

# KỸ NĂNG 1: DỊCH UML SANG C++ (TRANSLATION)

## Cấp độ 1: Cú pháp cơ bản (Basic Syntax)

Nhìn vào hình vẽ, bạn phải phản xạ ngay ra code C++.

Ký hiệu UML	Ý nghĩa	Code C++ tương ứng	Lưu ý Senior
<b>Class Name</b>	Tên lớp	<code>class Name { ... };</code>	Nhớ dấu chấm phẩy <code>;</code> cuối class! Quên là 0 điểm biên dịch.
<b>+ Attribute</b>	Public	<code>public:</code>	Thường dùng cho phương thức (Method).
<b>- Attribute</b>	Private	<code>private:</code>	Thường dùng cho thuộc tính (Data).
<b># Attribute</b>	Protected	<code>protected:</code>	Dùng khi có kế thừa (cho con xài ké).
<code>name: string</code>	Kiểu dữ liệu	<code>string name;</code>	Trong C++ là <code>std::string</code> (nhớ <code>#include &lt;string&gt;</code> ).
<code>&lt;u&gt;count&lt;/u&gt;</code>	Static	<code>static int count;</code>	Biến dùng chung cho cả lớp, không thuộc về object nào.
<code>Method()</code>	Abstract	<code>virtual void Method() = 0;</code>	Hàm thuần ảo. Class chứa nó thành Abstract Class.

## Cấp độ 2: Xử lý các Mối quan hệ (Relationships) - PHẦN QUAN TRỌNG NHẤT

Đây là chỗ sinh viên hay mất điểm nhất vì không phân biệt được Composition và Aggregation.

### 1. Kế thừa (Inheritance) - Mũi tên tam giác rỗng $\triangle$

- **UML:** `Student`  $\longrightarrow$  `Person`
- **Ý nghĩa:** Student LÀ Person (IS-A).
- **Code:**

C++

```
class Student : public Person { // Mặc định dùng public inheritance
    // ...
};
```

## 2. Kết tập (Aggregation) - Hình thoi rỗng

- **UML:** Classroom ◇ → Student
- **Ý nghĩa:** Classroom **CÓ** Student, nhưng Student tồn tại độc lập (HAS-A, weak). Nếu lớp học giải tán, học sinh vẫn còn đó.
- **Code:** Dùng Con trỏ (Pointer) hoặc Tham chiếu.

C++

```
class Classroom {
private:
    vector<Student*> students; // Chứa danh sách CON TRỎ đến Student
public:
    void addStudent(Student* s) { // Nhận Student từ bên ngoài vào
        students.push_back(s);
    }
};
```

## 3. Cấu thành (Composition) - Hình thoi đặc

- **UML:** Car ◆ → Engine
- **Ý nghĩa:** Car **BAO GỒM** Engine (Part-of, strong). Xe hỏng thì Động cơ đi tong (về mặt quản lý vòng đời trong phần mềm).

- **Code:** Dùng Biến thành viên (Instance variable) trực tiếp hoặc quản lý chặt chẽ việc new/delete .

C++

```
class Car {
private:
    Engine myEngine; // Cách 1: Khai báo trực tiếp (Dễ nhất)
    // Hoặc Cách 2: Pointer nhưng new trong Constructor, delete trong Destructor
public:
    Car() {
        // Engine tự được tạo khi Car được tạo
    }
};
```

#### 4. Phụ thuộc (Dependency) - Mũi tên nét đứt ---->

- **UML:** `Printer` ----> `Document`
- **Ý nghĩa:** `Printer` **DÙNG** `Document` để in, nhưng không sở hữu nó.
- **Code:** Truyền làm **Tham số hàm**.

C++

```
class Printer {  
public:  
    void print(const Document& doc) { // Truyền vào để xài  
        cout << doc.getContent();  
    }  
};
```

## KỸ NĂNG 2: NẠP CHỒNG TOÁN TỬ (OPERATOR OVERLOADING)

Đề thi 90% sẽ yêu cầu viết class `PhanSo` (Phân số), `SoPhuc` (Số phức), `NgayThang` (Date) hoặc `MyString` và yêu cầu chúng cộng trừ nhân chia được với nhau.

### 1. Phân biệt trả về: Tham trị (Value) vs Tham chiếu (Reference)

Rất nhiều bạn thi rớt hoặc mất điểm vì viết loạn xạ chỗ này. Hãy nhớ quy tắc "Sinh ra cái mới hay sửa cái cũ?".

