VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Kỹ Thuật Lập Trình (Cơ bản và nâng cao C++)

KTLT2 - HK242

TASK 8 Kiến trúc sư OOP - Kiểm soát và tổ chức

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 3/2025

Mục lục

1 Đóng gói (Encapsulation)

Khái niệm Đóng gói là một nguyên tắc cơ bản của OOP, nhằm **ẩn chi tiết triển khai bên trong** của một lớp và chỉ cung cấp giao diện (interface) để tương tác với lớp đó từ bên ngoài. Điều này được thực hiện bằng cách:

- Giới hạn quyền truy cập trực tiếp vào các thuộc tính (data members) của lớp.
- Sử dụng các phương thức (methods) để kiểm soát cách dữ liệu được truy cập hoặc thay đổi.
- **Đặc điểm chính**:
  - Kết hợp dữ liệu (attributes) và hành vi (methods) thành một đơn vị (lớp).
  - Sử dụng các mức độ truy cập (public, private, protected) để kiểm soát quyền truy cập.
  - Bảo vệ dữ liệu khỏi bị thay đổi không mong muốn (data hiding).

```
class Car {
private:
    string brand;  // Thuộc tính bị ẩn
    int speed;     // Thuộc tính bị ẩn
public:
    // Giao diện công khai để truy cập dữ liệu
    void setBrand(string b) {
        brand = b;
    }
    void setSpeed(int s) {
        if (s >= 0) speed = s;  // Kiểm soát dữ liệu
    }
    string getBrand() {
        return brand;
    }
    int getSpeed() {
        return speed;
    }
};
```

Mục đích của đóng gói

1. **Ẩn dữ liệu (Data Hiding)**: Ngăn chặn việc truy cập trực tiếp vào dữ liệu nội bộ từ bên ngoài lớp, giảm nguy cơ dữ liệu bị hỏng hoặc sử dụng sai.
2. **Kiểm soát truy cập**: Cho phép lớp tự quản lý cách dữ liệu được đọc hoặc ghi, thêm các điều kiện hoặc kiểm tra hợp lệ (validation).
3. **Tăng tính linh hoạt**: Có thể thay đổi triển khai bên trong lớp (ví dụ: cách lưu trữ dữ liệu) mà không ảnh hưởng đến mã sử dụng lớp.
4. **Bảo trì dễ dàng**: Giao diện công khai ổn định giúp mã bên ngoài không cần thay đổi khi nội bộ lớp thay đổi.

Cách triển khai đóng gói

- **Sử dụng mức độ truy cập**:
  - Đặt các thuộc tính ở mức private hoặc protected.
  - Cung cấp các phương thức public (getter và setter) để truy cập hoặc thay đổi dữ liệu.
- **Getter và Setter**:

- **Getter:** Lấy giá trị của thuộc tính.
- **Setter:** Thiết lập giá trị, thường kèm theo kiểm tra hợp lệ.
- const trong getter: Nên khai báo getter là const để đảm bảo không thay đổi dữ liệu:

```cpp
class Student {
private:
    string name;  // Thuộc tính ẩn
    int age;      // Thuộc tính ẩn
public:
    // Setter với kiểm tra hợp lệ
    void setName(string n) {
        if (!n.empty()) name = n;  // Chỉ cho phép tên không rỗng
    }
    void setAge(int a) {
        if (a >= 0 && a <= 150) age = a;  // Giới hạn tuổi hợp lý
    }
    // Getter
    string getName() {
        return name;
    }
    int getAge() {
        return age;
    }
};

int main() {
    Student s;
    s.setName("Alice");
    s.setAge(20);
    cout << "Name: " << s.getName() << ", Age: " << s.getAge() << endl;
    s.setAge(-5);  // Không thay đổi vì không hợp lệ
    cout << "Age after invalid set: " << s.getAge() << endl;  // Vẫn là 20
    return 0;
}
```

2 Hàm bạn (Friend Function)

Khái niệm Hàm bạn (friend function) là một hàm không phải thành viên (non-member) của lớp, nhưng được khai báo là friend trong lớp để có quyền truy cập vào các thành viên private và protected của lớp đó. Đây là một cơ chế đặc biệt trong C++ nhằm mở rộng quyền truy cập mà không phá vỡ hoàn toàn nguyên tắc đóng gói (encapsulation).

