National University
of computer and emerging sciences

Lab 12

Items	Description
Course Title	Object Oriented Programming
Lab Title	Classes (Inheritance)
Duration	3 Hours
Tools	Eclipse/ C++
Objective	To get familiar with the use of different concepts in classes in c++

Inheritance

Inheritance is one of the four pillars of Object-Oriented Programming (OOP), along with encapsulation, abstraction, and polymorphism. Inheritance enables a new class (called the derived class or subclass) to inherit properties and behaviors from an existing class (called the base class or superclass). This allows for code reusability and helps create a hierarchical classification of classes.

```
#include <iostream>

using namespace std;

// Base Class

class Person {

public:

   string name;
```

```
    int age;
    // Base class constructor
    Person(string n, int a) : name(n), age(a) {}
    // Base class member function
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
// Derived Class
class Student : public Person {
public:
    int studentID;
    // Derived class constructor
    Student(string n, int a, int id) : Person(n, a), studentID(id) {}
    // Derived class member function
    void displayStudentInfo() {
```

```
        displayInfo();  // Calling base class function
        cout << "Student ID: " << studentID << endl;
    }
};
int main() {
    // Create an object of the derived class
    Student s("Alice", 20, 12345);
    // Call derived class function
    s.displayStudentInfo();
    return 0;
}
```

Types of Inheritance in C++

There are five main types of inheritance in C++:

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example, it may include multiple inheritance and hierarchical inheritance combined. In hybrid inheritance, a class can derive from multiple classes that follow different inheritance patterns.

```cpp
#include <iostream>

using namespace std;

// Base class

class A {

public:

    void displayA() {

        cout << "Class A method" << endl;

    }

};
```

```
// Derived class B from A (Single Inheritance)
class B : public A {
public:
    void displayB() {
        cout << "Class B method" << endl;
    }
};
// Derived class C from A (Hierarchical Inheritance)
class C : public A {
public:
    void displayC() {
        cout << "Class C method" << endl;
    }
};
```

```
// Derived class D from B and C (Multiple Inheritance)
class D : public B, public C {
public:
    void displayD() {
        cout << "Class D method" << endl;
    }
};
int main() {
    D obj;
    // Access methods from different classes
    obj.displayB(); // From class B
    obj.displayC(); // From class C
    obj.displayD(); // From class D
    // Ambiguity: Which displayA() to call? From B or C?
    // obj.displayA(); // This will cause an error
 return 0;}
```

The Diamond Problem

The diamond problem occurs in multiple inheritance when two parent classes inherit from the same base class, and a derived class inherits from these two parent classes. This leads to ambiguity in resolving which version of the base class's members should be used.

```
#include <iostream>

using namespace std;

// Base class

class A {

public:

    void display() {

        cout << "Class A method" << endl;

    }

};

// Derived classes B and C use virtual inheritance

class B : virtual public A {

};

class C : virtual public A {

};
```

```
// Derived class D inherits from B and C
class D : public B, public C {
};
int main() {
    D obj;
    obj.display(); // Resolves ambiguity
    return 0;
}
```

Explanation of the Solution

- By using `virtual` keyword during inheritance, both `B` and `C` share a single instance of the base class `A`.
- When class `D` inherits from `B` and `C`, there is no ambiguity about which version of `A` to use, as there is only one shared copy.

Key Points

1. **Hybrid Inheritance** can combine multiple inheritance patterns.
2. The **diamond problem** arises from ambiguity in multiple inheritance.
3. **Virtual inheritance** ensures only one instance of the base class is shared among derived classes, solving the diamond problem.

National University
of computer and emerging sciences

Access Modifiers Overview

1. **Public**: Members are accessible from anywhere the object is visible.
2. **Protected**: Members are accessible within the class and by derived classes but not by objects of the class.
3. **Private**: Members are accessible only within the class itself.

When a class is inherited, the base class's access modifier (public, protected, or private inheritance) affects how its members are inherited by the derived class.

