DEVLINUX ACADEMY

Object-Oriented Programming

"Code Your Embeded Future"

devlinux.vn

https://devlinux.vn

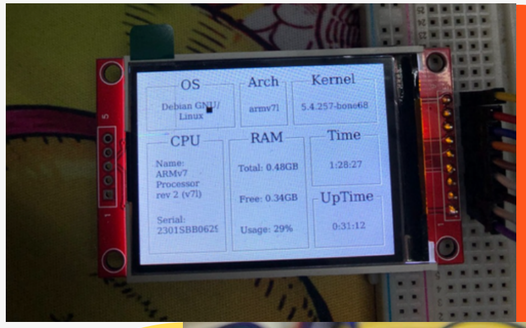
Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097

DEVLINUX ACADEMY

"Code Your Embedded Future"

NỘI DUNG BUỔI HỌC



- I. Nền tảng Hướng đối tượng
- II. Các tính chất OOP
- III. Contructors-Destructors
- IV. Nạp chồng toán tử

https://devlinux.vn

Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097

DEVLINUX ACADEMY

NỀN TẢNG HƯỚNG ĐỐI TƯỢNG

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

"Code Your Embedded Future"

Object-Oriented Programming - OOP

Lập trình Hướng đối tượng (**Object-Oriented Programming - OOP**) không chỉ là một tập hợp các tính năng của ngôn ngữ lập trình; nó là một cuộc cách mạng trong tư duy thiết kế phần mềm.

Trước khi có OOP, lập trình thủ tục (**Procedural Programming**) thống trị, nơi dữ liệu và các hàm xử lý dữ liệu tồn tại riêng rẽ. Điều này dẫn đến các hệ thống phức tạp, khó bảo trì và mở rộng, nơi một thay đổi nhỏ ở cấu trúc dữ liệu có thể gây ra hiệu ứng gợn sóng, đòi hỏi phải sửa đổi hàng loạt hàm liên quan.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Object-Oriented Programming - OOP

Tiêu chí so sánh	Lập trình Hướng Thủ tục (POP)	Lập trình Hướng Đối tượng (OOP)
Trọng tâm	Thủ tục, hàm, thuật toán	Dữ liệu, đối tượng
Cấu trúc chương trình	Chia thành các hàm	Chia thành các đối tượng/lớp
Bảo mật dữ liệu	Thấp, dữ liệu di chuyển tự do	Cao, dữ liệu được đóng gói và bảo vệ
Phạm vi truy cập	Không có công cụ sửa đổi truy cập	Có **public**, **private**, **protected**
Khả năng tái sử dụng	Thấp, khó tái sử dụng hàm	Cao, thông qua kế thừa
Quản lý độ phức tạp	Khó khăn với dự án lớn	Tốt hơn cho dự án lớn, phức tạp

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Object-Oriented Programming - OOP

```c
#include <stdio.h>

// Định nghĩa một cấu trúc (struct) để gom nhóm dữ liệu của hình chữ nhật
// Struct trong C chỉ chứa dữ liệu, không chứa hành vi (hàm)
struct HinhChuNhat {
    int chieu_dai;
    int chieu_rong;
};

// Hàm tính diện tích, nhận vào một struct HinhChuNhat để xử lý
int tinh_dien_tich(struct HinhChuNhat hcn) {
    return hcn.chieu_dai * hcn.chieu_rong;
}

// Hàm tính chu vi, cũng nhận vào một struct HinhChuNhat
int tinh_chu_vi(struct HinhChuNhat hcn) {
    return (hcn.chieu_dai + hcn.chieu_rong) * 2;
}

int main() {
    // 1. Tạo một biến cấu trúc và gán dữ liệu cho nó
    struct HinhChuNhat hcn1;
    hcn1.chieu_dai = 10;
    hcn1.chieu_rong = 5;

    // 2. Gọi các hàm bên ngoài và truyền dữ liệu (biến struct) vào
    int dien_tich = tinh_dien_tich(hcn1);
    int chu_vi = tinh_chu_vi(hcn1);

    printf("Ví dụ bằng C (Hướng thủ tục):\n");
    printf("Diện tích là: %d\n", dien_tich);
    printf("Chu vi là: %d\n", chu_vi);

    return 0;
}
```

```cpp
#include <iostream>

// Định nghĩa một lớp (class)
// Class trong C++ chứa cả dữ liệu (thuộc tính) và hành vi (phương thức)
class HinhChuNhat {
private:
    // Dữ liệu thường được đặt là private để đảm bảo tính đóng gói
    int chieu_dai;
    int chieu_rong;

public:
    // Hàm khởi tạo (Constructor) để gán giá trị khi đối tượng được tạo ra
    HinhChuNhat(int dai, int rong) {
        chieu_dai = dai;
        chieu_rong = rong;
    }

    // Phương thức tính diện tích. Nó đã có sẵn dữ liệu bên trong đối tượng
    int tinh_dien_tich() {
        return chieu_dai * chieu_rong;
    }

    // Phương thức tính chu vi
    int tinh_chu_vi() {
        return (chieu_dai + chieu_rong) * 2;
    }
};

int main() {
    // 1. Tạo một đối tượng từ lớp, dữ liệu được truyền vào ngay lúc tạo
    HinhChuNhat hcn1(10, 5);

    // 2. Gọi phương thức từ chính đối tượng đó.
    // Đối tượng hcn1 tự biết cách tính toán dựa trên dữ liệu nó đang giữ.
    int dien_tich = hcn1.tinh_dien_tich();
    int chu_vi = hcn1.tinh_chu_vi();

    std::cout << "Ví dụ bằng C++ (Hướng đối tượng):\n";
    std::cout << "Diện tích là: " << dien_tich << std::endl;
    std::cout << "Chu vi là: " << chu_vi << std::endl;

    return 0;
}
```


