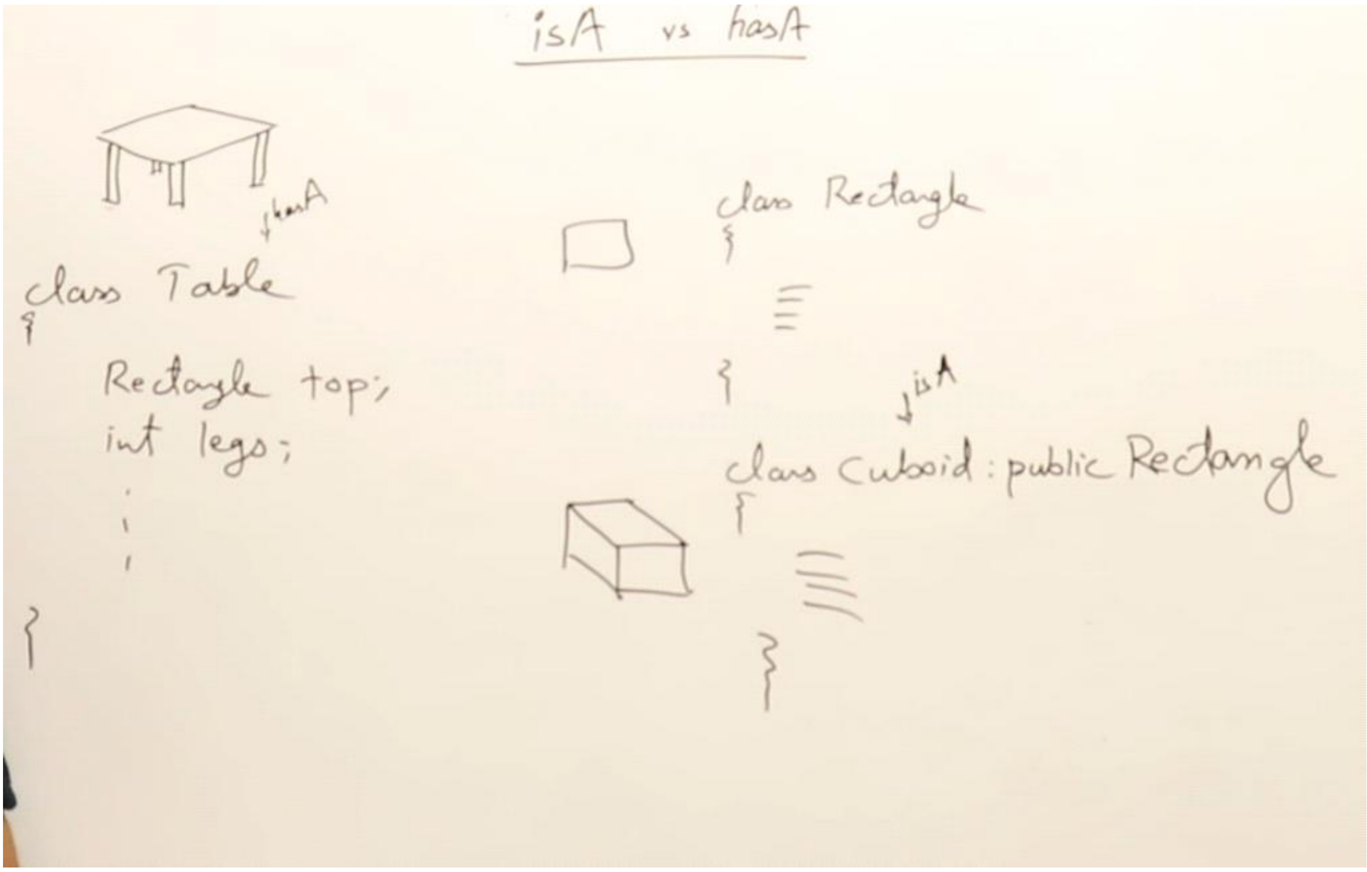
```
Base Class
Derived Class : 10
```

```
    Derived d(10,20);
```



```
Base Class : 10
Derived Class : 20
```

```
    return 0;
}
```



```
class Table
{
    Rectangle top;
    int legs;
    :
}
```

```
class Rectangle
{
    ≡
}
class Cuboid : public Rectangle
{
    ≡
}
```

