

## Table of Contents

5.1 Quan hệ giữa các lớp đối tượng	88
5.2 Kế thừa & tính chất	89
5.3. Định nghĩa lớp cơ sở và lớp dẫn xuất (Base class và Derived class)	90
5.4 Thành viên “Protected”	94
5.5 Các kiểu kế thừa	98
5.6 Hàm tạo và hàm hủy	112
5.7 Định nghĩa lại phương thức (Method Overriding)	115
5.8 Phép gán và con trỏ trong kế thừa	115

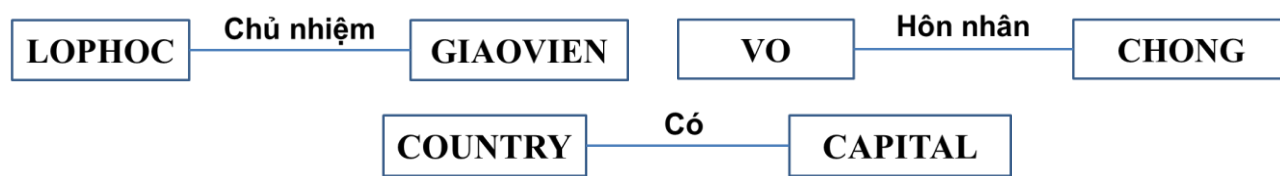
### 5.1 Quan hệ giữa các lớp đối tượng

❖ Giữa các lớp đối tượng có những loại quan hệ sau:

- **Quan hệ một một (1-1):** một đối tượng thuộc lớp này quan hệ với một đối tượng thuộc lớp kia và một đối tượng thuộc lớp kia có quan hệ duy nhất với một đối tượng thuộc lớp này.



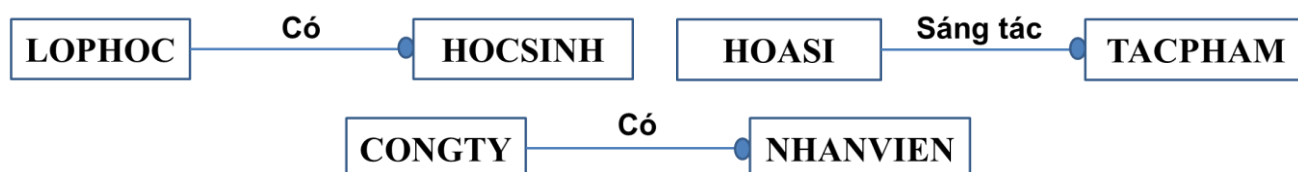
Ví dụ:



- **Quan hệ một nhiều (1-n):** một đối tượng thuộc lớp này quan hệ với nhiều đối tượng thuộc lớp kia và một đối tượng thuộc lớp kia có quan hệ duy nhất với một đối tượng thuộc lớp này.



Ví dụ:



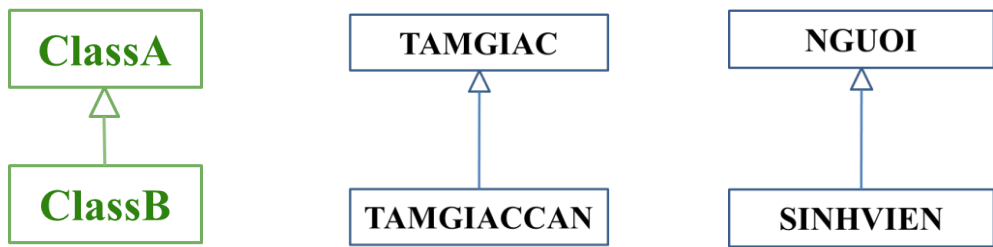
- **Quan hệ nhiều nhiều (n-n):** khi một đối tượng thuộc lớp này có quan hệ với nhiều đối tượng thuộc lớp kia và một đối tượng thuộc lớp kia cũng có quan hệ với nhiều đối tượng thuộc lớp này.



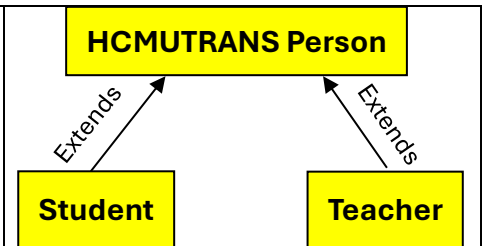
Ví dụ:



- **Quan hệ đặc biệt hóa, tổng quát hóa:** lớp đối tượng này là trường hợp đặc biệt của lớp đối tượng kia và lớp đối tượng kia là trường hợp tổng quát của lớp đối tượng này.



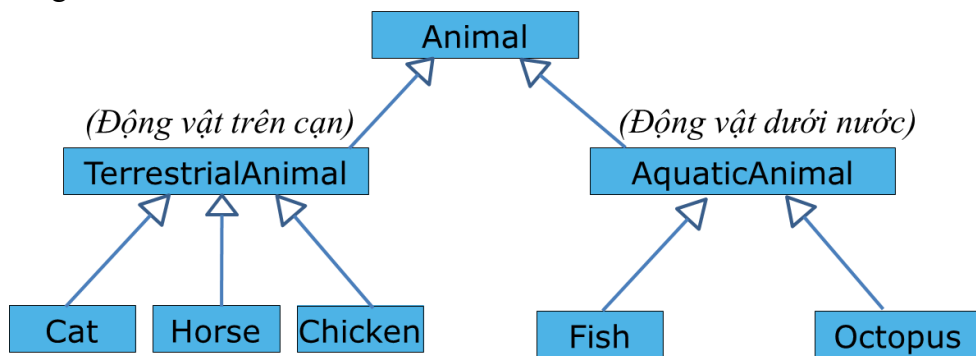
Dưới đây là một ví dụ khác về quan hệ này:

<p><b>People at HCMUTRANS = HCMUTRANS Teachers + HCMUTRANS Students</b> (Tạm thời chỉ đề cập đến 2 đối tượng này).</p> <p><b>Mọi người ở HCMUTRANS có những thuộc tính/hành vi chung nào?</b></p> <ul style="list-style-type: none"> <li>➤ Thuộc tính: name, ID, address</li> <li>➤ Hành vi: displayProfile(), changeAddr()</li> </ul> <p><b>Sinh viên cần có những thuộc tính/hành vi riêng nào?</b></p> <ul style="list-style-type: none"> <li>➤ Thuộc tính: group, year, course</li> <li>➤ Hành vi: changeGroup()</li> </ul> <p><b>Giảng viên cần có những thuộc tính/hành vi riêng nào?</b></p> <ul style="list-style-type: none"> <li>➤ Thuộc tính: rank, year, subject, salary,....</li> <li>➤ Hành vi: addSubj(), increaseSal()</li> </ul>	 <p><i>Mô hình phân cấp</i></p> <p>Student is Person Teacher is Person</p>
---	---

#### \* Cây kế thừa:

+ Các lớp với các đặc điểm tương tự nhau có thể được tổ chức thành một sơ đồ phân cấp kế thừa (cây kế thừa).

+ Là một cây đa nhánh thể hiện mối quan hệ đặc biệt hóa-tổng quát hóa giữa các lớp trong hệ thống, trong chương trình.



## 5.2 Kế thừa & tính chất

### ❖ Kế thừa:

- ✎ Một đặc điểm của ngôn ngữ dùng để biểu diễn mối quan hệ đặc biệt hóa – tổng quát hóa giữa các lớp. Các lớp được trừu tượng hóa và được tổ chức thành một sơ đồ phân cấp lớp.
- ✎ Nó cho phép một lớp có thể được thừa hưởng các thuộc tính, phương thức từ một lớp khác.
- ✎ Sự kế thừa là một mức cao hơn của trừu tượng hóa, cung cấp một cơ chế gom chung các lớp có liên quan với nhau thành một mức khái quát hóa đặc trưng cho toàn bộ các lớp nói trên.

### ❖ Tính chất:

- ✎ Cho phép tạo lớp mới (**derived class – lớp dẫn xuất**) từ lớp đã có (**base class – lớp cơ sở**); Nhiều lớp có thể dẫn xuất từ một lớp cơ sở; Một lớp có thể là dẫn xuất của nhiều lớp cơ sở

Lớp kế thừa từ một lớp khác thì được gọi là **lớp con (child class, subclass)** hay **lớp dẫn xuất (derived class)**. Lớp được các lớp khác kế thừa được gọi là **lớp cha (parent class, superclass)** hay **lớp cơ sở (base class)**.