Toán tử	Ý nghĩa	Trả về (Return Type)	Tại sao?
<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code>	Tính toán ra kết quả <b>MỚI</b>	<code>PhanSo</code> (Tham trị)	<code>c = a + b</code> . <code>a</code> , <code>b</code> không đổi. <code>c</code> là đối tượng mới sinh ra.
<code>+=</code> , <code>-=</code> , <code>++</code> (tiền tố)	<b>SỬA</b> trực tiếp đối tượng hiện tại	<code>PhanSo&amp;</code> (Tham chiếu)	<code>a += b</code> . <code>a</code> bị thay đổi. Trả về chính <code>a</code> để có thể gán tiếp.
<code>=</code> , <code>&lt;&lt;</code> , <code>&gt;&gt;</code>	Gán hoặc đẩy luồng	<code>PhanSo&amp;</code> , <code>ostream&amp;</code>	Để hỗ trợ chuỗi lệnh (chaining): <code>a = b = c</code> hoặc <code>cout &lt;&lt; a &lt;&lt; b</code> .
<code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&gt;</code>	So sánh	<code>bool</code>	Chỉ trả về đúng/sai.

## 2. Cái bẫy "Giao hoán": $ps + 1$ và $1 + ps$

Đây là lỗi kinh điển khiến sinh viên mất điểm oan.

- **Member function:** Chỉ hoạt động khi đối tượng lớp nằm bên trái.

C++

```
PhanSo a(1, 2);  
PhanSo b = a + 1; // OK: Tương đương a.operator+(1)
```

- **Vấn đề:** Nếu viết  $1 + a$  thì sao? Số  $1$  (int) không có hàm `operator+` nhận `PhanSo`.

C++

```
PhanSo c = 1 + a; // LỖI NẾU DÙNG MEMBER FUNCTION
```

**Giải pháp Senior:** Luôn ưu tiên viết toán tử 2 ngôi (+, -, \*, /) là **Friend Function** để hỗ trợ tính giao hoán.

C++

```
// Trong class PhanSo  
friend PhanSo operator+(const PhanSo& a, const PhanSo& b);  
  
// Triển khai bên ngoài  
PhanSo operator+(const PhanSo& a, const PhanSo& b) {  
    // Logic cộng...  
    // Nhờ constructor chuyển đổi (PhanSo(int t)), số 1 tự động biến thành PhanSo  
    // Nên code này chạy được cả: ps + ps, ps + 1, và 1 + ps  
}
```

## 3. Cái bẫy "Tăng giảm": $++a$ (Tiền tố) và $a++$ (Hậu tố)

Đề thi nâng cao thường yêu cầu: "Cài đặt toán tử  $++$  để tăng giá trị phân số lên 1 đơn vị". Bạn phải cài cả hai loại.

**Tiền tố (++a):** Tăng rồi mới dùng -> Trả về tham chiếu.

**Hậu tố (a++):** Dùng rồi mới tăng -> Trả về tham trị (bản sao cũ) -> Cần tham số `int` giả (dummy) để phân biệt.

```
// 1. Tiền tố (++a)
PhanSo& operator++() {
    tu += mau; // Cộng thêm 1 đơn vị (tử + mẫu)
    return *this; // Trả về chính mình sau khi tăng
}

// 2. Hậu tố (a++) - Chú ý tham số int dummy
PhanSo operator++(int) {
    PhanSo temp = *this; // Lưu lại bản sao cũ
    tu += mau;           // Tăng giá trị hiện tại
    return temp;         // Trả về bản sao cũ (chưa tăng)
}
```

#### 4. Code mẫu "Thần thánh" (Cheat Sheet) cho tờ A4

```
class PhanSo {
private:
    int tu, mau;

    // Hàm phụ trợ (Helper) để rút gọn: Code sạch là phải thế này
    void rutGon() {
        if (mau == 0) return; // Hoặc throw exception
        int ucln = 1;
        // Logic tìm UCLN... (giả sử có hàm gcd)
        // int ucln = std::gcd(tu, mau);
        // tu /= ucln; mau /= ucln;
        if (mau < 0) { tu = -tu; mau = -mau; } // Đưa dấu trừ lên tử
    }
}
```

```

public:
    // Constructor: Hỗ trợ chuyển đổi ngầm định int -> PhanSo
    PhanSo(int t = 0, int m = 1) : tu(t), mau(m) { rutGon(); }

    // 1. IO STREAM (Bắt buộc Friend)
    friend ostream& operator<<(ostream& os, const PhanSo& p) {
        os << p.tu;
        if (p.mau != 1) os << "/" << p.mau;
        return os;
    }

    friend istream& operator>>(istream& is, PhanSo& p) {
        is >> p.tu >> p.mau;
        p.rutGon(); // Nhập xong nhớ rút gọn ngay
        return is;
    }

    // 2. TOÁN TỬ 2 NGÔI (Nên là Friend để hỗ trợ 1 + ps)
    friend PhanSo operator+(const PhanSo& a, const PhanSo& b) {
        PhanSo kq;
        kq.tu = a.tu * b.mau + b.tu * a.mau;
        kq.mau = a.mau * b.mau;
        kq.rutGon();
        return kq;
    }

    // Mẹo: Trừ, Nhân, Chia tương tự.
    // Nhân: tu*tu, mau*mau. Chia: tu*mau, mau*tu.

    // 3. TOÁN TỬ SO SÁNH
    bool operator==(const PhanSo& other) const {
        return (long long)tu * other.mau == (long long)other.tu * mau;
        // Ép kiểu long long tránh tràn số khi nhân chéo
    }

