TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR

Lab Report No: - 04

Title: - Inheritance

Submitted by: -	**Submitted To: -**
Name: - Bikram Panthi	**Department of Electronics Engg**
Roll NO: - 13	**Checked by: -**
	Date of submission: -

Objectives:

- To understand and implement various types of inheritance in C++.
- To explore how function overriding and virtual functions enable polymorphism.
- To apply concepts of runtime polymorphism using base class pointers.

Tools and Libraries Used:

- Programming Language: C++
- IDE: G++
- Libraries: include <iostream>, include <string>

Theory:

In C++, inheritance is a powerful feature of object-oriented programming (OOP) that allows a new class (called the derived class) to acquire the properties and behaviors of an existing class (called the base class). This promotes code reusability, hierarchical classification, and a more natural mapping of real-world entities into code.

Inheritance not only enables one class to reuse the functionality of another but also allows programmers to extend and customize the existing behaviors. The derived class inherits accessible attributes and methods of the base class, and it can also define its own members or override existing ones.

Types of Inheritance in C++

1. Single Inheritance:

   A derived class inherits from a single base class.

   Example: `class Circle : public Shape`
2. Multiple Inheritance:

   A derived class inherits from more than one base class.

   Example: `class Manager : public Person, public Employee`
3. Hierarchical Inheritance:

   Multiple derived classes inherit from a single base class.

   Example: `class Dog : public Animal, class Cat : public Animal`
4. Multilevel Inheritance:

   A derived class is itself used as a base class for another class.

   Example: `class ElectricCar : public Car : public Vehicle`
5. Hybrid Inheritance:

   A combination of two or more types of inheritance. This often leads to ambiguity and requires careful handling using virtual inheritance.

Function Overriding

Function overriding occurs when a derived class redefines a base class method with the same name, return type, and parameters. This allows the derived class to provide its own specific implementation of the method.

- The overridden function in the base class must be accessible (usually public) and is commonly declared as virtual to enable runtime polymorphism.

Virtual Functions

Virtual functions are key to achieving dynamic dispatch in C++. When a base class declares a method as virtual, it tells the compiler to wait until runtime to resolve which class's version of the method should be called—based on the type of object, not the pointer/reference.

Characteristics:

- Declared using the virtual keyword in the base class.
- Enable calling derived class methods through base class pointers or references.
- Must be overridden in the derived class to exhibit polymorphic behavior.
- Typically used with base class pointers or references.

Syntax:

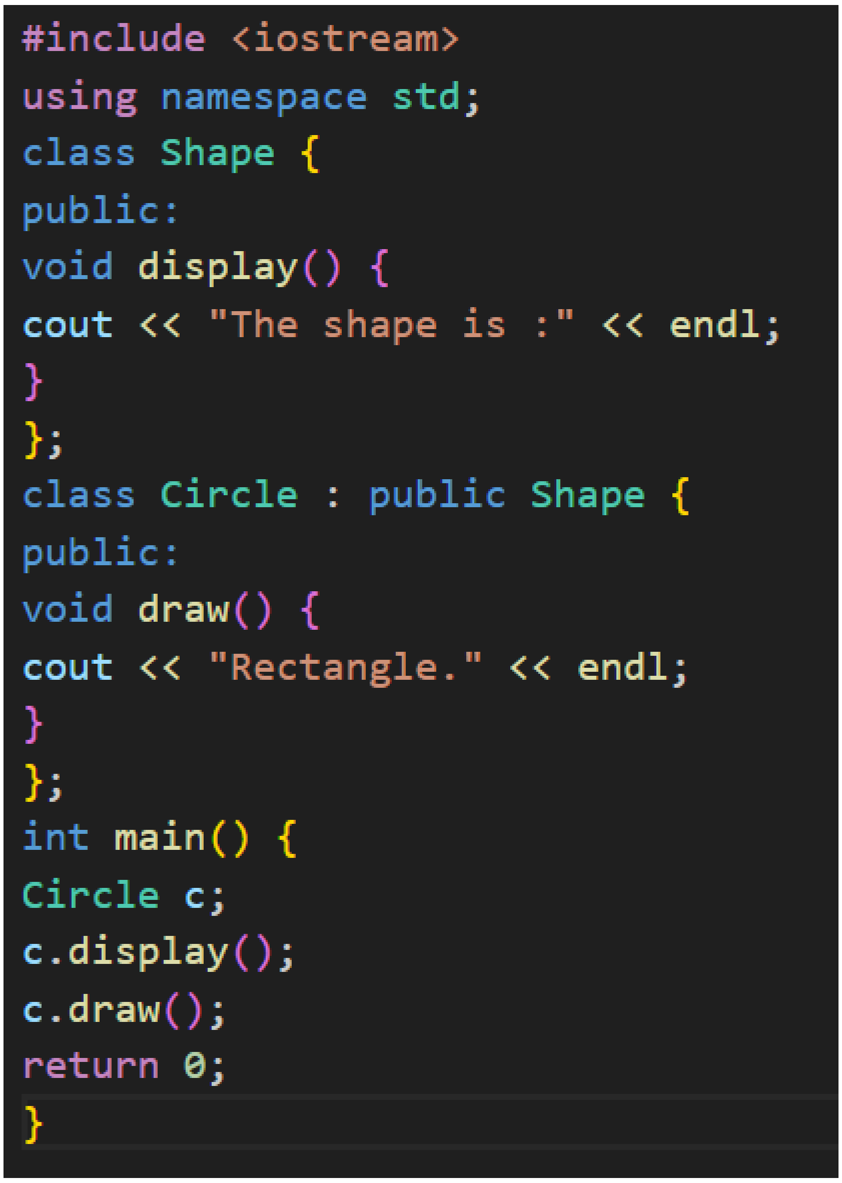
```
class Base {

public:

  virtual void show(); // virtual function

};
```

