Lab Sheet: Exploring Inheritance Types in C++

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.

Overview:

1. **Single Inheritance**

Single inheritance allows a derived class to inherit from a single base class. It is used when there is a one-to-one relationship between classes.

Syntax:

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

2. **Multiple Inheritance**

In multiple inheritance, a derived class inherits from more than one base class. This helps to combine features of multiple classes into one.

Syntax:

```
class Base1 {
    // base1 members
};
```

```
class Base2 {
    // base2 members
};
class Derived : public Base1, public Base2 {
    // derived class members
};
```

3. Multilevel Inheritance

Multilevel inheritance involves a class derived from another derived class, forming a chain of inheritance.

Syntax:

```
class Base {
    // base class members
};
class Intermediate : public Base {
    // intermediate derived class
};
class Final : public Intermediate {
    // final derived class
};
```

4. Hierarchical Inheritance

Hierarchical inheritance is when multiple derived classes inherit from a single base class.

Syntax:

```
class Base {
```

```
    // base class members
};
class Derived1 : public Base {
    // derived1 class
};
class Derived2 : public Base {
    // derived2 class
};
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance, such as multiple and multilevel. It can lead to complexity and requires careful handling.

Syntax:

```
class A {
    // base class
};
class B : public A {
    // single inheritance
};
class C {
    // another base class
};
class D : public B, public C {
    // hybrid derived class
```

};

- **Basic Virtual Function**

A virtual function allows a function to be overridden in the derived class. It ensures that the function call is resolved at runtime (dynamic binding), not compile-time. It enables **polymorphism**.

Syntax:

```
class Base {
public:
    virtual void show();  // virtual function
};

class Derived : public Base {
public:
    void show();  // overridden function
};
```

- **2. Virtual Destructor**

When a class has virtual functions, its destructor should be declared virtual to ensure proper **object cleanup** during deletion via a base class pointer.

Syntax:

```
class Base {
public:
    virtual ~Base();  // virtual destructor
};

class Derived : public Base {
public:
    ~Derived();
};
```

- **Pure Virtual Function & Abstract Class**

A pure virtual function is a virtual function with `= 0`. A class containing at least one pure virtual function becomes an **abstract class**, which cannot be instantiated directly.

Syntax:

```
class Shape {
public:
    virtual void draw() = 0;  // pure virtual function
```

```
};

class Circle : public Shape {
public:
    void draw();  // must override
};
```

- **Virtual Function with Multiple Inheritance**

In multiple inheritance, if both base classes have virtual functions, the derived class must handle overriding carefully to avoid ambiguity and ensure correct function binding.

Syntax:

```
class A {
public:
    virtual void show();
};

class B {
public:
    virtual void display();
};

class C : public A, public B {
public:
    void show();     // override A
    void display();  // override B
};
```

Lab Exercises:

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.

Code:

```cpp
#include<iostream>
using namespace std;
class shape{
public:
    void display(){
    cout<<"This is the shape."<<endl;
    }
};
class circle: public shape{
public:
    void draw(){
    cout<<"Drawing a circle."<<endl;
    }
};
int main(){
circle c;
c.display();
c.draw();
return 0;
}
```

Output:

```
This is the shape.
Drawing a circle.
```

Qn 2: Create two base classes Person and Employee with appropriate methods. Create a derived class Manager that inherits from both Person and Employee. Implement a main() function to demonstrate the usage of these classes.

```cpp
#include<iostream>
#include<string>
using namespace std;
class person{
public:
    void Name(){
    cout<<"Name: ArunSauden"<<endl;
}
};
class employee{
public:
    void role(){
cout<<"Role: Manager"<<endl;
}};
class manager: public person, public employee{
public:
    void detail(){
    cout<<"Manager details:"<<endl;
    Name();
    role();
    }
    };
int main(){
manager m;
m.detail();
return 0;
}
```

Output:

```
Manager details:
Name: ArunSauden
Role: Manager
```

Qn 3. **Create a base class Animal with a method speak(). Create two derived classes Dog and Cat that inherit from Animal and have their own speak() methods. Implement a main() function to demonstrate the usage of these classes.**

Code:

```cpp
#include<iostream>
using namespace std;
class Animal{
public:
void speak(){
cout<<"Animal speak."<<endl;
}
};
class Dog: public Animal{
public:
    void speak(){
    cout<<"Dog barks!!"<<endl;
    }
};
class cat: public Animal{
public:
    void speak(){
    cout<<"Cat meow!!"<<endl;}
    };
int main(){
Dog d;
cat c;
d.speak();
c.speak();
return 0;}
```

Output:

```
Dog barks!!
Cat meow!!
```

Qn 4: Create a base class Vehicle with a method drive(). Create a derived class Car that inherits from Vehicle and has an additional method start(). Create another derived class ElectricCar that inherits from Car and adds its own method charge(). Implement a main() function to demonstrate the usage of these classes.

Code:

```
#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 Patrolcar: public car{
public:
    void rif(){
    cout<<"Car is refiling."<<endl;
    }
    };
    int main(){
    Patrolcar c;
    c.drive();
    c.start();
    c.rif();
    return 0;
    }
```

Output:

```
Vehicle is driving.
Car started.
Car is refiling.
```

Qn 5. **Create a base class Vehicle and a base class Engine. Create a derived class Car that inherits from both Vehicle and Engine. Implement a main() function to demonstrate the usage of these classes.**

Code:

```cpp
#include <iostream>
using namespace std;
class Vehicle {
public:
 void drive() {
 cout << "Vehicle is driving." << endl;
 }
};

class Engine {
public:
    void warmup(){
    cout<<"Engine is warming up."<<endl;}

 void start() {
 cout << "Engine started." << endl;
 }
};
class Car : public Vehicle, public Engine {
public:
 void run() {
 start();
 warmup();
 drive();
 }
};
int main() {
 Car c;
 c.run();
 return 0;}
```

Output:

```
Engine started.
Engine is warming up.
Vehicle is driving.
```

Qn 6.

Create a base class Base with a virtual method display(). Create a derived class Derived that overrides the display() method. Implement a main() function where you create a Base pointer pointing to a Derived object and call the display() method.

Code:

```cpp
#include <iostream>
using namespace std;
class Base {
public:
    virtual void display() {
        cout << "Display from Base class" << endl;
    }
    virtual ~Base() {}
};
class Derived : public Base {
public:
    void display() override {
        cout << "Display from Derived class" << endl;
    }
};
int main() {
    Base* ptr = new Derived();
    ptr->display();
    delete ptr;
    return 0;
}
```

Output:

```
Display from Derived class
```

Discussion

In our C++ lab, we looked at how inheritance works basically, it lets one class use stuff from another class. There are a few types, like single, multiple, and hierarchical. Each has its own use. For example, with multiple inheritance, a class can get features from more than one parent, which can be powerful but tricky to manage. We can use different type of inheritance as per our need.

Conclusion

Overall, learning about inheritance in C++ really helped me understand how classes can share and reuse code. It's a useful concept that makes programs more organized and efficient. Seeing the different types—like single, multiple, and hierarchical—in action made it easier to grasp how they work in real scenarios. While multiple inheritance can get a bit confusing, it was interesting to see its power. This lab gave me a clearer picture of how object-oriented programming works in practice.