VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Kỹ Thuật Lập Trình (Cơ bản và nâng cao C++)

KTLT2 - HK242

TASK 7 Nền móng OOP - Xây dựng thế giới đối tượng

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 2/2025

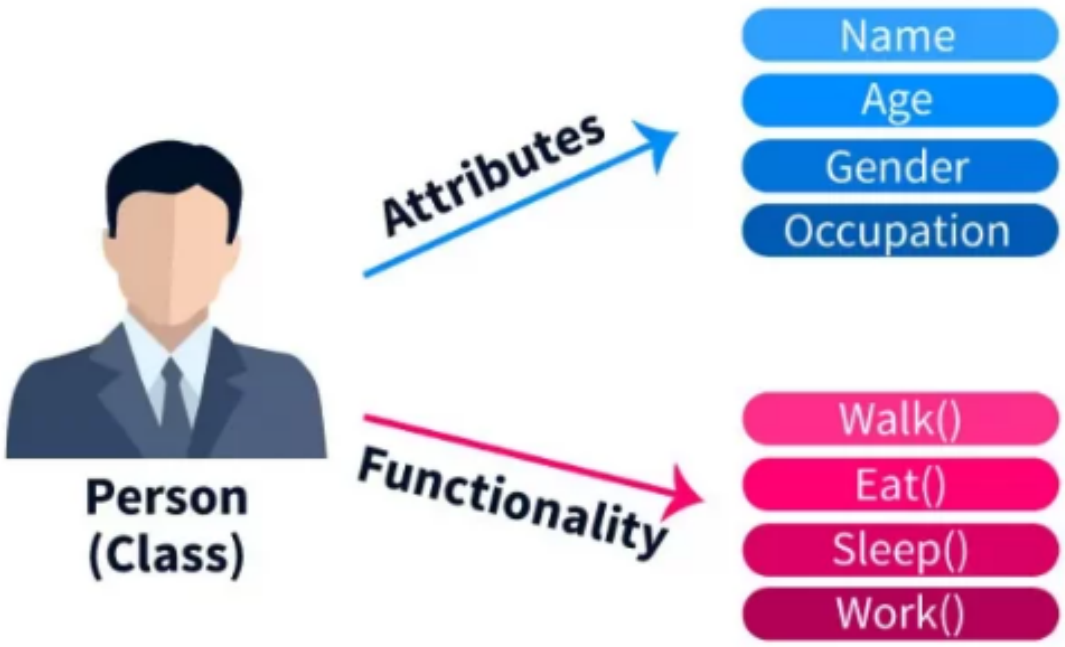
Mục lục

1 Lớp (Class) và đối tượng (Object)

1.1 Class

Định nghĩa

- **Class (lớp)** là một khuôn mẫu (blueprint) để tạo ra **đối tượng (object)**.
- Nó giúp định nghĩa **thuộc tính (dữ liệu)** và **phương thức (hành vi)** mà đối tượng có thể có.
- Trong lập trình hướng đối tượng (OOP), class giúp tổ chức và quản lý dữ liệu theo cách chặt chẽ, dễ hiểu



Ví dụ thực tế:

- Hãy tưởng tượng **class là một bản thiết kế** của một Person.
- Dựa vào bản thiết kế này, ta có thể tạo ra nhiều Persion (đối tượng), mỗi người có thể có Name, Age, ... khác nhau nhưng vẫn dựa trên cùng một cấu trúc.

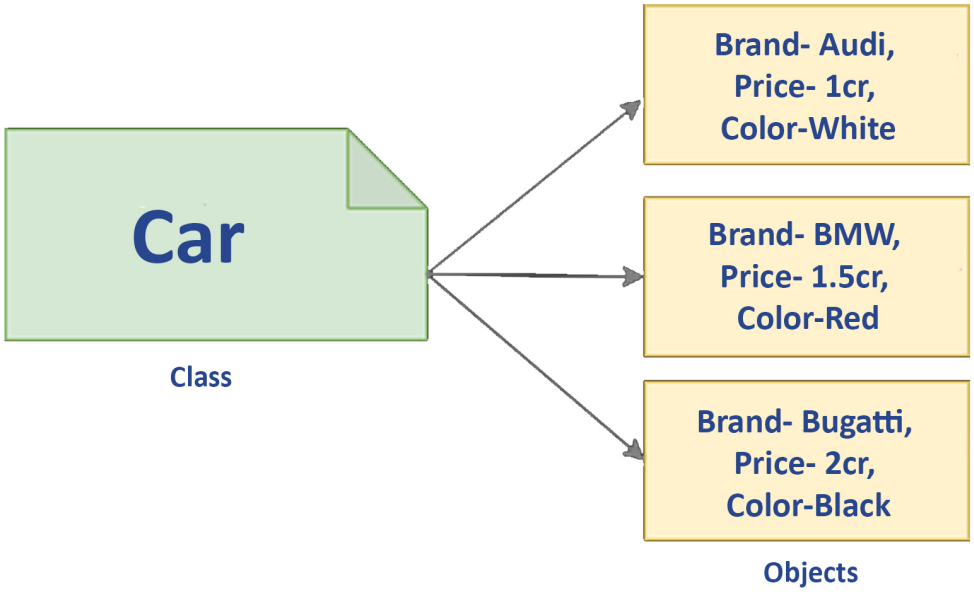
Vì sao lại dùng Class?