Exercise 1: Single Inheritance

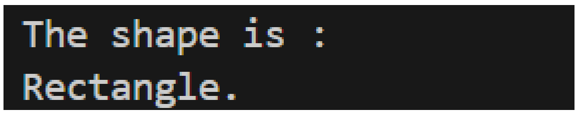
1. Create a base class Shape with a method display().

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

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



```cpp
#include <iostream>
using namespace std;
class Shape {
public:
void display() {
cout << "The shape is :" << endl;
}
};
class Circle : public Shape {
public:
void draw() {
cout << "Rectangle." << endl;
}
};
int main() {
Circle c;
c.display();
c.draw();
return 0;
}
```



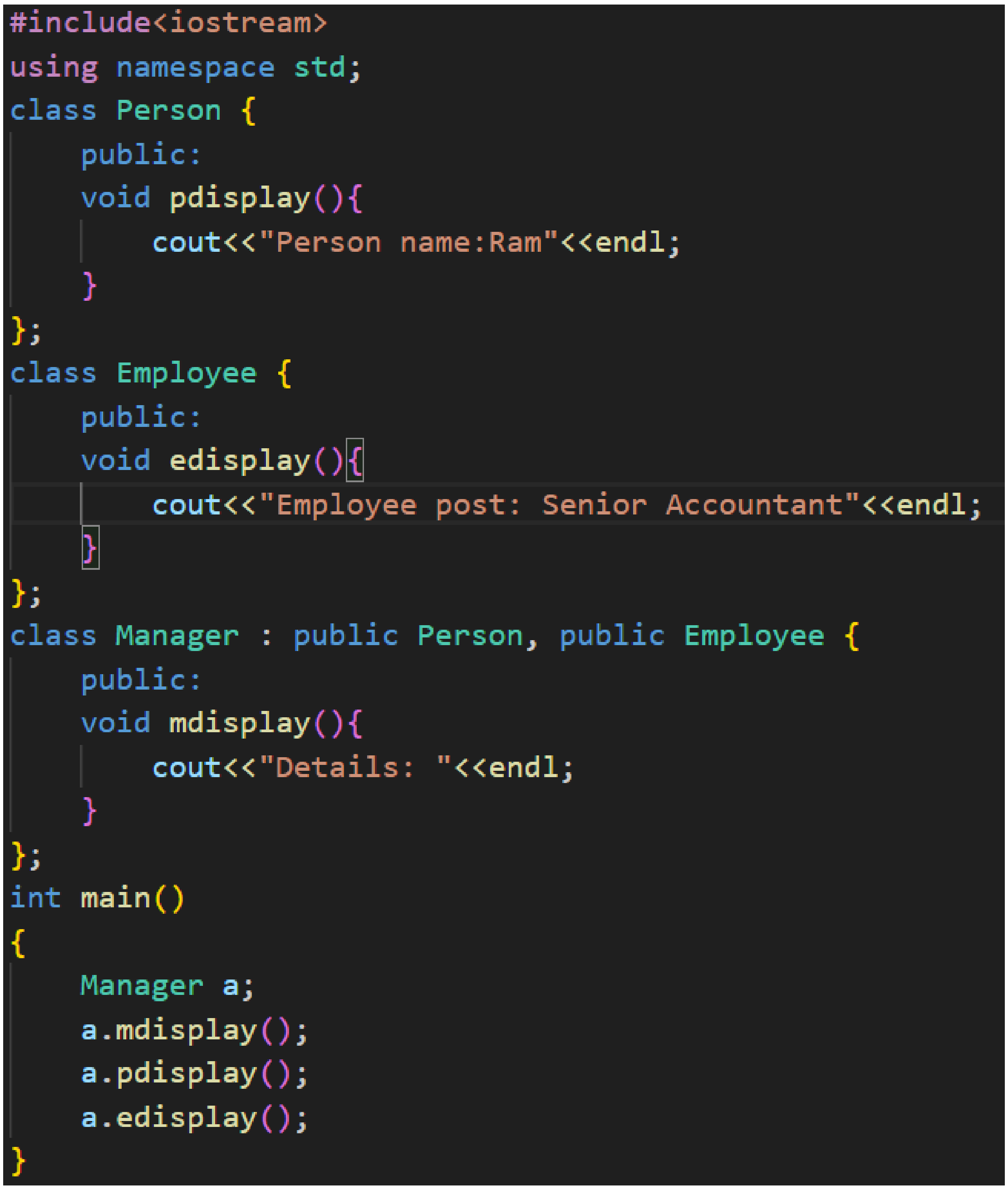
```
The shape is :
Rectangle.
```

Exercise 2: Multiple Inheritance

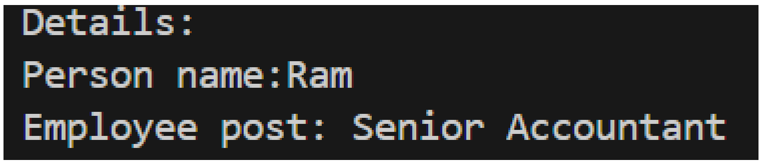
1. Create two base classes Person and Employee with appropriate methods.

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.



```cpp
#include<iostream>
using namespace std;
class Person {
    public:
    void pdisplay(){
        cout<<"Person name:Ram"<<endl;
    }
};
class Employee {
    public:
    void edisplay(){
        cout<<"Employee post: Senior Accountant"<<endl;
    }
};
class Manager : public Person, public Employee {
    public:
    void mdisplay(){
        cout<<"Details: "<<endl;
    }
};
int main()
{
    Manager a;
    a.mdisplay();
    a.pdisplay();
    a.edisplay();
}
```



