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 9 Nghệ thuật lập trình - Đa hình và Trừu tượng hóa

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 Đa hình (Polymorphism)

1.1 Khái niệm

Đa hình là khả năng của một đối tượng hoặc hàm trong C++ để thể hiện nhiều hình thức (forms) khác nhau tùy theo ngữ cảnh. Từ "polymorphism" trong tiếng Hy Lạp nghĩa là "nhiều hình dạng" (poly = nhiều, morph = hình dạng). Trong OOP, đa hình cho phép xử lý các đối tượng của các lớp khác nhau thông qua một giao diện chung.

Đặc điểm:

- Từ "polymorphism" trong tiếng Hy Lạp nghĩa là "nhiều hình dạng" (poly = nhiều, morph = hình dạng).
- Có hai loại chính trong C++:
  - **Đa hình thời gian biên dịch (Compile-time Polymorphism)**: Thông qua quá tải hàm và quá tải toán tử.
  - **Đa hình thời gian chạy (Runtime Polymorphism)**: Thông qua hàm ảo và kế thừa.

```cpp
class Animal {
public:
    virtual void sound() {  // Hàm ảo
        cout << "Some generic sound" << endl;
    }
    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    void sound() override {  // Ghi đè
        cout << "Woof!" << endl;
    }
};

class Cat : public Animal {
public:
    void sound() override {
        cout << "Meow!" << endl;
    }
};

Animal* animal = new Dog();
animal->sound();  // Output: Woof! (đa hình thời gian chạy)
delete animal;
animal = new Cat();
animal->sound();  // Output: Meow!
delete animal;
```

1.2 Đa hình tại thời điểm biên dịch (Compile-time Polymorphism)

Định nghĩa: Đây là loại đa hình được xác định và giải quyết ngay tại thời điểm chương trình được biên dịch (trước khi chạy). Trình biên dịch sẽ quyết định hàm nào được gọi dựa trên thông tin có sẵn (ví dụ: số lượng hoặc kiểu tham số).

- Được thực hiện thông qua **nạp chồng hàm (function overloading)** hoặc **nạp chồng toán tử (operator overloading)**.

- Ví dụ: Bạn có thể định nghĩa nhiều hàm cùng tên nhưng khác tham số.

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

Example obj;
obj.print(5);      // Gọi hàm với int
obj.print(5.5);    // Gọi hàm với double
```

1.3 Đa hình tại thời điểm chạy (Run-time Polymorphism)

Định nghĩa: Đây là loại đa hình được xác định trong thời gian chương trình chạy. Nó cho phép một giao diện chung (hàm ảo) được gọi, nhưng hành vi cụ thể sẽ phụ thuộc vào đối tượng thực tế đang được tham chiếu.

- Được thực hiện thông qua **hàm ảo (virtual functions)** và **kế thừa (inheritance)**.
- Cho phép gọi phương thức của lớp con thông qua con trỏ hoặc tham chiếu của lớp cha.

```
class Animal {
public:
    virtual void sound() { cout << "Some generic animal sound" << endl; }
};

class Dog : public Animal {
public:
    void sound() override { cout << "Woof Woof" << endl; }
};

Animal* animal = new Dog();
animal->sound(); // In ra: "Woof Woof"
delete animal;
```

2 Khởi tạo đối tượng

2.1 Khái niệm

Khởi tạo đối tượng là quá trình cấp phát bộ nhớ và thiết lập trạng thái ban đầu cho một đối tượng của một lớp hoặc kiểu dữ liệu trong C++. Quá trình này thường được thực hiện thông qua **hàm tạo (constructor)**, một hàm thành viên đặc biệt được gọi tự động khi đối tượng được tạo. Khởi tạo đảm bảo đối tượng ở trạng thái hợp lệ ngay từ khi tồn tại, tránh lỗi do truy cập dữ liệu chưa khởi tạo.

- Hàm tạo không có kiểu trả về, trùng tên với lớp.
- Nếu không cung cấp hàm tạo, trình biên dịch tạo hàm tạo mặc định (nhưng không khởi tạo thành viên nguyên thủy).
- Liên quan đến kế thừa, đa hình, và quản lý tài nguyên.