Đặc điểm:

- Không thuộc lớp (không phải phương thức thành viên), nhưng có thể truy cập dữ liệu nội bộ của lớp.
- Được khai báo bằng từ khóa friend bên trong lớp.
- Có thể là hàm độc lập hoặc thuộc một lớp khác.

```
class MyClass {
private:
    int secret;  // Dữ liệu ẩn
public:
    MyClass(int value) : secret(value) {}
    // Khai báo hàm bạn
    friend void printSecret(MyClass obj);
};

// Định nghĩa hàm bạn
void printSecret(MyClass obj) {
    cout << "Secret value: " << obj.secret << endl;  // Truy cập private
}

int main() {
    MyClass obj(42);
    printSecret(obj);  // Output: Secret value: 42
    return 0;
}
```

Mục đích của hàm bạn

1. **Truy cập dữ liệu nội bộ**: Khi một hàm bên ngoài cần làm việc trực tiếp với các thành viên private hoặc protected mà không cần qua getter/setter.
2. **Hỗ trợ tương tác giữa các lớp**: Cho phép một hàm thao tác trên nhiều lớp mà không cần thuộc về bất kỳ lớp nào.
3. **Tối ưu hóa**: Tránh gọi nhiều getter/setter, giúp mã nhanh hơn trong một số trường hợp cụ thể.

Cách sử dụng hàm bạn

1. **Hàm bạn độc lập** Hàm không thuộc lớp nào, được khai báo friend để truy cập thành viên của lớp.

```
class Box {
private:
    double width;
    double height;
public:
    Box(double w, double h) : width(w), height(h) {}
    // Khai báo hàm bạn
    friend double getArea(Box b);
```

```
};

// Hàm bạn tính diện tích
double getArea(Box b) {
    return b.width * b.height;  // Truy cập private
}

int main() {
    Box box(5.0, 3.0);
    cout << "Area: " << getArea(box) << endl;  // Output: Area: 15
    return 0;
}
```

2. **Hàm bạn từ một lớp khác** Một phương thức của lớp khác có thể được khai báo là friend để truy cập dữ liệu của lớp hiện tại.

```
class Printer;  // Khai báo trước vì Printer dùng trong Box

class Box {
private:
    double width;
public:
    Box(double w) : width(w) {}
    // Khai báo phương thức của lớp Printer làm bạn
    friend void Printer::printWidth(Box b);
};

class Printer {
public:
    void printWidth(Box b) {
        cout << "Width: " << b.width << endl;  // Truy cập private
    }
};

int main() {
    Box box(10.0);
    Printer p;
    p.printWidth(box);  // Output: Width: 10
    return 0;
}
```

3. **Hàm bạn với nhiều lớp** Một hàm bạn có thể truy cập dữ liệu của nhiều lớp nếu được khai báo là friend trong tất cả các lớp đó.

```
class ClassB;  // Khai báo trước

class ClassA {
private:
    int valueA;
public:
    ClassA(int v) : valueA(v) {}
    friend int sum(ClassA a, ClassB b);
};
```

```
10
11 class ClassB {
12 private:
13     int valueB;
14 public:
15     ClassB(int v) : valueB(v) {}
16     friend int sum(ClassA a, ClassB b);
17 };
18
19 // Hàm bạn cộng giá trị từ hai lớp
20 int sum(ClassA a, ClassB b) {
21     return a.valueA + b.valueB;  // Truy cập private của cả hai
22 }
23
24 int main() {
25     ClassA a(5);
26     ClassB b(10);
27     cout << "Sum: " << sum(a, b) << endl;  // Output: Sum: 15
28     return 0;
29 }
```

Ưu điểm

- **Linh hoạt**: Cho phép các hàm bên ngoài truy cập dữ liệu mà không cần phá vỡ cấu trúc lớp.
- **Hữu ích trong toán tử**: Thường dùng để định nghĩa các toán tử như +, « cần truy cập dữ liệu nội bộ (ví dụ: friend ostream& operator«(ostream& os, const MyClass& obj)).
- **Hiệu suất**: Tránh gọi nhiều phương thức công khai khi cần thao tác trực tiếp trên dữ liệu.

Nhược điểm

- **Giảm đóng gói**: Việc cho phép hàm bạn truy cập dữ liệu nội bộ làm suy yếu nguyên tắc ẩn dữ liệu (data hiding).
- **Phụ thuộc**: Hàm bạn trở nên phụ thuộc vào triển khai nội bộ của lớp, nếu lớp thay đổi (ví dụ: đổi tên thuộc tính), hàm bạn cũng phải thay đổi.
- **Dễ lạm dụng**: Sử dụng quá nhiều hàm bạn có thể làm mã khó bảo trì và mất tính mô-đun.

Lưu ý quan trọng

- **Không phải thành viên**: Hàm bạn không thuộc lớp, nên không dùng this và không kế thừa trong các lớp dẫn xuất.
- **Tham số truyền giá trị**: Trong ví dụ trên, đối tượng được truyền bằng giá trị (copy), nếu cần tránh sao chép, dùng tham chiếu (&):

```
1 friend void printSecret(const MyClass& obj);
```

3 Lớp bạn (Friend Class)

Khái niệm

Lớp bạn (friend class) là một lớp được khai báo là friend trong một lớp khác, cho phép tất cả các phương thức của lớp bạn truy cập vào các thành viên private và protected của lớp khai báo nó. Đây là cơ chế giúp hai lớp tương tác chặt chẽ với nhau mà không phá vỡ hoàn toàn nguyên tắc đóng gói (encapsulation).