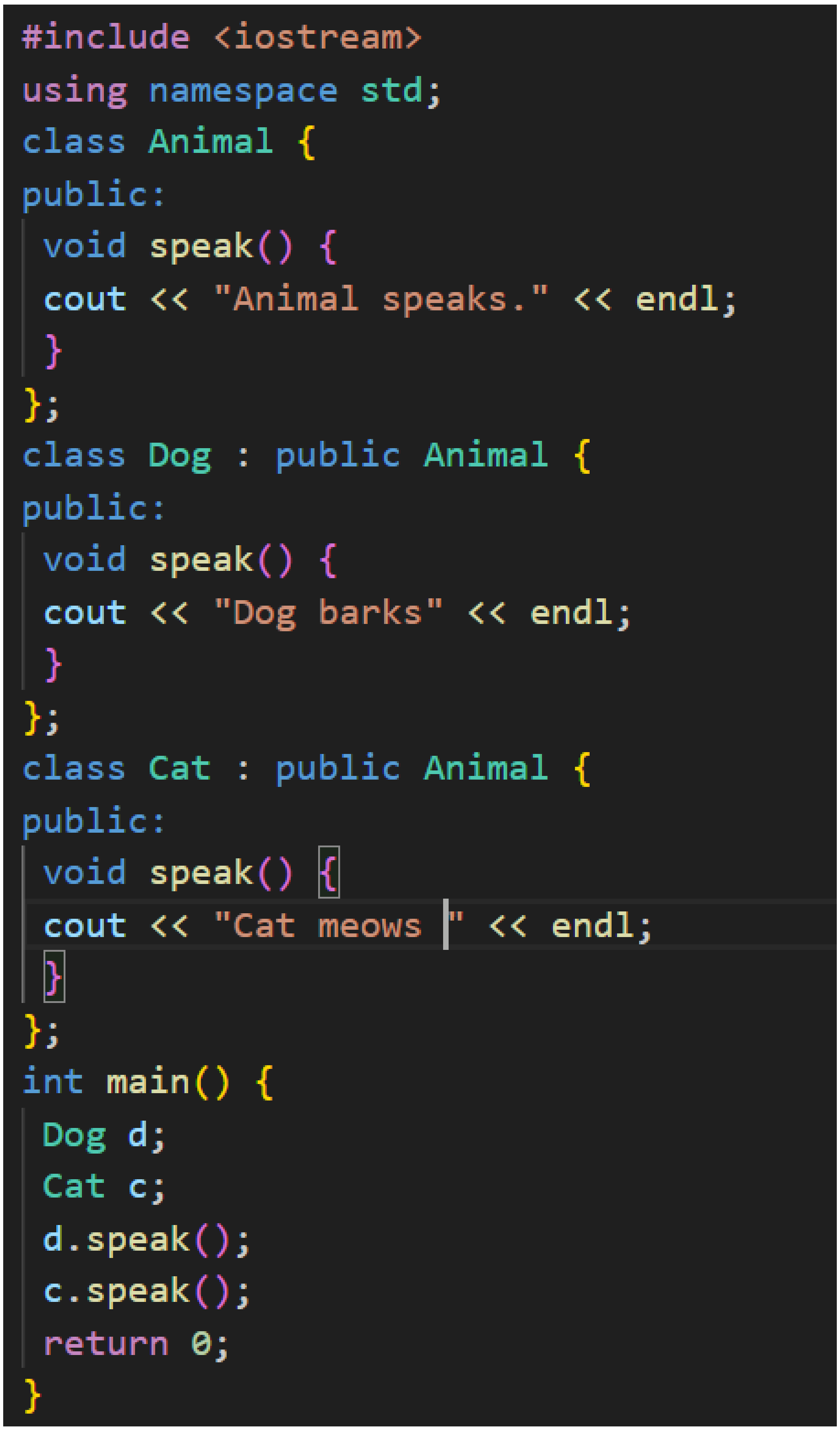
```
Details:
Person name:Ram
Employee post: Senior Accountant
```

Exercise 3: Hierarchical Inheritance

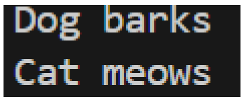
1. Create a base class Animal with a method speak().

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

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



```cpp
#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;
}
```



```
Dog barks
Cat meows
```

