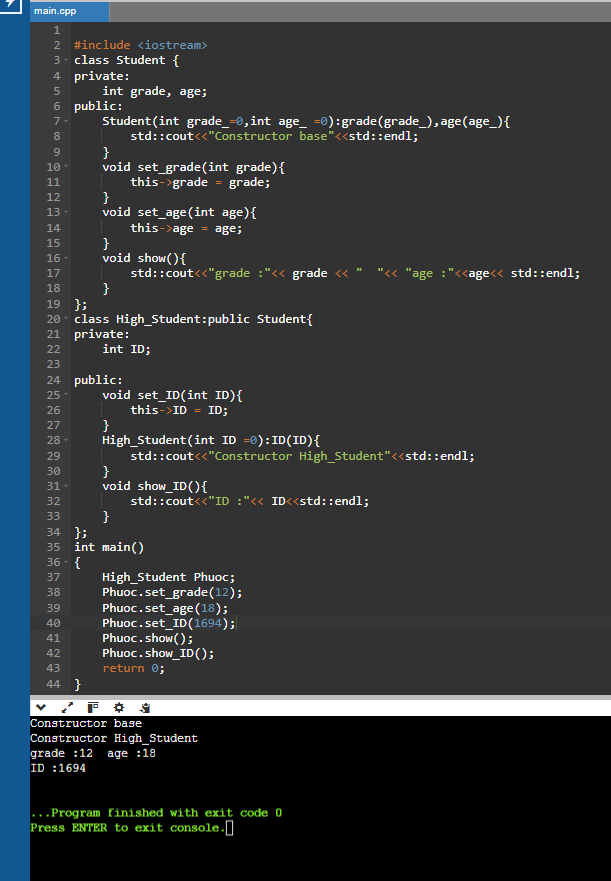
**III. Inheritance**

### Định nghĩa

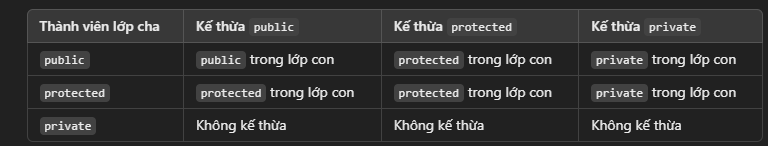
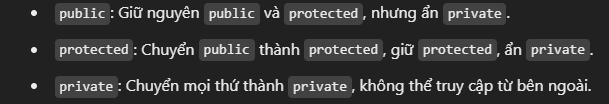
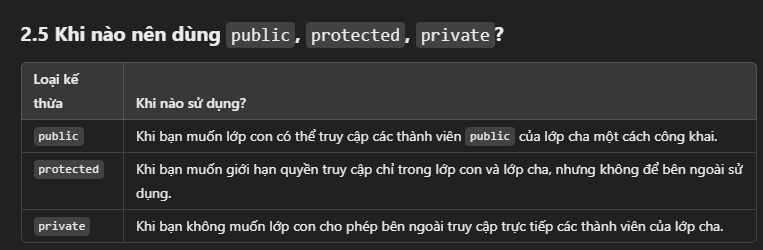
Kế thừa (Inheritance) là một khái niệm quan trọng trong lập trình hướng đối tượng (OOP), cho phép một lớp (class) con kế thừa các thuộc tính (attributes) và phương thức (methods) từ một lớp cha. Điều này giúp tái sử dụng mã nguồn, mở rộng chức năng và tổ chức hệ thống phân cấp lớp một cách hiệu quả.

### Mục đích của Kế thừa

* **Tái sử dụng mã nguồn (Code Reusability):** Giúp tránh việc viết lại mã bằng cách định nghĩa chức năng chung trong một lớp cơ sở.
* **Mở rộng tính năng (Extensibility):** Cho phép xây dựng các lớp chuyên biệt hơn dựa trên lớp tổng quát.
* **Tổ chức hệ thống phân cấp (Hierarchical Classification):** Giúp mô tả mối quan hệ thực tế (Ví dụ: Lớp "Người" có thể là lớp cha của "Sinh viên" và "Giảng viên").
* **Hỗ trợ Đa hình (Polymorphism):** Cho phép ghi đè (override) và mở rộng các phương thức theo nhu cầu của lớp con.
* Xem ví dụ dưới để hiểu cơ bản về kế thừa. Lớp con có các data của lớp cha, có data riêng của nó và cũng có thể gọi method của cha và method của nó. Ta sẽ đi sâu hơn sau.