*Ví dụ thực tế*

- Bạn có một lớp đối tượng con người, có các thuộc tính như họ tên, ngày sinh, quê quán, ta khai báo thêm một lớp sinh viên kế thừa từ lớp con người.
- Khi đó lớp sinh viên sẽ có các thuộc tính họ tên, ngày sinh, quê quán từ lớp con người mà không cần phải khai báo lại. Lớp con người được gọi là lớp cha và lớp sinh viên là lớp con
- Ngoài các thuộc tính của lớp cha, lớp con còn có thể có thêm các thuộc tính, phương thức của riêng nó. Ở ví dụ trên thì lớp sinh viên có thể có thêm các thuộc tính như MSSV, tên trường, chuyên ngành ...

\* **Thừa kế không chỉ giới hạn ở một mức:** Một lớp dẫn xuất có thể là lớp cơ sở cho các lớp dẫn xuất khác.

\* **Nhận xét:** Lớp dẫn xuất “mạnh mẽ hơn” lớp cơ sở; *lớp cơ sở chỉ là những thứ chung chung, sơ khai, tổng quát.*

- ✎ Làm tăng khả năng **tái sử dụng, sửa chữa, nâng cấp hệ thống.**
- ✎ Cho phép tạo ra sự phân loại theo cấp bậc
- ✎ Là công cụ cho phép mô tả cụ thể hóa các khái niệm theo nghĩa
- ✎ Định nghĩa sự tương thích giữa các lớp, nhờ đó ta có thể chuyển kiểu tự động.

### 5.3. Định nghĩa lớp cơ sở và lớp dẫn xuất (Base class và Derived class)

➤ Mọi quan hệ giữa Base class và Derived class là **“is-A”**

- ✎ *Một sinh viên là một người*
- ✎ *Một hình tròn là một hình ellipse*
- ✎ *Một tam giác là một đa giác*

➤ **Derived class:** kế thừa tất cả các thành viên của Base class.

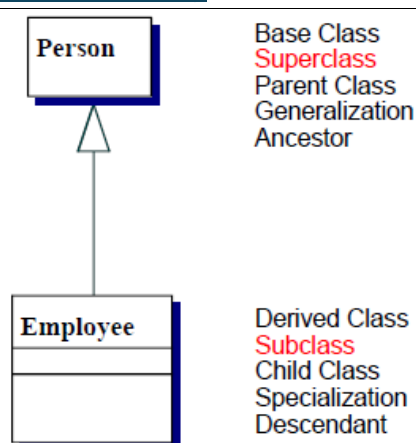
**Chú ý:**

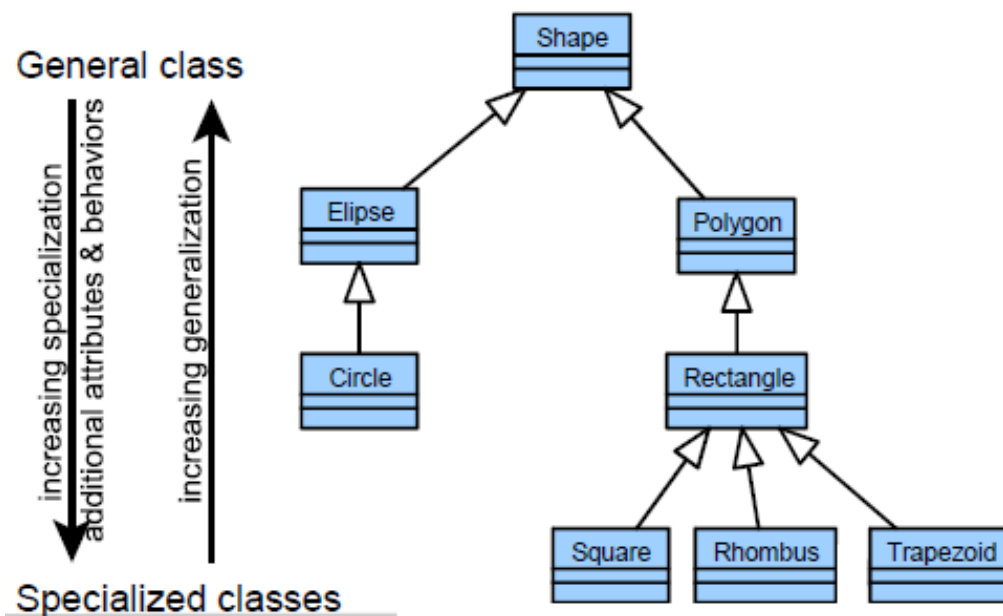
- ✎ **Hàm bạn không được kế thừa.** Tuy nhiên, nếu một hàm bạn được định nghĩa trong lớp cha, lớp con có thể định nghĩa lại hoặc sử dụng nó như là một hàm bạn riêng.
- ✎ **Hàm tạo (constructor) và hàm hủy (destructor)** không được kế thừa bởi lớp con. Tuy nhiên, lớp con có thể gọi trực tiếp hàm tạo của lớp cha trong danh sách khởi tạo của hàm tạo lớp con.

➤ Object của Derived class cũng chính là object của Base class

➤ Derived class có thể **định nghĩa lại các phương thức** của Base class

➤ **Ký hiệu hình vẽ:** Employee được dẫn xuất từ Person (mũi tên **tam giác trắng** từ Employee lên Person)





### - Xây dựng Derived class (lớp dẫn xuất)

- Cú pháp:

```
class SuperClass{
    //Thành phần của lớp cơ sở
};
class DerivedClass : <access-specifier> SuperClass{
    //Thành phần bổ sung của lớp dẫn xuất
};
```

**<access-specifier>:** kiểm soát truy xuất các thành viên được kế thừa

- **public:** có thể truy xuất từ bất cứ nơi nào.
- **private (default)** (nếu không viết gì vào vị trí <access-specifier>)
  - Là riêng tư của lớp đó
  - Chỉ có hàm thành phần của lớp và ngoại lệ các hàm bạn được phép truy xuất.
  - Các lớp con cũng không có quyền truy xuất
- **Protected:** Cho phép qui định một vài thành phần nào đó của lớp là bảo mật, theo nghĩa thế giới bên ngoài không được phép truy xuất, nhưng tất cả các lớp con, cháu... đều được phép truy xuất.

### Ví dụ:

```

1 //Point.h
2 #pragma once
3 #include <iostream>
4 using namespace std;
5 class Point
6 {
7 private:
8     float x;
9     float y;
10 public:
11     //constructor and destructor
12     Point(float x = 0, float y = 0);
13     ~Point(void);
14     //setters and getters
15     void setX(float);
16     float getX() const;
17     void setY(float);
18     float getY() const;
19     //print
20     friend ostream& operator<<(ostream &output, const Point &p );
21
22 };

```

```

1 //Point.cpp
2 #include "Point.h"
3 Point::Point( float x, float y )
4 {
5     setX(x);
6     setY(y);
7 }
8
9 void Point::setX(float x) { ... }
14 float Point::getX() const { ... }
19 void Point::setY(float y) { ... }
24 float Point::getY() const { ... }
29 ostream& operator<<(ostream &output, const Point &p )
30 {
31     output << "[" << p.x << ", " << p.y << "]";
32     return output;
33 }
34
35 Point::~~Point(void) { ... }

```

```

1 //circle.h
2 #pragma once
3 #include "Point.h"
4 class Circle: public Point //Lop Circle ke thua tu lop Point
5 {
6 //Lop Circle ke thua du lieu x,y tu lop Point
7 //va bo sung them du lieu radius
8 private:
9     float radius;
10 public:
11     Circle(float x = 0, float y = 0, float r = 0);
12     ~Circle(void);
13     void setR(float);
14     float getR() const;
15     float CalArea() const;
16 };