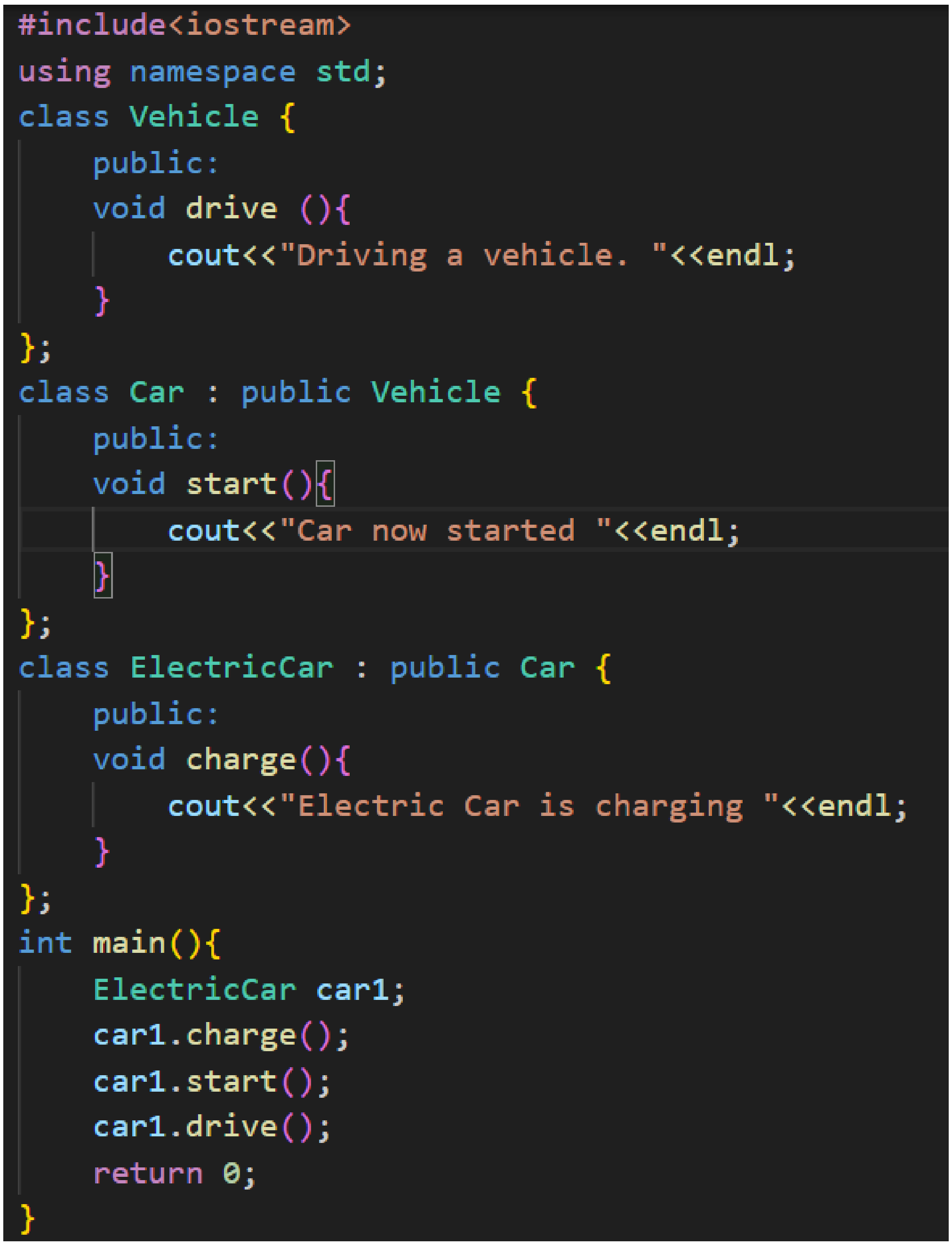
Exercise 4: Multilevel Inheritance

1. Create a base class Vehicle with a method drive().

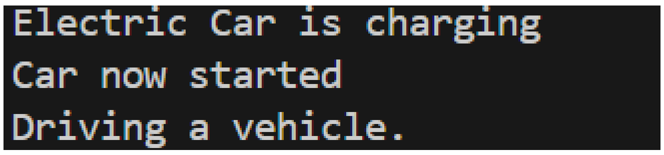
2. Create a derived class Car that inherits from Vehicle and has an additional method start().

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

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



```cpp
#include<iostream>
using namespace std;
class Vehicle {
    public:
    void drive (){
        cout<<"Driving a vehicle. "<<endl;
    }
};
class Car : public Vehicle {
    public:
    void start(){
        cout<<"Car now started "<<endl;
    }
};
class ElectricCar : public Car {
    public:
    void charge(){
        cout<<"Electric Car is charging "<<endl;
    }
};
int main(){
    ElectricCar car1;
    car1.charge();
    car1.start();
    car1.drive();
    return 0;
}
```