Đặc điểm:

- Lớp bạn không phải là thành viên của lớp gốc, nhưng có quyền truy cập toàn bộ dữ liệu nội bộ (bao gồm private và protected).
- Được khai báo bằng từ khóa friend theo sau là tên lớp.
- Quan hệ bạn bè là **một chiều** (không đối xứng): nếu lớp A là bạn của lớp B, thì B không tự động là bạn của A trừ khi được khai báo ngược lại.

```
class Box {
private:
    double width;
    double height;
public:
    Box(double w, double h) : width(w), height(h) {}
    // Khai báo lớp bạn
    friend class BoxPrinter;
};

class BoxPrinter {
public:
    void printArea(Box b) {
        double area = b.width * b.height;  // Truy cập private
        cout << "Area: " << area << endl;
    }
};

int main() {
    Box box(5.0, 3.0);
    BoxPrinter printer;
    printer.printArea(box);  // Output: Area: 15
    return 0;
}
```

4 Kế thừa (Inheritance)

4.1 Khái niệm

Kế thừa là một nguyên tắc cốt lõi của lập trình hướng đối tượng (OOP), cho phép một lớp mới (lớp dẫn xuất - derived class) thừa hưởng các đặc điểm (thuộc tính và phương thức) từ một lớp hiện có (lớp cơ sở - base class). Điều này nhằm mục đích tái sử dụng mã, thiết lập mối quan hệ "là một" (is-a), và hỗ trợ tính đa hình.

Mối quan hệ "là một" (is-a)

Ví dụ: "Car là một Vehicle", "Dog là một Animal".

Cú pháp:

```
class DerivedClass : accessSpecifier BaseClass {
    // Thành viên lớp con
};
```

Thành viên được kế thừa:

- public và protected: Lớp dẫn xuất có thể truy cập.
- private: Không được kế thừa trực tiếp, nhưng có thể truy cập gián tiếp qua phương thức của lớp cơ sở.

```
class Base {
public:
    int publicVar;  // Có thể truy cập ở mọi nơi

protected:
    int protectedVar;  // Chỉ truy cập được trong lớp cơ sở và lớp dẫn xuất

private:
    int privateVar;  // Chỉ có thể truy cập trong lớp cơ sở

public:
    // Hàm để gán giá trị privateVar (gián tiếp truy cập private từ lớp dẫn xuất)
    void setPrivateVar(int value) {
        privateVar = value;
    }

    // Hàm để lấy giá trị privateVar (gián tiếp truy cập private từ lớp dẫn xuất)
    int getPrivateVar() {
        return privateVar;
    }
};

// Lớp dẫn xuất sử dụng kế thừa public
class DerivedPublic : public Base {
public:
    void show() {
        cout << "publicVar = " << publicVar << endl;      // Truy cập được
        cout << "protectedVar = " << protectedVar << endl; // Truy cập được
        // cout << "privateVar = " << privateVar << endl; // Lỗi: Không truy cập
        ↪ được
    }
};
```

Mục đích

Kế thừa là một tính năng quan trọng trong Lập trình Hướng Đối Tượng (OOP) giúp một lớp (`lớp con` hoặc `lớp dẫn xuất`) có thể tái sử dụng các thuộc tính và phương thức từ một lớp khác (`lớp cha` hoặc `lớp cơ sở`). Mục đích chính của kế thừa:

- **Tái sử dụng mã nguồn**: Tránh việc lặp lại mã bằng cách sử dụng lại các phương thức và thuộc tính từ lớp cha.
- **Mở rộng tính năng**: Lớp con có thể thêm các thuộc tính và phương thức mới mà không cần thay đổi lớp cha.
- **Tổ chức mã theo phân cấp**: Dễ dàng quản lý các lớp có liên quan theo cấu trúc cha - con.
- **Tạo quan hệ "is-a"**: Giúp mô hình hóa mối quan hệ giữa các đối tượng (ví dụ: "Xe hơi" là một loại "Phương tiện").