```

```

1 //Circle.cpp
2 #include "Circle.h"
3 Circle::Circle(float x, float y, float r):Point(x,y)
4 {
5     setR(r);
6 }
7 void Circle::setR(float r)
8 {
9     radius = ( r > 0 ? r : 0 );
10 }
11 float Circle::getR() const { ... }
16 float Circle::CalArea() const
17 {
18     return 3.14159*radius*radius;
19 }
20
21 Circle::~~Circle(void)
22 {
23 }

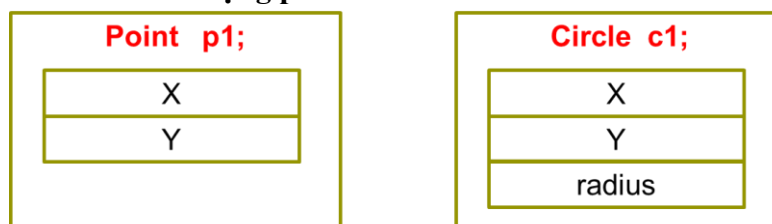
```

```

1 //main.cpp
2 #include "Point.h"
3 #include "Circle.h"
4 int main()
5 {
6     Point p1;
7     Circle c1;
8     // p1.CalArea(); invalid
9     cout<<p1<<endl;
10    cout<<c1<<endl; //goi operator<< thua ke tu lop Point
11    return 0;
12 }

```

→ Bộ nhớ được cấp phát cho 2 đối tượng p1 và c1 như sau:



## 5.4 Thành viên “Protected”

- Tương tự với cách mà người dùng sử dụng một lớp đối tượng, các lớp dẫn xuất có thể truy cập đến các thành viên **public** và không thể truy cập đến các thành viên **private** trong lớp cơ sở của nó. Tuy nhiên có những trường hợp mà ta muốn một vài thành viên của lớp cơ sở **có thể được truy cập ở lớp dẫn xuất** trong khi vẫn **giới hạn quyền truy cập từ người dùng**. Những thành viên như vậy sẽ được chỉ định là các thành viên **protected**.
  - Phạm vi truy xuất **protected** có thể được xem như là một sự kết hợp giữa **public** và **private**
  - Thành viên “private” không thể được truy xuất bên ngoài phạm vi lớp
  - **Derived class không thể truy xuất thành viên “private” của Base class**
- Sử dụng quyền truy xuất “Protected”
- Thành viên “Protected” của Base class có thể được truy xuất bởi:
- Thành viên và bạn của Base class
  - Thành viên và bạn của các lớp dẫn xuất từ Base class

```
5 class Point
6 {
7     private:
8         float x;
9         float y;
10    public:
11        //constructor and destructor
12        Point(float x = 0, float y = 0);
13        ~Point(void);
```

```
4 class Circle: public Point //Lop Circle ke thua tu lop Point
5 {
6     //Lop Circle ke thua du lieu x,y tu lop Point
7     //va bo sung them du lieu radius
8     private:
9         float radius;
10    public:
11        Circle(float x = 0, float y = 0, float r = 0);
12        ~Circle(void);
13        void setR(float);
14        float getR() const;
15        double CalArea() const;
16        friend ostream &operator<<( ostream &, const Circle & );
```

```
1 //Circle.cpp
2 #include "Circle.h"
3 Circle::Circle(float x, float y, float r) { ... }
7 void Circle::setR(float r) { ... }
11 float Circle::getR() const { ... }
16 double Circle::CalArea() const { ... }
21 ostream &operator<<( ostream &output, const Circle &c )
22 {
23     output<<"Center of circle: ["<<c.x<<" "<<c.y<<""]\\n";
24     output<<"Radius of circle: "<<c.radius<<"\\n";
25     return output;
26 }
27 Circle::~~Circle(void) { ... }
```

put

ow output from: Build

>Compiling...

>Circle.cpp

>h:\training\oop-lap trình huong doi tuong\bai giang\example\_inheritance\circle.cpp(23) : error C2248: 'Point::x' : cannot access private member declared in class 'Point'

### Cách khắc phục:

```
1 //Point.h
2 #pragma once
3 #include <iostream>
4 using namespace std;
5 class Point
6 {
7     protected:
8         float x;
9         float y;
10    public:
11        //constructor and destructor
12        Point(float x = 0, float y = 0);
13        ~Point(void);
14        //setters and getters
15        void setX(float);
16        float getX() const;
17        void setY(float);
18        float getY() const;
19        //print
20        friend ostream& operator<<(ostream &output, const Point &p );
```

#### ❑ Ưu điểm:

- Derived class có thể truy xuất trực tiếp các dữ liệu thành viên của Base class nếu các thành viên này có mức truy cập bảo vệ (protected) hoặc công khai (public)
- Giảm thiểu việc sử dụng get/set

#### ❑ Nhược điểm:

- Derived class có thể gây ra sự thay đổi không hợp lệ trên các dữ liệu thành viên của Base class

Giả sử chúng ta có một lớp cơ sở Base đại diện cho một đối tượng tài khoản ngân hàng, trong đó value là số dư tài khoản. Lớp dẫn xuất Derived được phép thay đổi số dư này. Tuy nhiên, nếu không có các biện pháp kiểm tra thích hợp, lớp dẫn xuất có thể vô tình hoặc cố ý đặt một số dư không hợp lệ (ví dụ như âm).

```
#include <iostream>

class Base {
protected:
    int balance; // Số dư tài khoản
public:
    Base(int initialBalance) : balance(initialBalance) {
        if (initialBalance < 0) {
            std::cout << "Initial balance cannot be negative!" << std::endl;
            balance = 0;
        }
    }

    void displayBalance() const {
        std::cout << "Current balance: " << balance << std::endl;
    }
};
```



```

class Derived : public Base {
public:
    Derived(int initialBalance) : Base(initialBalance) {}

    void changeBalance(int newBalance) {
        balance = newBalance; // Có thể đặt số dư không hợp lệ
    }
};

int main() {
    Derived account(100); // Tài khoản với số dư ban đầu là 100
    account.displayBalance(); // Hiển thị số dư: 100

    account.changeBalance(-500); // Đặt số dư không hợp lệ
    account.displayBalance(); // Hiển thị số dư: -500 (không hợp lệ)

    return 0;
}

```

**Lớp Base** đại diện cho một tài khoản ngân hàng với số dư ban đầu. Nó kiểm tra giá trị ban đầu để đảm bảo rằng không thể có số dư âm ngay từ khi tạo tài khoản.

**Lớp Derived** kế thừa từ Base và có thể thay đổi trực tiếp số dư bằng cách gán giá trị mới cho biến balance mà không có bất kỳ kiểm tra nào.

**Trong hàm main**, khi ta gọi changeBalance(-500), lớp dẫn xuất thay đổi số dư thành một giá trị không hợp lệ (âm). Điều này có thể gây ra những vấn đề nghiêm trọng trong một hệ thống tài chính thực tế.

➤ Phụ thuộc vào việc thực thi của lớp Base class

```

#include <iostream>

class Base {
protected:
    int fundValue; // Giá trị của quỹ
public:
    Base(int value) : fundValue(value) {}

    void setFundValue(int value) {
        fundValue = value;
    }

    int getFundValue() const {
        return fundValue;
    }
};

class Derived : public Base {
public:
    Derived(int value) : Base(value) {}

    int calculateInterest(int rate) {
        // Giả sử fundValue được sử dụng trực tiếp từ Base
        return fundValue * rate / 100;
    }
};

```

```

int main() {
    Derived investment(1000); // Giá trị quỹ ban đầu là 1000
    std::cout << "Interest: " << investment.calculateInterest(5) << std::endl; // In ra lãi
    suất: 50

    investment.setFundValue(2000); // Thay đổi giá trị quỹ
    std::cout << "Interest: " << investment.calculateInterest(5) << std::endl; // In ra lãi
    suất: 100

    return 0;
}

```

### Lớp Base:

- Lớp này đại diện cho một quỹ đầu tư với một thành viên dữ liệu fundValue, được bảo vệ (protected), đại diện cho giá trị của quỹ.
- Nó cũng cung cấp các phương thức để đặt (setFundValue) và lấy (getFundValue) giá trị của quỹ.

### Lớp Derived:

- Lớp này kế thừa từ Base và có một phương thức calculateInterest(int rate) để tính toán lãi suất dựa trên giá trị của quỹ. Nó truy cập trực tiếp vào thành viên dữ liệu fundValue từ lớp Base.

