Chương 06: Kế Thừa

**Câu 1:** Tại sao constructor lại gọi từ lớp cha đến lớp con trong khi destructor lại gọi từ lớp con đến lớp cha?

* Vì **khi xây dựng (constructor)** thì ta **xây từ gốc đến ngọn**, còn **khi hủy (destructor)** thì ta **phải gỡ từ ngọn đến gốc** để tránh lỗi và bảo vệ tài nguyên.

Ví dụ: Phải xây **tầng 1 trước** rồi mới có thể xây tầng 2. Còn khi phá thì phải phá từ tầng 2 trước nếu phá tầng 1 thì mọi thứ sẽ sập.

**Câu 2:** Tại sao trong **CDaThuc operator+(const CDaThuc& DT);** lại có cả **const** (giúp tránh thay đổi đối tượng gốc) và **&** (thường dùng để thay đổi đối tượng gốc)

Về bản chất thì & giúp truy cập và dùng trực tiếp đối tượng gốc, còn không có & thì có nghĩa là làm việc với phần sao chép của đối tượng gốc. **Khi đối tượng tốn quá nhiều bộ nhớ và tài nguyên để copy** thì chúng ta muốn làm việc với đối tượng gốc thay vì bản copy nên dùng &. Còn const thì để tránh việc chúng ta thay đổi đối tượng khi làm việc trực tiếp, complier sẽ báo lỗi.

**Câu 3:** Tại sao trong **CDiem::operator<=(const CDiem& d) const** lại có **const** ở cuối?

Const được dùng ở cuối để khi hàm không thay đổi dữ liệu và đặc biệt gọi một hàm có một đối tượng const

Ví dụ:

**float KhoangCach();** → this mặc định có kiểu CDiem\*

**bool CDiem::operator<=(const CDiem& d) const** {

return KhoangCach() <= d.KhoangCach();

}

→ this có kiểu **const** CDiem\*

nó sẽ gây ra lỗi “cannot convert 'this' pointer from 'const CDiem' to 'CDiem &'” vì C++ không cho phép gọi hàm KhoangCach() (phi const)

**Câu 4**: Tại sao **CDaThuc C = F;** lại dùng được mặc dù không có phép gán  
Dòng này thực hiện **copy initialization**, tức là gọi **copy constructor**. Nếu ta chưa định nghĩa CDaThuc(const CDaThuc&), trình biên dịch sẽ sinh ra bản mặc định, thực hiện sao chép **từng thành viên** (member-wise copy).

Khi nào cần dùng đa năng hóa toán tử gán  
- Lớp có con trỏ cấp phát động (new) → cần quản lý sâu

- Lớp quản lý tài nguyên hệ thống (file, socket,...)

**Câu 5:** Tại sao khi dùng **CPhanSo CPhanSo::RutGon()** thì nó vẫn truy cập được các thuộc tính **private**?

Private **ngăn chặn truy cập từ bên ngoài lớp**, không phải ngăn lớp đó **tự xử lý nội bộ dữ liệu của mình** nên các hàm thành viên được phép truy cập -> không mất tính đóng gói Ví dụ: Mặc dù xăng được đóng gói trong xe nhưng các chức năng của xe vẫn có thể sử dụng. Chỉ có bên ngoài mới không dùng xăng được.

**Câu 6:** Làm thế nào để dùng **Đa năng hóa >>** và **<<** cho **lớp con** khi lớp cha đã đa năng hóa? **(Kế thừa + Đa năng hóa)**

->Chúng ta sẽ **ép kiểu đối tượng lớp con về lớp cha** giúp gọi lại chính xác hàm operator>>

Ví dụ

**Lớp cha:**

friend istream &operator>>(istream &is, **MONHOC &Mon**) {}

friend ostream &operator<<(ostream &os, **const MONHOC &Mon**) {}

**Lớp con:**

friend istream &operator>>(istream &is, MONKIEMTRA &Mon)

    {

**is >> (MONHOC &)Mon;** // Ép kiểu về Mon về tham chiếu lớp cha MONHOC -> nó tương đương operator>>(istream &is, (MONHOC&) Mon);

        is >> Mon.He3;

        return is;

    }

    friend ostream &operator<<(ostream &os, const MONKIEMTRA &Mon)

    {

**os << (const MONHOC &)Mon << endl;**

        os << Mon.He3 << endl;

        return os;

    }

**Đối với các toán tử như +, -, \*, ...** thường hoạt động trực tiếp trên các thuộc tính dữ liệu (như số nguyên, phân số, số thực, v.v...), và nó không liên quan trực tiếp đến hàm như << và >> nên nếu lớp con tự định nghĩa lại toán tử này thì không cần ép kiểu về lớp cha để sử dụng.