4.2 Các loại kế thừa

Kế thừa đơn (Single Inheritance) Một lớp dẫn xuất kế thừa từ một lớp cơ sở duy nhất.

```
class Vehicle {
public:
    int speed;
    void move() {
        cout << "Moving at " << speed << " km/h" << endl;
    }
};

class Car : public Vehicle {
public:
    void honk() {
        cout << "Honking..." << endl;
    }
};

int main() {
    Car car;
    car.speed = 50;
    car.move();  // Output: Moving at 50 km/h
    car.honk();  // Output: Honking...
    return 0;
}
```

Kế thừa đa cấp (Multilevel Inheritance) Một chuỗi kế thừa: lớp A kế thừa từ lớp B, lớp B kế thừa từ lớp C.

```
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};

class Mammal : public Animal {
public:
    void walk() {
```

```
        cout << "Walking..." << endl;
    }
};

class Dog : public Mammal {
public:
    void bark() {
        cout << "Barking..." << endl;
    }
};

int main() {
    Dog dog;
    dog.eat();   // Từ Animal
    dog.walk();  // Từ Mammal
    dog.bark();  // Từ Dog
    return 0;
}
```

Kế thừa đa thừa (Multiple Inheritance) Một lớp dẫn xuất kế thừa từ nhiều lớp cơ sở.

```
class Engine {
public:
    void start() {
        cout << "Engine started" << endl;
    }
};

class Wheels {
public:
    void roll() {
        cout << "Wheels rolling" << endl;
    }
};

class Car : public Engine, public Wheels {
public:
    void drive() {
        cout << "Driving..." << endl;
    }
};

int main() {
    Car car;
    car.start();  // Từ Engine
    car.roll();   // Từ Wheels
    car.drive();  // Từ Car
    return 0;
}
```

Kế thừa phân cấp (Hierarchical Inheritance) Nhiều lớp dẫn xuất kế thừa từ cùng một lớp cơ sở.

```cpp
class Shape {
public:
    void draw() {
        cout << "Drawing a shape" << endl;
    }
};

class Circle : public Shape {
public:
    void drawCircle() {
        cout << "Drawing a circle" << endl;
    }
};

class Square : public Shape {
public:
    void drawSquare() {
        cout << "Drawing a square" << endl;
    }
};

int main() {
    Circle c;
    Square s;
    c.draw();       // Từ Shape
    c.drawCircle(); // Từ Circle
    s.draw();       // Từ Shape
    s.drawSquare(); // Từ Square
    return 0;
}
```

Kế thừa lai (Hybrid Inheritance) Kết hợp nhiều loại kế thừa (thường liên quan đến vấn đề kim cương, sẽ giải thích phần sau).

4.3 Mức độ truy cập trong kế thừa

1. **Kế thừa public**:
   - public trong lớp cơ sở → public trong lớp dẫn xuất.
   - protected trong lớp cơ sở → protected trong lớp dẫn xuất.
2. **Kế thừa protected**:
   - public trong lớp cơ sở → protected trong lớp dẫn xuất.
   - protected giữ nguyên.
3. **Kế thừa private**:
   - public và protected trong lớp cơ sở → private trong lớp dẫn xuất.
4. **private không kế thừa**: Lớp dẫn xuất không truy cập trực tiếp được thành viên private, cần dùng getter/setter.

```cpp
class Base {
public:
    int publicVar;
```

Thành viên trong lớp cơ sở	Kế thừa public	Kế thừa protected	Kế thừa private
public	public	protected	private
protected	protected	protected	private
private	**Không kế thừa trực tiếp**		

Bảng 1: *Mức độ truy cập trong kế thừa*

```
protected:
    int protectedVar;
private:
    int privateVar;
public:
    Base() : publicVar(1), protectedVar(2), privateVar(3) {}
    void setPrivate(int v) { privateVar = v; }
    int getPrivate() { return privateVar; }
};

class PublicDerived : public Base {
public:
    void access() {
        publicVar = 10;
        protectedVar = 20;
        setPrivate(30);
        cout << "PublicDerived: " << publicVar << " " << protectedVar << " " <<
        ↪  getPrivate() << endl;
    }
};

class ProtectedDerived : protected Base {
public:
    void access() {
        publicVar = 11;
        protectedVar = 21;
        setPrivate(31);
        cout << "ProtectedDerived: " << publicVar << " " << protectedVar << " " <<
        ↪  getPrivate() << endl;
    }
};

class PrivateDerived : private Base {
public:
    void access() {
        publicVar = 12;
        protectedVar = 22;
        setPrivate(32);
        cout << "PrivateDerived: " << publicVar << " " << protectedVar << " " <<
        ↪  getPrivate() << endl;
    }
};

int main() {
    PublicDerived pub;
    ProtectedDerived pro;
    PrivateDerived pri;

```

```
    pub.publicVar = 100;  // OK: public inheritance
    pub.access();         // Output: PublicDerived: 10 20 30

    // pro.publicVar = 200; // Lỗi: protected inheritance
    pro.access();         // Output: ProtectedDerived: 11 21 31

    // pri.publicVar = 300; // Lỗi: private inheritance
    pri.access();         // Output: PrivateDerived: 12 22 32

    return 0;
}
```

4.4 Hàm tạo và hàm hủy trong kế thừa

Trong kế thừa, khi một đối tượng của lớp dẫn xuất (derived class) được tạo hoặc bị hủy, cả lớp cơ sở (base class) và lớp dẫn xuất đều có vai trò trong việc khởi tạo và dọn dẹp tài nguyên. Điều này dẫn đến sự phối hợp giữa hàm tạo và hàm hủy của các lớp trong chuỗi kế thừa.

