Himalaya College of Engineering

Advanced C++ Programming Lab Report

Lab 6: Inheritance in C++

Prepared By : Nawnit Paudel (HCE081BEI024)

Subject : Object-Oriented Programming (OOP)

Program : Bachelor of Electronics, Communication and Information Engineering

Institution : Himalaya College of Engineering

Date : June 21, 2025

OBJECTIVE

- To understand the concept of inheritance in C++.

BACKGROUND THEORY

Inheritance in C++

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class (derived class or subclass) to inherit properties and behaviors (data members and member functions) from an existing class (base class or superclass). This mechanism promotes code reusability, reduces redundancy, and establishes a hierarchical relationship between classes.

Types of Inheritance

C++ supports several types of inheritance. They are described in brief below:

1. **Single Inheritance** In single inheritance, a class inherits from only one base class.

Syntax:

```
class DerivedClass : access_specifier BaseClass {
   // members of DerivedClass
};
```

Example:

```
class Animal {
public:
   void eat() {
      // ...
   }
};
class Dog : public Animal { // Dog inherits from Animal
public:
   void bark() {
      // ...
   }
};
```

2. **Multiple Inheritance** In multiple inheritance, a class can inherit from multiple base classes. This allows a derived class to combine features from several distinct classes.

Syntax:

```
class DerivedClass : access_specifier BaseClass1, access_specifier BaseClass2 {
   // members of DerivedClass
};
```

Example:

```
class LivingBeing { /* ... */ };
class Swimmer { /* ... */ };
class Frog : public LivingBeing, public Swimmer { // Frog inherits from LivingBeing and Swimmer
   //
};
```

3. **Multilevel Inheritance** In multilevel inheritance, a class inherits from a base class, and then another class inherits from this derived class, forming a chain of inheritance

Syntax:

```
class Grandparent { /* ... */ };
class Parent : access_specifier Grandparent { /* ... */ };
class Child : access_specifier Parent { /* ... */ };
```

Example:

C++

```
class Vehicle { /* ... */ };
class Car : public Vehicle { /* ... */ }; // Car inherits from Vehicle
class SportsCar : public Car { /* ... */ }; // SportsCar inherits from Car
```

4. **Hierarchical Inheritance** In hierarchical inheritance, multiple derived classes inherit from a single base class.

Syntax:

```
class BaseClass {
/* ... */ };
class DerivedClass1 : access_specifier BaseClass { /* ... */ };
class DerivedClass2 : access_specifier BaseClass { /* ... */ };
```

Example:

```
class Shape { /* ... */ };

class Circle : public Shape { /* ... */ };

class Square : public Shape { /* ... */ };
```

5. **Hybrid Inheritance** Hybrid inheritance is a combination of two or more types of inheritance. A common scenario is combining multiple and hierarchical inheritance, which can lead to the "diamond problem". The diamond problem occurs when a class inherits from two classes that have a common base class, leading to ambiguity in inheriting members of the common base class. This can be resolved using virtual inheritance.

Example (Virtual Inheritance):

```
class Animal {

public:

   void eat() {

      // ...

   }

};

class Mammal : virtual public Animal { // Virtual inheritance

   /* ... */

};

class Bird : virtual public Animal { // Virtual inheritance

};

class Bat : public Mammal, public Bird { // Bat inherits from Mammal and Bird

   /* ... */

};
```

In this example, Bat would have only one Animal sub object due to virtual inheritance, resolving the ambiguity.

Modes of Inheritance (Access Specifiers)

The access specifier used during inheritance controls how the members of the base class are accessed in the derived class.

1. **Public Inheritance**

    - Public members of the base class remain public in the derived class.
    - Protected members of the base class remain protected in the derived class.

- Private members of the base class are inaccessible in the derived class.
- **Purpose:** Establishes an "is-a" relationship, where the derived class is a specialized type of base class.

2. **Protected Inheritance**

    - Public members of the base class become protected in the derived class.
    - Protected members of the base class remain protected in the derived class.
    - Private members of the base class are inaccessible in the derived class.
    - **Purpose:** Useful when you want to expose base class members to further derived classes but not to external code.

3. **Private Inheritance**

    - Public members of the base class become private in the derived class.
    - Protected members of the base class become private in the derived class.
    - Private members of the base class are inaccessible in the derived class.
    - **Purpose:** Establishes a "has-a" or "implemented-in-terms-of" relationship, where the base class's functionality is used internally by the derived class.

Constructors and Destructors in Inheritance

- **Constructors:** When an object of a derived class is created, the base class constructor is called first, followed by the derived class constructor.
- **Destructors:** When an object of a derived class is destroyed, the derived class destructor is called first, followed by the base class destructor. It's crucial to declare base class destructors as virtual when dealing with polymorphism to ensure proper cleanup of derived class objects through base class pointers.

Overriding Member Functions

Derived classes can provide their own implementation for a base class member function, a concept known as **function overriding**. This is achieved when a function in the derived class has the same name, return type, and parameters as a function in the base class.

Virtual Functions and Polymorphism

Polymorphism, meaning "many forms," allows objects of different classes to be treated as objects of a common base class. This is primarily achieved through **virtual functions**.

- A virtual function is a member function in the base class that is declared with the virtual keyword.
- When a virtual function is called through a pointer or reference to the base class, the actual function executed depends on the type of the object being pointed to or referred to at runtime, not the type of the pointer/reference. This is known as **runtime polymorphism**.