```cpp
class Point {
public:
    int x, y;
    Point() : x(0), y(0) {}  // Hàm tạo mặc định
    Point(int xVal, int yVal) : x(xVal), y(yVal) {}  // Hàm tạo có tham số
};

int main() {
    Point p1;          // Khởi tạo mặc định
    Point p2(3, 4);    // Khởi tạo có tham số
    cout << p1.x << "," << p1.y << endl;  // Output: 0,0
    cout << p2.x << "," << p2.y << endl;  // Output: 3,4
    return 0;
}
```

2.2 Khởi tạo trong stack đối tượng

- Khởi tạo trên **ngăn xếp (stack)** là khi đối tượng được tạo trong bộ nhớ tự động (automatic storage). Đối tượng tồn tại trong phạm vi khai báo và tự động được hủy khi ra khỏi phạm vi (scope) nhờ cơ chế RAII (Resource Acquisition Is Initialization).
- Nhanh hơn khởi tạo trên heap vì không cần cấp phát động, nhưng bị giới hạn bởi kích thước stack (thường vài MB).
- Bộ nhớ được cấp phát tại compile-time, hàm tạo được gọi ngay khi đối tượng được khai báo.
- Hàm hủy được gọi tự động khi đối tượng ra khỏi phạm vi.

```cpp
class Box {
public:
    int size;
    Box(int s) : size(s) { cout << "Box created with size " << size << endl; }
    ~Box() { cout << "Box destroyed" << endl; }
};

int main() {
    Box b(10);  // Khởi tạo trên stack
    cout << "Size: " << b.size << endl;  // Output: Box created with size 10 \n
    ↪  Size: 10
    return 0;   // Tự động gọi ~Box() -> Output: Box destroyed
}
```

2.3 Khởi tạo trong heap đối tượng

- Khởi tạo trên **đống (heap)** là khi đối tượng được tạo trong bộ nhớ động (dynamic storage) bằng toán tử new. Đối tượng tồn tại cho đến khi được hủy thủ công bằng delete, cho phép kiểm soát vòng đời linh hoạt hơn stack.
- Chậm hơn stack do cấp phát động, nhưng không bị giới hạn kích thước (chỉ giới hạn bởi RAM).
- new cấp phát bộ nhớ trên heap, sau đó gọi hàm tạo với tham số (nếu có), trả về con trỏ.
- delete gọi hàm hủy và giải phóng bộ nhớ.

```cpp
class Circle {
public:
    int radius;
    Circle(int r) : radius(r) { cout << "Circle created with radius " << radius << endl; }
    ~Circle() { cout << "Circle destroyed" << endl; }
};

int main() {
    Circle* c = new Circle(5);  // Khởi tạo trên heap
    cout << "Radius: " << c->radius << endl;  // Output: Circle created with radius 5 \n Radius: 5
    delete c;  // Gọi ~Circle() -> Output: Circle destroyed
    return 0;
}
```

2.4 Khởi tạo thừa kế bằng con trỏ

- Trong kế thừa, khi khởi tạo đối tượng của lớp dẫn xuất thông qua con trỏ lớp cơ sở, hàm tạo của lớp cơ sở được gọi trước, sau đó đến hàm tạo của lớp dẫn xuất. Điều này đảm bảo đối tượng được khởi tạo đầy đủ theo hệ thống phân cấp.
- Dùng con trỏ hỗ trợ **đa hình thời gian chạy (runtime polymorphism)**, cho phép gọi hàm ảo của lớp dẫn xuất thông qua giao diện lớp cơ sở.
- Khi dùng new Derived(), trình biên dịch gọi chuỗi hàm tạo từ lớp cơ sở đến lớp dẫn xuất.
- Con trỏ lớp cơ sở trỏ đến đối tượng dẫn xuất, nhưng chỉ có thể truy cập thành viên của lớp cơ sở trừ khi ép kiểu.

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

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

int main() {
    Base* b = new Derived();  // Khởi tạo thừa kế bằng con trỏ
```

```
    b->show();  // Output: Base constructor \n Derived constructor \n Derived show
    delete b;   // Output: Derived destructor \n Base destructor
    return 0;
}
```