- **Hàm tạo (Constructor)**:
  - Dùng để khởi tạo đối tượng, bao gồm cả các thành viên kế thừa từ lớp cơ sở.
  - Trong kế thừa, hàm tạo của lớp cơ sở được gọi trước, sau đó đến hàm tạo của lớp dẫn xuất.
- **Hàm hủy (Destructor)**:
  - Dùng để dọn dẹp tài nguyên (như bộ nhớ động) khi đối tượng bị hủy.
  - Trong kế thừa, hàm hủy của lớp dẫn xuất được gọi trước, sau đó đến hàm hủy của lớp cơ sở.

Thứ tự gọi hàm tạo

- **Quy tắc**: Hàm tạo được gọi từ lớp cơ sở thấp nhất trong chuỗi kế thừa lên đến lớp dẫn xuất cuối cùng.
- **Cơ chế**:
  - Khi tạo một đối tượng của lớp dẫn xuất, trình biên dịch tự động gọi hàm tạo của lớp cơ sở trước.
  - Nếu lớp cơ sở có tham số, lớp dẫn xuất phải cung cấp giá trị cho hàm tạo của lớp cơ sở thông qua **danh sách khởi tạo (initializer list)**.

```
class Base {
public:
    Base() {
        cout << "Base default constructor" << endl;
    }
    Base(int x) {
        cout << "Base parameterized constructor with " << x << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived default constructor" << endl;
    }
    Derived(int x) : Base(x) {  // Gọi hàm tạo của Base với tham số
```

```
        cout << "Derived parameterized constructor" << endl;
    }
};

int main() {
    cout << "Creating Derived with default constructor:" << endl;
    Derived d1;
    cout << "\nCreating Derived with parameterized constructor:" << endl;
    Derived d2(10);
    return 0;
}
```

```
Output:
Creating Derived with default constructor:
Base default constructor
Derived default constructor

Creating Derived with parameterized constructor:
Base parameterized constructor with 10
Derived parameterized constructor
```

Giải thích:

- Với d1, hàm tạo mặc định của Base được gọi trước, sau đó là Derived.
- Với d2, lớp Derived gọi hàm tạo có tham số của Base qua danh sách khởi tạo, sau đó thực thi thân hàm tạo của mình.

Thứ tự gọi hàm hủy

- **Quy tắc**: Hàm hủy được gọi ngược lại với hàm tạo, từ lớp dẫn xuất xuống lớp cơ sở.
- **Cơ chế**: Khi một đối tượng ra khỏi phạm vi (scope) hoặc bị delete, trình biên dịch đảm bảo tài nguyên của lớp dẫn xuất được dọn dẹp trước, sau đó mới đến lớp cơ sở.

```
class Base {
public:
    Base() {
        cout << "Base constructor" << endl;
    }
    ~Base() {
        cout << "Base destructor" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived constructor" << endl;
    }
    ~Derived() {
        cout << "Derived destructor" << endl;
    }
};

int main() {
    Derived d;
```

```
    return 0;
}
```

Output:
```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

Giải thích:

- Hàm tạo: Base -> Derived.
- Hàm hủy: Derived -> Base. Điều này đảm bảo tài nguyên của lớp dẫn xuất được giải phóng trước khi lớp cơ sở bị hủy.

Hàm tạo với tham số và danh sách khởi tạo Nếu lớp cơ sở có hàm tạo yêu cầu tham số, lớp dẫn xuất phải cung cấp giá trị thông qua danh sách khởi tạo. Nếu không, trình biên dịch sẽ báo lỗi.

```
class Person {
protected:
    string name;
public:
    Person(string n) : name(n) {
        cout << "Person constructor: " << name << endl;
    }
};

class Student : public Person {
private:
    int id;
public:
    Student(string n, int i) : Person(n), id(i) {  // Gọi hàm tạo của Person
        cout << "Student constructor: ID " << id << endl;
    }
};

int main() {
    Student s("Alice", 123);
    return 0;
}
```

5 Kế thừa nhiều lớp (Multiple Inheritance)

Khái niệm Kế thừa nhiều lớp là khả năng một lớp dẫn xuất (derived class) thừa hưởng các đặc điểm (thuộc tính và phương thức) từ nhiều lớp cơ sở (base classes) cùng lúc trong C++. Điều này cho phép một lớp kết hợp các tính năng từ nhiều nguồn khác nhau, thể hiện mối quan hệ "là một" với nhiều thực thể.