2. Terminilogy and Notation : Thuật ngữ và kts hiệu:

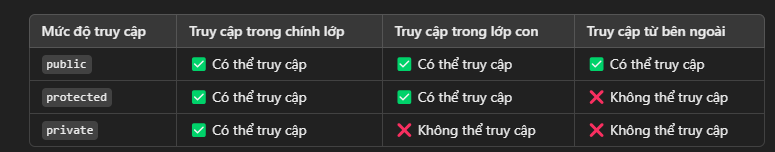
* 
* Cái này cẩn thận không lại bị nhầm, nói dễ hiểu thì:   
  + Không kế thừa được private của lớp cha. Chỉ kế thừa public và protected.   
  + Kế thừa kiểu public thì các access giữ nguyên của lớp cha  
  + Kế thừa kiểu protected thì acess là prottected cho tất cả.   
  + Kế thừa kiểu private thì acess là private cho tất cả
* 
* 

**3. Inheritance vs. Composition**

* Ta chỉ so sánh cơ bản giữa 2 kiểu này: Về cơ bản thì đều là các phương pháp chính để tái sử dụng mã nguồn và thiết lập cấu trúc lớp.
* Sự khác nhau cơ bản là Inheritance có mối quan hệ “Is a” giữa các class, còn ở composition là “Has a”. Kiểu như class Nhân Viên được kế thừa từ class con người, nhưng giữa class con người và class địa chỉ là quan hệ Has a. Kiểu thế.

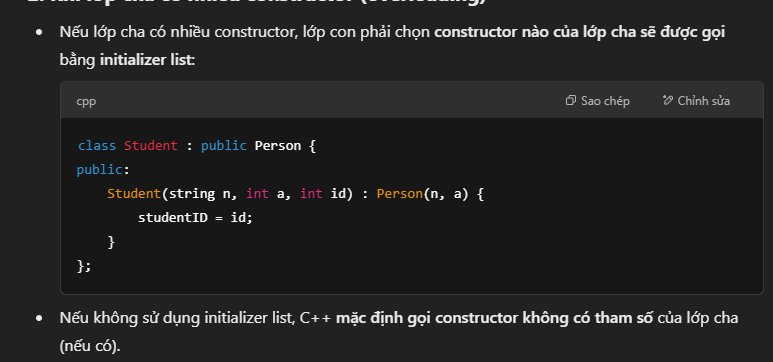
**4. Các mức độ truy cập trong Kế thừa:**

* Đứng dưới góc nhìn của class cha thì:



**5. Constructor và Destructor trong Kế thừa:**

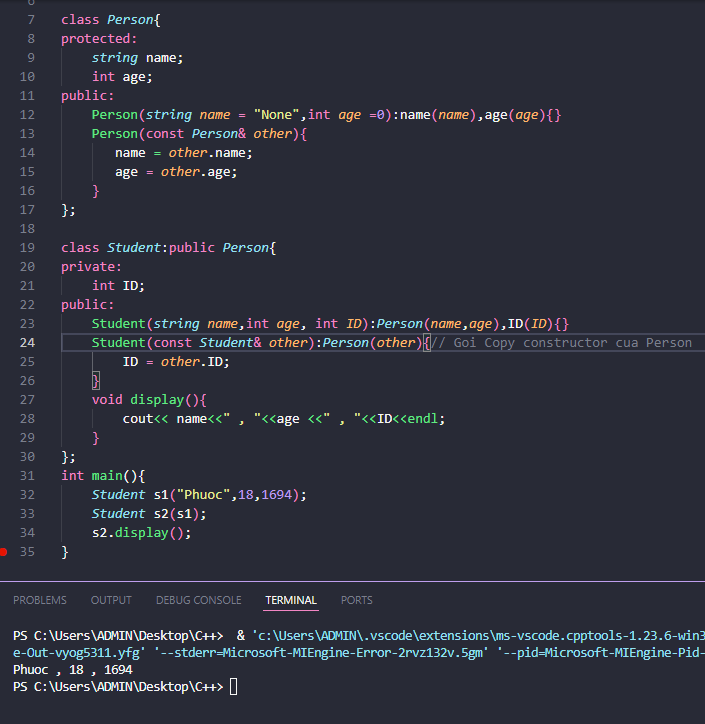
* Constructor của lớp cha được gọi trước rồi mới gọi constructor của lớp con.
* Destructor của lớp con được gọi trước rồi mới đên Destructor của lớp cha.
* Nếu lớp cha không có Default constructor Base() (các constructor của cha đều có tham số)thì lớp con phải gọi constructor của lớp cha bằng initializer list từ constructor của lớp con.