https://devlinux.vn

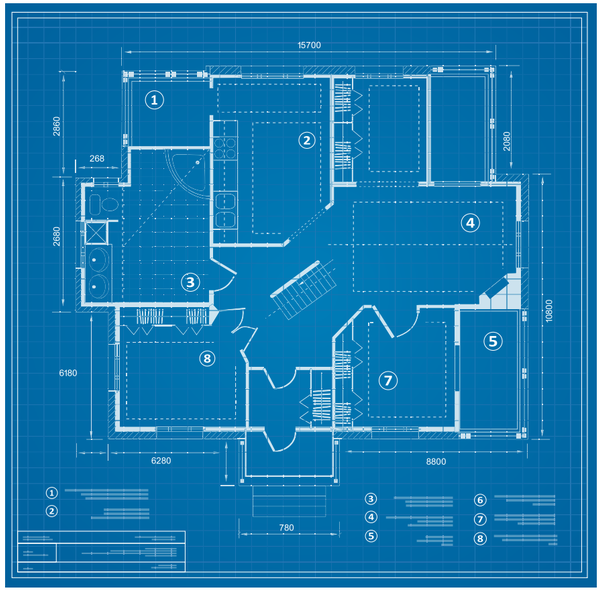
Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097

Lớp (Class)

Trong OOP, **Lớp (Class)** là khái niệm nền tảng, được định nghĩa như một khuôn mẫu (template) hay một bản thiết kế (blueprint). Nó mô tả cấu trúc và hành vi chung cho một tập hợp các đối tượng có cùng bản chất. Một lớp định nghĩa hai thành phần chính:

- **Thuộc tính (Attributes/Properties)**: Là các biến thành viên dùng để lưu trữ dữ liệu, đại diện cho trạng thái của đối tượng. Ví dụ, một lớp Student có thể có các thuộc tính như name và age.
- **Phương thức (Methods/Behaviors)**: Là các hàm thành viên định nghĩa các hành động mà đối tượng có thể thực hiện. Các phương thức này thường thao tác trên dữ liệu của chính đối tượng đó. Ví dụ, lớp Student có thể có phương thức display() để hiển thị thông tin.



Về bản chất, **class** là một thực thể logic, một định nghĩa kiểu dữ liệu do người dùng tạo ra. Nó không chiếm bất kỳ không gian bộ nhớ nào cho đến khi một **đối tượng** cụ thể được tạo ra từ nó.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Đối tượng (Object)

Đối tượng (Object) là một thể hiện (instance) cụ thể của một lớp. Nếu lớp là bản thiết kế, thì đối tượng chính là sản phẩm được tạo ra từ bản thiết kế đó. Mỗi đối tượng có trạng thái (state) và hành vi (behavior) riêng, được xác định bởi lớp của nó. Trạng thái của một đối tượng là tập hợp các giá trị của các thuộc tính tại một thời điểm nhất định.

Không giống như lớp, đối tượng là một thực thể vật lý, nó chiếm một vùng không gian trong bộ nhớ máy tính khi được tạo ra. Các đối tượng trong lập trình tương ứng với các thực thể trong thế giới thực, chẳng hạn như một chiếc xe hơi cụ thể, một sinh viên cụ thể, hay một tài khoản ngân hàng cụ thể.

DEVLINUX ACADEMY

CÁC TÍNH CHẤT OOP

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

"Code Your Embedded Future"

Khai báo Class - Tạo ra "Bản thiết kế"

```cpp
#include <iostream>
#include <string>

// --- STEP 1: DECLARE THE CLASS (Create the blueprint) ---
class Car {
private:
    // 1. Attributes: Characteristics of the object.
    // We make these private to protect the data from arbitrary changes from outside.
    std::string brand;
    std::string color;
    int year;

public:
    // 2. Methods: Behaviors of the object.
    // We make these public so they can be called from outside the class.
    void displayInfo() {
        std::cout << "--- Car Info ---" << std::endl;
        std::cout << "Brand: " << brand << std::endl;
        std::cout << "Color: " << color << std::endl;
        std::cout << "Year: " << year << std::endl;
    }
}; // End of class declaration
```

private: Các thành viên trong mục này chỉ có thể được truy cập từ bên trong class. Đây là nơi an toàn để lưu trữ dữ liệu.

public: Các thành viên trong mục này có thể được truy cập từ bất cứ đâu, giúp đối tượng giao tiếp với thế giới bên ngoài.