```
class Derived : public Base1, public Base2 {
    // Nội dung lớp dẫn xuất
};
```

Một lớp dẫn xuất có thể "là một" của nhiều lớp cơ sở (ví dụ: một AmphibiousVehicle vừa là Car vừa là Boat).

Ưu điểm của kế thừa nhiều lớp

1. **Kết hợp chức năng**: Cho phép một lớp tích hợp các đặc điểm từ nhiều nguồn (ví dụ: vừa có khả năng chạy trên đất liền vừa trên nước).
2. **Tái sử dụng mã linh hoạt**: Tận dụng các lớp cơ sở đã có mà không cần viết lại hoặc tổ hợp phức tạp.
3. **Hỗ trợ mô hình hóa phức tạp**: Phù hợp với các tình huống thực tế nơi một thực thể có nhiều vai trò (ví dụ: một nhân viên vừa là Employee vừa là TaxPayer).

```
class Employee {
protected:
    string name;
public:
    Employee(string n) : name(n) {}
    void work() { cout << name << " is working" << endl; }
};

class TaxPayer {
protected:
    double income;
public:
    TaxPayer(double i) : income(i) {}
    void payTax() { cout << "Paying tax on " << income << endl; }
};

class Manager : public Employee, public TaxPayer {
public:
    Manager(string n, double i) : Employee(n), TaxPayer(i) {}
    void manage() {
        work();
        payTax();
        cout << name << " is managing" << endl;
    }
};

int main() {
    Manager mgr("Alice", 80000);
    mgr.manage();
    return 0;
}
// Output:
```

```
// Alice is working
// Paying tax on 80000
// Alice is managing
```

Nhược điểm và thách thức

1. **Độ phức tạp tăng**:Việc quản lý nhiều lớp cơ sở làm mã khó đọc và dễ xảy ra lỗi.
2. **Xung đột tên (Name Collision)**: Nếu hai lớp cơ sở có thành viên trùng tên, lớp dẫn xuất phải giải quyết xung đột bằng cách chỉ định rõ ràng (dùng ::).
3. **Vấn đề kim cương (Diamond Problem)**: Khi hai lớp cơ sở kế thừa từ cùng một lớp gốc, lớp dẫn xuất có thể nhận nhiều bản sao của lớp gốc, gây xung đột hoặc trùng lặp dữ liệu.

6 Quá tải hàm (Function Overloading)

Khái niệm Quá tải hàm là một tính năng trong C++ cho phép định nghĩa nhiều hàm cùng tên trong cùng một phạm vi (scope), miễn là các hàm này khác nhau về danh sách tham số (số lượng, kiểu dữ liệu, hoặc thứ tự tham số). Điều này giúp tăng tính linh hoạt và dễ đọc của mã bằng cách cho phép một tên hàm thực hiện nhiều hành vi khác nhau tùy theo cách gọi.

Đặc điểm:

- Tên hàm giống nhau, nhưng chữ ký (signature) phải khác nhau.
- Chữ ký bao gồm: số lượng tham số, kiểu dữ liệu tham số, và thứ tự tham số (không bao gồm kiểu trả về).
- Trình biên dịch chọn hàm phù hợp dựa trên đối số (argument) được truyền khi gọi.

```cpp
class Example {
public:
    void print(int x) {
        cout << "Integer: " << x << endl;
    }
    void print(double x) {
        cout << "Double: " << x << endl;
    }
    void print(string s) {
        cout << "String: " << s << endl;
    }
};

int main() {
    Example ex;
    ex.print(5);       // Gọi print(int)
    ex.print(3.14);    // Gọi print(double)
    ex.print("Hello"); // Gọi print(string)
    return 0;
}
// Output:
// Integer: 5
// Double: 3.14
// String: Hello
```

Mục đích của quá tải hàm

- **Tăng tính dễ đọc (Readability)**: Dùng cùng tên hàm cho các tác vụ liên quan thay vì đặt tên khác nhau (ví dụ: printInt, printDouble).
- **Tính linh hoạt (Flexibility)**: Cho phép xử lý nhiều kiểu dữ liệu hoặc tình huống khác nhau mà không cần logic điều kiện phức tạp.
- **Hỗ trợ đa dạng giao diện (Interface Variety)**: Cung cấp nhiều cách gọi hàm phù hợp với nhu cầu người dùng.
- **Tái sử dụng tên (Name Reuse)**: Tận dụng tên hàm quen thuộc trong các ngữ cảnh khác nhau.

Cách hoạt động Chữ ký hàm (Function Signature):

- Trình biên dịch phân biệt các hàm dựa trên danh sách tham số, không dựa trên kiểu trả về.
- Ví dụ: void print(int) và int print(int) không được coi là quá tải vì chỉ khác kiểu trả về.