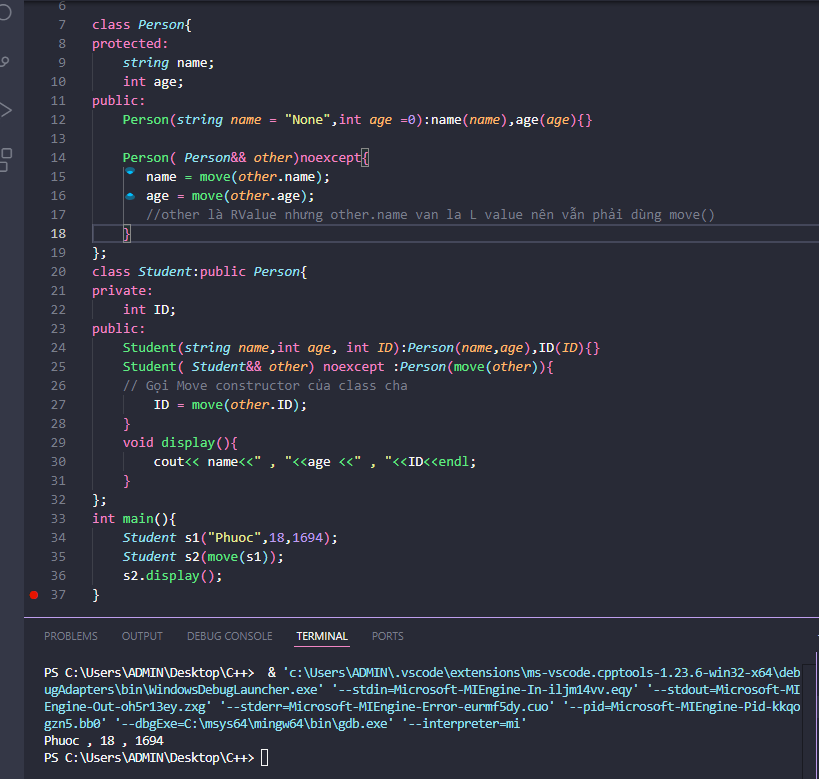


**6. Copy Constructor, Move Constructor và Toán tử = trong Lớp Kế Thừa**

* Đầu tiên là ta nên hiểu tổng quan thế này, Lớp con kế thừa từ 1 lớp cha, thì object tạo từ lớp con sẽ có những data, method của cả lớp cha và con (tùy loại kế thừa nữa). Thì khi ta sử dụng các phép như copy, move hay gán bằng toán tử = thì bản chất là đang copy/move toàn bộ data của lớp con, vì vậy tương tự như Constructor bình thường, copy constructor, move constructor và overload = cũng phải gọi đến copy/move… constructor tương ứng của lớp cha. Còn tùy từng loại gọi ra sao thì ta xem sau đây:



* Ta có thể thấy constructor và copy constructor ở Student đều gọi đến constructor và copy constructor của Person.



* Trường hợp với Move constructor cũng tương tự.