Class (lớp) là một khái niệm quan trọng trong lập trình hướng đối tượng (OOP), giúp lập trình viên tổ chức và quản lý dữ liệu một cách hiệu quả. Dưới đây là những lý do chính khiến chúng ta sử dụng **class** thay vì chỉ dùng kiểu lập trình thủ tục (procedural programming).

1. **Đóng gói dữ liệu (Encapsulation)** Giúp bảo vệ dữ liệu và chỉ cho phép truy cập thông qua các phương thức được xác định.
2. **Tái sử dụng mã nguồn (Code Reusability)** Giảm lặp lại code, giúp viết ít mà vẫn hiệu quả.
3. **Dễ bảo trì và mở rộng (Maintainability & Scalability)** Khi chương trình lớn lên, class giúp tổ chức code gọn gàng, dễ tìm kiếm và sửa lỗi.
4. **Tính trừu tượng (Abstraction)** Che giấu chi tiết không cần thiết, chỉ cung cấp giao diện cần thiết.

1.2 Object

Object (đối tượng) là một thực thể cụ thể của **class**. Nếu **class** là bản thiết kế, thì **object** là phiên bản được tạo ra từ bản thiết kế đó.



Ví dụ thực tế:

- **Class Car** (Xe hơi) mô tả một chiếc xe có **tên hãng, màu sắc, tốc độ**.
- **Object Audi** là một chiếc xe **Audi, giá 1cr, màu while**.
- **Object BMW** là một chiếc xe **BMW, giá 1.5cr, màu red**.
- **Object Bugatti** là một chiếc xe **Bugatti, giá 2cr, màu black**.

1.3 Syntax Class

Cấu trúc cơ bản của một Class Một class trong C++ được khai báo bằng từ khóa class, bao gồm thuộc tính (biến) và phương thức (hàm thành viên).

```
class ClassName {
public:
    // Thuộc tính (biến thành viên)
    DataType variableName;

    // Phương thức (hàm thành viên)
    ReturnType methodName(Parameters) {
        // Code xử lý
    }
};
```

Ví Dụ

```
class Car {
public:
    string brand;  // Thuộc tính
    int speed;

    void showInfo() {  // Phương thức
```

```
        cout << "Brand: " << brand << ", Speed: " << speed << " km/h" << endl;
    }
};
```

Tạo đối tượng (Object) từ Class Sau khi khai báo class, ta có thể tạo object và sử dụng chúng.

```
int main() {
    Car toyota;  // Tạo object toyota từ class Car
    toyota.brand = "Toyota";
    toyota.speed = 120;

    toyota.showInfo(); // Output: Brand: Toyota, Speed: 120 km/h
}
```

1.4 Các bộ nhớ lưu trữ

Kích thước của đối tượng trong C++, chúng ta cần xem xét cấu trúc bên trong của đối tượng, cách các thành viên được sắp xếp trong bộ nhớ, và ảnh hưởng của các yếu tố như **căn chỉnh bộ nhớ (alignment)**, **thành viên tĩnh (static members)**, và **kế thừa (inheritance)**. Dưới đây là giải thích chi tiết.

