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

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**.

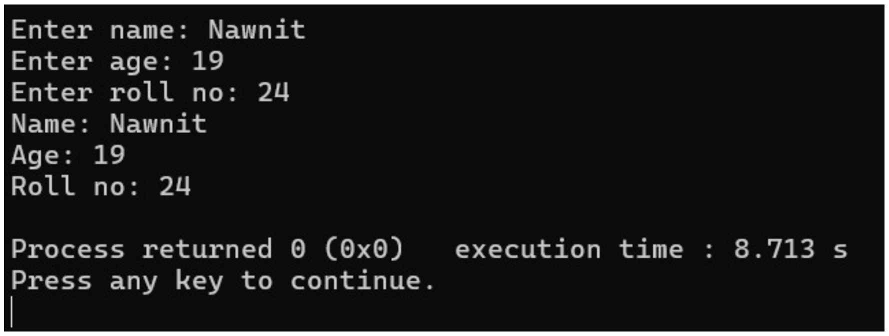
LAB ASSIGNMENTS:

1.Write a C++ program to create a base class Person with attributes name and age. Derive a class Student that adds rollNo. Use constructors to initialize all attributes. Create objects of both classes and display their details to show how Student inherits Person members.

Source Code:

```
#include<iostream>
using namespace std;
 class Person{     public:
string name;     int age;
void input(){
cout<<"Enter name: ";
cin>>name;
cout<<"Enter age: ";
cin>>age;
} }; class Student:public
Person{     public:      int
roll_no;     void input(){
Person::input();
cout<<"Enter roll no: ";
cin>>roll_no;
    }
    void output(){
        cout<<"Name: "<<name<<endl<< "Age: "<<age<<endl;
cout<<"Roll no: "<<roll_no<<endl;
    } }; int main(){
Student s;
    s.input();
    s.output();
return 0;  }
```

Output:



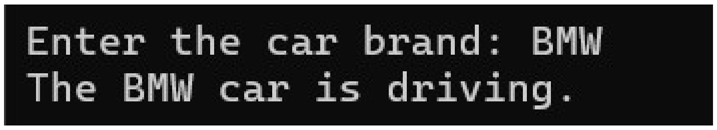
```
Enter name: Nawnit
Enter age: 19
Enter roll no: 24
Name: Nawnit
Age: 19
Roll no: 24

Process returned 0 (0x0)   execution time : 8.713 s
Press any key to continue.
```

2. Create a C++ program with a base class Vehicle having a function move(). Derive a class Car that overrides move() to indicate driving. Use a base class pointer to call move() on a Car object initialized with user input for attributes like brand. Show that Car is a Vehicle.

Source Code:

```
#include <iostream>

#include <string>

using namespace std;

class Vehicle { public:

    virtual void move() {        cout <<

"Vehicle is moving." << endl;

    } }; class Car : public Vehicle

{ private:

    string brand; public:

    Car(string b) : brand(b) {}     void move() override {         cout

<< "The " << brand << " car is driving." << endl;

    } }; int main() {     string

carBrand;     cout << "Enter the car

brand: ";     getline(cin, carBrand);

    Car myCar(carBrand);     Vehicle*

vehiclePtr = &myCar;     vehiclePtr-

>move();     return 0;

}
```

Output:



```
Enter the car brand: BMW
The BMW car is driving.
```

3. Implement a C++ program with a class Engine having an attribute horsepower. Create a class Car that contains an Engine object (composition) and an attribute model. Initialize all attributes with user input and display details to show that Car has an Engine.

Source Code:

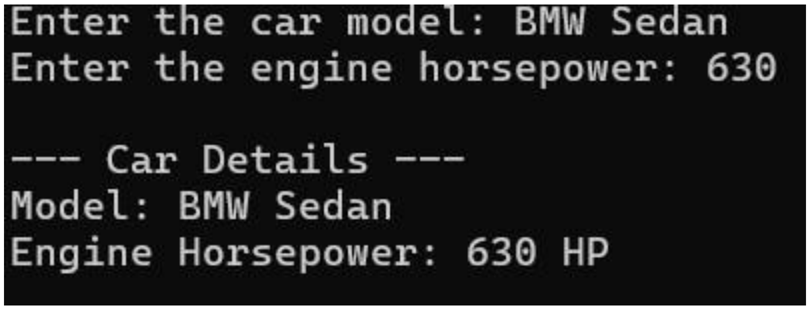
```
#include <iostream>
#include <string> using
namespace std; class
Engine { public:      int
horsepower;
Engine(int hp = 0) {
horsepower = hp;
    } }; class
Car {
public:
    string model;
    Engine engine;
    Car(string m, int hp) : model(m), engine(hp) {}     void displayDetails() {
cout << "\n--- Car Details ---" << endl;         cout << "Model: " << model << endl;
cout << "Engine Horsepower: " << engine.horsepower << " HP" << endl;
    } }; int main() {
string carModel; int
engineHp;
cout << "Enter the
car model: ";
getline(cin,
carModel);     cout
<< "Enter the
engine horsepower:
";     cin >>
engineHp;
```