Tính Đóng gói (Encapsulation)

Tính đóng gói là nguyên tắc gói gọn dữ liệu (thuộc tính) và các phương thức xử lý dữ liệu đó vào trong một đơn vị duy nhất là Lớp. Mục đích chính của nó là:

- **Che giấu dữ liệu (Data Hiding)**, tức là bảo vệ trạng thái nội tại của một đối tượng khỏi sự truy cập và thay đổi trực tiếp từ bên ngoài.

Tính đóng gói được thực hiện thông qua các công cụ sửa đổi truy cập (access modifiers):

- **private**: Các thành viên được khai báo là private chỉ có thể được truy cập từ bên trong chính lớp đó. Đây là mức độ bảo vệ cao nhất và là lựa chọn mặc định được khuyến nghị cho các thuộc tính để đảm bảo tính toàn vẹn dữ liệu.
- **protected**: Các thành viên protected có thể được truy cập bên trong lớp đó và bởi các lớp con kế thừa từ nó.
- **public**: Các thành viên public có thể được truy cập từ bất kỳ đâu trong chương trình.

Để cho phép thế giới bên ngoài **tương tác một cách có kiểm soát với dữ liệu private**, người ta thường cung cấp các phương thức public được gọi là **Getters** và **Setters**. Phương thức getter dùng để truy xuất (đọc) giá trị của một thuộc tính, còn phương thức setter dùng để thay đổi (ghi) giá trị, thường đi kèm với logic kiểm tra tính hợp lệ của dữ liệu đầu vào

Tính Kế thừa (Inheritance)

Tính kế thừa cho phép một lớp mới (lớp con) thừa hưởng các thuộc tính và phương thức từ một lớp đã tồn tại (lớp cha). Mối quan hệ này được gọi là quan hệ "IS-A" (một - một), ví dụ: một **Dog** "là một" **Animal**.

Mục đích chính của kế thừa là để tái sử dụng mã nguồn, tránh lặp lại mã và tạo ra một hệ thống phân cấp các lớp có tổ chức, logic. Lớp con có thể sử dụng lại các chức năng của lớp cha, đồng thời có thể mở rộng bằng cách thêm các thuộc tính, phương thức mới hoặc thay đổi hành vi của phương thức đã có.

Các loại kế thừa:

- **Đơn kế thừa (Single Inheritance)**: Một lớp con chỉ kế thừa từ một lớp cha duy nhất.
- **Đa kế thừa (Multiple Inheritance)**: Một lớp con có thể kế thừa từ nhiều lớp cha.
- **Kế thừa đa cấp (Multilevel Inheritance)**: Một chuỗi kế thừa, ví dụ: lớp C kế thừa từ lớp B, và lớp B kế thừa từ lớp A.
- **Kế thừa phân cấp (Hierarchical Inheritance)**: Nhiều lớp con cùng kế thừa từ một lớp cha duy nhất.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Tính Đa hình (Polymorphism)

"Polymorphism" có nguồn gốc từ tiếng Hy Lạp, có nghĩa là "nhiều hình dạng". Trong OOP, tính đa hình cho phép một đối tượng có thể được đối xử như một đối tượng của lớp cha nó, nhưng khi một phương thức được gọi, hành vi thực thi sẽ là của lớp con. Nói cách khác, cùng một hành động có thể được thực hiện theo nhiều cách khác nhau tùy thuộc vào đối tượng thực thi nó.

Hai loại chính:

- **Đa hình tại thời điểm biên dịch (Compile-time Polymorphism)**: Còn được gọi là nạp chồng phương thức (**Method Overloading**). Trong cùng một lớp, có thể có nhiều phương thức cùng tên nhưng khác nhau về danh sách tham số (số lượng tham số hoặc kiểu dữ liệu của chúng). Trình biên dịch sẽ quyết định phương thức nào được gọi dựa trên các đối số được truyền vào tại thời điểm biên dịch.
- **Đa hình tại thời điểm chạy (Run-time Polymorphism)**: Còn được gọi là ghi đè phương thức (**Method Overriding**). Một lớp con cung cấp một triển khai cụ thể cho một phương thức đã được định nghĩa ở lớp cha. Quyết định về việc phương thức của lớp nào (cha hay con) sẽ được thực thi chỉ được đưa ra tại thời điểm chạy, dựa trên kiểu đối tượng thực tế mà biến tham chiếu đang trỏ tới.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Tính Trừu tượng (Abstraction)

Tính trừu tượng là quá trình che giấu các chi tiết triển khai phức tạp bên trong và chỉ hiển thị các chức năng thiết yếu, cần thiết cho người dùng. Nó giúp giảm độ phức tạp của hệ thống bằng cách cho phép chúng ta tập trung vào **một đối tượng làm "cái gì"**, thay vì **"như thế nào"** nó làm điều đó.