1. Hàm sizeof trả về kích thước (tính bằng byte) của đối tượng trong bộ nhớ, bao gồm:
   - Tổng kích thước của tất cả các thuộc tính không tĩnh.
   - Padding thêm vào để căn chỉnh.
2. **Không bao gồm**:
   - Kích thước của phương thức.
   - Kích thước của thành viên tĩnh.
   - Dữ liệu động mà con trỏ trỏ tới (chỉ tính kích thước của con trỏ).
3. Căn chỉnh bộ nhớ (Alignment)
   - Máy tính yêu cầu các kiểu dữ liệu được căn chỉnh tại các địa chỉ bộ nhớ phù hợp để truy cập hiệu quả (ví dụ: int thường cần căn chỉnh tại bội số của 4 byte, double tại bội số của 8 byte trên hệ 64-bit).
   - Trình biên dịch tự động thêm **padding** (byte đệm) giữa các thành viên để đảm bảo căn chỉnh, điều này làm tăng kích thước của đối tượng

```
class MyClass {
public:
    int* ptr;   // 8 byte (trên hệ 64-bit)
    int x;      // 4 byte
};

int main() {
    MyClass obj;
    obj.ptr = new int(10);
    cout << "Size of MyClass: " << sizeof(obj) << endl;  // Output: 16
    delete obj.ptr;
    return 0;
}
```

- int* ptr: 8 byte (kích thước con trỏ trên hệ 64-bit), dữ liệu mà ptr trỏ tới nằm trên heap và không được tính.
- int x: 4 byte.

- Padding: 4 byte để căn chỉnh tổng kích thước thành bội số của 8 (yêu cầu của con trỏ).
- Tổng: 8 (ptr) + 4 (x) + 4 (padding) = 16 byte.

```cpp
class Empty {
};

int main() {
    Empty obj;
    cout << "Size of Empty: " << sizeof(obj) << endl;  // Output: 1
    return 0;
}
```

- Lớp không có thuộc tính, nhưng sizeof trả về 1 byte.
- Đây là quy định của C++ để đảm bảo mỗi đối tượng có một địa chỉ riêng biệt (không thể có kích thước 0).

Sơ đồ minh họa (ví dụ lớp có int và double)

```
class MyClass { int a; double b; };
Đối tượng MyClass trên bộ nhớ (16 byte):
[ a: 4 byte ][ padding: 4 byte ][ b: 8 byte ]
```

2 Thuộc tính (Attributes) và phương thức (Methods)

2.1 Thuộc tính (Attributes)

Khái niệm

Thuộc tính (còn gọi là biến thành viên) là dữ liệu được lưu trữ bên trong một lớp (class). Chúng đại diện cho trạng thái (state) hoặc đặc điểm (characteristics) của đối tượng..

```
class Car {
public:
    string brand;  // Thuộc tính: hãng xe
    int year;      // Thuộc tính: năm sản xuất
};
```

Các mức độ truy cập của thuộc tính

- **Public**: Có thể truy cập từ bên ngoài lớp.
- **Private**: Chỉ có thể truy cập từ bên trong lớp.
- **Protected**: Có thể truy cập trong lớp và các lớp kế thừa.

```
class Car {
public:
    string brand;  // Có thể truy cập từ bên ngoài lớp
private:
    int year;      // Chỉ có thể truy cập từ bên trong lớp
};

Car car;
car.year = 2020; // Lỗi: year là private
```

Truy cập trực tiếp qua toán tử dấu chấm (.) Nếu một đối tượng được khai báo trực tiếp và thuộc tính đó có phạm vi truy cập public, bạn có thể truy cập bằng dấu chấm.

```
class MyClass {
public:
    int myProperty; // Thuộc tính public
};

int main() {
    MyClass obj;
    obj.myProperty = 10; // Truy cập thuộc tính
    cout << obj.myProperty << endl; // In giá trị
    return 0;
}
```

Truy cập qua con trỏ bằng toán tử mũi tên (->) Nếu bạn làm việc với con trỏ (pointer) tới một đối tượng, sử dụng -> để truy cập thuộc tính.

```
class MyClass {
public:
```

```
    int myProperty;
};

int main() {
    MyClass* ptr = new MyClass();
    ptr->myProperty = 20; // Truy cập qua con trỏ
    cout << ptr->myProperty << endl;
    delete ptr; // Đừng quên giải phóng bộ nhớ
    return 0;
}
```

Sự khác biệt giữa . và ->: . dùng cho đối tượng trực tiếp (l-value), trong khi -> là cú pháp ngắn gọn để truy cập qua con trỏ (tương đương (*ptr).member).

2.2 Phương thức (Methods)

Khái niệm

Phương thức (còn gọi là hàm thành viên) là các hàm được định nghĩa bên trong một lớp (class). Chúng đại diện cho hành vi (behavior) hoặc chức năng (functionality) của đối tượng. Phương thức thường thao tác trên các thuộc tính của lớp để thực hiện các nhiệm vụ cụ thể.