```cpp
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
    string add(string a, string b) {
        return a + b;
    }
    int add(int a, int b, int c) {  // Quá tải với 3 tham số
        return a + b + c;
    }
};

int main() {
    Calculator calc;
    cout << "Int: " << calc.add(3, 4) << endl;          // Output: Int: 7
    cout << "Double: " << calc.add(3.5, 2.5) << endl;   // Output: Double: 6
    cout << "String: " << calc.add("Hello ", "World") << endl; // Output: String:
     ↪ Hello World
    cout << "Three ints: " << calc.add(1, 2, 3) << endl; // Output: Three ints: 6
    return 0;
}
```

Ẩn hàm (Function Hiding) Nếu lớp dẫn xuất định nghĩa một hàm cùng tên với lớp cơ sở nhưng khác chữ ký, các phiên bản quá tải trong lớp cơ sở sẽ bị ẩn (hide) trừ khi được đưa vào phạm vi bằng using.

```cpp
class Base {
public:
    void print(int x) {
        cout << "Base int: " << x << endl;
    }
    void print(double x) {
        cout << "Base double: " << x << endl;
    }
};

class Derived : public Base {
public:
    using Base::print;  // Đưa các hàm từ Base vào phạm vi
    void print(string s) {
        cout << "Derived string: " << s << endl;
    }
};

int main() {
    Derived d;
    d.print(5);       // Base int: 5
    d.print(3.14);    // Base double: 3.14
    d.print("Hello"); // Derived string: Hello
    return 0;
```

```
}
```

Liên hệ với kế thừa nhiều lớp Trong kế thừa nhiều lớp, quá tải có thể phức tạp hơn nếu các lớp cơ sở có hàm cùng tên. Cần dùng :: hoặc using để giải quyết.

```
class A {
public:
    void process(int x) { cout << "A int: " << x << endl; }
};

class B {
public:
    void process(double x) { cout << "B double: " << x << endl; }
};

class C : public A, public B {
public:
    void process(string s) { cout << "C string: " << s << endl; }
};

int main() {
    C c;
    c.A::process(5);     // A int: 5
    c.B::process(3.14);  // B double: 3.14
    c.process("Hello");  // C string: Hello
    return 0;
}
```

Lưu ý quan trọng

- **Kiểu trả về không ảnh hưởng:** Quá tải không dựa trên kiểu trả về, chỉ dựa trên danh sách tham số.

```
void func(int x) {}
// int func(int x) {}  // Lỗi: trùng chữ ký, chỉ khác kiểu trả về
```

- **Tham số mặc định (Default Arguments)** Có thể gây nhầm lẫn khi kết hợp với quá tải.

```
void func(int x) { cout << "One: " << x << endl; }
void func(int x, int y = 0) { cout << "Two: " << x << " " << y << endl; }

int main() {
    func(5);  // Lỗi: không rõ ràng (gọi func(int) hay func(int, int)?)
    return 0;
}
```

- **Ép kiểu (Type Conversion)** Trình biên dịch có thể chọn hàm gần khớp nhất nếu không có khớp chính xác.

```cpp
void print(int x) { cout << "Int: " << x << endl; }
void print(double x) { cout << "Double: " << x << endl; }

int main() {
    print(5.5f);  // Gọi print(double) qua ép kiểu từ float
    return 0;
}
// Output: Double: 5.5
```

- **Hiệu suất** Quá tải không gây chậm vì được giải quyết tại thời điểm biên dịch (compile-time).

7 Quá tải toán tử (Operator Overloading)

Khái niệm

Quá tải toán tử là kỹ thuật trong C++ cho phép bạn gán ý nghĩa mới cho các toán tử có sẵn khi chúng được sử dụng với các đối tượng của lớp do bạn định nghĩa. Điều này giúp mã trở nên trực quan và giống với cách sử dụng toán tử trên các kiểu dữ liệu cơ bản (như int, double).

Đặc điểm:

- Toán tử được định nghĩa lại thông qua hàm thành viên hoặc hàm bạn (friend function).
- Cú pháp: return_type operator(symbol)(parameters).
- Không thay đổi ưu tiên (precedence) hoặc tính kết hợp (associativity) của toán tử.