Car myCar(carModel, engineHp);   myCar.displayDetails();

return 0;

}

Output:



```
Enter the car model: BMW Sedan
Enter the engine horsepower: 630

--- Car Details ---
Model: BMW Sedan
Engine Horsepower: 630 HP
```

4. Create a C++ program with a base class Animal having a virtual function sound(). Derive classes Dog and Cat that override sound() to print specific sounds. Use a base class pointer array to call sound() on Dog and Cat objects created with user input, showing runtime polymorphism.

Source Code:

#include<iostream>

using namespace std;

class Animal{

protected: string

name; public:

virtual void Sound(){

cout<<name<<"

makes a

sound."<<endl;

}

};

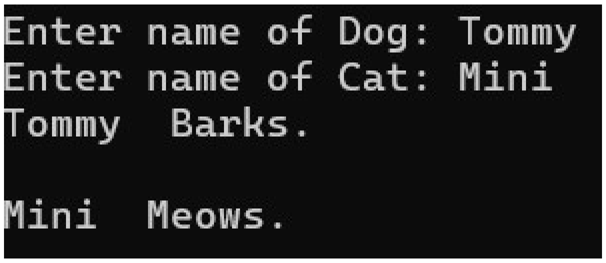
class Dog:public Animal{ public:

Dog(string *s*){ name=s;

}

```
    void Sound(){     cout<<name<<"

Barks."<<endl;}

};
class Cat:public Animal{ public:

Cat (string s){     name=s;

    }

    void Sound(){     cout<<name<<"  Meows."<<endl;}

}; int main(){     string name;

Animal *animal[2];

cout<<"Enter name of Dog: ";

cin>>name; animal[0]= new

Dog(name); cout<<"Enter name

of Cat: "; cin>>name;

animal[1]= new Cat(name);

for(int i=0;i<2;i++){

animal[i]->Sound();

cout<<endl;} return 0; }
```

Output:



```
Enter name of Dog: Tommy
Enter name of Cat: Mini
Tommy  Barks.

Mini  Meows.
```

5. Write a C++ program with two base classes Battery and Screen, each with a function showStatus().Derive a class Smartphone that inherits from both. Resolve ambiguity when calling showStatus()using the scope resolution operator. Initialize attributes with user input and display details.

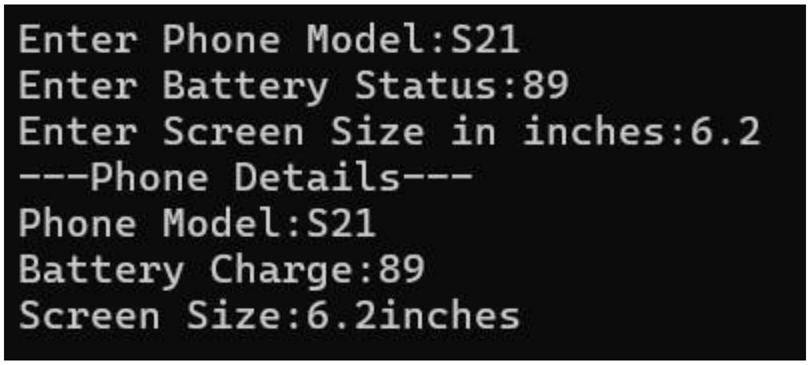
Source Code:

```
#include<iostream>
```

```
using namespace std;

class Battery{     int

Charge; public:      void

SetCharge(int c){

        Charge=c;

    }

    void ShowStatus(){     cout<<"Battery Charge:"<<Charge<<endl;

    }     };      class

Screen{     float

Size; public:

    void SetSize(float s){

        Size=s;

    }

    void ShowStatus(){ cout<<"Screen

    Size:"<<Size<<"inches"<<endl;
    } }; class Smartphone:public Battery,public

    Screen{ private:


        string model; public:     void

SetName(string n){

model=n;

    }

    void Display(){         cout<<"Phone

Model:"<<model<<endl;

    Battery::ShowStatus();

    Screen::ShowStatus();

    }     };     int main(){

Smartphone S;
```

```
    int c;     float s;     string n;
cout<<"Enter Phone Model:";
cin>>n;     S.SetName(n);
cout<<"Enter Battery Status:";
cin>>c;
    S.SetCharge(c); cout<<"Enter Screen
    Size in inches:"; cin>>s; S.SetSize(s);
    cout<<"---Phone Details---"<<endl;
    S.Display(); return 0;
     }
```

