1. Kế thừa (Inheritance)

Khái niệm: Là cơ chế cho phép xây dựng một lớp mới (Lớp con - Derived Class) dựa trên các thuộc tính và phương thức của một lớp đã có (Lớp cha - Base Class).

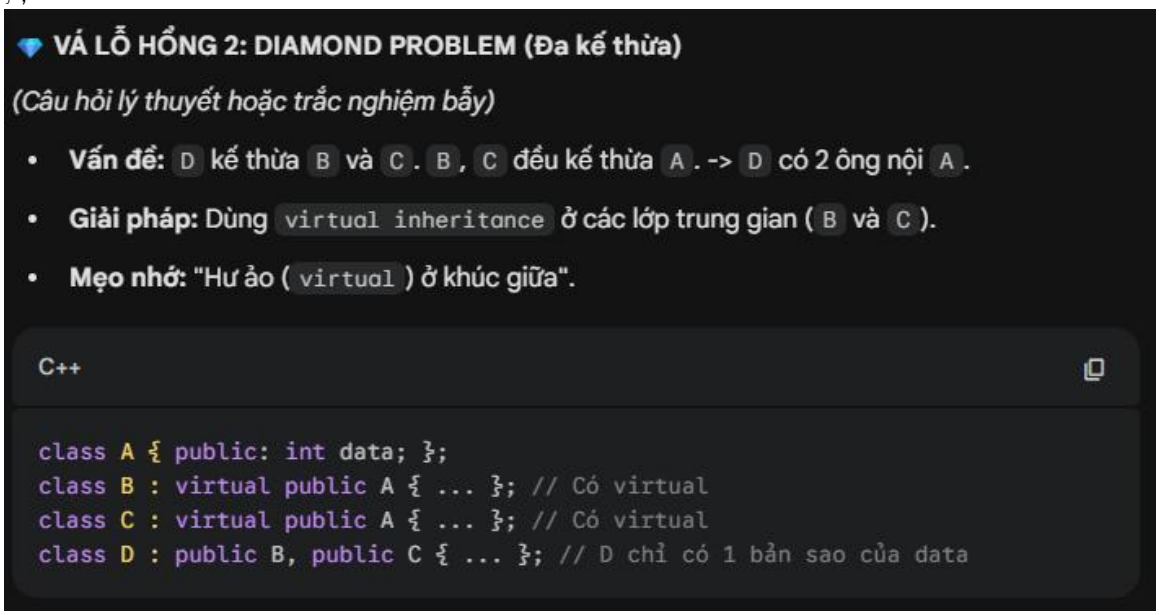
Mục đích: Tái sử dụng mã nguồn (Reusability) và tạo nên cấu trúc phân cấp (Hierarchy).

Quan hệ: Thể hiện mối quan hệ IS-A (Là một).

Ví dụ minh họa:

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

// Lớp con kế thừa lớp cha
class Cat : public Animal { // Cat IS-A Animal
public:
    void meow() { cout << "Meow..."; }
};
```



2. Đa hình (Polymorphism)

Khái niệm: Là khả năng các đối tượng thuộc các lớp khác nhau (trong cùng cây kế thừa) phản ứng theo những cách khác nhau đối với cùng một thông điệp (lời gọi hàm).

Các thể hiện (Manifestations):

Đa hình động (Runtime Polymorphism): Sử dụng virtual function và con trỏ/tham chiếu lớp cha (Base*). Quyết định gọi hàm nào xảy ra lúc chạy chương trình.

Đa hình tĩnh (Compile-time Polymorphism): Function Overloading (Nạp chồng hàm) và Template.

Điều kiện Đa hình động:

1. Có kế thừa.
2. Hàm ở lớp cha là virtual.

3. Gọi qua con trỏ/tham chiếu lớp cha.

Ví dụ:

```
  // 1. Lớp cha (Abstract Class) - Định nghĩa giao diện chung
class PaymentMethod {
public:
    // virtual là chìa khóa của Đa hình
    virtual void pay(double amount) = 0; // Pure virtual function
    virtual ~PaymentMethod() {} // Luôn nhớ virtual destructor!
};

  // 2. Các lớp con (Concrete Classes) - Thực hiện hành vi cụ thể
class Cash : public PaymentMethod {
public:
    void pay(double amount) override {
        cout << "-> Tra " << amount << "k bang TIEN MAT.\n";
    }
};

class CreditCard : public PaymentMethod {
public:
    void pay(double amount) override {
        cout << "-> Quet the VISA so tien: " << amount << "k (Phi 2%).\n";
    }
};

class Momo : public PaymentMethod {
public:
    void pay(double amount) override {
        cout << "-> Tru vi MOMO: " << amount << "k (Hoan tien 5%).\n";
    }
};