Vai trò của phương thức

- **Thao tác dữ liệu**: Cập nhật hoặc truy xuất giá trị của thuộc tính (ví dụ: getter/setter).
- **Thực hiện hành vi**: Mô phỏng hành động của đối tượng (ví dụ: drive() cho lớp Car).
- **Ẩn chi tiết triển khai**: Kết hợp với private để bảo vệ dữ liệu (encapsulation).

```
class Car {
public:
    string brand;  // Thuộc tính
    int year;      // Thuộc tính
    void displayInfo() {  // Phương thức: hiển thị thông tin
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};
```

Các mức độ truy cập của thuộc tính

- **Public**: Có thể gọi từ bên ngoài lớp.
- **Private**: Chỉ có thể gọi từ bên trong lớp.
- **Protected**: Có thể gọi trong lớp và các lớp kế thừa.

```
class Car {
public:
    string brand;
private:
    int year;
    void setYear(int y) {  // Phương thức private
        year = y;
    }
public:
```

```
10      void displayInfo() {  // Phương thức public
11          cout << "Brand: " << brand << ", Year: " << year << endl;
12      }
13  };
14
15  Car car;
16  car.brand = "Toyota";
17  car.displayInfo();  // OK: displayInfo là public
18  car.setYear(2020);  // Lỗi: setYear là private
```

Gọi phương thức qua toán tử dấu chấm (.) giống thuộc tính

Gọi phương thức qua con trỏ bằng toán tử mũi tên (->) giống thuộc tính

3 Con trỏ this

Khái niệm

Con trỏ this là một con trỏ đặc biệt trong C++, được tự động cung cấp bởi trình biên dịch cho mọi phương thức không tĩnh (non-static) của một lớp. Nó trỏ đến **đối tượng hiện tại** mà phương thức đang được gọi trên đó. Nói cách khác, this đại diện cho địa chỉ của đối tượng đang thực thi phương thức.

- **Kiểu dữ liệu**: this là một con trỏ có kiểu ClassName* const, trong đó ClassName là tên của lớp. Điều này có nghĩa là nó là một con trỏ hằng (constant pointer), không thể bị thay đổi để trỏ tới đối tượng khác.
- **Mục đích**: Dùng để truy cập các thành viên (thuộc tính hoặc phương thức) của đối tượng hiện tại hoặc để phân biệt giữa tham số và thuộc tính cùng tên.

Ứng dụng của con trỏ this

1. **Tránh xung đột tên biến** Nếu tham số của phương thức trùng tên với biến thành viên, ta dùng this để phân biệt.

```
class Car {
private:
    string brand;  // Biến thành viên
public:
    void setBrand(string brand) {
        // this->brand (biến thành viên) = brand (tham số)
        this->brand = brand;
    }
    string getBrand() { return this->brand; } // this có thể bỏ đi
};

Car car;
car.setBrand("Toyota");
cout << car.getBrand(); // In ra: Toyota
```

2. **Trả về chính đối tượng (return *this)** Cho phép gọi chuỗi (method chaining) để gọi nhiều phương thức liên tiếp. Dùng return *this; để trả về chính đối tượng hiện tại.

```
class Car {
private:
    string brand;
    int year;
public:
    Car& setBrand(string b) {
        this->brand = b;
        return *this;  // Trả về đối tượng hiện tại
    }
    Car& setYear(int y) {
        this->year = y;
        return *this;
    }
    void show() { cout << brand << " - " << year << endl; }
};
```

3. **So sánh hai đối tượng (this với đối tượng khác)** Dùng this để so sánh với con trỏ của một đối tượng khác.

```cpp
class Car {
public:
    bool isSame(Car* other) {
        return this == other;  // So sánh địa chỉ
    }
};

int main() {
    Car car1, car2;
    cout << car1.isSame(&car2); // false (0) vì car1 và car2 khác nhau
    cout << car1.isSame(&car1); // true (1) vì so sánh với chính nó
}
```

4. **Tránh gọi phương thức trên đối tượng null (nullptr)** Nếu đối tượng chưa được cấp phát (nullptr), gọi phương thức có thể gây lỗi.

```cpp
class Car {
public:
    void show() {
        if (this == nullptr) {
            cout << "Invalid object!" << endl;
            return;
        }
        cout << "Car exists!" << endl;
    }
};

int main() {
    Car* car = nullptr;
    car->show(); // Lỗi nếu không kiểm tra `this`
}
```

