

Base and Derived Class Concept in C++

In C++, the **base and derived class concept** is central to object-oriented programming (OOP) and is primarily used to **implement inheritance**—a mechanism where a class (called a derived or child class) can inherit members (data and functions) from another class (called a base or parent class).

1. Defining a Base and Derived Class

```
class Base {  
    // Base class members  
};  
  
class Derived : public Base {  
    // Derived class members  
};
```

In **Object-Oriented Programming (OOP)**,

- a **base class** (also called a **parent class**) is a general class that provides common functionality.
- a **derived class** (also called a **child class**) extends the base class, adding or modifying behavior.

1. Defining a Base and Derived Class

```
class Base {  
    // Base class members  
};
```

```
class Derived : public Base {  
    // Derived class members  
};
```

// Base class

```
class Animal {  
public:  
    void eat() {  
        cout << "This animal eats food." << endl;  
    }  
};
```

// Derived class

```
class Dog : public Animal {  
public:  
    void bark() {  
        cout << "The dog barks." << endl;  
    }  
};
```

```
int main() {  
    Dog d;  
    d.eat(); // Inherited from Base class  
    d.bark(); // Defined in Derived class  
    return 0;  
}
```

This animal eats food.
The dog barks.

Types of Inheritance in C++

Type	Syntax	Access in Derived Class
Public	class Derived : public Base	Inherits public and protected members as they are.
Protected	class Derived : protected Base	Inherits public as protected, protected remains protected.
Private	class Derived : private Base	Inherits public and protected as private.

Inheritance Type	public members of base become	protected members become	private members become
public inheritance	public	protected	Not accessible
protected inheritance	protected	protected	Not accessible
private inheritance	private	private	Not accessible

```
class Base {  
    public:  
        int pub;  
    protected:  
        int prot;  
    private:  
        int priv;  
};
```

```
class DerivedPublic : public Base {  
    // pub -> public  
    // prot -> protected  
    // priv -> not accessible  
};
```

```
class DerivedProtected : protected Base {  
    // pub -> protected  
    // prot -> protected  
    // priv -> not accessible  
};
```

```
class DerivedPrivate : private Base {  
    // pub -> private  
    // prot -> private  
    // priv -> not accessible  
};
```

- **Private members** of the base class are **never accessible** directly in the derived class, regardless of inheritance type.
- Use **public inheritance** when you're modeling an "is-a" relationship (e.g., a Dog *is an* Animal).
- Use **protected/private inheritance** when you're using inheritance for code reuse but **don't want a full public interface exposed**.

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

class Animal {
public:
    void breathe() { cout << "Breathing..." << endl; }

protected:
    void run() { cout << "Running..." << endl; }

private:
    void sleep() { cout << "Sleeping..." << endl; } // Not
inherited
};
```

Public Inheritance: "is-a" relationship

```
class Dog : public Animal {
public:
    void showAbilities() {
        breathe(); // OK (public stays public)
        run();    // OK (protected stays protected)
        // sleep(); // Error: Not accessible
    }
};

Dog d;
d.breathe(); // Public
// d.run();  // Can't call directly (still protected)
d.showAbilities();
```

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

class Animal {
public:
    void breathe() { cout << "Breathing..." << endl; }

protected:
    void run() { cout << "Running..." << endl; }

private:
    void sleep() { cout << "Sleeping..." << endl; } // Not
inherited
};
```

Protected Inheritance

```
class Cat : protected Animal {
public:
    void showAbilities() {
        breathe(); // Now protected
        run();    // Still protected
    }
};
```

```
Cat c;
// c.breathe(); // Not accessible outside (became
protected)
c.showAbilities();
```

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

class Animal {
public:
    void breathe() { cout << "Breathing..." << endl; }

protected:
    void run() { cout << "Running..." << endl; }

private:
    void sleep() { cout << "Sleeping..." << endl; } // Not
inherited
};
```

Private Inheritance

```
class Tiger : private Animal {
public:
    void showAbilities() {
        breathe(); // Now private
        run();    // Now private
    }
};

Tiger t;
// t.breathe(); // Inaccessible, became private
t.showAbilities();
```


Class

Dog (public)


Cat (protected)

Tiger (private)

Can access breathe() publicly?

 Yes

 No (protected)

 No (private)

Can access run() inside derived class?

 Yes

 Yes

 Yes

Constructor Calling Order

When you create an object of a derived class:

- 1.The **base class constructor** is called **first**.
- 2.Then the **derived class constructor** is called.
- 3.**Derived class destructor runs first** before the base class destructor.

```
class Animal {  
public:  
    Animal() {  
        cout << "Animal constructor called." << endl;  
    }  
};
```

```
class Dog : public Animal {  
public:  
    Dog() {  
        cout << "Dog constructor called." << endl;  
    }  
};
```

```
int main() {  
    Dog d;  
    return 0;  
}
```

Animal constructor called.
Dog constructor called.

```
class Base {  
public:  
    Base() { cout << "Base Constructor\n"; }  
    ~Base() { cout << "Base Destructor\n"; }  
};
```

```
class Derived : public Base {  
public:  
    Derived() { cout << "Derived Constructor\n"; }  
    ~Derived() { cout << "Derived Destructor\n"; }  
};
```

```
int main() {  
    Derived d;  
    return 0;  
}
```

Base Constructor
Derived Constructor
Derived Destructor
Base Destructor

Parameterized Constructors with Base and Derived Classes

```
class Animal {  
public:  
    Animal(string name) {  
        cout << "Animal constructor: " << name << endl;  
    }  
};
```

```
class Dog : public Animal {  
public:  
    Dog(string name) : Animal(name) {  
        cout << "Dog constructor: " << name << endl;  
    }  
};
```

```
int main() {  
    Dog d("Buddy");  
    return 0;  
}
```

Animal constructor: Buddy
Dog constructor: Buddy

- Destructors are called in **reverse order**: derived → base.
- If the base class **only has a parameterized constructor**, you **must call it explicitly** in the derived class initializer list.
- If the base class has a **default constructor**, it will be called automatically if not specified.

Function Overriding in Derived Class

Function overriding allows a derived class to **redefine** a function from the base class. Function overriding occurs when a **derived class** provides a **specific implementation** of a function that is already defined in its **base class**.

```
class Base {
public:
    void show() {
        cout << "Base class show function\n";
    }
};

class Derived : public Base {
public:
    void show() { // Overriding function
        cout << "Derived class show function\n";
    }
};

int main() {
    Derived d;
    d.show(); // Calls the Derived class function
    return 0;
}
```

Derived class show function

- **Base Class** – The general class providing common properties.
- **Derived Class** – The specific class that extends the base class.
- **Inheritance Types** – Public, Protected, Private.
- **Function Overriding** – Allows redefining base class methods.
- **Constructor & Destructor Execution Order** – Base class first, then derived class.

Why Use Base and Derived Classes?

- Promotes **code reusability**.
- Helps with **polymorphism** (especially with virtual functions).
- Makes it easier to **manage hierarchical relationships**.