2.5 Ép kiểu

Ép kiểu (type casting) trong C++ là quá trình chuyển đổi một đối tượng hoặc con trỏ từ kiểu dữ liệu này sang kiểu dữ liệu khác. Trong bối cảnh lập trình hướng đối tượng (OOP), ép kiểu thường được sử dụng để xử lý các mối quan hệ giữa các lớp trong hệ thống phân cấp kế thừa, đặc biệt với **đa hình (polymorphism)**. C++ cung cấp nhiều cách ép kiểu, từ kiểu truyền thống (C-style) đến các toán tử ép kiểu hiện đại (static_cast, dynamic_cast, reinterpret_cast, const_cast), mỗi loại có mục đích và mức độ an toàn riêng.

- **Upcasting**: An toàn, tự động, chuyển từ lớp dẫn xuất lên lớp cơ sở.
- **Downcasting**: Không an toàn trừ khi dùng dynamic_cast, chuyển từ lớp cơ sở xuống lớp dẫn xuất.
- **Cross-casting**: Ép kiểu giữa các lớp không trực tiếp liên quan trong hệ thống phân cấp (thường cần dynamic_cast).
- **Ép kiểu không liên quan đến kế thừa**: Dùng reinterpret_cast hoặc const_cast, ít phổ biến trong OOP.

2.5.1 Upcasting

1. **Lý thuyết**:
   - Upcasting là quá trình chuyển đổi từ lớp dẫn xuất lên lớp cơ sở, luôn an toàn vì lớp dẫn xuất "là một" (is-a) lớp cơ sở theo quan hệ kế thừa.
   - Thường không cần toán tử ép kiểu rõ ràng, diễn ra tự động.
2. **Cơ chế bên dưới**:
   - Con trỏ/tham chiếu được điều chỉnh để trỏ đến phần lớp cơ sở của đối tượng, không thay đổi bố trí bộ nhớ.
   - Duy trì đa hình nhờ vtable nếu có hàm ảo.

```
class Animal {
public:
    virtual void sound() { cout << "Generic sound" << endl; }
    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    void sound() override { cout << "Woof!" << endl; }
};

int main() {
    Dog d;
    Animal* a = &d;       // Upcasting tự động
    a->sound();           // Output: Woof! (đa hình)
    return 0;
}
```

2.5.2 Downcasting với static_cast

1. **Lý thuyết**:
   - Downcasting chuyển từ lớp cơ sở xuống lớp dẫn xuất, không an toàn trừ khi lập trình viên chắc chắn về kiểu thực tế của đối tượng.
   - static_cast kiểm tra mối quan hệ kế thừa tại compile-time, không kiểm tra runtime.
2. **Cơ chế bên dưới**: Điều chỉnh con trỏ dựa trên bố trí bộ nhớ của lớp, không dùng RTTI, nhanh nhưng có thể gây lỗi nếu ép kiểu sai.

```
class Base {
public:
    virtual void show() { cout << "Base" << endl; }
    virtual ~Base() {}
};

class Derived : public Base {
public:
    void show() override { cout << "Derived" << endl; }
    void extra() { cout << "Extra" << endl; }
};

int main() {
    Derived d;
    Base* b = &d;               // Upcasting
    Derived* dPtr = static_cast<Derived*>(b);  // Downcasting
    dPtr->extra();              // Output: Extra
    return 0;
}
```

Nếu b không thực sự trỏ đến Derived, hành vi không xác định (undefined behavior).