Một Sô tính chất về con trỏ this

- **this không chiếm bộ nhớ trong đối tượng** Mỗi đối tượng của một lớp chỉ lưu trữ các **thuộc tính (data members)** của nó trong bộ nhớ. Con trỏ this không phải là một phần của cấu trúc dữ liệu của đối tượng, nên nó không làm tăng kích thước của đối tượng (kiểm chứng bằng sizeof).
- **this là một tham số ẩn được truyền bởi trình biên dịch** khi bạn gọi một phương thức không tĩnh của một đối tượng, trình biên dịch tự động truyền con trỏ this như một tham số ẩn (hidden parameter) vào hàm đó. Về mặt triển khai nội bộ, phương thức của lớp được biên dịch thành một hàm thông thường, và this được truyền vào để xác định đối tượng cụ thể mà phương thức đang thao tác.

```cpp
class MyClass {
public:
    int x;
    void setX(int value) {
        this->x = value;
    }
};

// Trình biên dịch biến đổi setX
```

```
10 void setX(MyClass* this, int value) {
11     this->x = value;
12 }
```

4 Hàm tạo (Constructor) và hàm hủy (Destructor)

4.1 Hàm tạo (Constructor)

Khái niệm

Hàm tạo (constructor) là một phương thức đặc biệt trong lớp, được tự động gọi khi một đối tượng của lớp được tạo ra. Mục đích chính của hàm tạo là **khởi tạo các thuộc tính** của đối tượng hoặc thực hiện các thiết lập ban đầu cần thiết.

Đặc điểm:

- Có cùng tên với lớp.
- Không có kiểu trả về (không kể cả void).
- Có thể có tham số hoặc không (tùy loại hàm tạo).
- Nếu bạn không định nghĩa hàm tạo, trình biên dịch sẽ tự động cung cấp một **hàm tạo mặc định (default constructor)** không tham số.

```
class Car {
public:
    string brand;
    int year;
    // Hàm tạo với tham số
    Car(string b, int y) {
        brand = b;
        year = y;
    }
};
```

Các loại hàm tạo

1. **Hàm tạo mặc định (Default Constructor)**
   - Là **constructor không có tham số** hoặc tất cả tham số có giá trị mặc định.
   - Nếu **không có constructor nào được định nghĩa**, trình biên dịch **tự động tạo một constructor mặc định**.
   - Dùng để **khởi tạo giá trị mặc định** cho đối tượng.

   Constructor mặc định do trình biên dịch tạo

```
class Car {
    string brand;
public:
    Car() { // Constructor mặc định
        cout << "Default Constructor called!" << endl;
        brand = "Unknown";
    }
    void show() { cout << "Brand: " << brand << endl; }
};

int main() {
    Car c1; // Gọi constructor mặc định
    c1.show();
}
```

Constructor mặc định do trình biên dịch tạo

```
class Car {
    string brand;
public:
    void show() { cout << "Brand: " << brand << endl; }
};

int main() {
    Car c1; // Gọi constructor mặc định do trình biên dịch tạo
    c1.show();
}
```

2. **Hàm tạo có tham số (Parameterized Constructor)**
   - Là **constructor có tham số**, cho phép truyền giá trị khi tạo đối tượng.
   - Dùng để **khởi tạo đối tượng với giá trị cụ thể** thay vì giá trị mặc định.
   - Nếu đã có **constructor có tham số**, trình biên dịch **không tạo constructor mặc định**.
   - Nếu muốn dùng cả constructor mặc định và có tham số, cần **tạo cả hai**.

```
class Car {
    string brand;
public:
    Car(string b) { // Constructor có tham số
        brand = b;
        cout << "Parameterized Constructor called!" << endl;
    }
    void show() { cout << "Brand: " << brand << endl; }
};

int main() {
    Car c1("Toyota"); // Gọi constructor có tham số
    c1.show();
}
```

3. **Hàm tạo sao chép (Copy Constructor)**
   - Dùng để **tạo một đối tượng bằng cách sao chép một đối tượng khác cùng lớp**.
   - **Tham số là một tham chiếu hằng (const ClassName&)** để tránh vòng lặp vô hạn.
   - Nếu không định nghĩa, C++ sẽ tự động tạo một **copy constructor mặc định**.
   - Nếu lớp có **con trỏ hoặc tài nguyên cấp phát động**, cần định nghĩa copy constructor để **tránh lỗi sao chép vùng nhớ**.