Tính trừu tượng thường được triển khai bằng hai cơ chế chính:

- **Lớp trừu tượng (Abstract Class):** Là một lớp không thể được dùng để tạo đối tượng trực tiếp (không thể khởi tạo). Nó được khai báo với từ khóa abstract và có thể chứa cả các phương thức trừu tượng (phương thức không có thân hàm) và các phương thức thông thường. Bất kỳ lớp con nào kế thừa từ một lớp trừu tượng đều phải cung cấp triển khai cho tất cả các phương thức trừu tượng của lớp cha.
- **Giao diện (Interface):** Là một cấu trúc hoàn toàn trừu tượng, về cơ bản chỉ định nghĩa một "hợp đồng" về các hành vi mà một lớp phải thực hiện. Một lớp có thể triển khai (implements) nhiều giao diện.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Nguyên tắc Thiết kế Hướng đối tượng SOLID

SOLID là một từ viết tắt cho năm nguyên tắc thiết kế cơ bản trong lập trình hướng đối tượng, được đề xuất và phổ biến bởi Robert C. Martin (còn được biết đến với biệt danh "Uncle Bob"). Đây không phải là các quy tắc cứng nhắc mà là những kim chỉ nam giúp lập trình viên viết ra những đoạn mã dễ hiểu, linh hoạt, dễ bảo trì và có khả năng mở rộng theo thời gian.

- **S - Single Responsibility Principle (Nguyên tắc Trách nhiệm Đơn lẻ)**
  - Một lớp chỉ nên có một và chỉ một lý do để thay đổi. Điều này có nghĩa là một lớp chỉ nên chịu trách nhiệm cho một chức năng hoặc một khía cạnh duy nhất của chương trình.
  - Nguyên tắc này giúp tạo ra các lớp có tính gắn kết cao (high cohesion) và khớp nối lỏng (low coupling). Khi các chức năng được tách biệt rõ ràng, mã nguồn trở nên dễ hiểu, dễ kiểm thử và dễ bảo trì hơn, vì một thay đổi ở một chức năng sẽ không ảnh hưởng đến các chức năng khác.
- **O - Open/Closed Principle (Nguyên tắc Đóng/Mở)**
  - Các thực thể phần mềm (lớp, module, hàm) nên "mở" cho việc mở rộng (open for extension) nhưng "đóng" cho việc sửa đổi (closed for modification).
  - Nguyên tắc này cho phép thêm các chức năng mới vào hệ thống mà không cần phải thay đổi mã nguồn hiện có đã được kiểm thử và đang hoạt động ổn định. Điều này giảm thiểu rủi ro gây ra lỗi và làm cho hệ thống linh hoạt hơn.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Nguyên tắc Thiết kế Hướng đối tượng SOLID

- **L - Liskov Substitution Principle (Nguyên tắc Thay thế Liskov)**
  - Các đối tượng của một lớp con phải có thể thay thế được cho các đối tượng của lớp cha mà không làm thay đổi tính đúng đắn của chương trình.
  - Nguyên tắc này đảm bảo rằng tính kế thừa được sử dụng một cách hợp lý và logic. Nó giúp duy trì sự nhất quán trong hệ thống phân cấp lớp, đảm bảo rằng các lớp con không phá vỡ "hợp đồng" đã được thiết lập bởi lớp cha, từ đó giúp tính đa hình hoạt động đúng như mong đợi.
- **I - Interface Segregation Principle (Nguyên tắc Phân tách Giao diện)**
  - Thay vì sử dụng một giao diện (interface) lớn và cồng kềnh, nên tách nó thành nhiều giao diện nhỏ và chuyên biệt hơn. Các client không nên bị buộc phải phụ thuộc vào các phương thức mà chúng không sử dụng.
  - Nguyên tắc này giúp giảm sự phụ thuộc không cần thiết, làm cho hệ thống trở nên linh hoạt hơn. Khi một client chỉ phụ thuộc vào những gì nó thực sự cần, mã nguồn sẽ dễ tái sử dụng và ít bị ảnh hưởng bởi những thay đổi ở các phần không liên quan.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Nguyên tắc Thiết kế Hướng đối tượng SOLID

- **D - Dependency Inversion Principle (Nguyên tắc Đảo ngược Phụ thuộc)**
  - Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả hai nên phụ thuộc vào sự trừu tượng (abstractions).
  - Sự trừu tượng không nên phụ thuộc vào chi tiết. Chi tiết nên phụ thuộc vào sự trừu tượng.
  - Nguyên tắc này giúp giảm đáng kể sự khớp nối giữa các thành phần trong hệ thống. Bằng cách phụ thuộc vào các giao diện trừu tượng thay vì các lớp cụ thể, mã nguồn trở nên linh hoạt, dễ dàng thay thế các thành phần và đặc biệt là dễ dàng kiểm thử (testability) hơn.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