  // 3. Hàm xử lý chung (Không quan tâm loại thanh toán là gì)
void processCheckout(PaymentMethod* method, double money) {
    cout << "Dang xu ly giao dich...\n";
    method->pay(money); // ĐA HÌNH XẢY RA Ở ĐÂY
    // Dù code giống hệt nhau, nhưng chạy khác nhau tùy đối tượng truyền vào
}
```

3. Phân biệt Override và Overloading

	Overloading (Nạp chồng)	Overriding (Ghi đè/Phủ quyết)
Phạm vi	Trong cùng một class (hoặc global).	Giữa lớp Cha và lớp Con (Kế thừa).
Tên hàm	Giống nhau.	Giống nhau.
Tham số	BẮT BUỘC KHÁC NHAU (số lượng/kiểu).	BẮT BUỘC GIỐNG NHAU hoàn toàn.
Từ khóa	Không cần virtual.	Cần virtual ở lớp cha (nên có override ở lớp con).
Cơ chế	Compile-time (Tĩnh).	Runtime (Động).

Ví dụ minh họa:

```
class Base {
```

```
public:
    // Overloading: Cùng tên func, khác tham số
    void func(int x) { cout << "Int"; }
    void func(double y) { cout << "Double"; }

    virtual void show() { cout << "Base show"; }
};

class Derived : public Base {
public:
    // Overriding: Viết lại hàm show y chang của Base
    void show() override { cout << "Derived show"; }
};
```

Tại sao cái này hay? Giả sử sếp bảo: "Thêm tính năng thanh toán bằng Bitcoin".

Nếu không có đa hình: Em phải vào hàm processCheckout, viết thêm if (type == BITCOIN) {...}. Sửa code cũ -> Rủi ro bug.

Có đa hình: Em chỉ cần tạo class Bitcoin kế thừa PaymentMethod. Hàm processCheckout giữ nguyên 100%. Đây là nguyên lý Open/Closed (Mở để mở rộng, Đóng để sửa đổi).

4. Tính đóng gói (Encapsulation) & Blackbox Rule

Tính đóng gói: Là kỹ thuật gom dữ liệu (Data) và các thao tác trên dữ liệu (Methods) vào một đơn vị duy nhất (Class), đồng thời che giấu chi tiết cài đặt bên trong.

Sử dụng private để giấu dữ liệu.

Sử dụng public (Getter/Setter) để cung cấp cổng giao tiếp.

Blackbox Rule (Quy tắc hộp đen): * Người sử dụng đối tượng (Client) không cần biết bên trong đối tượng hoạt động thế nào (Logic phức tạp bị ẩn đi).

Họ chỉ cần biết đầu vào (Input) đưa vào cổng public và nhận lại kết quả (Output).

Lợi ích: Giúp thay đổi code bên trong mà không ảnh hưởng đến người dùng bên ngoài.

Chúng ta phải để private và chỉ cho phép truy cập qua Getter/Setter (public) vì 3 lý do sống còn sau:

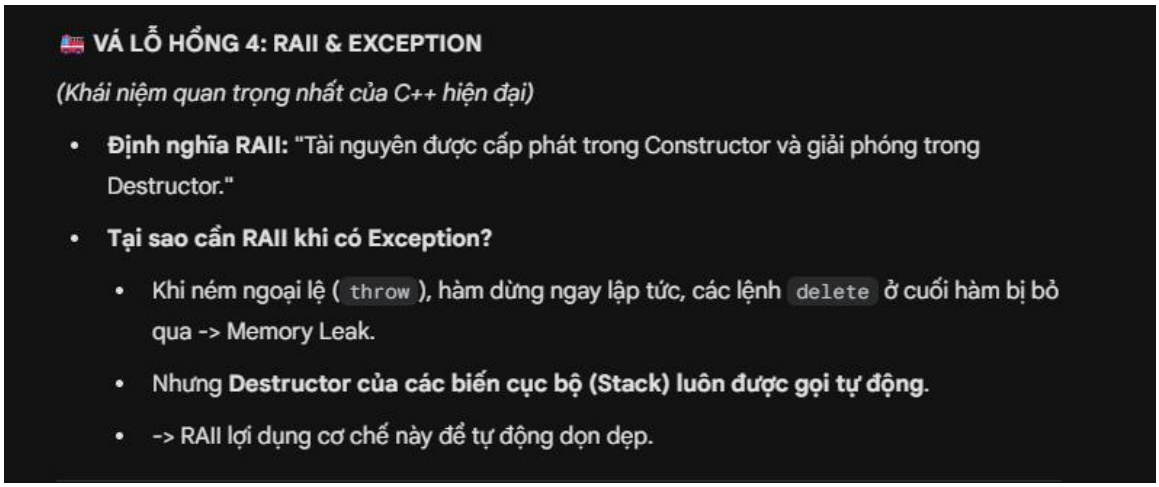
Lý do 1: Bảo vệ toàn vẹn dữ liệu (Data Integrity). Thực tế: Ngăn chặn data rác (tuổi âm, ngày tháng thứ 32, lương < 0) làm sập hệ thống tính toán phía sau.

Lý do 2: Ẩn giấu chi tiết xử lý (Abstraction). Người dùng class không cần biết dữ liệu được lưu thế nào, họ chỉ cần kết quả.

Lý do 3: Kiểm soát quyền truy cập (Read-only / Write-only)

Nếu để public, người ta vừa đọc vừa sửa được.

Nếu để private, em có thể chỉ viết hàm getSalary() (xem lương) mà không viết hàm setSalary() (sửa lương). -> Thực tế: Nhân viên chỉ được xem lương mình, không được tự sửa lương mình lên 1 tỷ.



5. Quy tắc "The Big Three" (3 Big Rule)

Nội dung: Nếu một class cần phải viết thủ công Hàm hủy (Destructor) (thường do có cấp phát động), thì class đó cũng cần phải viết thủ công Copy Constructor và Assignment Operator (operator=).

Tại sao (Why): * Trình biên dịch mặc định chỉ thực hiện Shallow Copy (Sao chép nông - chỉ copy địa chỉ con trỏ).

Khi 2 đối tượng cùng trỏ vào 1 vùng nhớ, nếu 1 đối tượng bị hủy (delete), vùng nhớ đó mất đi. Đối tượng kia truy cập vào sẽ lỗi hoặc khi hủy sẽ bị Double Free (Giải phóng 2 lần).

-> Cần cài đặt "The Big Three" để thực hiện Deep Copy (Sao chép sâu - tạo vùng nhớ mới riêng biệt).

Ví dụ Code:

```
class MyString {
    char* str;
public:
    // 1. Destructor
    ~MyString() { delete[] str; }

    // 2. Copy Constructor
    MyString(const MyString& other) {
        str = new char[strlen(other.str) + 1];
        strcpy(str, other.str);
    }

    // 3. Assignment Operator
    MyString& operator=(const MyString& other) {
        if (this != &other) { // Chống tự gán
            delete[] str;     // Xóa cũ
            str = new char[strlen(other.str) + 1]; // Cấp mới
            strcpy(str, other.str); // Copy dữ liệu
        }
        return *this;
    }
};
```

6. Lớp Trừu tượng (Abstract Class)

Khái niệm: Là lớp chứa ít nhất một hàm thuần ảo (pure virtual function).

Đặc điểm:

Không thể khởi tạo đối tượng (Instance) từ lớp này. (VD: Không thể new Shape(), chỉ có thể new Circle()).

Dùng làm khung mẫu (Interface) để các lớp con bắt buộc phải định nghĩa lại các hàm đó.

Cú pháp hàm thuần ảo: virtual void func() = 0;

7. Static Binding vs Dynamic Binding

Static Binding (Liên kết tĩnh): Hàm được quyết định gọi ngay lúc biên dịch (Compile-time). Xảy ra khi dùng hàm thường hoặc Overloading. (Nhanh nhưng cứng nhắc).

Dynamic Binding (Liên kết động): Hàm được quyết định gọi lúc chạy (Runtime) dựa vào đối tượng thực tế. Xảy ra khi dùng virtual + Con trỏ/Tham chiếu. (Linh hoạt).

8. Quan hệ HAS-A (Thành phần/Composition)

Khái niệm: Đối tượng này chứa đối tượng kia như một phần của nó.

Phân biệt:

IS-A (Kế thừa): Student là một Person.

HAS-A (Thành phần): Car có một Engine. (Xe hơi không là động cơ).

Tại sao quan trọng? Đề thi hay hỏi: "Khi nào nên dùng Kế thừa, khi nào nên dùng Thành phần?".

Trả lời: Dùng Kế thừa khi muốn tái sử dụng hành vi (behavior) và quan hệ là "Là một". Dùng Thành phần khi muốn kết hợp các bộ phận lại.

9. Từ khóa static & friend

static member:

Là tài sản chung của cả lớp, không thuộc về riêng một đối tượng nào.

Dùng để đếm số lượng đối tượng (count), hoặc hằng số chung.

friend function:

Hàm bạn không phải là thành viên của lớp nhưng được phép truy cập vào private của lớp đó.

Thường dùng để nạp chồng toán tử nhập xuất (>>, <<).

🔄 VÁ LỖ HỔNG 5: CASTING (Ép kiểu)

(Tránh mất điểm oan vì dùng sai loại cast)

- **Quy tắc bất di bất dịch:** Khi làm việc với Đa hình (Inheritance + Virtual), **TUYỆT ĐỐI KHÔNG** dùng ép kiểu kiểu C `(Type*)ptr`.
- **Phải dùng:**
  1. **Upcasting (Con -> Cha):** Tự động. `Base* b = new Derived();`
  2. **Downcasting (Cha -> Con):** Dùng `dynamic_cast`.
     - **Điều kiện:** Class cha phải có ít nhất 1 hàm `virtual`.
     - **Cơ chế:** Kiểm tra lúc chạy (Runtime). Nếu sai kiểu trả về `nullptr`.

C++

```
Base* b = new Base();
Derived* d = dynamic_cast<Derived*>(b); // Trả về nullptr vì b không phải Derive
if (d) { /* An toàn để dùng d */ }
```