2.5.3 Downcasting với static_cast

1. **Lý thuyết**:
   - dynamic_cast là cách ép kiểu an toàn trong OOP, kiểm tra kiểu thực tế của đối tượng tại runtime nhờ RTTI. Nó trả về nullptr (cho con trỏ) hoặc ném ngoại lệ (cho tham chiếu) nếu ép kiểu thất bại.
   - Yêu cầu lớp có ít nhất một hàm ảo.
2. **Cơ chế bên dưới**: Dùng RTTI và vtable để xác định kiểu thực tế, chậm hơn static_cast nhưng an toàn.

```
class Base {
public:
    virtual void show() { cout << "Base" << endl; }
    virtual ~Base() {}
};

class Derived : public Base {
public:
    void show() override { cout << "Derived" << endl; }
    void extra() { cout << "Extra" << endl; }
};
```

```cpp

int main() {
    Base* b = new Derived();
    Derived* d = dynamic_cast<Derived*>(b);  // Downcasting an toàn
    if (d) {
        d->extra();  // Output: Extra
    }

    Base* b2 = new Base();
    Derived* d2 = dynamic_cast<Derived*>(b2);  // Thất bại
    if (!d2) {
        cout << "Cast failed" << endl;  // Output: Cast failed
    }
    delete b;
    delete b2;
    return 0;
}
```

3 Hàm ảo (Virtual Functions)

Hàm ảo là một cơ chế trong C++ cho phép thực hiện **đa hình thời gian chạy (runtime polymorphism)** trong lập trình hướng đối tượng (OOP). Khi một hàm được khai báo là virtual trong lớp cơ sở (base class), nó có thể được **ghi đè (override)** bởi các lớp dẫn xuất (derived classes), và lời gọi hàm sẽ được xác định tại thời điểm chạy dựa trên kiểu thực tế của đối tượng, thay vì kiểu của con trỏ/tham chiếu.

Đặc điểm:

- Dùng từ khóa virtual trong khai báo hàm ở lớp cơ sở.
- Liên quan chặt chẽ đến kế thừa và đa hình, cho phép một giao diện chung gọi các hành vi khác nhau.
- C++ dùng **bảng hàm ảo (vtable)** để quản lý các lời gọi hàm ảo tại runtime.

```
class Animal {
public:
    virtual void sound() {  // Hàm ảo
        cout << "Generic sound" << endl;
    }
    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    void sound() override {  // Ghi đè
        cout << "Woof!" << endl;
    }
};

Animal* a = new Dog();
a->sound();  // Output: Woof! (đa hình thời gian chạy)
delete a;
```

Lưu ý quan trọng

- **Không có virtual, không đa hình**: Nếu hàm không được khai báo virtual, lời gọi dựa trên kiểu con trỏ/tham chiếu, không phải kiểu thực tế.
- **Hiệu suất**: Hàm ảo dùng vtable, tăng chi phí bộ nhớ (8 byte/con trỏ vtable trên 64-bit) và thời gian tra cứu so với hàm thông thường.
- **Từ khóa override**: Dùng override để đảm bảo hàm ghi đè đúng hàm ảo, tránh lỗi khi chữ ký không khớp.
- **Hàm hủy ảo**: Luôn khai báo hàm hủy ảo trong lớp cơ sở nếu có kế thừa để tránh rò rỉ tài nguyên.

3.1 Hàm ảo cơ bản

Khai báo hàm với từ khóa virtual trong lớp cơ sở, cung cấp triển khai mặc định (hoặc không).