Nếu lớp Base thay đổi cách lưu trữ hoặc xử lý giá trị quỹ (fundValue), lớp Derived sẽ gặp vấn đề. Chẳng hạn, giả sử Base thay đổi cách tính giá trị quỹ dựa trên nhiều yếu tố khác nhau, thay vì chỉ lưu trữ trực tiếp giá trị fundValue:

```

class Base {
protected:
    int fundValue;
    int riskFactor; // Hệ số rủi ro ảnh hưởng đến giá trị quỹ
public:
    Base(int value, int risk) : fundValue(value), riskFactor(risk) {}

    void setRiskFactor(int risk) {
        riskFactor = risk;
    }

    int getFundValue() const {
        // Giả sử giá trị quỹ bây giờ phụ thuộc vào một hệ số rủi ro
        return fundValue - (fundValue * riskFactor / 100);
    }
};

```

Trong ví dụ này, nếu lớp Derived vẫn tiếp tục truy cập trực tiếp vào fundValue để tính toán lãi suất mà không sử dụng phương thức getFundValue() mới, thì kết quả sẽ không còn chính xác, vì giá trị thực sự của quỹ đã thay đổi dựa trên hệ số rủi ro (riskFactor). Điều này dẫn đến lớp Derived phải thay đổi cho phù hợp:

```

#include <iostream>

class Base {
protected:
    int fundValue;
    int riskFactor; // Hệ số rủi ro ảnh hưởng đến giá trị quỹ
public:
    Base(int value, int risk) : fundValue(value), riskFactor(risk) {}

    void setRiskFactor(int risk) {
        riskFactor = risk;
    }

    int getFundValue() const {
        // Giá trị quỹ bây giờ phụ thuộc vào một hệ số rủi ro
        return fundValue - (fundValue * riskFactor / 100);
    }
};

class Derived : public Base {
public:
    Derived(int value, int risk) : Base(value, risk) {}

    int calculateInterest(int rate) {
        // Thay vì truy cập trực tiếp vào fundValue, ta sử dụng getFundValue()
        return getFundValue() * rate / 100;
    }
};

int main() {
    Derived investment(1000, 10); // Giá trị quỹ ban đầu là 1000 với hệ số rủi ro là 10%
    std::cout << "Interest: " << investment.calculateInterest(5) << std::endl; // In ra lãi
    suất dựa trên giá trị quỹ thực tế

    investment.setRiskFactor(20); // Thay đổi hệ số rủi ro
    std::cout << "Interest: " << investment.calculateInterest(5) << std::endl; // In ra lãi
    suất dựa trên giá trị quỹ mới

    return 0;
}

```

## 5.5 Các kiểu kế thừa

### ❑ Dựa trên quyền truy xuất

#### ➤ public Inheritance

```

class MyDerived : public MyBase{
    /* Lớp MyDerived sẽ kế thừa các thành viên “protected” và “public” của lớp
    MyBase */
};

```

#### ➤ protected Inheritance

```

class MyDerived : protected MyBase{
    /*Lớp MyDerived sẽ kế thừa các thành viên “protected” và “public” của lớp
    MyBase */
};

```

➤ private Inheritance

```
class MyDerived : private MyBase{
/*Lớp MyDerived sẽ kế thừa các thành viên “protected” và “public” của lớp
MyBase */
};
```

Ta có bảng quy tắc kế thừa như sau:

Phạm vi truy xuất được chỉ định ở lớp cơ sở:	Từ khóa dẫn xuất khi khai báo lớp con:	Phạm vi truy xuất ở lớp con được chuyển thành:
Public	public	Public
Protected		protected
Private		Không truy cập được
Public	protected	protected
Protected		protected
Private		Không truy cập được
Public	private	private
Protected		private
Private		Không truy cập được

Từ bảng có thể rút ra được các thông tin:

- Thành phần **private** ở lớp cha thì không truy xuất được ở lớp con
- Lớp con kế thừa kiểu **public** từ lớp cha thì các thành phần **protected** của lớp cha trở thành **protected** của lớp con, các thành phần **public** của lớp cha trở thành **public** của lớp con.
- Lớp con kế thừa kiểu **private** từ lớp cha thì các thành phần **protected** và **public** của lớp cha trở thành **private** của lớp con.
- Lớp con kế thừa kiểu **protected** từ lớp cha thì các thành phần **protected** và **public** của lớp cha trở thành **protected** của lớp con.

Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
Public	<b>public</b> in derived class. Can be accessed directly by any non- <b>static</b> member functions, <b>friend</b> functions and non-member functions.	<b>protected</b> in derived class. Can be accessed directly by all non- <b>static</b> member functions and <b>friend</b> functions.	<b>private</b> in derived class. Can be accessed directly by all non- <b>static</b> member functions and <b>friend</b> functions.
Protected	<b>protected</b> in derived class. Can be accessed directly by all non- <b>static</b> member functions and <b>friend</b> functions.	<b>protected</b> in derived class. Can be accessed directly by all non- <b>static</b> member functions and <b>friend</b> functions.	<b>private</b> in derived class. Can be accessed directly by all non- <b>static</b> member functions and <b>friend</b> functions.
Private	Hidden in derived class. Can be accessed by non- <b>static</b> member functions and <b>friend</b> functions through <b>public</b> or <b>protected</b> member functions of the base class.	Hidden in derived class. Can be accessed by non- <b>static</b> member functions and <b>friend</b> functions through <b>public</b> or <b>protected</b> member functions of the base class.	Hidden in derived class. Can be accessed by non- <b>static</b> member functions and <b>friend</b> functions through <b>public</b> or <b>protected</b> member functions of the base class.

- Mặc dù một đối tượng của lớp con được thừa hưởng các thuộc tính từ lớp cha, nó **không nên trực tiếp khởi tạo dữ liệu** cho các thuộc tính đó (trong vài trường hợp thì là không thể).

- Giả sử có ví dụ:

```
class Quote {
private:
    string bookNo; // mã số sách

protected:
    double price; // giá của một quyển sách

public:
    // phương thức thiết lập:
    Quote();
    Quote(const string& book, double salesPrice);

    // phương thức truy vấn:
    string getBookNo() { return bookNo; }
};

// lớp BulkQuote kế thừa từ lớp Quote
class BulkQuote : public Quote {
private:
    double discount; // tỉ lệ phần trăm được giảm
    int minQty; // số lượng mua tối thiểu để được áp mã

public:
    BulkQuote();
    BulkQuote(const string& book, double salesPrice, double disc, int cnt);
};
```

- Lớp **BulkQuote** kế thừa 2 thuộc tính là **bookNo** và **price** từ **Quote**, tuy nhiên **bookNo** là thành phần **private** của **Quote**, kể cả lớp con cũng không truy cập được thuộc tính này, dẫn đến việc **BulkQuote** không thể trực tiếp gán dữ liệu cho **bookNo**.
- Cho dù các thuộc tính này đều là **protected** và lớp con có thể truy cập được đi chăng nữa, lớp con cũng không nên khởi tạo dữ liệu trực tiếp cho chúng (bởi vì lớp cha có cung cấp các giao diện để tương tác với các thuộc tính của nó, lớp con nên tôn trọng và sử dụng giao diện này).
- Tóm lại, lớp con nên **sử dụng phương thức thiết lập của lớp cha** để khởi tạo dữ liệu cho các thuộc tính chung mà nó được thừa kế. Cách sử dụng như sau (lấy ví dụ là phương thức thiết lập cho lớp **BulkQuote**):

```
//Phương thức thiết lập của Quote
Quote::Quote(string id, double price)
    : bookNo(id), price(price) {}

//Phương thức thiết lập của Bulk_quote
BulkQuote::BulkQuote(string id, double price, double disc, int n)
    : Quote(id, price), discount(disc), minQty(n) {}
```

