Object-Oriented Programming

CL-1004

Lab 06

Inheritance, Modes of Inheritance, Inheritance Types, and Resolving the Diamond Problem

National University of Computer and Emerging Sciences–NUCES – Karachi

Contents

1. Inheritance

Capability of a class to derive properties and characteristics from **another class** is known as **Inheritance**. The existing class is called the **base** class (or sometimes **super** class) and the new class is referred to as the **derived** class (or sometimes **subclass**). For e.g.: The **car** is a **vehicle**, so any attributes and behaviors of a **vehicle** are also attributes and behaviors of a **car**.

Base Class: It is the class from which features are to be inherited into another class.

Derived Class: It is the class in which the **base** class features are inherited. A derived class can have additional properties and methods not present in the parent class that distinguishes it and provides additional functionality.

Syntax:

```
class derived_class_name : access_mode base_class_name {
    //body of subclass
};
```

2. is-a Relationship

The "**is-a**" relationship is a fundamental concept in object-oriented programming (OOP) that defines a subtyping hierarchy between classes. It asserts that a derived class (**subclass**) is a specialized version of a base class (**superclass**), inheriting its interface and behavior while potentially adding or modifying functionality.

An "**is-a**" relationship exists when:

1. **Public Inheritance** is used in C++ (class **Derived** : public **Base**).
2. The derived class can substitute the base class in all contexts.
    - Any function or operation expecting a **Base** object should work correctly with a **Derived** object.
3. The derived class represents a more specific type of the base class.
    - Example: Dog **is-a** Animal, Car **is-a** Vehicle.

3. Modes of Inheritance

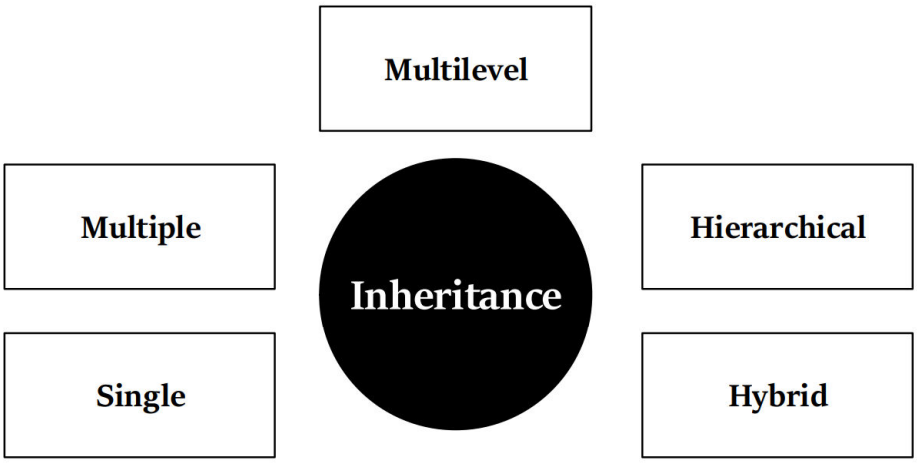
Public Mode: If we derive a subclass from a **public** base class. Then the **public** member of the base class will become **public** in the derived class and **protected** members of the base class will become **protected** in the derived class.

Protected Mode: If we derive a subclass from a **Protected** base class. Then both public members and **protected** members of the base class will become **protected** in the derived class.

Private Mode: If we derive a subclass from a **Private** base class. Then both public members and **protected** members of the base class will become **Private** in the derived class.

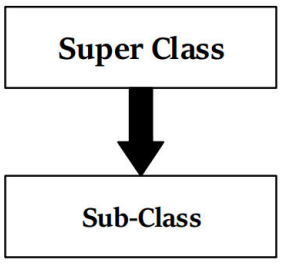
Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (hidden)	Not accessible (hidden)	Not accessible (hidden)

4. Types of Inheritance



4.1. Single Inheritance

In single inheritance, a class is allowed to inherit from only i.e., **one subclass** is inherited by **one base class** only.




```
class sub_class_name : access_mode base_class_name {
    //body of subclass
};
```