- Trình biên dịch tạo một vtable cho lớp Shape, lưu trữ địa chỉ của draw().
- Khi Circle ghi đè, vtable của Circle trỏ đến phiên bản draw() của nó.
- Lời gọi qua con trỏ s tra cứu vtable để xác định hàm đúng.

```cpp
class Shape {
public:
    virtual void draw() {  // Hàm ảo với triển khai mặc định
        cout << "Drawing a generic shape" << endl;
    }
    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    void draw() override {  // Ghi đè
        cout << "Drawing a circle" << endl;
    }
};

Shape* s = new Circle();
s->draw();  // Output: Drawing a circle
delete s;
```

3.2 Hàm ảo với tham chiếu

Đa hình qua tham chiếu cũng hoạt động tương tự như con trỏ.

```cpp
class Base {
public:
    virtual void show() { cout << "Base" << endl; }
    virtual ~Base() {}
};

class Derived : public Base {
public:
    void show() override { cout << "Derived" << endl; }
};

void display(Base& obj) {
    obj.show();  // Đa hình
}

Derived d;
display(d);  // Output: Derived
```

3.3 Hàm hủy ảo (Virtual Destructor)

Hàm hủy cần được khai báo ảo trong lớp cơ sở để đảm bảo hủy đúng các lớp dẫn xuất khi dùng con trỏ lớp cơ sở. **Không dùng virtual**: Chỉ Base() được gọi, gây rò rỉ tài nguyên nếu Derived cấp phát động.

```cpp
class Base {
public:
    virtual void show() { cout << "Base" << endl; }
    virtual ~Base() { cout << "Base destructor" << endl; }
};

```

```
class Derived : public Base {
public:
    void show() override { cout << "Derived" << endl; }
    ~Derived() override { cout << "Derived destructor" << endl; }
};

Base* b = new Derived();
delete b;  // Gọi cả ~Derived() và ~Base() nhờ virtual
// Output:
// Derived destructor
// Base destructor
```

4 Hàm thuần ảo (Pure Virtual Function)

Hàm thuần ảo là một loại hàm ảo đặc biệt trong C++, được khai báo trong lớp cơ sở (base class) với cú pháp = 0, không cung cấp triển khai (implementation). Lớp chứa hàm thuần ảo trở thành **lớp cơ sở trừu tượng (abstract base class - ABC)** và không thể khởi tạo trực tiếp. Hàm thuần ảo buộc các lớp dẫn xuất (derived classes) phải ghi đè (override) để cung cấp triển khai cụ thể, hỗ trợ mạnh mẽ **đa hình thời gian chạy (runtime polymorphism)**.

- Cú pháp: virtual return_type function_name() = 0;.
- Lớp chứa hàm thuần ảo là trừu tượng, không thể tạo đối tượng từ nó.
- Dùng để định nghĩa giao diện (interface) chung mà các lớp dẫn xuất phải tuân theo.

```
class Vehicle {
public:
    virtual void move() = 0;  // Hàm thuần ảo
    virtual void stop() {     // Hàm ảo có triển khai
        cout << "Vehicle stops" << endl;
    }
    void honk() {             // Hàm thường
        cout << "Beep!" << endl;
    }
    virtual ~Vehicle() {}
};

class Car : public Vehicle {
public:
    void move() override {
        cout << "Car drives" << endl;
    }
    void stop() override {  // Ghi đè tùy chọn
        cout << "Car brakes" << endl;
    }
};

Vehicle* v = new Car();
v->move();   // Output: Car drives
v->stop();   // Output: Car brakes
v->honk();   // Output: Beep!
    delete v;
```

Lưu ý quan trọng

- **Lớp trừu tượng**: Lớp có hàm thuần ảo không thể khởi tạo trực tiếp (Animal a; sẽ lỗi).
- **Bắt buộc ghi đè**: Nếu lớp dẫn xuất không ghi đè hàm thuần ảo, nó vẫn là lớp trừu tượng.
- **Hiệu suất**: Giống hàm ảo thông thường, dùng vtable, chi phí nhỏ (8 byte/con trỏ vtable trên 64-bit).
- **Hàm hủy ảo**: Lớp trừu tượng vẫn nên có hàm hủy ảo để đảm bảo hủy đúng.