```

```

    }

    bool operator>(const PhanSo& other) const {
        return (double)tu/mau > (double)other.tu/other.mau;
    }

    // 4. TOÁN TỬ GÁN CỘNG (+=) - Member Function
    PhanSo& operator+=(const PhanSo& other) {
        *this = *this + other; // Tái sử dụng toán tử + đã viết
        return *this;
    }

    // 5. TĂNG GIẢM (Optional - Nếu đề hỏi khó)
    PhanSo& operator++(); // Prefix
    PhanSo operator++(int); // Postfix
};

```

## KỸ NĂNG 3: QUẢN LÝ BỘ NHỚ (THE BIG THREE)

Nếu đề bài yêu cầu viết class **MyString**, **IntArray** (Mảng động), hoặc **Vector**... tóm lại là có con trỏ (\*) -> **BẬT CHẾ ĐỘ CẢNH BÁO ĐỎ**.

Em phải viết đủ 3 hàm sau. Thiếu 1 hàm = Trừ 50% số điểm câu đó.

### 1. Bản chất: Tại sao phải "Deep Copy"?

Hãy hình dung:

- **Shallow Copy (Mặc định):** Bạn có một chìa khóa nhà. Bạn copy cho bạn mình một cái chìa khóa y hệt. Cả hai dùng chung một căn nhà. Nếu bạn bán nhà (delete), bạn mình ra đường ở.
- **Deep Copy (Big Three):** Bạn xây một căn nhà mới y hệt, nội thất y hệt. Bạn mình có chìa khóa nhà mới đó. Bạn bán nhà cũ không ảnh hưởng gì tới nhà mới.

Nếu không viết Copy Constructor, C++ sẽ mặc định dùng Shallow Copy (chỉ copy địa chỉ con trỏ). Khi hàm kết thúc, Destructor gọi 2 lần trên cùng 1 vùng nhớ -> Lỗi **Double Free** -> Crash chương trình.



## 2. Kỹ thuật Senior: Refactoring (Tránh lặp code)

Trong code mẫu của bạn, logic "Cấp phát mới & Copy dữ liệu" lặp lại y hệt ở Copy Constructor và Operator=. Đi thi viết 2 lần vừa mỏi tay vừa dễ sai.

**Giải pháp:** Tách logic đó ra thành hàm private tên là `init` hoặc `copyFrom`.

```
class IntArray {
private:
    int* data;
    int size;

    // Hàm phụ trợ: Giúp code gọn gàng, tránh lặp lại (DRY Principle)
    void copyFrom(const IntArray& other) {
        this->size = other.size;
        if (other.size > 0) {
            this->data = new int[other.size];
            for (int i = 0; i < size; i++) {
                this->data[i] = other.data[i];
            }
        } else {
            this->data = nullptr; // Xử lý trường hợp mảng rỗng
        }
    }
};
```



```

    }
}

public:
    IntArray(int n) {
        size = n;
        data = (n > 0) ? new int[n] : nullptr;
    }

    ~IntArray() { delete[] data; } // delete[] nullptr là an toàn, không cần if

    // 2. Copy Constructor: Chỉ cần gọi hàm phụ
    IntArray(const IntArray& other) {
        copyFrom(other);
    }

public:
    IntArray(int n) {
        size = n;
        data = (n > 0) ? new int[n] : nullptr;
    }

    ~IntArray() { delete[] data; } // delete[] nullptr là an toàn, không cần if

    // 2. Copy Constructor: Chỉ cần gọi hàm phụ
    IntArray(const IntArray& other) {
        copyFrom(other);
    }

    // 3. Assignment Operator: Xóa cũ -> Gọi hàm phụ
    IntArray& operator=(const IntArray& other) {
        if (this == &other) return *this; // 1. Chống tự gán

        delete[] data; // 2. Xóa dữ liệu cũ
        copyFrom(other); // 3. Copy dữ liệu mới từ hàm phụ

        return *this;
    }
};

```

### 3. Biến thể nguy hiểm: Class MyString

Nếu đề ra IntArray, bạn chỉ cần loop for. Nhưng nếu đề ra MyString (quản lý char\*), bạn phải dùng thư viện <cstring> (<string.h>).