Constructor sao chép do lập trình viên định nghĩa

```
class Car {
    string brand;
public:
    Car(string b) { brand = b; }
    Car(const Car &c) { // Copy constructor
        brand = c.brand;
```

```
        cout << "Copy Constructor called!" << endl;
    }
    void show() { cout << "Brand: " << brand << endl; }
};

int main() {
    Car c1("Honda");
    Car c2 = c1; // Gọi constructor sao chép
    c2.show();
}
```

Copy constructor do trình biên dịch tạo

```
class Car {
    string brand;
public:
    Car(string b) { brand = b; }
    void show() { cout << "Brand: " << brand << endl; }
};

int main() {
    Car c1("Toyota");
    Car c2 = c1; // Gọi constructor sao chép mặc định
    c2.show();
}
```

4. **Hàm tạo danh sách khởi tạo (Initializer List Constructor)**
   - **Dùng danh sách khởi tạo (initializer list) thay vì gán giá trị trong thân constructor.**
   - **Tăng hiệu suất**, đặc biệt với **biến const hoặc tham chiếu.**

`ClassName(parameters) : member1(value1), member2(value2) { }`

Cách gán giá trị trong thân constructor:

```
class Car {
    string brand;
public:
    Car(string b) { // Gán giá trị trong thân constructor (chậm hơn)
        brand = b;
    }
};
```

Dùng danh sách khởi tạo (tốt hơn):

```
class Car {
    string brand;
public:
    Car(string b) : brand(b) { } // Dùng initializer list (tối ưu)
};
```

4.2 Hàm hủy (Destructor)

Khái niệm

Hàm hủy (destructor) là một phương thức đặc biệt trong lớp, được tự động gọi khi một đối tượng của lớp bị hủy hoặc ra khỏi phạm vi (scope). Mục đích chính của hàm hủy là **giải phóng tài nguyên** mà đối tượng đã cấp phát (như bộ nhớ động, tệp, hoặc kết nối mạng) để tránh rò rỉ tài nguyên.

Đặc điểm:

- Có tên giống tên lớp, nhưng bắt đầu bằng ký tự ~
- Không có tham số và không có giá trị trả về.
- Chỉ có duy nhất một hàm hủy cho mỗi lớp (không thể quá tải).
- Nếu bạn không định nghĩa hàm hủy, trình biên dịch sẽ tự động cung cấp một **hàm hủy mặc định** (default destructor) thực hiện hành vi cơ bản.

```
class Car {
public:
    string* brand;  // Con trỏ động
    Car(string b) {  // Hàm tạo
        brand = new string(b);  // Cấp phát động
    }
    ~Car() {  // Hàm hủy
        delete brand;  // Giải phóng bộ nhớ
        cout << "Destructor called" << endl;
    }
};
```

Khi nào hàm hủy được gọi?

- **Đối tượng trên stack**: Khi đối tượng ra khỏi phạm vi (scope) nơi nó được khai báo.
- **Đối tượng trên heap**: Khi bạn gọi delete trên con trỏ trỏ tới đối tượng.
- **Đối tượng tạm thời**: Khi đối tượng tạm thời không còn được sử dụng.

Cách sử dụng hàm hủy

- **Giải phóng bộ nhớ động** Nếu lớp sử dụng new để cấp phát bộ nhớ, hàm hủy cần dùng delete để giải phóng.

```
class MyClass {
public:
    int* ptr;
    MyClass() {
        ptr = new int(10); // Cấp phát động
    }
    ~MyClass() {
        delete ptr; // Giải phóng bộ nhớ
    }
};

MyClass* obj = new MyClass();
delete obj;  // Gọi hàm hủy
```

- **Quản lý tài nguyên khác** Ngoài bộ nhớ, hàm hủy có thể đóng tệp, giải phóng khóa mạng, hoặc thực hiện các tác vụ dọn dẹp khác.

Lưu ý quan trọng

- **Hàm hủy mặc định**: Nếu bạn không định nghĩa hàm hủy, trình biên dịch sẽ cung cấp một hàm hủy mặc định, nhưng nó không giải phóng tài nguyên động (như bộ nhớ do new cấp phát). Điều này có thể dẫn đến rò rỉ bộ nhớ (memory leak).
- **Không gọi thủ công**: Bạn không nên gọi hàm hủy trực tiếp (ví dụ: obj.~MyClass();), trừ trường hợp đặc biệt khi quản lý bộ nhớ thủ công (như sử dụng placement new).