5 Lớp trừu tượng (Abstract Class)

Lớp trừu tượng trong C++ là một khái niệm trung tâm của lập trình hướng đối tượng (OOP), đại diện cho một mức độ **trừu tượng hóa (abstraction)** cao trong việc thiết kế phần mềm. Nó là một lớp không hoàn chỉnh (incomplete) về mặt triển khai, chứa ít nhất một **hàm thuần ảo (pure virtual function)**, được khai báo bằng cú pháp virtual function() = 0;. Điều này biến lớp thành một **lớp cơ sở trừu tượng (Abstract Base Class - ABC)**, không thể khởi tạo trực tiếp, mà chỉ tồn tại để cung cấp một **giao diện (interface)** hoặc **khung hành vi (behavioral skeleton)** cho các lớp dẫn xuất.

- Lớp trừu tượng là một biểu hiện của nguyên tắc **trừu tượng hóa dữ liệu (data abstraction)** trong OOP, nơi chi tiết triển khai được ẩn đi, chỉ để lại giao diện cần thiết.
- Nó liên kết chặt chẽ với **đa hình thời gian chạy (runtime polymorphism)**, cho phép các lớp dẫn xuất ghi đè (override) hành vi của hàm thuần ảo, từ đó tạo ra các thực thể cụ thể (concrete) từ một khuôn mẫu chung.
- Trong mô hình bộ nhớ của C++, lớp trừu tượng không khác biệt về mặt cấu trúc so với lớp thông thường, nhưng trình biên dịch áp đặt hạn chế khởi tạo dựa trên sự hiện diện của hàm thuần ảo.

```cpp
class Animal {
public:
    virtual void sound() = 0;  // Hàm thuần ảo
    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    void sound() override { cout << "Woof!" << endl; }
};

// Animal a;  // Lỗi: không thể khởi tạo lớp trừu tượng
Animal* a = new Dog();
a->sound();  // Output: Woof!
delete a;
```

6 Lớp cơ sở ảo (Virtual Base Class)

Lớp cơ sở ảo là một cơ chế trong C++ được sử dụng để giải quyết **vấn đề kim cương (diamond problem)** trong kế thừa nhiều lớp (multiple inheritance). Khi một lớp được khai báo là cơ sở ảo bằng từ khóa virtual trong khai báo kế thừa, nó đảm bảo rằng chỉ có **một bản sao duy nhất** của lớp cơ sở đó tồn tại trong hệ thống phân cấp, bất kể nó được kế thừa qua bao nhiêu đường dẫn.

- Dùng từ khóa virtual trước từ khóa truy cập (public, protected, private) khi kế thừa.
- Giải quyết vấn đề trùng lặp dữ liệu và xung đột tên trong kế thừa nhiều lớp.
- Liên quan đến việc quản lý bộ nhớ và truy cập thành viên trong hệ thống phân cấp phức tạp.

```
class Base {
public:
    int value = 10;
    void show() { cout << "Base: " << value << endl; }
};

class B : virtual public Base {};
class C : virtual public Base {};
class D : public B, public C {};

D d;
d.show();  // Output: Base: 10 (chỉ một bản sao của Base)
```

Phần 5: Lưu ý quan trọng

1. **Khởi tạo lớp cơ sở ảo**: Lớp dẫn xuất cuối cùng (như D) chịu trách nhiệm gọi hàm tạo của lớp cơ sở ảo, bỏ qua các lớp trung gian.
2. **Hiệu suất**: Kế thừa ảo phức tạp hơn kế thừa thông thường, thêm chi phí nhỏ để quản lý một bản sao duy nhất (thường qua con trỏ nội bộ).
3. **Không cần thiết trong kế thừa đơn**: Chỉ dùng lớp cơ sở ảo khi có kế thừa nhiều lớp và nguy cơ trùng lặp.
4. **Truy cập thành viên**: Không còn mơ hồ khi truy cập thành viên của lớp cơ sở ảo, nhưng cần chú ý nếu kết hợp với kế thừa không ảo.
5. **Hàm hủy ảo**: Lớp cơ sở ảo vẫn nên có hàm hủy ảo nếu dùng đa hình để đảm bảo hủy đúng.