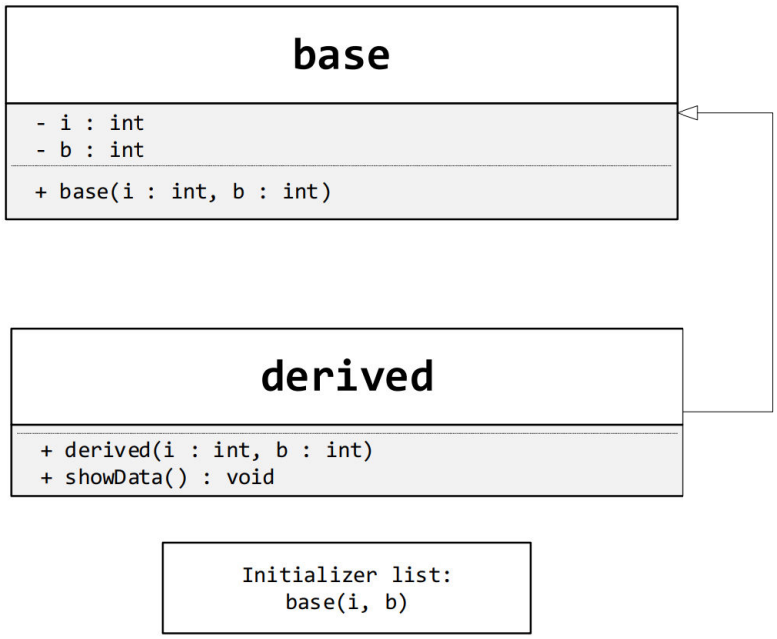
Example:

```
#include <iostream>
using namespace std;

class base {
public:
    int i, b;

    // constructor to initialize i and b
    base(int i, int b) {
        this->i = i;
        this->b = b;
    }
};

class derived : public base {
public:
    // constructor to call the base class constructor
    derived(int i, int b) : base(i, b) {}

    void showData() {
        cout << "i = " << i << endl << "b = " << b;
    }
};

int main() {
    // create an object of derived class and initialize it using the
constructor
    derived d1(4, 5);
    d1.showData();
```

```
    return 0;
}
```

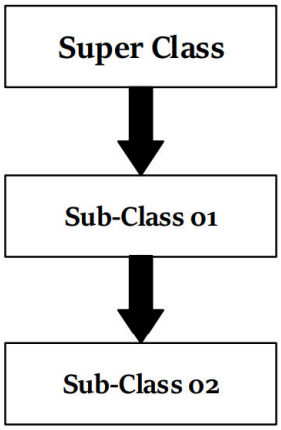
Output

```
i = 4
b = 5
```



4.2. Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from **another derived class**.




```
class second_derived_class : access_mode first_derived_class{
    // body of second_derived_class
};
```

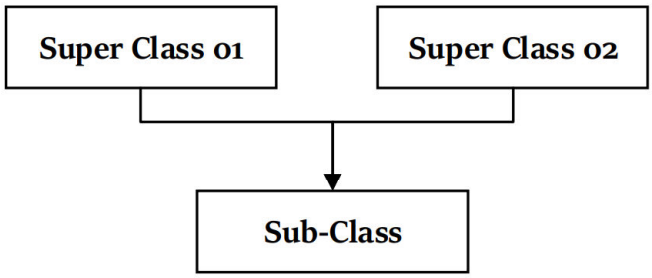
Example:

```
#include <iostream>
using namespace std;

class base {
public:
    int i, b;

    // constructor to initialize i and b
    base(int i, int b) {
        this->i = i;
        this->b = b;
    }
};

class derived1 : public base {
public:
    int j, c;

    // constructor to initialize base class and derived1 members
    derived1(int i, int b, int j, int c) : base(i, b) {
        this->j = j;
        this->c = c;
    }

    void showData1() {
        cout << "i = " << i << endl << "b = " << b << endl;
    }
};

class derived2 : public derived1 {
public:
    // constructor to initialize derived1 class
    derived2(int i, int b, int j, int c) : derived1(i, b, j, c) {}

    void showData2() {
        cout << "j = " << j << endl << "c = " << c << endl;
    }
};

int main() {
    // create an object of derived2 and initialize it using the constructor
    derived2 d2(4, 5, 6, 7);
    d2.showData1();
    d2.showData2();
    return 0;
}
```