DEVLINUX ACADEMY

CONSTRUCTORS-DESTRUCTORS

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

"Code Your Embedded Future"

Vòng Đời Của Một Đối Tượng

Mọi đối tượng trong chương trình của bạn đều có một vòng đời:

- **Sinh ra (Creation)**: Khi đối tượng được tạo, Hàm khởi tạo (Constructor) được gọi để thiết lập trạng thái ban đầu, cấp phát tài nguyên (bộ nhớ, file,...).
- **Sống (Lifetime)**: Đối tượng tồn tại và các phương thức của nó được gọi để thực hiện các công việc.
- **Chết đi (Destruction)**: Khi đối tượng sắp bị xóa khỏi bộ nhớ, Hàm hủy (Destructor) được gọi để "dọn dẹp" sạch sẽ mọi thứ mà đối tượng đã tạo ra.

Hàm khởi tạo mặc định (Default Constructor)

Hàm khởi tạo mặc định là một hàm khởi tạo có thể được gọi mà không cần bất kỳ đối số nào. Điều này có thể xảy ra theo hai cách:

- **Không có tham số:** Hàm khởi tạo được định nghĩa không có tham số nào.
- **Tất cả tham số đều có giá trị mặc định:** Hàm khởi tạo có tham số, nhưng tất cả chúng đều được gán giá trị mặc định.

Sự "Can Thiệp" Của Trình Biên Dịch

- **Hàm Khởi Tạo Mặc Định do Trình biên dịch Tự động Tạo**: Nếu bạn không định nghĩa bất kỳ hàm khởi tạo nào trong lớp của mình, trình biên dịch C++ sẽ tự động tạo ra một hàm khởi tạo mặc định public và inline cho bạn. Hàm khởi tạo này được gọi là implicitly-defined default constructor.
- Hàm khởi tạo mặc định do trình biên dịch tạo ra sẽ không khởi tạo giá trị cho các thành viên kiểu nguyên thủy (như int, float, char, con trỏ). Các biến này sẽ mang giá trị rác. Tuy nhiên, các thành viên là đối tượng của lớp khác sẽ được khởi tạo bằng hàm khởi tạo mặc định của lớp đó.

Hàm khởi tạo mặc định (Default Constructor)

Khi Nào Trình biên dịch KHÔNG Tự động Tạo?

- Đây là một quy tắc vàng trong C++**:** Nếu bạn tự định nghĩa BẤT KỲ một hàm khởi tạo nào khác (ví dụ: hàm khởi tạo có tham số), trình biên dịch sẽ KHÔNG tự động tạo hàm khởi tạo mặc định nữa.

Cú pháp Hiện đại (C++11 trở đi) **= default** và **= delete**

- **Yêu cầu Trình biên dịch Tạo**: = default
  - Nếu bạn đã định nghĩa một hàm khởi tạo khác nhưng vẫn muốn trình biên dịch tạo ra hàm khởi tạo mặc định một cách tự động, bạn có thể yêu cầu tường minh bằng = default.
- **Vô hiệu hóa Hàm khởi tạo Mặc định:** = delete
  - Đôi khi, bạn muốn ngăn chặn việc một đối tượng được tạo ra mà không có tham số. Trong trường hợp này, bạn có thể vô hiệu hóa (xóa) hàm khởi tạo mặc định bằng = delete.

Hàm Khởi Tạo Có Tham Số

Hàm khởi tạo có tham số là một hàm khởi tạo chấp nhận một hoặc nhiều đối số (tham số) khi được gọi. Mục đích của nó là sử dụng các đối số này để khởi tạo các thuộc tính của đối tượng ngay tại thời điểm tạo ra nó.

Đặc điểm chính:

- **Cùng tên với lớp**: Giống như mọi hàm khởi tạo khác.
- **Không có kiểu trả về**: Kể cả void.
- **Có danh sách tham số**: Đây là điểm khác biệt cốt lõi so với hàm khởi tạo mặc định.

Tại sao cách này lại tốt hơn?

- **Hiệu quả hơn**: Nó thực hiện việc khởi tạo trực tiếp, chỉ tốn một bước. Cách gán trong thân hàm thực hiện hai bước: khởi tạo mặc định trước, sau đó mới gán giá trị mới. Đối với các kiểu dữ liệu phức tạp, điều này tiết kiệm được một lệnh gọi hàm không cần thiết.
- **Bắt buộc trong một số trường hợp**: Nếu thuộc tính của bạn là một hằng số (const) hoặc một tham chiếu (&), bạn bắt buộc phải khởi tạo chúng trong danh sách này.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Danh sách Khởi tạo Thành viên (Member Initializer List)

```
Car::Car(std::string newBrand, std::string newColor, int newYear)
    // This is the member initializer list.
    // It initializes members directly instead of assigning them in the body.
    // This is more efficient, especially for complex types.
    : brand(newBrand), color(newColor), year(newYear)
{
    td::cout << "-> Parameterized constructor called to create a " << brand << std::endl;
    // The constructor body can be empty or contain other logic.
}
```

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Hàm Khởi Tạo Sao Chép

Hàm khởi tạo sao chép là một hàm khởi tạo đặc biệt, có nhiệm vụ tạo ra một đối tượng mới bằng cách sao chép dữ liệu từ một đối tượng đã tồn tại của cùng một lớp.

