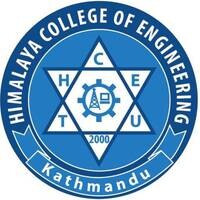


**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**



**HIMALAYA COLLEGE OF ENGINEERING**

**CHYASAL, LALITPUR**

**Lab Report No: - 4**

**Title: - Inheritance in C++**

**Submitted by: - Submitted To: -**

**Name: - Atul Bhattarai Department Of Computer and Electronics**

**Roll NO: - HCOE 081 BEI 011 Checked by: -**

**Date of submission: -**

**Objective:**

To understand and implement various types of inheritance in C++ through practical examples. You will gain hands-on experience with single, multiple, hierarchical, multilevel, and hybrid inheritance.

**Theory: Exploring Inheritance Types in C++**

Inheritance is one of the core features of object-oriented programming (OOP) in C++. It allows a class (called the derivedclass) to inherit properties and behaviors (i.e., datamembers and memberfunctions) from another class (called the baseclass). This promotes codereusability, reduces redundancy, and allows for better organization and management of large codebases.

Inheritance establishes an “is-a”relationship. For example, if we have a base class Animal and a derived class Dog, then a dog “is an” animal. This kind of logical relationship allows the derived class to access and reuse the base class's members, without having to rewrite the same code.

C++ supports several types of inheritance, each useful in different scenarios:

**1. Single Inheritance**

In single inheritance, one class inherits from one base class. It is the simplest form of inheritance and is used when there is a direct relationship between two classes.

**Syntax:**

class Base {

// base members

};

class Derived : public Base {

// additional members

};

**2. Multiple Inheritance**

In multiple inheritance, a derived class inherits from more than one base class. This allows the derived class to combine features from multiple sources. However, it can introduce ambiguity if the same member exists in more than one base class.

**Syntax:**

class A {

};

class B {

};

class C : public A, public B {

};

**3. Multilevel Inheritance**

Multilevel inheritance forms a chain where a class inherits from a derived class, which itself inherits from another base class. This creates a hierarchy of classes.

**Syntax:**

class A {

};

class B : public A {

};

class C : public B {

};

**4. Hierarchical Inheritance**

In hierarchical inheritance, multiple classes are derived from a single base class. This reflects real-world systems where many different objects share a common base.

**Syntax:**

class Base {

};

class Derived1 : public Base {

};

class Derived2 : public Base {

};

**5. Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance. For example, a class could inherit from two base classes, one of which is itself derived from another base class. Hybrid inheritance can lead to the **“**diamond problem”, where ambiguity arises if a derived class inherits the same base class through multiple paths. This issue is resolved using virtual inheritance.

**Syntax:**

class A {

};

class B : virtual public A {

};

class C : virtual public A {

};

class D : public B, public C {

};

**Access Specifiers in Inheritance**

* **Public Inheritance:** Public members of the base class remain public in the derived class.
* **Protected Inheritance:** Public and protected members of the base class become protected in the derived class.
* **Private Inheritance:** Public and protected members of the base class become private in the derived class.

**Advantages of Inheritance**

* **Code Reusability:** Common functionality is written once in the base class and reused by derived classes.
* **Extensibility:** New features can be added to existing classes without modifying them.
* **Modularity:** Inheritance encourages clean class hierarchies and better software design.
* **Polymorphism:** In combination with virtual functions, inheritance supports run-time polymorphism.

This theoretical foundation helps in understanding how inheritance structures are formed and how C++ facilitates code organization, clarity, and scalability in software development.

**Lab assignment:**

**Exercise 1: Single Inheritance**

1. Create a base class Shapewith a method display().

2. Create a derived class Circlethat inherits from Shape and has an additional method draw().

3. Implement a main() function to demonstrate the usage of these classes.

#include <iostream>

using namespace std;

class Shape {

public:

void display() {

cout << "This is a shape." << endl;

}

};

class Circle : public Shape {

public:

void draw() {

cout << "Drawing a circle." << endl;

}

};

int main() {

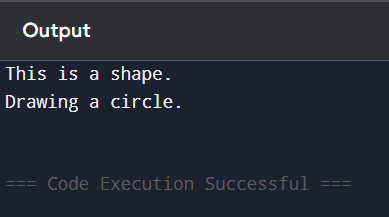
Circle c;

c.display();

c.draw();

return 0;

}



**Exercise 2: Multiple Inheritance**

1. Create two base classes Person and Employeewith appropriate methods.

2. Create a derived class Manager that inherits from both Person and Employee.

3. Implement a main() function to demonstrate the usage of these classes.

#include <iostream>

using namespace std;

class Person {

public:

void name() {

cout << "Name: ram" << endl;