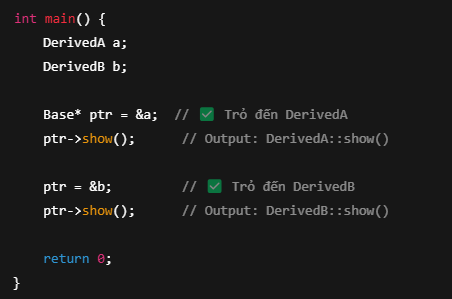


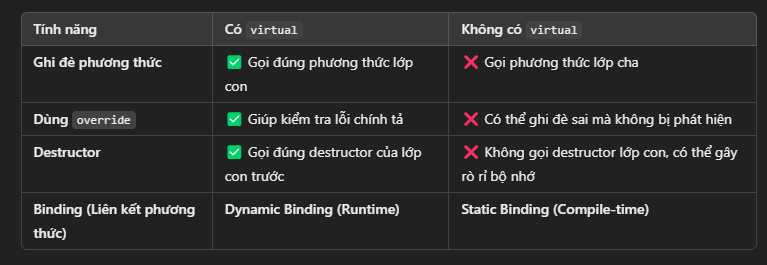
* Ta cungx phải gọi overload operator= của lớp cha trong overload operator= của lớp con.

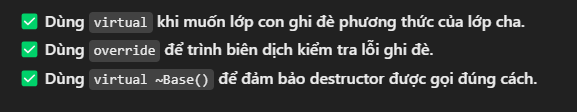
**7. Ghi đè phương thức của lớp cha (Overriding Base Class Methods)**

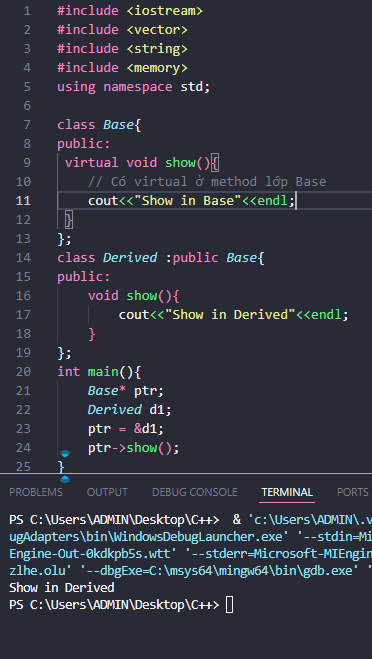
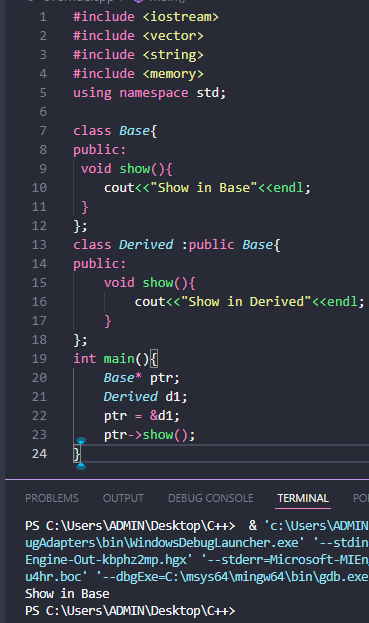
* Overriding ( ghi đè) là ghi lớp con định nghĩa lại 1 method của lớp cha để thay đổi cách hoạt động của nó. Tất nhiên là phải có cùng cả tham số.

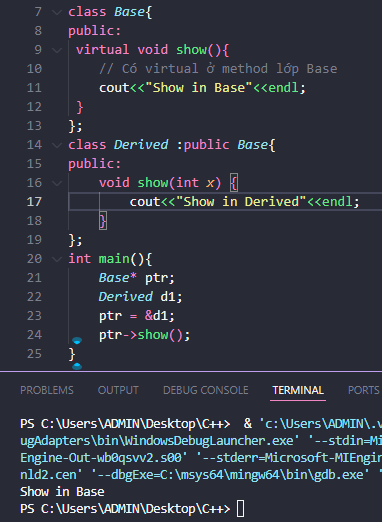
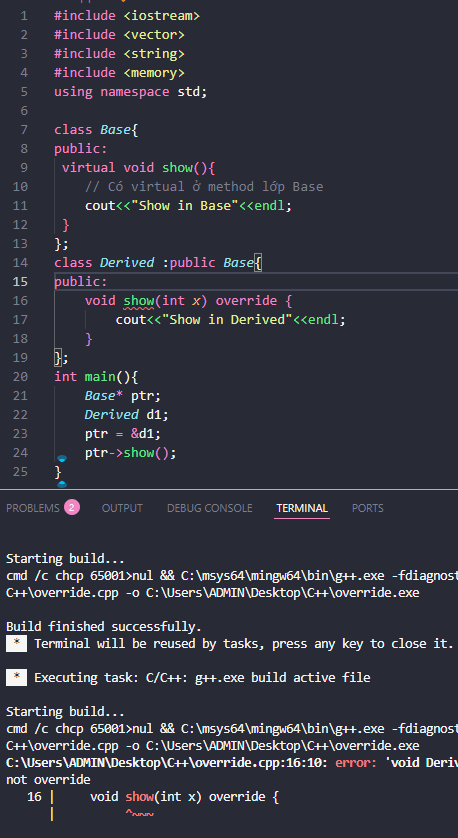
-> Lớp con có thể thay đổi hành vi của lớp cha mà không thay đổi lớp cha.