Output	i = 4 b = 5 j = 6 c = 7

4.3. Multiple Inheritance

In Multiple Inheritance a class can inherit from **more than one** class. i.e., **one sub class** is inherited from **more than one base class**.




```
class base_class_1 {
    // base_class_1
};
class base_class_2 {
    // body of base_class_2
};
class derived_class : access_mode base_class_1, access_mode base_class_2{
        // body of derived_class
};
```

Example:

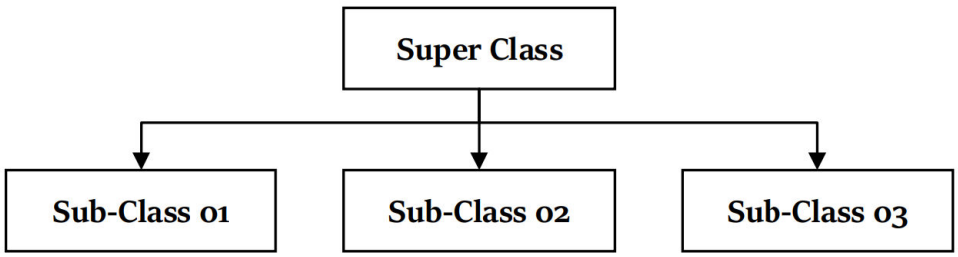
```
#include <iostream>
using namespace std;

class base1 {
public:
    int i, b;

    // constructor to initialize i and b
    base1(int i, int b) {
        this->i = i;
        this->b = b;
    }
};

class base2 {
public:
    int j, c;
```

```
    // constructor to initialize j and c
    base2(int j, int c) {
        this->j = j;
        this->c = c;
    }
};

class derived : public base1, public base2 {
public:
    // constructor to initialize base1 and base2
    derived(int i, int b, int j, int c) : base1(i, b), base2(j, c) {}

    void showData1() {
        cout << "i = " << i << endl << "b = " << b << endl;
    }

    void showData2() {
        cout << "j = " << j << endl << "c = " << c << endl;
    }
};

int main() {
    // create an object of derived and initialize it using the constructor
    derived d1(4, 5, 6, 7);
    d1.showData1();
    d1.showData2();
    return 0;
}
```

Output	i = 4 b = 5 j = 6 c = 7

4.4. Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving **more than one class** from a **base class**.



```
class base_class {
    // base_class
};
class derived_class_1 : access_mode base_class {
    // body of derived_class_1
};
class derived_class_2 : access_mode base_class {
    // body of derived_class_2
};
class derived_class_3 : access_mode base_class {
    // body of derived_class_3
};
```

Example:

```
#include <iostream>
using namespace std;

class base {
public:
    int i, b;

    // Constructor to initialize i and b
    base(int i, int b) {
        this->i = i;
        this->b = b;
    }
};

class derived1 : public base {
public:
    // constructor to call the base class constructor
    derived1(int i, int b) : base(i, b) {}

    void showData1() {
        cout << "This is the data of Base class set by derived class 1" <<
endl;
        cout << "i = " << i << endl << "b = " << b << endl;
    }
};

class derived2 : public base {
public:
    // constructor to call the base class constructor
    derived2(int i, int b) : base(i, b) {}

    void showData2() {
        cout << "This is the data of Base class set by derived class 2" <<
endl;
```

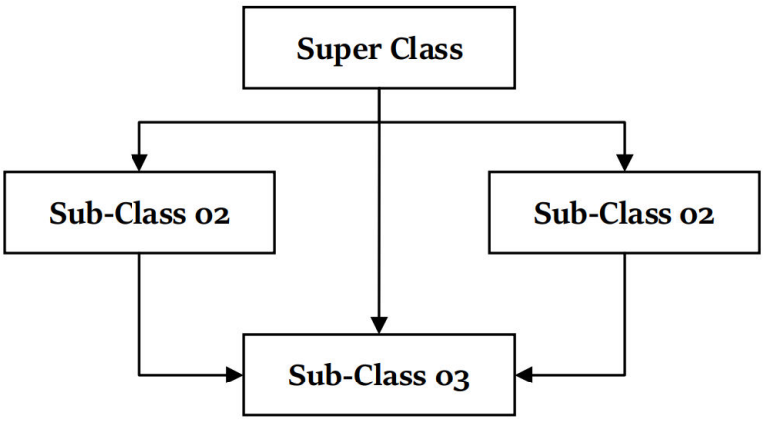
```
        cout << "i = " << i << endl << "b = " << b << endl;
    }
};

class derived3 : public base {
public:
    // constructor to call the base class constructor
    derived3(int i, int b) : base(i, b) {}

    void showData3() {
        cout << "This is the data of Base class set by derived class 3" <<
endl;
        cout << "i = " << i << endl << "b = " << b << endl;
    }
};

int main() {
    // create objects of derived classes and initialize them using
constructors
    derived1 d1(4, 5);
    derived2 d2(6, 7);
    derived3 d3(8, 9);

    d1.showData1();
    d2.showData2();
    d3.showData3();

    return 0;
}
```