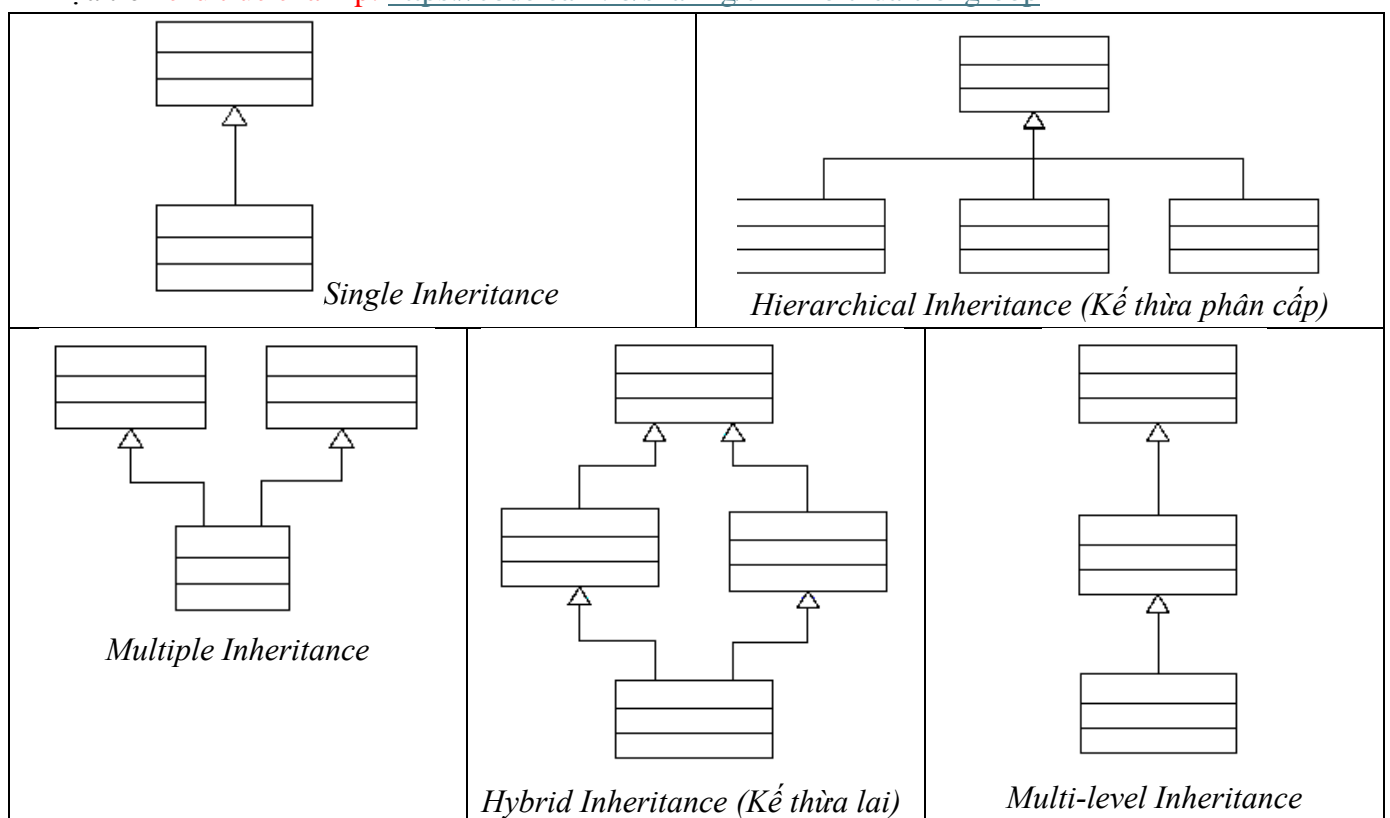
- Một điểm mới trong ví dụ trên chính là việc sử dụng **danh sách khởi tạo** trong phương thức thiết lập (**constructor initializer list**)

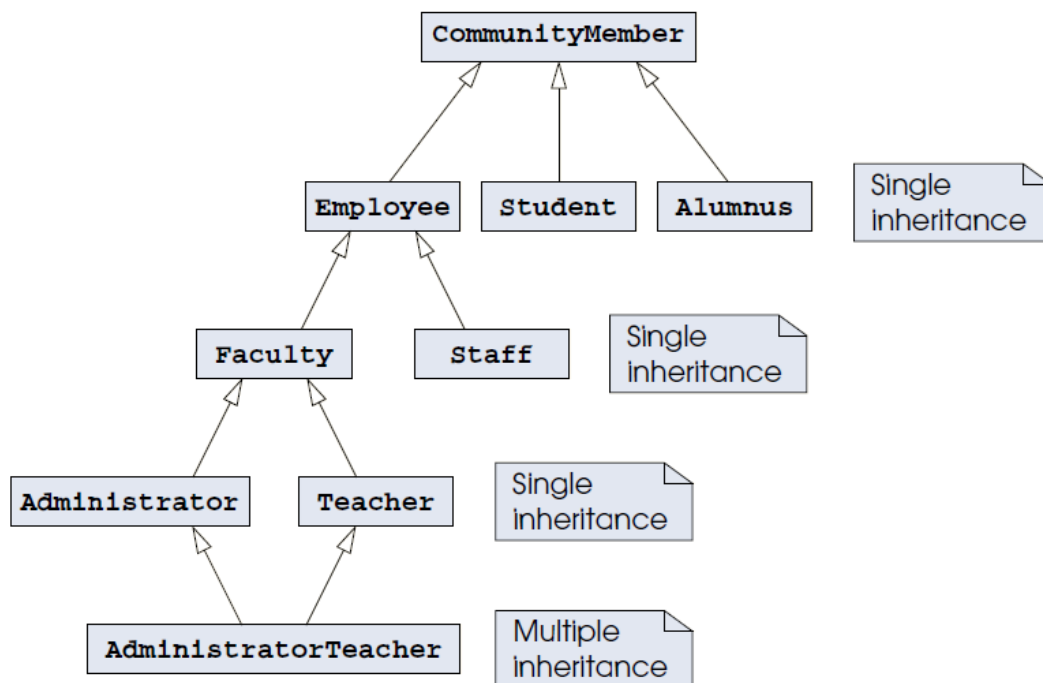
- Với cách định nghĩa như trên, khi một phương thức thiết lập của **BulkQuote** được gọi với 4 đối số đầu vào:

```
BulkQuote derived("893-523-52-2211-3", 150000, 0.2, 3);
```

2 đối số đầu tiên sẽ được dùng để gọi phương thức thiết lập của **Quote**, sau khi phương thức này thực hiện xong nhiệm vụ và các thuộc tính chung đã được khởi tạo, hai đối số còn lại sẽ lần lượt được dùng để khởi tạo cho giá trị cho 2 thuộc tính riêng của **BulkQuote**.

❑ Dựa trên **cấu trúc của lớp**: <https://codelearn.io/sharing/tinh-ke-thua-trong-oop>





- ❑ **Single Inheritance - Đơn kế thừa:** một lớp chỉ được kế thừa từ đúng một lớp khác. Hay nói cách khác, lớp con chỉ có duy nhất một lớp cha.

**Cú pháp khai báo đơn kế thừa:**

```

class lopcon : phạmviđulieu lopcha
{
    // nội dung lớp con
};
  
```

**Ví dụ:**

```

#include <iostream>
using namespace std;

// Lớp cha
class Mayvitinh
{
public:
    Mayvitinh()
    {
        cout << "This is a computer" << endl;
    }
};

// Lớp con kế thừa từ lớp cha
class mayAcer : public Mayvitinh
{
};

// main function
int main()
{
    mayAcer may1;
    return 0;
}
  
```

**Kết quả:** This is a computer

- ❑ **Multiple Inheritance - Đa kế thừa:** một lớp có thể kế thừa từ nhiều hơn một lớp khác. Nghĩa là một lớp con được kế thừa từ nhiều hơn một lớp cơ sở.

**Cú pháp khai báo đa kế thừa:**

```
class lopcon : phamvitruycap lopcha1, phamvitruycap lopcha2, ....
{
    // nội dung của lớp con
};
```

Ở đây, các lớp cơ sở sẽ được *phân tách bằng dấu phẩy*, và phạm vi truy cập cho mọi lớp cơ sở phải được chỉ định.