}

};

class Employee {

public:

void role() {

cout << "Role: Manager" << endl;

}

};

class Manager : public Person, public Employee {

public:

void details() {

cout << "Manager Details:" << endl;

name();

role();

};

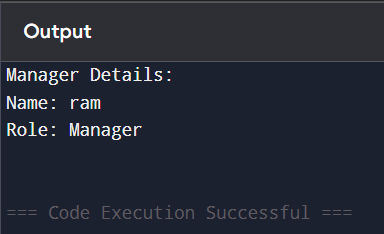
int main() {

Manager m;

m.details();

return 0;

}



**Exercise 3: Hierarchical Inheritance**

1. Create a base class Animalwith a method speak().

2. Create two derived classes Dog and Catthat inherit from Animal and have their own speak() methods.

3. Implement a main() function to demonstrate the usage of these classes.

#include <iostream>

using namespace std;

class Animal {

public:

void speak() {

cout << "Animal speaks." << endl;

}

};

class Dog : public Animal {

public:

void speak() {

cout << "Dog barks." << endl;

}

};

class Cat : public Animal {

public:

void speak() {

cout << "Cat meows." << endl;

}

};

int main() {

Dog d;

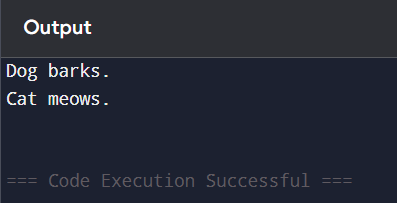
Cat c;

d.speak();

c.speak();

return 0;

}



**Exercise 4: Multilevel Inheritance**

1. Create a base class Vehiclewith a method drive().

2. Create a derived classCarthat inherits from Vehicle and has an additional method start().

3. Create another derived class ElectricCarthat inherits from Car and adds its own method charge().

4. Implement a main() function to demonstrate the usage of these classes.

#include <iostream>

using namespace std;

class Vehicle {

public:

void drive() {

cout << "Vehicle is driving." << endl;

}

};

class Car : public Vehicle {

public:

void start() {

cout << "Car started." << endl;

}

};

class ElectricCar : public Car {

public:

void charge() {

cout << "Electric car is charging." << endl;

}

};

int main() {

ElectricCar eCar;

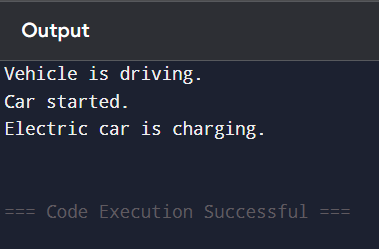
eCar.drive();

eCar.start();

eCar.charge();

return 0;

}



**Exercise 5: Hybrid Inheritance**

1. Create a base classVehicleand a base class Engine.

2. Create a derived class Carthat inherits from both Vehicle and Engine.

3. Implement a main() function to demonstrate the usage of these classes.

#include <iostream>

using namespace std;

class Vehicle {

public:

void drive() {

cout << "Vehicle is driving." << endl;

}

};

class Engine {

public:

void start() {

cout << "Engine started." << endl;

}

};

class Car : public Vehicle, public Engine {

public:

void run() {

start();

drive();

};

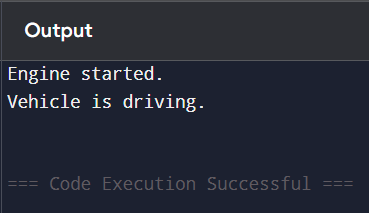
int main() {

Car c;

c.run();

return 0;

}



### **Discussion**

This lab provided a practical understanding of the different types of inheritance in C++. By implementing single, multiple, multilevel, hierarchical, and hybrid inheritance, we observed how derived classes inherit and extend features from base classes. Each type illustrated how code reusability and modularity are achieved. Special attention was given to multiple and hybrid inheritance due to their complexity. Overall, the exercises strengthened our grasp of class relationships and object-oriented design.

### **Conclusion**

In conclusion, the lab successfully demonstrated the power and flexibility of inheritance in C++. By implementing different inheritance models, we reinforced our understanding of object-oriented programming principles such as code reuse, modularity, and extensibility. Each inheritance type serves a unique purpose, and understanding their behavior helps in designing efficient class hierarchies. This foundational knowledge of inheritance will be essential as we build more complex systems using object-oriented programming techniques.