Types of Inheritance with Access Modifiers

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 not accessible in the derived class directly.
2. **Protected Inheritance**:
    - Public and protected members of the base class become protected in the derived class.
    - Private members of the base class are not accessible directly in the derived class.
3. **Private Inheritance**:
    - Public and protected members of the base class become private in the derived class.
    - Private members of the base class are not accessible directly in the derived class.

```
#include <iostream>

using namespace std;



class Base {

public:
```

```
    int publicVar = 1;
protected:
    int protectedVar = 2;
private:
    int privateVar = 3;


public:
    void showVars() {
        cout << "Base Class - Public Var: " << publicVar << endl;
        cout << "Base Class - Protected Var: " << protectedVar << endl;
        cout << "Base Class - Private Var: " << privateVar << endl;
    }
};

// Public inheritance example
```

```
class PublicDerived : public Base {
public:
    void showInheritedVars() {
        cout << "PublicDerived - Public Var: " << publicVar << endl;
        cout << "PublicDerived - Protected Var: " << protectedVar << endl;
        // cout << privateVar; // Error: privateVar is not accessible
    }
};
// Protected inheritance example
class ProtectedDerived : protected Base {
public:
    void showInheritedVars() {
        cout << "ProtectedDerived - Public Var: " << publicVar << endl;
        cout << "ProtectedDerived - Protected Var: " << protectedVar << endl;
```

```
        // cout << privateVar; // Error: privateVar is not accessible
    }
};
// Private inheritance example
class PrivateDerived : private Base {
public:
    void showInheritedVars() {
        cout << "PrivateDerived - Public Var: " << publicVar << endl;
        cout << "PrivateDerived - Protected Var: " << protectedVar << endl;
        // cout << privateVar; // Error: privateVar is not accessible
    }
};

int main() {
```

```
    PublicDerived publicObj;

    publicObj.showInheritedVars();

    // publicObj.publicVar; // Accessible in public inheritance

    ProtectedDerived protectedObj;

    protectedObj.showInheritedVars();

    // protectedObj.publicVar; // Error: publicVar is protected in
protected inheritance

    PrivateDerived privateObj;

    privateObj.showInheritedVars();

    // privateObj.publicVar; // Error: publicVar is private in private
inheritance

    return 0;}
```

Explanation of Each Inheritance Type

- **Public Inheritance** (`PublicDerived`):
  - `publicVar` remains public and can be accessed outside the class.
  - `protectedVar` remains protected and accessible within derived classes.
  - `privateVar` is not accessible in `PublicDerived`.
- **Protected Inheritance** (`ProtectedDerived`):

    - Both `publicVar` and `protectedVar` from `Base` are now protected in `ProtectedDerived`.
    - `publicVar` and `protectedVar` can be accessed within the derived class but not from outside.
- **Private Inheritance** (`PrivateDerived`):
    - Both `publicVar` and `protectedVar` become private in `PrivateDerived`.
    - Neither can be accessed from outside the class. Only `PrivateDerived` class methods can access them directly.

Key Takeaway

- **Public inheritance** is typically used when you want a "is-a" relationship where the derived class should publicly expose the base class's functionality.
- **Protected inheritance** is less commonly used but is sometimes used to hide base class functionality from outside access while allowing derived classes to access it.
- **Private inheritance** is used to model a "has-a" relationship where you want to reuse the base class's implementation without exposing it publicly.

Friend Functions and Classes in Inheritance

In C++, **friend functions** and **friend classes** are used to access the private or protected members of a class. When inheritance is involved, friend functions and classes behave slightly differently based on the access specifier (public, protected, or private) used for inheritance.

Friend Function in Inheritance

A friend function can access private and protected members of the base and derived classes it is declared a friend of.


```cpp
#include <iostream>

using namespace std;

class Base {

private:

    int basePrivate;

protected:

    int baseProtected;

public:

    int basePublic;

    Base() : basePrivate(10), baseProtected(20), basePublic(30)
{}

    // Declare a friend function

    friend void showBaseDetails(const Base& obj);

};
```

```
// Derived class
class Derived : public Base {
private:
    int derivedPrivate;
protected:
    int derivedProtected;
public:
    int derivedPublic;
    Derived() : derivedPrivate(40), derivedProtected(50),
derivedPublic(60) {}
    // Declare a friend function
    friend void showDerivedDetails(const Derived& obj);
};
```

```
// Friend function of Base
void showBaseDetails(const Base& obj) {
    cout << "Base Private: " << obj.basePrivate << endl;
    cout << "Base Protected: " << obj.baseProtected << endl;
    cout << "Base Public: " << obj.basePublic << endl;
}
// Friend function of Derived
void showDerivedDetails(const Derived& obj) {
    cout << "Base Protected: " << obj.baseProtected << endl;
    cout << "Base Public: " << obj.basePublic << endl;
    cout << "Derived Private: " << obj.derivedPrivate << endl;
    cout << "Derived Protected: " << obj.derivedProtected <<
endl;
    cout << "Derived Public: " << obj.derivedPublic << endl;
}
```

```
int main() {
    Base baseObj;
    Derived derivedObj;
    cout << "Details of Base class:\n";
    showBaseDetails(baseObj);
    cout << "\nDetails of Derived class:\n";
    showDerivedDetails(derivedObj);
    return 0;
}
```

Explanation:

- `showBaseDetails` is a friend of the `Base` class and can access its private and protected members.
- `showDerivedDetails` is a friend of the `Derived` class and can access both `Base`'s inherited protected and public members as well as `Derived`'s private and protected members.