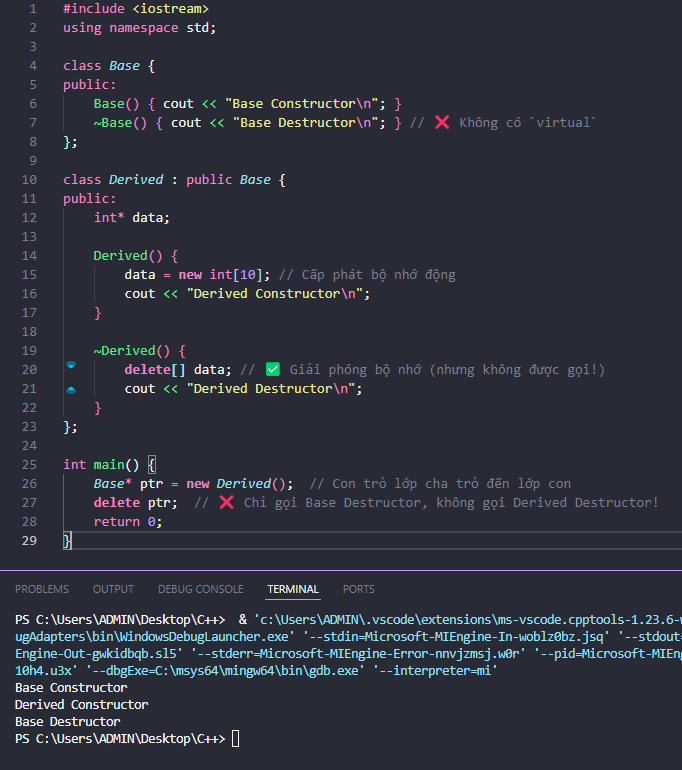
* **Dùng con trỏ Base\* ptr hoặc tham chiếu Base& ref để gọi đúng phương thức con.**
* Câu hỏi đặt ra là tại sao lại phải sử dụng con trỏ lớp Base trong khi nếu đơn muốn gọi method của lớp con ta tạo object lớp con là được rồi ?
* Câu trả lời là ta làm vậy vì tính đa hình, nghĩa là không chỉ có 1 lớp con kế thừa từ Base mà là nhiều lớp con thì sao ? Nếu chỉ tạo đối tượng lớp con thì chỉ gọi được method lớp đó thôi. Với 1 con trỏ base ta có thể gọi được nhiều method ghi đè ở nhiều lớp con khác nhau. 
* Ta sẽ tóm tắt nội dung trước rồi đi chi tiết:



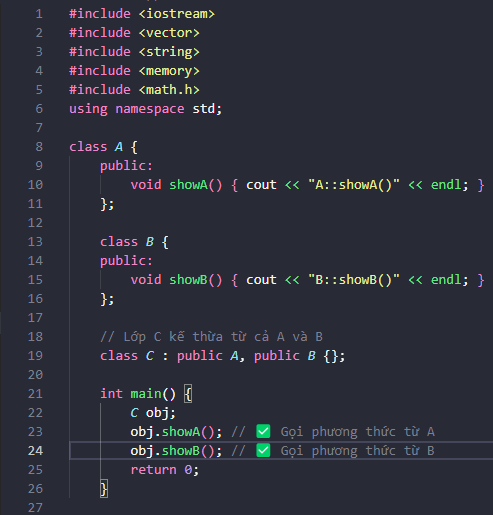


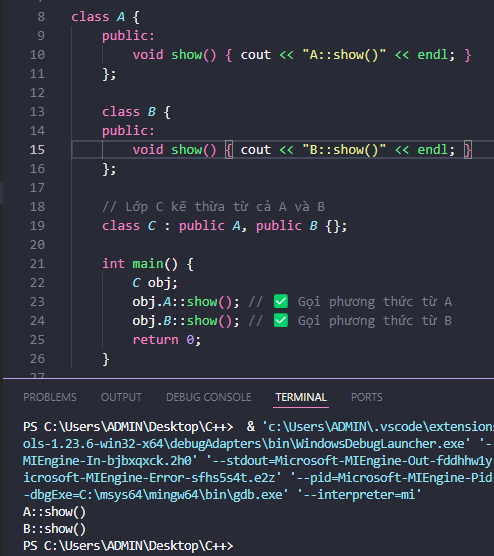
* Ta sẽ có các từ khóa cần hiểu rõ ở phần này: **virtual, override, Virtual ~Base(), Dynamic Binding, Static Binding.**
* Đầu tiên là virtual, nếu không có virtual thì phương thức lớp cha sẽ được gọi chứ không phải lớp con. 
* Chưa hiểu sâu nhưng giải thích như sau: Bình thường không có virtual, C++ dùng Static Binding tức là gọi phương thức theo kiểu của con trỏ Base ( tức là lớp Base).
* Dùng Virtual để bật tính năng Dynamic Binding -> Gọi đến method class Derived
* Tiếp theo là **Override**:
  + Vấn đề xảy ra khi ta override lại method trong lớp con nhưng lại làm thay dổi tham sô truyền vào. Điều này là không mong muốn và method lớp base sẽ được gọi. Tuy nhiên compiler sẽ không báo lỗi này.
  + Vì vậy ta sử dụng override để nếu sai chính tả thì sẽ báo lỗi.

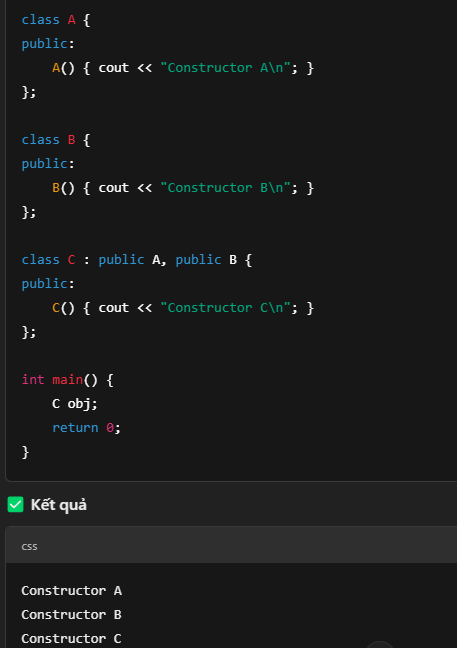
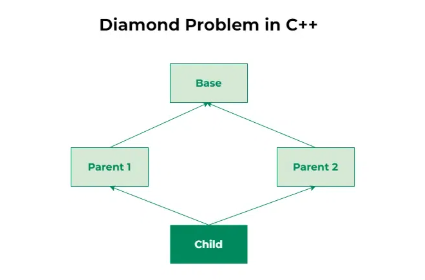


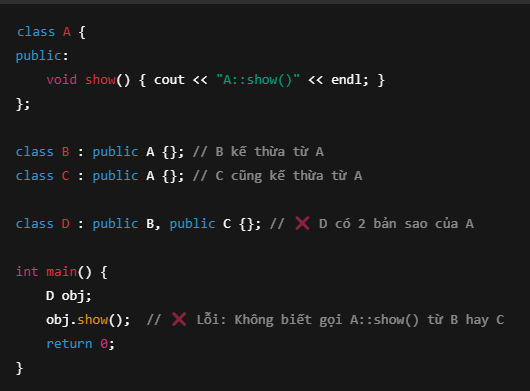
* Ghi đè Destructor với **virtual ~Base():**
* Vấn đề xảy ra khi ta sử dụng con trỏ lớp Base để sử dụng, khi out scope thì con trỏ này bị hủy tuy nhiên nó chỉ gọi đến Destructor của lớp cha chứ **không gọi đến Destructor lớp con**, cái này khá là nguy hiểm, Vì vậy ở lớp Base ta phải luôn dùng virtual ~Base() khi dùng con trỏ base để tránh trường hợp này.
* ****
* ****