isA is relationship refers to **inheritance**, where one class is a more specific version of another class. The derived (or subclass) is said to be a type of the base (or superclass).

```
class Animal {
public:
    void makeSound() {
        cout << "Animal sound!" << endl;
    }
};
class Dog : public Animal {  // Dog isA Animal
public:
    void bark() {
        cout << "Bark!" << endl;
    }
};
```

hasA

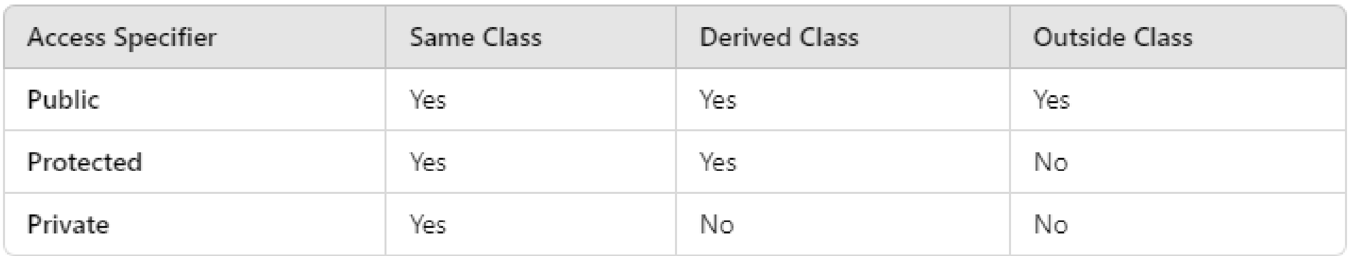
- relationship refers to **composition** or **aggregation**, where a class contains an object of another class as one of its members.
- This relationship describes ownership or usage, where one class has a reference to another class but does not inherit from it.

```
class Engine {
public:
    void start() {
        cout << "Engine started" << endl;
    }
};
class Car {
private:
    Engine engine;  // Car hasA Engine
public:
    void startCar() {
        engine.start();
        cout << "Car is running" << endl;
    }
};
```

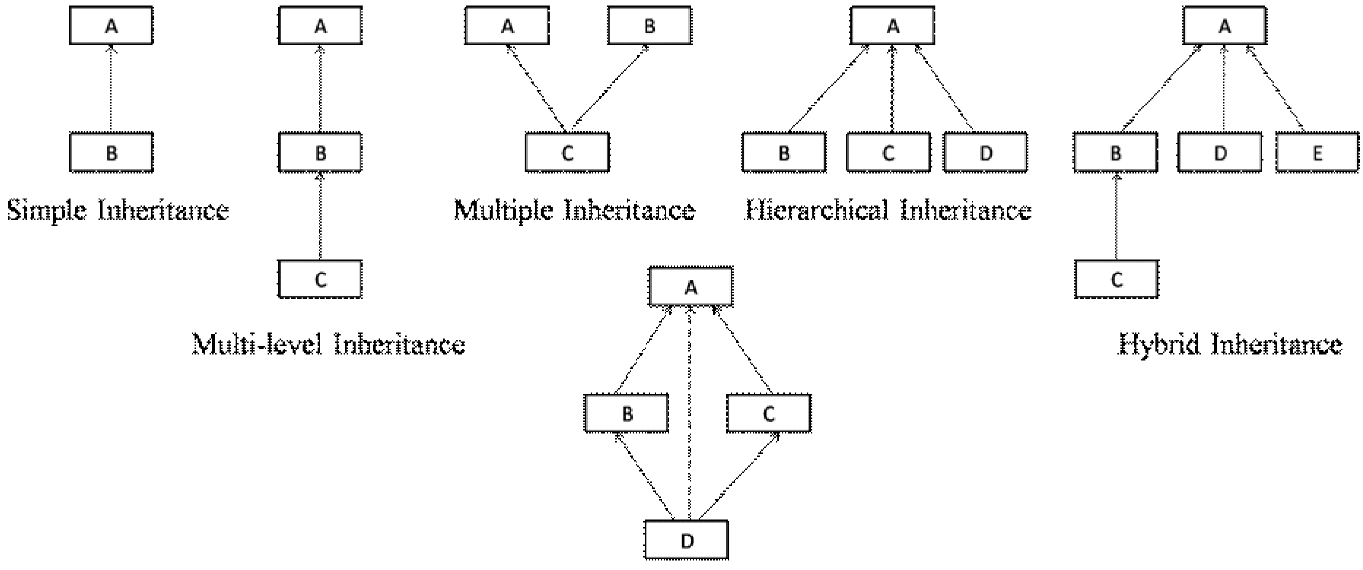
Access Specifier in Inheritance:

When a class inherits from another, the access specifiers can affect the accessibility of the base class members in the derived class. This is controlled through the inheritance mode (`public`, `protected`, `private`).

1. Public Inheritance:
   - The public and protected members of the base class remain public and protected in the derived class, respectively.
   - Private members of the base class are inaccessible in the derived class.
2. Protected Inheritance:
   - The public and protected members of the base class become protected in the derived class.
   - Private members of the base class remain inaccessible.
3. Private Inheritance:
   - The public and protected members of the base class become private in the derived class.
   - Private members of the base class remain inaccessible.



Access Specifier	Same Class	Derived Class	Outside Class
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No



Simple Inheritance

Multi-level Inheritance

Multiple Inheritance

Hierarchical Inheritance

Hybrid Inheritance

Multipath Inheritance

1. Single Inheritance

```
class Animal {
public:
    void sound() {
        cout << "Animal makes a sound" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
```

2. Multiple Inheritance

```
class Engine {
public:
    void startEngine() {
        cout << "Engine started" << endl;
    }
};
class Transmission {
public:
    void startTransmission() {
        cout << "Transmission started" << endl;
    }
```

```cpp
};
class Car : public Engine, public Transmission {
    // Car inherits from both Engine and Transmission
};
```

3. Multilevel Inheritance

```cpp
class LivingBeing {
public:
    void grow() {
        cout << "Living being grows" << endl;
    }
};
class Animal : public LivingBeing {
public:
    void move() {
        cout << "Animal moves" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
```

4. Hierarchical Inheritance

```cpp
class Animal {
public:
    void breathe() {
        cout << "Animal breathes" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
class Cat : public Animal {
public:
    void meow() {
        cout << "Cat meows" << endl;
    }
};
```

5. Hybrid Inheritance

```cpp
class Animal {
public:
    void breathe() {
        cout << "Animal breathes" << endl;
    }
};
class Mammal : public Animal {
public:
    void feedMilk() {
        cout << "Mammal feeds milk" << endl;
    }
};
class Bird : public Animal {
public:
    void layEggs() {
        cout << "Bird lays eggs" << endl;
    }
};
class Bat : public Mammal, public Bird {
public:
    void fly() {
        cout << "Bat flies" << endl;
    }
};
```

6. Virtual/ Multipath/ Diamond Inheritance

```cpp
class Animal {
public:
    void speak() {
        cout << "Animal speaks" << endl;
    }
};
class Mammal : virtual public Animal {
    // Virtual inheritance ensures only one copy of Animal is inherited
};
class Bird : virtual public Animal {
    // Virtual inheritance ensures only one copy of Animal is inherited
};
class Bat : public Mammal, public Bird {
    // Only one copy of Animal is inherited, no ambiguity
};
```