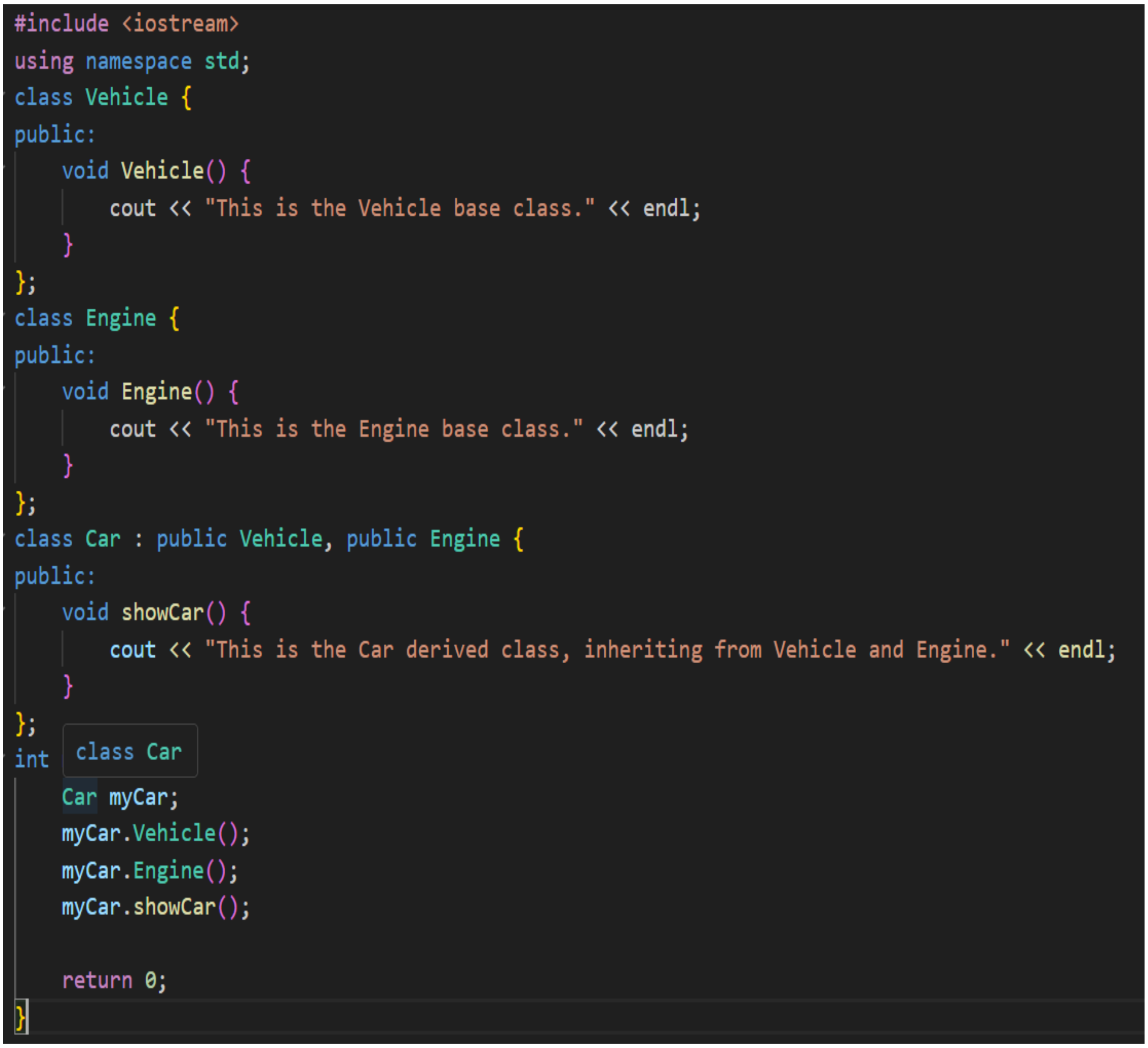
```
Electric Car is charging
Car now started
Driving a vehicle.
```

Exercise 5: Hybrid Inheritance

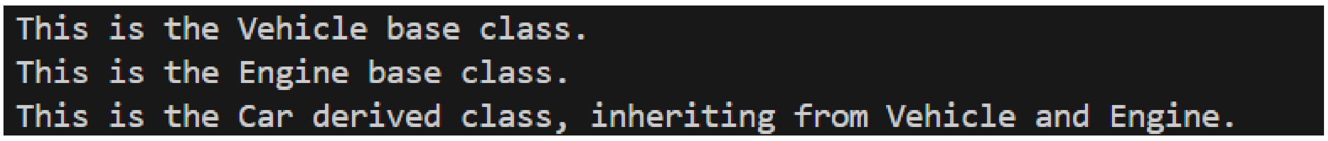
1. Create a base class Vehicle and a base class Engine.

2. Create a derived class Car that 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 Vehicle() {
        cout << "This is the Vehicle base class." << endl;
    }
};
class Engine {
public:
    void Engine() {
        cout << "This is the Engine base class." << endl;
    }
};
class Car : public Vehicle, public Engine {
public:
    void showCar() {
        cout << "This is the Car derived class, inheriting from Vehicle and Engine." << endl;
    }
};
int  class Car
    Car myCar;
    myCar.Vehicle();
    myCar.Engine();
    myCar.showCar();

    return 0;
}
```



```
This is the Vehicle base class.
This is the Engine base class.
This is the Car derived class, inheriting from Vehicle and Engine.
```

Discussion

We learned about the type of inheritance. We handled all type of problems and gained all concepts about inheritance and all its types. The exploration of inheritance types in C++ reveals the language's sophisticated approach to object-oriented programming through its three distinct access specifiers: public, private, and protected inheritance. Public inheritance maintains the "is-a" relationship, preserving the accessibility of base class members and enabling polymorphism.

Conclusion

Understanding C++ inheritance types is fundamental for effective object-oriented design and programming. Public inheritance serves as the backbone of polymorphic hierarchies and interface implementation, while private and protected inheritance offer specialized tools for composition-like relationships and controlled access patterns. The strategic application of different inheritance types allows for more flexible and secure code architectures, ultimately leading to better software design that balances functionality, security, and maintainability in complex C++ applications.