Output

```
This is the data of Base class set by derived class 1
i = 4
b = 5
This is the data of Base class set by derived class 2
i = 6
b = 7
This is the data of Base class set by derived class 3
i = 8
b = 9
```

4.5. Hybrid Inheritance

Hybrid inheritance is a combination of **more than one type of inheritance**. For example: Combining **Hierarchical** inheritance and **Multiple** Inheritance.




```
class base_class {
    // base_class
};
class derived_class_1 : access_mode base_class {
        // body of derived_class_1
};
class derived_class_2 : access_mode base_class {
        // body of derived_class_2
};
class derived_class_3 : access_mode base_class, access_mode derived_class_1,
access_mode derived_class_2 {
        // body of derived_class_3
};
```

Example:

```
#include "iostream"
using namespace std;

class base{

public:
    int i, b;
    void setData(int i, int b){
        this->i = i;
        this->b = b;
    }
};
class derived1 : public base{
public:
    int x, y;
    void setXY(int x, int y){
        this->x = x;
        this->y = y;
```

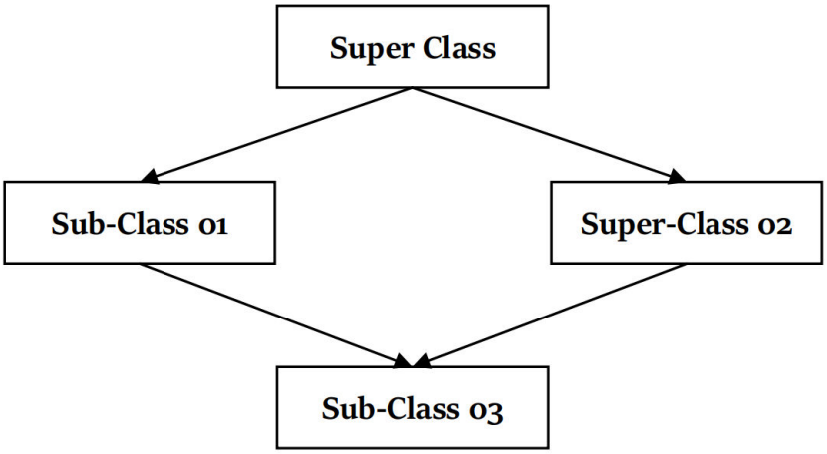
```
    }
    void showData1(){
        cout << "This is the data of Base class set by derived class 1" <<
endl;
        cout << "i = " << i << endl << "b = " << b << endl;
    }
};
class derived2 : public base{
public:
    int u, v;
    void setUV(int u, int v){
        this->u = u;
        this->v = v;
    }
    void showData2(){
        cout << "This is the data of Base class set by derived class 2" <<
endl;
        cout << "i = " << i << endl << "b = " << b << endl;
    }
};
class derived3 : public base, public derived1, public derived2{
public:
    void showData3(){
        cout << "This is the data of derived class 1 set by derived class 3"
<< endl;
        cout << "x = " << x << endl << "y = " << y << endl;
        cout << "This is the data of derived class 2 set by derived class 3"
<< endl;
        cout << "u = " << u << endl << "v = " << v << endl;
    }
};

int main(){

    derived1 d1;
    derived2 d2;
    derived3 d3;

    d1.setData(4,5);
    d1.showData1();

    d2.setData(6,7);
    d2.showData2();


    d3.setXY(10, 11);
    d3.setUV(12, 13);
    d3.showData3();
    return 0;
}
```

Output	This is the data of Base class set by derived class 1 i = 4 b = 5 This is the data of Base class set by derived class 2

```
i = 6
b = 7
This is the data of derived class 1 set by derived class 3
x = 10
y = 11
This is the data of derived class 2 set by derived class 3
u = 12
v = 13
```

5. Diamond Problem

The **diamond problem** is a common issue in C++ that arises in **multiple inheritance** when a class inherits from **two** or **more** classes that have a **common base class**. This creates ambiguity for the compiler because it cannot determine which path to take to access the members of the common base class.