Output:



```
Enter Phone Model:S21
Enter Battery Status:89
Enter Screen Size in inches:6.2
---Phone Details---
Phone Model:S21
Battery Charge:89
Screen Size:6.2inches
```

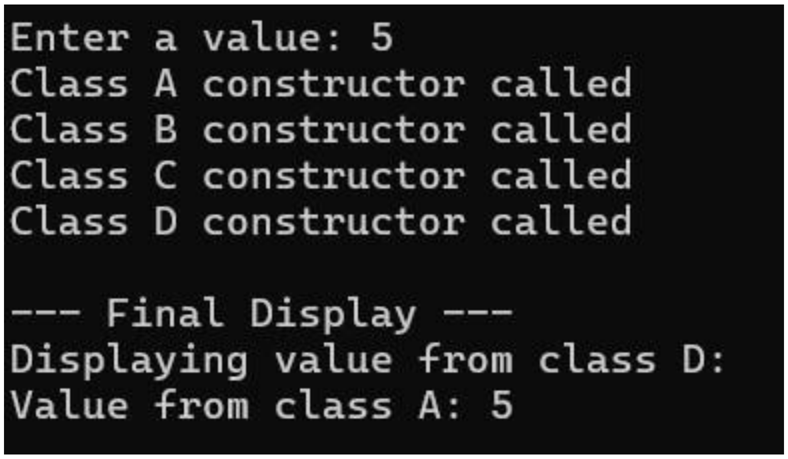
6. Create a C++ program with a base class A having an attribute value. Derive classes B and C from A, and derive class D from both B and C. Use virtual inheritance to avoid duplication of A's members.Initialize value with user input and display it from D to show ambiguity resolution.

Source Code:

```
#include  <iostream>  using  namespace  std;
class A { protected:   int value;  public:    A()
{        cout << "Enter a value: ";
cin >> value;        cout << "Class A constructor called"
<< endl;
```

```
   }
   void displayValue() {         cout << "Value from
class A: " << value << endl;
   } }; class B : virtual public A { public:    B() {         cout
<< "Class B constructor called" << endl;
   } }; class C : virtual public A { public:    C() {         cout
<< "Class C constructor called" << endl;
   } }; class D : public B, public C { public:    D() {
cout << "Class D constructor called" << endl;
  }
   void showValue() {         cout << "Displaying value from class
D:" << endl;         displayValue(); // No ambiguity due to virtual
inheritance
   } }; int main() {    D obj;    cout << "\n--
Final Display ---" << endl;
obj.showValue();    return 0;
}
```

Output:



```
Enter a value: 5
Class A constructor called
Class B constructor called
Class C constructor called
Class D constructor called

--- Final Display ---
Displaying value from class D:
Value from class A: 5
```

DISCUSSION

In this lab we were able to understand the concept of Inheritance in C++ language, we were able to do various inheritance using programming where we did Multiple, Multilevel, Hierarchical and diamond Inheritance. These practical examples helped us understand how classes can inherit properties and behaviors from other classes, enhancing code reusability and demonstrating the power of object-oriented programming in C++.

CONCLUSION

In this lab, we learned how one class in C++ can get features from another class using something called inheritance. We tried different types like multiple, multilevel, hierarchical, and diamond inheritance. By doing this, we saw how it helps us write less code and keep our programs neat and easy to understand. It was helpful in clarifying the core concept of OOP i.e. Inheritance.