Khi Nào Hàm Khởi Tạo Sao Chép Được Gọi?

- **Khi khai báo và khởi tạo một đối tượng từ một đối tượng khác cùng loại**
- **Khi truyền một đối tượng vào hàm bằng giá trị (pass-by-value)**
- **Khi trả về một đối tượng từ hàm bằng giá trị (return-by-value)**

Sao Chép Nông vs. Sao Chép Sâu

Hàm Khởi Tạo Sao Chép Mặc Định (Sao Chép Nông - Shallow Copy)

- Nếu bạn không tự viết, trình biên dịch sẽ cung cấp một hàm khởi tạo sao chép mặc định. Hàm này thực hiện một việc rất đơn giản: sao chép từng thành viên một (member-wise copy).
- Với các kiểu dữ liệu đơn giản (int, double, bool,...) và các lớp an toàn (như std::string), việc này hoàn toàn ổn.

Vấn đề bắt đầu khi lớp của bạn quản lý tài nguyên động, đặc biệt là con trỏ thô.

- Khi hàm khởi tạo sao chép mặc định hoạt động, nó chỉ sao chép giá trị của con trỏ name. Tức là cả hai đối tượng (gốc và bản sao) sẽ có hai con trỏ name cùng trỏ đến **MỘT vùng nhớ duy nhất**.

Hậu quả:

- Lỗi Double Free
- Dữ liệu bị thay đổi không mong muốn

```
class Student {
private:
    char* name; // heap pointer
public:
    // ...
};
```

Sao Chép Nông vs. Sao Chép Sâu

Tự Viết Hàm Khởi Tạo Sao Chép (Sao Chép Sâu - Deep Copy)

- Để giải quyết vấn đề trên, chúng ta phải tự định nghĩa lại hàm khởi tạo sao chép để thực hiện Sao chép sâu.

Sao chép sâu có nghĩa là:

- Tạo ra một vùng nhớ mới cho đối tượng bản sao.
- Sao chép nội dung từ vùng nhớ của đối tượng gốc sang vùng nhớ mới này.

```
class Student {
private:
    char* name; // heap pointer
public:
    // ...
};
```

Hàm Khởi Tạo Ủy Quyền

Delegating Constructor là một tính năng cho phép một hàm khởi tạo có thể gọi một hàm khởi tạo khác trong cùng một lớp. Một hàm khởi tạo sẽ "ủy quyền" (delegate) công việc cho một hàm khởi tạo khác.

- **Không thể vừa ủy quyền vừa khởi tạo thành viên**: Một hàm khởi tạo nếu đã ủy quyền cho một hàm khởi tạo khác thì không được phép khởi tạo bất kỳ thành viên nào trong danh sách khởi tạo của chính nó. Toàn bộ việc khởi tạo phải do hàm khởi tạo mục tiêu đảm nhiệm.
- **Thứ tự thực thi**: Hàm khởi tạo mục tiêu (target) sẽ chạy xong hoàn toàn, sau đó phần thân của hàm khởi tạo ủy quyền (delegating) mới được thực thi.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Khi Nào Hàm Hủy Được Gọi?

Việc này phụ thuộc vào cách đối tượng được tạo ra:

- **Đối tượng cục bộ (trên stack)**: Hàm hủy được gọi tự động khi đối tượng đi ra khỏi phạm vi (scope) của nó.
- **Đối tượng động (trên heap)**: Hàm hủy CHỈ được gọi khi bạn gọi toán tử delete (cho đối tượng đơn) hoặc delete[] (cho mảng đối tượng) một cách tường minh.
- **Đối tượng toàn cục hoặc static**: Hàm hủy được gọi khi chương trình kết thúc (sau khi hàm main trả về).

The Rule of Three

Đây là một quy tắc kinh điển trong C++:

- Nếu một lớp cần bạn tự định nghĩa một trong ba phương thức sau:
  - Hàm hủy (Destructor)
  - Hàm khởi tạo sao chép (Copy Constructor)
  - Toán tử gán sao chép (Copy Assignment Operator)

thì rất có thể nó cần cả ba.

Lý do là nhu cầu tự định nghĩa một trong ba thường xuất phát từ việc quản lý tài nguyên động, và việc này ảnh hưởng đến cả ba thao tác: hủy, khởi tạo-sao-chép, và gán-sao-chép.