### Ví dụ 1:

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

// Lớp cơ sở thứ nhất
class Mayvitinh
{
public:
    Mayvitinh()
    {
        cout << "This is a computer's brand" << endl;
    }
};

// Lớp cơ sở thứ hai
class Maylaptop
{
public:
    Maylaptop()
    {
        cout << "This is a laptop's brand" << endl;
    }
};

// Lớp con kế thừa từ 2 lớp cha
class mayAcer : public Mayvitinh, public Maylaptop
{
};

// main function
int main()
{
    mayAcer may1;
    return 0;
}
```

### **Kết quả:**

```
This is a computer's brand
This is a laptop's brand
```



## Ví dụ 2:

```
1 //Date.h
2 #pragma once
3 class Date
4 {
5     protected:
6         int day;
7         int month;
8         int year;
9     public:
10         Date(void);
11         Date(int d, int m, int y);
12         ~Date(void);
13         int getDay() const;
14         void setDay(int d);
15         int getMonth() const;
16         void setMonth(int m);
17         int getYear() const;
18         void setYear(int y);
19 };
```

```
1 //Time.h
2 #pragma once
3 class Time
4 {
5     protected:
6         int hour;
7         int min;
8         int sec;
9     public:
10         Time();
11         Time(int h, int m, int s);
12         ~Time(void);
13         int getHour() const;
14         void setHour(int h);
15         int getMin() const;
16         void setMin(int m);
17         int getSecond() const;
18         void setSecond(int s);
19 };
```

```
1 //Date.cpp
2 #include "Date.h"
3 Date::Date(void)
4 {
5     day = 1;
6     month = 1;
7     year = 1900;
8 }
9 Date::Date(int d, int m, int y)
10 {
11     setDay(d);
12     setMonth(m);
13     setYear(y);
14 }
15 int Date::getDay() const
16 { return day; }
17
18 int Date::getMonth() const
19 { return month; }
20
21 int Date::getYear() const
22 { return year; }
23 void Date::setDay(int d) { ... }
24 void Date::setMonth(int m) { ... }
25 void Date::setYear(int y) { ... }
26 Date::~Date(void)
```

```
1 //Time.cpp
2 #include "Time.h"
3 Time::Time(void)
4 {
5     hour = 0;
6     min = 0;
7     sec = 0;
8 }
9 Time::Time(int h, int m, int s)
10 {
11     setHour(h);
12     setMin(m);
13     setSecond(s);
14 }
15 int Time::getHour() const { ... }
16 int Time::getMin() const { ... }
17 int Time::getSecond() const { ... }
18 void Time::setHour(int h) { ... }
19 void Time::setMin(int m) { ... }
20 void Time::setSecond(int s) { ... }
21 Time::~Time(void) { ... }
```

```

1 //Datetime.h
2 #pragma once
3 #include <iostream>
4 #include "Date.h"
5 #include "Time.h"
6 using namespace std;
7 const int SIZE = 20;
8 class DateTime: public Date, public Time
9 {
10 private:
11     char dateTimeString[SIZE];
12 public:
13     DateTime(void);
14     DateTime(int, int, int, int, int, int);
15     const char* getDateTime() const;
16     ~DateTime(void);
17     friend ostream& operator <<(ostream &, const DateTime &);
18 };
19
20 //Datetime.cpp
21 #include "DateTime.h"
22 DateTime::DateTime(): Date(), Time()
23 {
24     strcpy(dateTimeString, "1/1/1900 0:0:0");
25 }
26 DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc):
27     Date(dy, mon, yr), Time(hr, mt, sc)
28 {
29     char temp[SIZE/2];
30     //Dinh dang MM/DD/YY
31     strcpy(dateTimeString, itoa(day, temp, SIZE/2));
32     strcat(dateTimeString, "/");
33     strcat(dateTimeString, itoa(month, temp, SIZE/2));
34     strcat(dateTimeString, "/");
35     strcat(dateTimeString, itoa(year, temp, SIZE/2));
36     strcat(dateTimeString, " ");
37     // Dinh dang HH:MM:SS
38     strcat(dateTimeString, itoa(hour, temp, SIZE/2));
39     strcat(dateTimeString, ":");
40     strcat(dateTimeString, itoa(min, temp, SIZE/2));
41     strcat(dateTimeString, ":");
42     strcat(dateTimeString, itoa(sec, temp, SIZE/2));
43 }

```

## Một số vấn đề với đa kế thừa

### ● Trùng tên

- Những phương thức trùng nhau của các lớp cơ sở
- Những dữ liệu trùng nhau của các lớp cơ sở

```

1 #pragma once
2 class Student
3 {
4 protected:
5     int id;
6 public:
7     Student(void);
8     int getID() const;
9     ~Student(void);
10 };
11
12 #pragma once
13 class Employee
14 {
15     class Employee
16 protected:
17     int id;
18 public:
19     Employee(void);
20     int getID() const;
21     ~Employee(void);
22 };

```

```

1 #pragma once
2 #include "Student.h"
3 #include "Employee.h"
4 class TeachingAssistant: public Student, public Employee
5 {
6 public:
7     TeachingAssistant(void);
8     ~TeachingAssistant(void);
9     void changeID(int _id);
10 };

```

```

1 #include "TeachingAssistant.h"
2
3 TeachingAssistant::TeachingAssistant(void): Student(), Employee()
4 {
5 }
6
7 void TeachingAssistant::changeID(int _id)
8 {
9     id = _id;
10 }

```

error C2385: ambiguous access of 'id'  
could be the 'id' in base 'Student'  
or could be the 'id' in base 'Employee'

```

3 //main.cpp
4 #include <iostream>
5 #include "TeachingAssistant.h"
6 using namespace std;
7 int main()
8 {
9     TeachingAssistant *mark = new TeachingAssistant();
10     cout<<mark->getID();
11     return 0;
12 }

```

### Giải pháp: Sử dụng toán tử phạm vi ::

```

void TeachingAssistant::changeID(int _id){
    Employee::id = _id;
}
int main()
{
    TeachingAssistant *mark = new TeachingAssistant();
    cout<<mark->Employee::getID();
    return 0;
}

```

- Một lớp được thừa kế nhiều lần (xung đột kế thừa - Multiple Inheritance Conflict)

#### ➤ Lớp D thừa kế lớp A hai lần

```

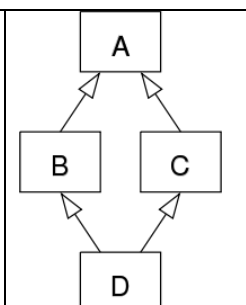
class Person
{..};

class Employee : public Person
{..};

class Student: public Person
{..};

class TeachingAssistant: public Student, public Employee
{..};

```



- TeachingAssistant kế thừa từ cả Student và Employee.
- Cả Student và Employee đều kế thừa từ Person.

Điều này dẫn đến một vấn đề gọi là “**diamond problem**”:

**Nhiều bản sao của lớp cơ sở:**

- Khi TeachingAssistant kế thừa từ cả Student và Employee, nó sẽ **nhận được hai bản sao của lớp Person** – một từ Student và một từ Employee. Điều này có thể dẫn đến sự mâu thuẫn khi truy cập các thành viên của Person.
- Ví dụ, nếu Person có một biến thành viên name, **TeachingAssistant sẽ có hai biến name, một từ Student::Person và một từ Employee::Person.**

**Gây mơ hồ (Ambiguity):**

- Khi bạn cố gắng truy cập một thành viên của Person từ đối tượng của TeachingAssistant, trình biên dịch sẽ không biết bạn muốn truy cập thành viên từ bản sao nào của Person, dẫn đến lỗi mơ hồ.

**Giải pháp: Thừa kế ảo (virtual inheritance)**

- Tất cả các thành phần của lớp A chỉ tổ hợp một lần duy nhất trong lớp D.
- Khi một lớp dẫn xuất kế thừa ảo từ lớp cơ sở, chỉ một bản sao duy nhất của lớp cơ sở sẽ tồn tại trong hệ thống phân cấp kế thừa.

```
class Employee : public virtual Person
{..};
class Student: public virtual Person
{..};
class TeachingAssistant: public Student, public Employee
{..};
```

TeachingAssistant sẽ chỉ có một bản sao của Person, và sự mơ hồ sẽ không còn xảy ra.

#### \* Kế thừa ảo

- Cơ chế giải quyết vấn đề "diamond problem" trong trường hợp kế thừa nhiều lớp.
- Khi một lớp dẫn xuất kế thừa từ nhiều lớp cơ sở mà các lớp cơ sở này đều kế thừa từ cùng một lớp tổ tiên, việc sử dụng kế thừa ảo đảm bảo rằng chỉ có một bản sao của lớp tổ tiên được sử dụng trong hệ thống phân cấp kế thừa.

**Khi nào cần sử dụng kế thừa ảo?** Khi bạn có một cấu trúc kế thừa mà một lớp cơ sở bị kế thừa nhiều lần thông qua các lớp trung gian, và bạn muốn tránh việc có nhiều bản sao của lớp cơ sở đó trong lớp dẫn xuất cuối cùng.

**Cách khai báo kế thừa ảo:** Thêm từ khóa `virtual` trước tên lớp cơ sở trong danh sách kế thừa

```
class Base { /* ... */ };

class Derived1 : public virtual Base { /* ... */ };

class Derived2 : public virtual Base { /* ... */ };

class Final : public Derived1, public Derived2 { /* ... */ };
```

- Derived1 và Derived2 kế thừa ảo từ Base, do đó, Final sẽ chỉ có một bản sao duy nhất của Base.

**Thứ tự gọi hàm tạo và hàm hủy với kế thừa ảo**

- **Hàm tạo:** Trong trường hợp kế thừa ảo, hàm tạo của lớp cơ sở ảo sẽ được gọi bởi lớp dẫn xuất cuối cùng, không phải bởi các lớp dẫn xuất trung gian.

- **Hàm hủy:** Hàm hủy của lớp cơ sở ảo sẽ được gọi sau hàm hủy của các lớp dẫn xuất trung gian, tương tự như thứ tự thực thi của hàm hủy trong kế thừa thông thường.