**7.Kế thừa đa lớp (Multiple Inheritance) trong C++**

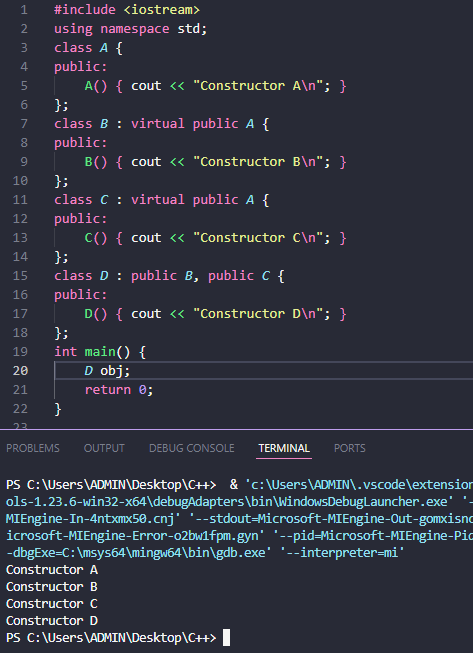
* Kế thừa Đa lớp là trường hợp 1 lớp con kế thừa từ nhiều lớp cha khác nhau cùng lúc.
* Cần lưu ý xung đột giữa các lớp cha.
* 
* Có thể gọi phương thức từ cả 2 lớp cha.
* Trường hợp 2 method của 2 lớp cha cùng tên. thì ta phải gọi cụ thể nếu không compiler sẽ báo lỗi.



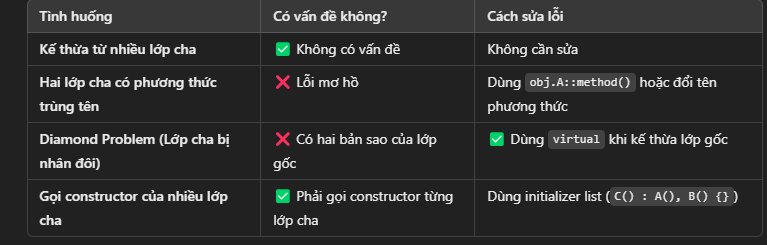
* Ở đây ta sẽ lưu ý 1 điều, đó là muốn gọi method của lớp cha từ lớp con ta có thể sử dụng **Base::method\_name()** như ví dụ trên.
* **Constructor trong kế thừa đa lớp:**
* 
* Được gọi theo thứ tự của Initialization list.
* **Diamond problem:**
  + 
* Xảy ra khi một lớp con kế thừa từ 2 lớp cha, mà 2 lớp cha này cùng kế thừa từ 1 lớp ông nội.
* Điều này dẫn đến lớp con có đến 2 bản sao của lớp ông nội, gây dư thừa bộ nhớ và lỗi mơ hồ.



* Để giải quyết vấn đề này, ta sẽ dùng “**virtual**” khi kế thừa A của các lớp B,C -> Điều này sẽ giúp chỉ có 1 bản sao của A trong Dn nghĩa là các lớp B,C dùng chung 1 bản sao của lớp A. Sâu hơn thì tìm hiểu sau

-

- Trong Diamond problem khi dùng virtual, Constructor của A chỉ đc gọi 1 lần.



Chủ đề nâng cao sau này rảnh học tiếp:

### 1️⃣ Lớp trừu tượng & Pure Virtual Function (= 0)

* **Lớp trừu tượng (Abstract Class)** và phương thức **pure virtual** (= 0).
* Cách dùng lớp trừu tượng để **bắt buộc các lớp con phải triển khai một số phương thức**.
* So sánh **lớp trừu tượng với Interface trong Java/C#**.

👉 **Từ khóa quan trọng:** virtual void draw() = 0;

### 2️⃣ Interface trong C++ (Lớp chỉ có Pure Virtual Function)

* Xây dựng **Interface** bằng cách sử dụng **pure virtual functions**.
* Tại sao C++ không có từ khóa interface như Java, nhưng vẫn có thể tạo Interface?
* Khi nào nên dùng Interface thay vì kế thừa thông thường?