(*Trong C++11 trở đi, quy tắc này được mở rộng thành Quy Tắc Số Năm, bổ sung thêm Move Constructor và Move Assignment Operator.*)

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Đối Tượng Động

Một đối tượng động là một đối tượng mà bộ nhớ của nó được cấp phát trên Heap tại thời điểm chương trình đang chạy (runtime).

Đặc điểm quan trọng nhất:

- Vòng đời của một đối tượng động không bị ràng buộc bởi phạm vi (scope). Nó sẽ tồn tại từ lúc bạn tạo ra nó bằng new cho đến khi và chỉ khi bạn hủy nó một cách tường minh bằng delete.
- Nó có thể "sống sót" ngay cả khi hàm tạo ra nó đã kết thúc. Đây vừa là sức mạnh, vừa là nguồn gốc của nhiều lỗi nguy hiểm.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Cú Pháp: new và delete

Toán tử new (Để tạo đối tượng)

- Yêu cầu hệ điều hành cấp phát một vùng nhớ đủ lớn trên Heap.
- Gọi hàm khởi tạo (constructor) của đối tượng để thiết lập vùng nhớ đó.
- Trả về một con trỏ trỏ đến địa chỉ của vùng nhớ vừa được cấp phát.

Toán tử delete (Để hủy đối tượng)

- Toán tử delete làm hai việc
- Gọi hàm hủy (destructor) của đối tượng để dọn dẹp tài nguyên
- Trả lại vùng nhớ cho hệ điều hành

Lưu ý: Dùng delete cho mảng cấp phát bằng new[] hoặc delete[] cho đối tượng đơn sẽ gây ra lỗi không xác định (undefined behavior)!

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Con Trỏ Thông Minh (Smart Pointers)

Nhận thấy sự nguy hiểm của con trỏ thô (raw pointers), C++11 đã giới thiệu con trỏ thông minh (std::unique_ptr, std::shared_ptr).

- Đây là các lớp bao bọc (wrapper classes) con trỏ thô.
- Chúng tự động gọi delete khi đi ra khỏi phạm vi, nhờ vào cơ chế hàm hủy (RAII).
- Chúng kết hợp sự linh hoạt của Heap với sự an toàn của Stack.

Con trỏ this

Con trỏ this là một từ khóa đặc biệt trong C++. Bên trong một phương thức thành viên (member function) không phải static của một lớp, **this là một con trỏ trỏ đến chính đối tượng đã gọi phương thức đó.**

Hãy tưởng tượng bạn có nhiều đối tượng của cùng một lớp:

- Car car1;
- Car car2;

Khi bạn gọi car1.displayInfo();, làm thế nào để phương thức displayInfo() biết nó phải hiển thị thông tin của car1 chứ không phải car2?

this là một con trỏ trỏ đến chính đối tượng đã gọi phương thức đó:

- Khi **car1.displayInfo()** được gọi, bên trong displayInfo, this sẽ tương đương với **&car1**.
- Khi **car2.displayInfo()** được gọi, bên trong displayInfo, this sẽ tương đương với **&car2**.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Các Trường Hợp Sử Dụng Phổ Biến

Trong thực tế, bạn sẽ thường xuyên sử dụng this trong các trường hợp sau:

- **Giải Quyết Sự Nhập Nhằng về Tên (Disambiguation)**
- **Trả về Chính Đối Tượng Hiện Tại để "Gọi Chuỗi" (Method Chaining)**
- **Chuyển Đối Tượng Hiện Tại cho một Hàm Khác**

DEVLINUX ACADEMY

NẠP CHỒNG TOÁN TỬ

Hàm Bạn (Friend Functions) - Phá Vỡ Quy Tắc Đóng Gói

Một trong những trụ cột của Lập trình Hướng đối tượng (OOP) là **Tính Đóng Gói (Encapsulation)**. Nguyên tắc này nói rằng dữ liệu nội bộ của một đối tượng (các thành viên private và protected) phải được bảo vệ khỏi sự truy cập trực tiếp từ bên ngoài. Bất kỳ tương tác nào cũng phải thông qua một giao diện public có kiểm soát.

Tuy nhiên, trong một số trường hợp đặc biệt, có những hàm không phải là thành viên của lớp nhưng lại có mối quan hệ mật thiết với lớp đó và cần truy cập vào dữ liệu private để hoạt động hiệu quả.

C++ cung cấp một cơ chế để "nới lỏng" quy tắc đóng gói một cách có kiểm soát: **từ khóa friend**. Nó cho phép một lớp "tin tưởng" và cấp cho một hàm hoặc một lớp khác "chìa khóa" để truy cập vào các thành viên riêng tư của mình.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Hàm Bạn (Friend Functions) Là Gì?

Một **hàm bạn** của một lớp là một **hàm không phải là phương thức thành viên** của lớp đó, nhưng được cấp quyền truy cập vào các thành viên **private** và **protected** của lớp.