```
#include <iostream>

class Person {
public:
    Person() {
        std::cout << "Person constructor called" << std::endl;
    }
    ~Person() {
        std::cout << "Person destructor called" << std::endl;
    }
};

class Employee : public virtual Person {
public:
    Employee() {
        std::cout << "Employee constructor called" << std::endl;
    }
    ~Employee() {
        std::cout << "Employee destructor called" << std::endl;
    }
};

class Student : public virtual Person {
public:
    Student() {
        std::cout << "Student constructor called" << std::endl;
    }
    ~Student() {
        std::cout << "Student destructor called" << std::endl;
    }
};

class TeachingAssistant : public Employee, public Student {
public:
    TeachingAssistant() {
        std::cout << "TeachingAssistant constructor called" << std::endl;
    }
    ~TeachingAssistant() {
        std::cout << "TeachingAssistant destructor called" << std::endl;
    }
};

int main() {
    TeachingAssistant ta;
    return 0;
}
```

**Kết quả:**

```
Person constructor called
Employee constructor called
Student constructor called
TeachingAssistant constructor called
TeachingAssistant destructor called
Student destructor called
Employee destructor called
Person destructor called
```

- Khi tạo đối tượng TeachingAssistant, hàm tạo của Person chỉ được gọi **một lần** dù Person là lớp cơ sở của cả Employee và Student.

- Thứ tự gọi hàm tạo và hàm hủy được xác định dựa trên kế thừa ảo: hàm tạo `Person` được gọi trước, và hàm hủy `Person` được gọi sau cùng.

#### Ưu điểm:

- Giải quyết vấn đề xung đột kế thừa:** Tránh sự mơ hồ và xung đột khi truy cập thành viên của lớp cơ sở.
- Tăng tính linh hoạt:** Giúp quản lý tốt hơn các hệ thống phân cấp kế thừa phức tạp, đặc biệt khi lớp cơ sở có dữ liệu cần được dùng chung giữa các lớp dẫn xuất.

#### Nhược điểm:

- Phức tạp hơn:** Cấu trúc kế thừa trở nên phức tạp hơn, và việc quản lý các hàm tạo, hàm hủy cũng trở nên khó khăn hơn.
- Hiệu suất:** Kế thừa ảo có thể gây ra một chút overhead trong việc quản lý các con trỏ ảo (virtual pointer) và gọi các hàm ảo, nhưng sự khác biệt này thường không đáng kể.

- ❑ **Multilevel Inheritance - Kế thừa đa cấp:** một lớp dẫn xuất được tạo từ một lớp dẫn xuất khác.

#### Ví dụ:

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

// Lớp cha
class Mayvitinh
{
public:
    Mayvitinh()
    {
        cout << "This is a computer's brand" << endl;
    }
};

// Lớp con kế thừa từ lớp cha
class Maylaptop : public Mayvitinh
{
public:
    Maylaptop()
    {
        cout << "This is a laptop's brand" << endl;
    }
};

// Lớp con kế thừa từ lớp cha thứ 2
class mayAcer : public Maylaptop
{
public:
    mayAcer() {
        cout << "This brand is Acer" << endl;
    }
};

// main function
int main()
{
    mayAcer may1;
    return 0;
}
```

#### Kết quả:

```
This is a computer's brand
This is a laptop's brand
This brand is Acer
```

- ❑ **Hierarchical Inheritance - Kế thừa phân cấp:** sẽ có nhiều hơn một lớp con được kế thừa từ một lớp cha duy nhất.

**Ví dụ:**

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

// Lớp cha
class Mayvitinh
{
public:
    Mayvitinh()
    {
        cout << "This is a computer's brand" << endl;
    }
};

// Lớp con thứ nhất
class mayAsus : public Mayvitinh
{
};

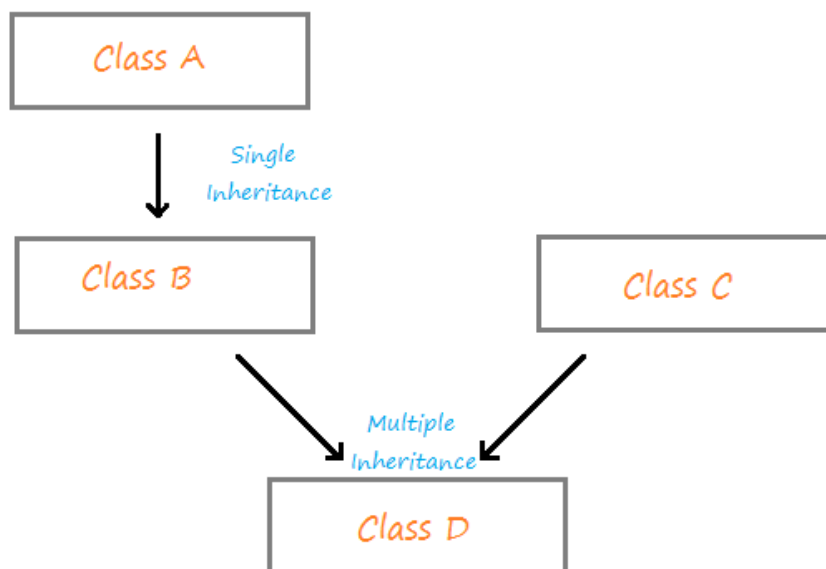
// Lớp con thứ hai
class mayAcer : public Mayvitinh
{
};

// main function
int main()
{
    mayAcer may1;
    mayAsus may2;
    return 0;
}
```

**Kết quả:**

This is a computer's brand  
This is a computer's brand

- ❑ **Hybrid Inheritance - Kế thừa lai:** kết hợp nhiều hơn một loại thừa kế.



Hybrid Inheritance

### Ví dụ:

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

// Lớp cha
class Mayvitinh
{
public:
    Mayvitinh()
    {
        cout << "This is a computer's brand" << endl;
    }
};

// Lớp cha
class Maylaptop
{
public:
    Maylaptop()
    {
        cout << "This is a laptop's brand" << endl;
    }
};

// Lớp con thứ nhất
class mayAcer : public Mayvitinh
{
};

// Lớp con thứ hai
class mayAsus : public Mayvitinh, public Maylaptop
{
};

// main function
int main()
{
    mayAsus may1;
    mayAcer may2;
    return 0;
}
```

### **Kết quả:**

```
This is a computer's brand
This is a laptop's brand
This is a computer's brand
```

**Lớp mayAsus** kế thừa từ cả hai lớp cha là Mayvitinh và Maylaptop. Đây là ví dụ điển hình của *đa kế thừa*, nơi một lớp con kế thừa từ nhiều hơn một lớp cha.

**Lớp mayAcer** chỉ kế thừa từ một lớp cha là Mayvitinh, đây là ví dụ của *đơn kế thừa*.

Khi tạo đối tượng mayAsus, cả hai constructor của Mayvitinh và Maylaptop sẽ được gọi, và bạn sẽ thấy cả hai thông báo "This is a computer's brand" và "This is a laptop's brand" xuất hiện. Ngược lại, khi tạo đối tượng mayAcer, chỉ có constructor của Mayvitinh được gọi.



## 5.6 Hàm tạo và hàm hủy

### □ Hàm tạo

- **Không được kế thừa** nhưng có thể được gọi trong lớp con
- Khi tạo ra object của Derived class, thứ tự thực thi hàm tạo là:

**Hàm tạo của Base class → Hàm tạo của Derived class**

Điều này đảm bảo rằng tất cả các thành viên của lớp cơ sở được khởi tạo trước khi khởi tạo các thành viên của lớp dẫn xuất.