### 3️⃣ Static Binding vs Dynamic Binding & Cơ chế VTable

* **Static Binding:** Khi C++ **quyết định phương thức nào được gọi tại compile-time**.
* **Dynamic Binding:** Khi C++ **dùng Virtual Table (VTable) để quyết định phương thức nào gọi tại runtime**.
* Hiểu cách C++ **tạo Virtual Table (VTable) & Virtual Pointer (VPTR)** để hỗ trợ virtual functions.

👉 **Từ khóa quan trọng:** VTable, VPTR, runtime polymorphism

### 4️⃣ Covariant Return Type (Kiểu trả về đồng biến)

* Khi ghi đè phương thức ảo (virtual), **kiểu trả về của lớp con có thể khác lớp cha** (nếu nó là một kiểu con).
* Giúp tránh ép kiểu con trỏ (Base\* -> Derived\*).

👉 **Từ khóa quan trọng:** Base\* clone() override;

### 5️⃣ Composition vs Inheritance – Khi nào nên dùng gì?

* **Khi nào nên dùng Composition thay vì Kế thừa?**
* **Tại sao kế thừa không phải lúc nào cũng là lựa chọn tốt nhất?**
* **Ưu và nhược điểm của Composition và Inheritance.**

👉 **Từ khóa quan trọng:** has-a vs is-a relationship

### 6️⃣ Smart Pointers & virtual ~Base()

* Dùng **std::unique\_ptr**, **std::shared\_ptr** để **quản lý bộ nhớ tự động** khi dùng kế thừa.
* Tránh dùng delete thủ công, giúp mã an toàn hơn và tránh memory leaks.

👉 **Từ khóa quan trọng:** std::unique\_ptr<Base>, std::shared\_ptr<Base>, make\_unique<>

### 7️⃣ Diamond Problem – Các cách giải quyết ngoài virtual

* **Tại sao không dùng virtual mà vẫn tránh được Diamond Problem?**
* **Dùng Composition thay vì Multiple Inheritance để tránh xung đột.**

👉 **Từ khóa quan trọng:** Avoiding Diamond Problem, Composition over Inheritance

### 8️⃣ Destructor thứ tự gọi như thế nào khi có Virtual Inheritance?

* Khi có **Kế thừa Ảo (virtual inheritance)**, **Destructor được gọi theo thứ tự nào?**
* **Hiểu cách C++ quản lý bộ nhớ khi dùng Virtual Base Class**.

👉 **Từ khóa quan trọng:** virtual inheritance, destructor call sequence

### 9️⃣ Kế thừa và Overloading Operators (operator=, operator<<, ...)

* Khi kế thừa, **cách ghi đè toán tử operator= đúng cách để tránh lỗi sao chép không mong muốn.**
* Cách **ghi đè operator<< để hỗ trợ in đối tượng ra màn hình**.

👉 **Từ khóa quan trọng:** operator overloading, operator<<, operator=

### 🔟 Multiple Inheritance và virtual có ảnh hưởng đến hiệu suất không?

* Khi dùng **Kế thừa đa lớp**, C++ phải **dùng thêm VTable**, điều này có ảnh hưởng đến hiệu suất không?
* So sánh hiệu suất **Kế thừa Ảo vs Kế thừa Thông thường**.

👉 **Từ khóa quan trọng:** Virtual Table (VTable), Performance Impact

### 📌 🚀 Tổng kết – Bạn chưa học các chủ đề sau:

✅ **1. Lớp trừu tượng & Pure Virtual Function (= 0)** ✅ **2. Interface trong C++** ✅ **3. Static Binding vs Dynamic Binding & Cơ chế VTable** ✅ **4. Covariant Return Type** ✅ **5. Composition vs Inheritance – Khi nào nên dùng gì?** ✅ **6. Smart Pointers & virtual ~Base()** ✅ **7. Diamond Problem – Các cách giải quyết ngoài virtual** ✅ **8. Destructor thứ tự gọi như thế nào khi có Virtual Inheritance?** ✅ **9. Kế thừa và Overloading Operators (operator=, operator<<, ...)** ✅ **10. Multiple Inheritance và virtual có ảnh hưởng đến hiệu suất không?**