Here:

- Class **A** is the **base class**.
- Classes **B** and **C** inherit from **A**.
- Class **D** inherits from **both B** and **C**.

This creates a **diamond-shaped** inheritance structure, hence the name "**diamond problem**."

Example:

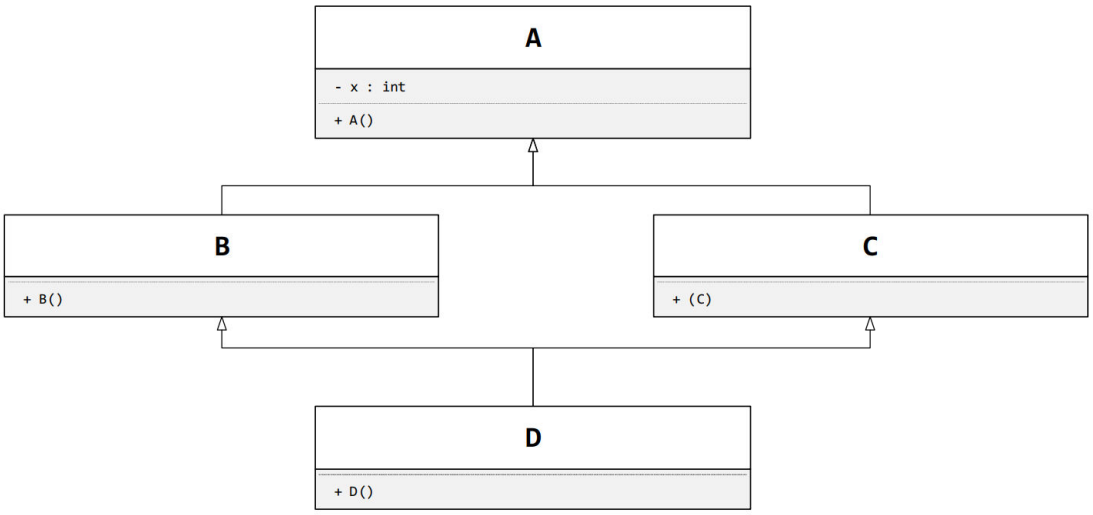
```cpp
#include <iostream>
using namespace std;

class A {
public:
    int x;
    A() { x = 10; }
};
```

```cpp
class B : public A {};

class C : public A {};

class D : public B, public C {};

int main() {
    D d;
    cout << d.x; // Error: Ambiguity - which 'x' to access? From B or C?
    return 0;
}
```

Error

```
lab06.cpp: In function 'int main()':
lab06.cpp:18:15: error: request for member 'x' is ambiguous
     cout << d.x; // Error: Ambiguity - which 'x' to access?
From B or C?
               ^
lab06.cpp:6:9: note: candidates are: int A::x
     int x;
         ^
lab06.cpp:6:9: note:                 int A::x
```

- The class **D** inherits two copies of **x**:
  - One from **B** (which inherits from **A**).
  - One from **C** (which also inherits from **A**).
- When you try to access **d.x**, the compiler doesn't know which **x** to use (from **B** or **C**), resulting in a **compilation error**.



5.1. Resolving Diamond Problem | Virtual Inheritance

To **resolve** the **diamond problem**, C++ provides **virtual inheritance**. When you use **virtual** inheritance, only **one instance** of the **common base** class **(A)** is shared among **all** the derived classes (**B** and **C**). This ensures that there is no ambiguity when accessing members of the **base class**.

Example:

```cpp
#include <iostream>
using namespace std;

class A {
public:
    int x;
    A() { x = 10; }
};

class B : virtual public A {}; // virtual inheritance

class C : virtual public A {}; // virtual inheritance

class D : public B, public C {};

int main() {
    D d;
    cout << d.x; // no ambiguity - only one 'x' exists
    return 0;
}
```

Output	10

- When **B** and **C** inherit **virtually** from **A**, the compiler ensures that only **one instance** of **A** is created in the most derived class (**D**).
- This avoids duplication of the base class members and resolves the ambiguity.