- Cách gọi hàm tạo của Base class trong Derived class:
  - Ngầm định (hàm tạo mặc định được gọi): Nếu bạn không chỉ định rõ ràng việc gọi hàm tạo của lớp cơ sở trong hàm tạo của lớp dẫn xuất, C++ sẽ tự động gọi hàm tạo mặc định (nếu có) của lớp cơ sở.
  - Tường minh

➤ **Cách gọi hàm tạo của Base class tường minh:**

```
<Hàm tạo lớp dẫn xuất>(tham số) : <Hàm tạo lớp cơ sở>(đối số)
    //gọi hàm tạo của lớp cơ sở
{
    ...
}
```

### Ví dụ:

```
class Base {
public:
    Base(int x) {
        // code
    }
};

class Derived : public Base {
public:
    Derived(int x, int y) : Base(x) {
        // code
    }
};
```

- **Multiple Inheritance:** hàm tạo lớp cha được thực thi theo trình tự được khai báo trong lớp con

```
class Base1 {
public:
    Base1() {
        // code
    }
};

class Base2 {
public:
    Base2() {
        // code
    }
};

class Derived : public Base1, public Base2 {
public:
    Derived() : Base1(), Base2() {
        // code
    }
};
```

Trong ví dụ trên, hàm tạo Base1 sẽ được gọi trước, sau đó đến hàm tạo Base2, và cuối cùng là hàm tạo của Derived.

- **Multilevel inheritance:** hàm tạo được thực thi theo trình tự kế thừa (Khi một lớp dẫn xuất kế thừa từ một lớp trung gian, mà lớp trung gian này lại kế thừa từ một lớp khác, hàm tạo sẽ được thực thi theo thứ tự kế thừa từ lớp cơ sở cao nhất đến lớp dẫn xuất cuối cùng)

```
class Base {
public:
    Base() {
        // code
    }
};

class Intermediate : public Base {
public:
    Intermediate() : Base() {
        // code
    }
};

class Derived : public Intermediate {
public:
    Derived() : Intermediate() {
        // code
    }
};
```

Trong ví dụ này, thứ tự gọi hàm tạo sẽ là: **Base → Intermediate → Derived**.

## ❑ Hàm hủy

- **Không được kế thừa** nhưng có thể được gọi trong lớp con. Tuy nhiên, hàm hủy của lớp cơ sở sẽ được tự động gọi khi đối tượng của lớp dẫn xuất bị hủy. Vì vậy, *lớp con không cần và cũng không được thực hiện các thao tác dọn dẹp cho các thành phần thuộc lớp cha*.
- Được thực thi theo trình tự **ngược lại với hàm tạo**:

**Hàm hủy của Derived class → Hàm hủy của Base class**

Điều này đảm bảo rằng các thành viên của lớp dẫn xuất được giải phóng trước, sau đó đến các thành viên của lớp cơ sở.

**Ví dụ:**

```
#include <iostream>

class Base {
public:
    Base() {
        std::cout << "Base Constructor\n";
    }
    ~Base() {
        std::cout << "Base Destructor\n";
    }
};

class Derived : public Base {
public:
    Derived() {
        std::cout << "Derived Constructor\n";
    }
    ~Derived() {
        std::cout << "Derived Destructor\n";
    }
};
```

```
int main() {
    Derived d;
    return 0;
}
```

**Kết quả:**

Base Constructor

Derived Constructor

Derived Destructor

Base Destructor

Trong ví dụ trên:

- Khi Derived d được tạo, hàm tạo của Base sẽ được gọi trước, sau đó đến hàm tạo của Derived.
- Khi đối tượng d bị hủy, hàm hủy của Derived sẽ được gọi trước, sau đó đến hàm hủy của Base.

**Ví dụ:**

```
2 | #include "Circle.h"
3 | Circle::Circle(float x, float y, float r):Point(x,y)
4 | {
5 |     cout<<"Constructor of Circle class is called\n";
6 |     setR(r);
7 | }
8 | void Circle::setR(float r) { ... }
12 | float Circle::getR() const { ... }
17 | double Circle::CalArea() const { ... }
22 | ostream &operator<<( ostream &output, const Circle &c ) { ... }
28 | Circle::~Circle(void)
29 | {
30 |     cout<<"Destructor of Circle class is called\n";
31 | }
1 | //main.cpp
2 | #include "Point.h"
3 | #include "Circle.h"
4 | int main()
5 | {
6 |     Point p1;
7 |     Circle c1;
8 |     // p1.CalArea(); invalid
9 |     cout<<p1<<endl;
10 |    cout<<c1<<endl; //goi operator<< thua ke tu lop Point
11 |    return 0;
12 | }
```

**Kết quả:**

Constructor of Point class is called

Constructor of Point class is called

Constructor of Circle class is called

[0, 0]

Center of circle: [3,4]

Radius of circle: 0

Destructor of Circle class is called

Destructor of Point class is called

Destructor of Point class is called

## 5.7 Định nghĩa lại phương thức (Method Overriding)

- Định nghĩa lại phương thức của Base class trong Derived class
- Xảy ra khi phương thức của Base class *không còn phù hợp* với Derived class
- **Khác với định nghĩa chồng hàm**

<pre>class Point { protected:     float x;     float y; public:     void Draw()     {         cout&lt;&lt;"Draw a Point: ["&lt;&lt;x&lt;&lt;" "&lt;&lt;y&lt;&lt;"]\n";     } }</pre>	<pre>int main() {     Point p1;     Circle c1(4,5,1);     p1.Draw();     c1.Draw(); }</pre>
<pre>class Circle: public Point //Lop Circle kế thừa tu lop Point { private:     float radius; public:     void Draw() //method overriding     {         cout&lt;&lt;"Draw a Circle with center: ["&lt;&lt;x&lt;&lt;" "&lt;&lt;y;         cout&lt;&lt;"] and radius: "&lt;&lt;radius&lt;&lt;"\n";     } }</pre>	

## 5.8 Phép gán và con trỏ trong kế thừa

- Thông thường, một biến con trỏ chỉ có thể giữ địa chỉ của các đối tượng có cùng kiểu với nó, tuy nhiên trong kế thừa có một ngoại lệ: **một con trỏ đối tượng thuộc lớp cơ sở có thể giữ địa chỉ của một đối tượng thuộc lớp dẫn xuất**. Dẫu vậy, một con trỏ đối tượng thuộc lớp dẫn xuất lại *không thể* giữ địa chỉ của một đối tượng thuộc lớp cơ sở, như ví dụ dưới đây

```
Quote base;
BulkQuote derived;
Quote* basePtr = &derived; // Đúng
BulkQuote* derivedPtr = &base; // Sai
```

- Việc có thể gán địa chỉ của một đối tượng thuộc lớp dẫn xuất cho một biến con trỏ đối tượng thuộc lớp cơ sở dẫn đến một kết quả quan trọng: Khi sử dụng một con trỏ đối tượng thuộc lớp cơ sở, chúng ta **không biết chắc** được *đối tượng mà con trỏ đó sẽ giữ có kiểu dữ liệu gì*, nó có thể là *một đối tượng thuộc lớp cơ sở hoặc lớp dẫn xuất*. Điều này góp phần tạo nên tính đa hình trong OOP (Chương sau).

### - Phép gán trực tiếp giữa các đối tượng

+ Ta cũng có thể **gán trực tiếp một đối tượng thuộc lớp con cho một đối tượng thuộc lớp cha**. Ngược lại, không thể gán một đối tượng của lớp cha cho một đối tượng thuộc lớp con.

+ Tuy nhiên có một vài điểm cần lưu ý: Khi dùng một biến có kiểu lớp con khởi tạo cho một đối tượng thuộc lớp cha, chương trình sẽ chỉ **sao chép những thuộc tính chung** giữa 2 lớp mà **không sao chép các thuộc tính riêng** của lớp con. Đơn giản là vì lớp cha thì không thể biết được các lớp con của nó có các thuộc tính mới nào, nó chỉ biết được những thuộc tính đã được định nghĩa sẵn bên trong mình mà thôi.

### Ví dụ:

```
Quote item1;
BulkQuote item2("13hd", 50000, 0.2, 3);
item1 = item2;
```

Sau dòng lệnh trên, *item1* lúc này chỉ chứa 2 thuộc tính là *bookNo = 13hd* và *price = 50000*, 2 thuộc tính *discount = 0.2* và *minQty = 3* trong *item2* đã bị lược bỏ.