```
class Point {
private:
    int x, y;
public:
    Point(int x = 0, int y = 0) : x(x), y(y) {}
    // Quá tải toán tử +
    Point operator+(const Point& other) {
        return Point(x + other.x, y + other.y);
    }
    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point p1(2, 3), p2(4, 5);
    Point p3 = p1 + p2;  // Gọi operator+
    p3.display();        // Output: (6, 8)
    return 0;
}
```

Mục đích của quá tải toán tử

1. **Tăng tính trực quan**: Cho phép sử dụng toán tử theo cách tự nhiên (ví dụ: p1 + p2 thay vì p1.add(p2)).
2. **Tính nhất quán**: Làm cho lớp của bạn hoạt động giống kiểu dữ liệu cơ bản.
3. **Hỗ trợ biểu thức phức tạp**: Dễ dàng viết các biểu thức như obj1 + obj2 * obj3.
4. **Tính linh hoạt**: Mở rộng chức năng của toán tử cho các kiểu dữ liệu mới.

Cách triển khai quá tải toán tử

1. **Là hàm thành viên** Toán tử được định nghĩa trong lớp, với đối số đầu tiên là đối tượng hiện tại (qua this).

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}
    // Quá tải toán tử + làm hàm thành viên
    Complex operator+(const Complex& other) {
```

```
        return Complex(real + other.real, imag + other.imag);
    }
    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 2), c2(1, 4);
    Complex c3 = c1 + c2;
    c3.display();  // Output: 4 + 6i
    return 0;
}
```

2. **Là hàm bạn (Friend Function)** Dùng khi toán tử cần truy cập trực tiếp vào thành viên private của hai đối tượng, hoặc khi đối số đầu tiên không phải là đối tượng của lớp.

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}
    friend Complex operator+(const Complex& a, const Complex& b);
    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.real + b.real, a.imag + b.imag);
}

int main() {
    Complex c1(3, 2), c2(1, 4);
    Complex c3 = c1 + c2;
    c3.display();  // Output: 4 + 6i
    return 0;
}
```

Các toán tử thường được quá tải

- **Toán tử số học**: +, -, *, /, %.
- **Toán tử so sánh**: ==, !=, <, >, <=, >=.
- **Toán tử gán**: =.
- **Toán tử nhập/xuất**: «, ».
- **Toán tử tăng/giảm**: ++, –.
- **Toán tử truy cập**: [], ().

```
class Vector {
private:
    int x, y;
```

```
public:
    Vector(int x = 0, int y = 0) : x(x), y(y) {}
    Vector operator+(const Vector& v) { return Vector(x + v.x, y + v.y); }
    bool operator==(const Vector& v) { return x == v.x && y == v.y; }
    friend ostream& operator<<(ostream& os, const Vector& v);
};

ostream& operator<<(ostream& os, const Vector& v) {
    os << "(" << v.x << ", " << v.y << ")";
    return os;
}

int main() {
    Vector v1(1, 2), v2(3, 4), v3(1, 2);
    Vector v4 = v1 + v2;
    cout << v4 << endl;          // Output: (4, 6)
    cout << (v1 == v3) << endl; // Output: 1 (true)
    cout << (v1 == v2) << endl; // Output: 0 (false)
    return 0;
}
```

Kế thừa toán tử từ lớp cơ sở

```
class Base {
protected:
    int value;
public:
    Base(int v = 0) : value(v) {}
    Base operator+(const Base& other) {
        return Base(value + other.value);
    }
    void display() { cout << value << endl; }
};

class Derived : public Base {
public:
    Derived(int v = 0) : Base(v) {}
};

int main() {
    Derived d1(5), d2(10);
    Derived d3 = d1 + d2;  // Sử dụng operator+ từ Base
    d3.display();          // Output: 15
    return 0;
}
```

Lưu ý quan trọng

1. **Không thể quá tải một số toán tử**: :: (phạm vi), . (truy cập thành viên), .*, sizeof, typeid, alignof.
2. **Ít nhất một toán hạng phải là kiểu người dùng định nghĩa**: Không thể quá tải int + int, nhưng có thể quá tải MyClass + int.

3. **Giữ ngữ nghĩa tự nhiên**: + nên biểu thị cộng hoặc kết hợp, không nên dùng cho trừ để tránh nhầm lẫn.
4. **Trả về tham chiếu khi cần**: Với toán tử như =, «, trả về tham chiếu để hỗ trợ chuỗi phép toán (chaining).

```cpp
class Number {
private:
    int value;
public:
    Number(int v = 0) : value(v) {}
    Number& operator=(const Number& other) {
        value = other.value;
        return *this;  // Trả về tham chiếu để chaining
    }
    friend ostream& operator<<(ostream& os, const Number& n);
};

ostream& operator<<(ostream& os, const Number& n) {
    os << n.value;
    return os;
}

int main() {
    Number n1(5), n2, n3;
    n3 = n2 = n1;  // Chaining nhờ trả về tham chiếu
    cout << n3 << endl;  // Output: 5
    return 0;
}
```