7 Truy cập thành viên tĩnh

Trong C++, **thành viên tĩnh (static members)** là các thành viên (biến hoặc hàm) của một lớp được khai báo với từ khóa static. Chúng thuộc về **lớp (class)** chứ không thuộc về **đối tượng (object)** cụ thể, nghĩa là chỉ tồn tại một bản sao duy nhất trong suốt chương trình, bất kể có bao nhiêu đối tượng của lớp được tạo ra. **Truy cập thành viên tĩnh** là cách sử dụng các thành viên này thông qua tên lớp hoặc (ít phổ biến hơn) qua đối tượng.

1. **Lý thuyết cơ bản**:
   - Thành viên tĩnh được liên kết tĩnh (statically bound) tại thời điểm biên dịch, không phụ thuộc vào trạng thái của đối tượng hay cơ chế đa hình thời gian chạy (runtime polymorphism).
   - Biến tĩnh (static variables) được lưu trong vùng nhớ tĩnh (static storage), tồn tại suốt vòng đời chương trình.
   - Hàm tĩnh (static functions) không có quyền truy cập đến con trỏ this, vì chúng không hoạt động trên một đối tượng cụ thể.
2. **Đặc điểm**:
   - Truy cập qua toán tử phạm vi (::) với tên lớp (ví dụ: ClassName::staticMember).
   - Có thể truy cập mà không cần khởi tạo đối tượng.
   - Không tham gia vào đa hình (không thể khai báo virtual).

```cpp
class Counter {
public:
    static int instances;  // Khai báo biến tĩnh
    Counter() { instances++; }
    ~Counter() { instances--; }
};

int Counter::instances = 0;  // Định nghĩa biến tĩnh

int main() {
    Counter c1, c2;
    cout << Counter::instances << endl;  // Output: 2
    return 0;
}
```

```cpp
class Math {
public:
    static int add(int a, int b) {
        return a + b;
    }
};

int main() {
    int sum = Math::add(3, 4);
    cout << sum << endl;  // Output: 7
    return 0;
}
```

8 Giao diện (Interface)

Trong Java, **giao diện (interface)** là một cấu trúc cú pháp riêng biệt, được khai báo bằng từ khóa interface, dùng để định nghĩa một **hợp đồng (contract)** về hành vi mà các lớp triển khai (implementing classes) phải tuân theo. Giao diện chỉ chứa các **phương thức trừu tượng (abstract methods)** (trước Java 8) hoặc kết hợp với các phương thức mặc định (default methods) và phương thức tĩnh (static methods) từ Java 8 trở đi. Nó là công cụ chính để thực hiện **đa hình (polymorphism)** và **trừu tượng hóa (abstraction)** trong Java.

Lý thuyết cơ bản:

- Giao diện là một **kiểu tham chiếu (reference type)**, tương tự như lớp, nhưng không thể khởi tạo trực tiếp (tức là không có new Interface()).
- Trước Java 8, giao diện chỉ chứa các phương thức trừu tượng công khai (public abstract) và hằng số (public static final). Từ Java 8, nó được mở rộng để hỗ trợ triển khai mặc định, tăng tính linh hoạt.
- Giao diện hỗ trợ **kế thừa nhiều (multiple inheritance)** trong Java, khắc phục hạn chế của lớp (Java không cho phép kế thừa nhiều lớp).

```
interface Printable {
    void print();  // Phương thức trừu tượng
}

class Document implements Printable {
    public void print() {
        System.out.println("Printing document");
    }
}

public class Main {
    public static void main(String[] args) {
        Printable p = new Document();
        p.print();  // Output: Printing document
    }
}
```