Friend Class in Inheritance

A friend class can access all private and protected members of another class. If a class declares another class as its friend, the friend class can also access private and protected members of the base class in case of inheritance.

```
#include <iostream>
using namespace std;
class Base {
private:
    int basePrivate;

protected:
    int baseProtected;

public:
    int basePublic;

    Base() : basePrivate(10), baseProtected(20), basePublic(30)
{}
    // Declare a friend class
    friend class FriendClass;
};
```

```
// Derived class
class Derived : public Base {
private:
    int derivedPrivate;
protected:
    int derivedProtected;
public:
    int derivedPublic;
    Derived() : derivedPrivate(40), derivedProtected(50),
derivedPublic(60) {}
    // Declare a friend class
    friend class FriendClass;
};
// Friend class
class FriendClass {
public:
    void showDetails(const Base& baseObj, const Derived&
derivedObj) {
        // Accessing Base class members
        cout << "Base Private: " << baseObj.basePrivate << endl;
        cout << "Base Protected: " << baseObj.baseProtected <<
endl;
        cout << "Base Public: " << baseObj.basePublic << endl;
```

```
        // Accessing Derived class members
        cout << "Base Protected (from Derived): " <<
derivedObj.baseProtected << endl;
        cout << "Base Public (from Derived): " <<
derivedObj.basePublic << endl;
        cout << "Derived Private: " << derivedObj.derivedPrivate
<< endl;
        cout << "Derived Protected: " <<
derivedObj.derivedProtected << endl;
        cout << "Derived Public: " << derivedObj.derivedPublic <<
endl;
    }
};
int main() {
    Base baseObj;
    Derived derivedObj;
    FriendClass friendClassObj;

    friendClassObj.showDetails(baseObj, derivedObj);

    return 0;
}
```

Explanation:

- `FriendClass` is a friend of both `Base` and `Derived`, so it can access their private and protected members.
- In the `showDetails` function of `FriendClass`, we can see members from both `Base` and `Derived` classes.

Key Points

1. A **friend function** of a class can access private and protected members of that class.
2. A **friend function** of the derived class can access inherited protected and public members.
3. A **friend class** can access all private, protected, and public members of the class it is a friend of, even in an inheritance hierarchy.
4. Friendship is **not inherited**. If `Base` declares `FriendClass` as a friend, `Derived` does not automatically inherit the friendship. However, you can explicitly declare `FriendClass` as a friend of `Derived`.

National University of computer and emerging sciences

Lab Task

Create Header files

Task 1: Library Management System

Objective: Implement a system to manage books and users, utilizing multiple inheritance and friend functions.

Details:

- Create a base class `LibraryItem` with protected attributes like `itemID` and `title`. Include a method to display these details.
- Create two derived classes:
    - `Book` (inherits publicly from `LibraryItem`) with additional attributes `author` and `publisher`.
    - `Magazine` (inherits privately from `LibraryItem`) with additional attributes `volume` and `issue`.
- Use virtual inheritance to ensure no ambiguity arises if another class inherits both `Book` and `Magazine`.
- Create a friend function `showItemDetails` that can access private and protected attributes of both `Book` and `Magazine`.

Task 2: Employee Management System

Objective: Implement a system to manage employees and departments, showcasing different types of inheritance and access modifiers.

Details:

1. Create a base class `Person` with protected attributes `name` and `age`. Include a method to display these details.
2. Create a class `Employee` that inherits publicly from `Person` and has additional attributes `employeeID` and `designation`.
3. Create another class `Manager` that inherits privately from `Employee` and has an additional attribute `department`.

4. Use a friend function `showManagerDetails` to display all the private and protected data of a `Manager` object.
5. Use protected inheritance for another class `Contractor` that inherits from `Person`, restricting the accessibility of `Person`'s attributes to further derived classes.

Task 3: Virtual University System

Objective: Implement a system to simulate a university structure using virtual inheritance and friend functions.

Details:

1. Create a base class `UniversityMember` with attributes like `memberID` and `name`.
2. Create two derived classes:
    - `Student` (inherits virtually and publicly from `UniversityMember`) with additional attributes `course` and `CGPA`.
    - `Faculty` (inherits virtually and publicly from `UniversityMember`) with additional attributes `subject` and `designation`.
3. Create another class `TeachingAssistant` that inherits publicly from both `Student` and `Faculty`.
4. Add a friend function `showTeachingAssistantDetails` to access and display all attributes of a `TeachingAssistant`.

Task 4: Vehicle System

Objective: Implement a hierarchy for vehicles with mixed inheritance and access modifiers.

Details:

1. Create a base class `Vehicle` with protected attributes `make` and `model`, and a public method to display these details.
2. Create a derived class `Car` (inherits publicly from `Vehicle`) with additional attributes `engineCapacity` and `numDoors`.
3. Create another derived class `Truck` (inherits protectedly from `Vehicle`) with additional attributes `cargoCapacity` and `numWheels`.

4. Use virtual inheritance to avoid ambiguity in a class `AmphibiousVehicle` that inherits from both `Car` and `Truck`.
5. Add a friend function `showAmphibiousDetails` to display all private and protected attributes of an `AmphibiousVehicle`.