Đặc điểm cốt lõi:

- Nó là một hàm thông thường (hàm toàn cục hoặc là phương thức của một lớp khác).
- Nó không có con trỏ this của lớp mà nó làm bạn.
- Nó phải được truyền đối tượng một cách tường minh qua tham số để có thể tác động lên đối tượng đó.
- Việc "làm bạn" là đơn phương. Nếu hàm f() là bạn của lớp A, điều đó không có nghĩa lớp A có quyền truy cập vào các thành viên riêng tư của hàm f() (nếu có).

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Hàm Bạn (Friend Functions) Là Gì?

```cpp
class MyClass {
private:
    int secretValue;

public:
    MyClass(int val) : secretValue(val) {}

    // Step 1: Declare the friend function inside the class
    friend void showSecret(const MyClass& obj);
};

// Step 2: Define the friend function outside the class
void showSecret(const MyClass& obj) {
    // Because showSecret is a friend of MyClass,
    // it can directly access the private member 'secretValue'.
    std::cout << "The secret value is: " << obj.secretValue << std::endl;
}
```

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Khi Nào Nên Dùng Friend?

Nạp chồng toán tử (Operator Overloading)

- Đây là ứng dụng phổ biến và quan trọng nhất. Đặc biệt là các toán tử hai ngôi như +, -, * và các toán tử xuất/nhập <<, >>.

Các hàm "cầu nối" giữa hai lớp khác nhau:

- Khi một hàm cần truy cập vào dữ liệu riêng tư của hai lớp hoàn toàn khác nhau. Việc đặt hàm này làm thành viên của một trong hai lớp cũng không giải quyết được vấn đề. Giải pháp là làm cho nó trở thành friend của cả hai lớp.

Nạp Chồng Toán Tử (Operator Overloading).

Giúp Đối Tượng "Nói" Ngôn Ngữ Tự Nhiên

- Trình biên dịch không hiểu phải làm gì khi bạn dùng toán tử + hay << với các đối tượng Point
- **Nạp chồng toán tử** là một tính năng mạnh mẽ của C++ cho phép chúng ta định nghĩa lại ý nghĩa của các toán tử này khi chúng được dùng với các kiểu dữ liệu do người dùng định nghĩa.

```
int a = 5, b = 10;
int sum = a + b; // Completely natural and easy
std::cout << "Sum: " << sum << std::endl;
```

```
Point p1(10, 20);
Point p2(5, 7);
Point p3 = p1 + p2; // COMPILE ERROR!
std::cout << p3;    // COMPILE ERROR!
```

Nạp Chồng Toán Tử (Operator Overloading).

Về bản chất, nạp chồng toán tử chỉ là một **"lớp vỏ cú pháp" (syntactic sugar)** cho một lời gọi hàm. Khi bạn viết p1 + p2, trình biên dịch sẽ tìm kiếm một hàm đặc biệt, có thể là p1.operator+(p2) hoặc operator+(p1, p2).

Cú pháp chung:

```
return_type operator<symbol>(parameters);
```

Các Loại Toán Tử Phổ Biến và Quy Tắc

- Toán tử số học (+, -, *, /): Thường trả về một đối tượng mới.
- Toán tử gán (=, +=, -=): Phải là phương thức thành viên.
- Toán tử so sánh (==, !=, <, >): Thường là phương thức thành viên, trả về bool.
- Toán tử xuất/nhập (<<, >>): Phải là hàm tự do friend.
- Toán tử tăng/giảm (++, --): Có hai dạng, tiền tố (++obj) và hậu tố (obj++).

Những điều cần tránh:

- Không nạp chồng toán tử với ý nghĩa khó hiểu (ví dụ + để trừ).
- Không thể thay đổi độ ưu tiên của toán tử.
- Không thể tạo ra toán tử mới.
- Không nên nạp chồng &&, ||, và , vì bạn sẽ làm mất đi các hành vi đặc biệt của chúng.

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Getters và Setters

Chúng ta đã học về Tính Đóng Gói (Encapsulation), và biết rằng việc khai báo các thành viên dữ liệu là private là một thông lệ tốt. Tại sao?

- **Bảo vệ dữ liệu**: Ngăn chặn việc thay đổi dữ liệu một cách tùy tiện từ bên ngoài, có thể dẫn đến trạng thái không hợp lệ của đối tượng.
- **Che giấu chi tiết triển khai**: Cho phép bạn thay đổi cách lưu trữ dữ liệu bên trong mà không ảnh hưởng đến code bên ngoài đang sử dụng lớp của bạn.

Getter là một phương thức public có nhiệm vụ duy nhất là trả về giá trị của một thuộc tính private.
Setter là một phương thức public có nhiệm vụ thay đổi giá trị của một thuộc tính private.

DEVLINUX ACADEMY

Thanks for Your Attention!

Liên hệ với chúng tôi

https://devlinux.vn

Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097