**Cạm bẫy chết người:** Quên ký tự kết thúc chuỗi `\0` và quên cộng thêm 1 khi cấp phát `new`.

```
class MyString {  
private:  
    char* str;  
public:  
    // Constructor  
    MyString(const char* s = "") {  
        if (s) {  
            // LỖI KINH ĐIỂN: Quên +1 cho ký tự '\0'  
            str = new char[strlen(s) + 1];  
            strcpy(str, s);  
        } else {  
            str = new char[1];  
            str[0] = '\0';  
        }  
    }  
};
```

```
~MyString() { delete[] str; }  
  
MyString(const MyString& other) {  
    str = new char[strlen(other.str) + 1]; // Deep Copy  
    strcpy(str, other.str);  
}  
  
MyString& operator=(const MyString& other) {  
    if (this == &other) return *this;  
  
    delete[] str; // Xóa chuỗi cũ  
  
    // Cấp phát mới và copy  
    str = new char[strlen(other.str) + 1];  
    strcpy(str, other.str);  
}
```

```
    return *this;  
}  
};
```

#### 4. Phân tích sâu: Tại sao phải kiểm tra `if (this == &other)`?

Đây là câu hỏi lấy điểm 10 hoặc câu hỏi vấn đáp.

Giả sử bạn viết `a = a;` (Tự gán). Nếu **không** có dòng `if (this == &other)`:

1. Hàm chạy dòng `delete[] data;` -> Dữ liệu của `a` bị xóa sạch.
2. Sau đó chạy dòng `copy other.data` (nhưng `other` chính là `a`!).
3. Kết quả: Copy dữ liệu rác hoặc chương trình bị sập.

-> **Luôn luôn kiểm tra tự gán trước khi delete.**

#### 5. Checklist sinh tồn cho bài thi

Khi gặp bài toán quản lý bộ nhớ, hãy rà soát danh sách này trên tờ nháp:

1. ☐ Có **Con trỏ** trong Class không? -> Nếu có, bắt buộc viết Big Three.
2. ☐ **Destructor**: Có `delete[]` chưa?
3. ☐ **Copy Constructor**: Có new vùng nhớ mới trước khi copy giá trị không?
4. ☐ **Operator=**:
  - o Có kiểm tra tự gán (`this == &other`) không?
  - o Có `delete[]` dữ liệu cũ không?
  - o Có trả về `*this` không?
5. ☐ **MyString**: Có `strlen(...) + 1` chưa?

## 🚫 VÁ LỖ HỒNG 1: DESIGN PATTERN (Câu 3.3)

(Phần này bắt buộc có trong Câu 3)

### A. Singleton (Đối tượng duy nhất)

- Code chuẩn (Ngắn gọn để chép):

```
C++  
  
class Logger {  
private:  
    static Logger* instance;  
    Logger() {} // 1. Private Constructor  
public:  
    // 2. Chống copy  
    Logger(const Logger&) = delete;  
    void operator=(const Logger&) = delete;  
  
    // 3. Static method lấy instance  
    static Logger* getInstance() {  
        if (instance == nullptr) instance = new Logger();  
        return instance;  
    }  
};  
Logger* Logger::instance = nullptr; // 4. Khởi tạo bên ngoài
```

### B. Composite (Cấu trúc cây)

- **Dấu hiệu nhận biết:** "Thư mục chứa File và Thư mục con", "Tổ chức có Nhân viên và Quản lý (quản lý nhân viên khác)".
- **Từ khóa:** `Component` (Cha chung), `Leaf` (Lá - File), `Composite` (Cành - Folder).
- **Cơ chế:** Dùng `vector<Component*> children` trong lớp `Composite`.

### 3. Thêm: Kỹ thuật `dynamic_cast` (Loại danh sách)

Dùng cho câu hỏi: "Tính lương riêng cho Quản lý trong danh sách nhân viên hỗn hợp".

C++

```
void xuLyRieng(vector<NhanVien*> list) {  
    for (auto nv : list) {  
        // Cố ép kiểu về QuanLy  
        QuanLy* ql = dynamic_cast<QuanLy*>(nv);  
        if (ql != nullptr) {  
            // Nếu thành công -> Đây là Quản lý  
            ql->tinhThuongLanhDao();  
        }  
    }  
}
```

### 4. Thêm: Từ khóa `explicit`

Để tránh bị trừ điểm cú pháp lật vật.

C++

```
class PhanSo {  
public:  
    // Cấm trình biên dịch tự ý đổi số 5 thành PhanSo(5)  
    explicit PhanSo(int tu) { ... }  
};
```