**Các loại phương thức trong C++:**

1. **Các phương thức công khai (Public Methods)**:
   * Là các phương thức mà các đối tượng bên ngoài có thể gọi để thực hiện các thao tác trên đối tượng của lớp.
   * Các phương thức công khai thường được sử dụng để cung cấp khả năng truy

A screen shot of a computer code

Description automatically generated

1. **Các phương thức riêng tư (Private Methods)**:
   * Là các phương thức chỉ có thể được gọi từ bên trong cùng lớp hoặc từ các phương thức công khai của cùng lớp.
   * Các phương thức riêng tư thường được sử dụng để triển khai các thao tác nội bộ và bảo vệ dữ liệu của lớp.

A screen shot of a computer program

Description automatically generated

1. **Các phương thức bảo vệ (Protected Methods)**:
   * Tương tự như phương thức riêng tư, nhưng chỉ có thể truy cập từ bên trong lớp hoặc từ các lớp dẫn xuất từ lớp hiện tại.
   * Các phương thức bảo vệ thường được sử dụng trong kế thừa để cho phép lớp con truy cập và thay đổi dữ liệu mà không cần phải công khai chúng.

A screen shot of a computer program

Description automatically generated

**Các tính chất của phương thức:**

* **const Method**: Được gắn thẻ với const ở cuối, chỉ ra rằng phương thức không thay đổi trạng thái nội tại của đối tượng.

A black screen with white text

Description automatically generated

* **Static Method**: Là phương thức thuộc về lớp chứ không phải đối tượng cụ thể và có thể được gọi mà không cần tạo đối tượng.

A black background with white text

Description automatically generated

* **Inline Method**: Phương thức có thể được định nghĩa và triển khai ngay trong phần khai báo của lớp, thường dùng cho các phương thức đơn giản.

A screen shot of a computer program

Description automatically generated

Các phương thức là cách chính để định nghĩa hành vi của các đối tượng trong C++. Chúng cung cấp sự tương tác và quản lý dữ liệu hiệu quả cho các lớp và đóng vai trò quan trọng trong việc thiết kế và triển khai các ứng dụng phần mềm.

**4. Setter getter data**

 Sử dụng danh sách khởi tạo để khởi tạo hiệu quả các biến thành viên.

 Cung cấp các hàm lấy giá trị để truy cập an toàn đến các biến thành viên riêng tư.

 Bao đóng dữ liệu và tránh sự thay đổi không cần thiết bằng cách thiết lập dữ liệu trong hàm tạo mà không cần sử dụng hàm thiết lập

A computer screen with white text

Description automatically generated

**1. Constructor (Student(int id, string name): id\_{id}, name\_{name} {})**

* **Danh sách khởi tạo (Initializer List)**: Trong hàm tạo Student(int id, string name), các biến thành viên id\_ và name\_ được khởi tạo bằng danh sách khởi tạo id\_{id} và name\_{name}. Đây là cách tiếp cận được ưa thích trong C++ vì nó giúp tránh phép gán không cần thiết và có thể hiệu quả hơn.

**2. Các hàm thành viên công khai (Public Member Functions)**

* **Các hàm lấy giá trị (Getter Functions)**:
  + int id() const { return id\_; }: Hàm này trả về giá trị của biến thành viên id\_, là mã số sinh viên. Hàm này trả về kiểu int.
  + const string& name() const { return name\_; }: Hàm này trả về một tham chiếu hằng đến biến thành viên name\_. Bằng cách sử dụng const string&, nó đảm bảo rằng chuỗi trả về không thể bị thay đổi vô tình bởi người gọi hàm, cung cấp cả hiệu suất và an toàn.

**3. Các biến thành viên riêng tư (Private Member Variables)**

* **Bao đóng (Encapsulation)**:
  + int id\_;: Biến thành viên này lưu trữ mã số sinh viên và được đánh dấu là riêng tư (private:), có nghĩa là chỉ có thể truy cập trực tiếp trong lớp Student và không thể từ bên ngoài.
  + string name\_;: Tương tự, name\_ là biến thành viên riêng tư lưu trữ tên của sinh viên. Các biến thành viên riêng tư giúp bao đóng trạng thái nội tại của lớp và cung cấp quyền truy cập được kiểm soát thông qua các hàm thành viên công khai.

**Các Thực Tiễn Tốt Được Giải Thích**

* **Thiết lập và Lấy Dữ Liệu**:
  + Dữ liệu (mã số sinh viên và tên) được thiết lập trong quá trình tạo đối tượng bằng các tham số của hàm tạo. Điều này hiệu quả và tuân thủ nguyên tắc của việc khởi tạo đối tượng khi tạo ra.
* **Sử dụng Danh Sách Khởi Tạo (Initializer List)**:
  + Danh sách khởi tạo (id\_{id}, name\_{name}) khởi tạo các biến thành viên trực tiếp. Phương pháp này được ưa thích hơn so với gán giá trị trong thân hàm tạo vì nó khởi tạo biến trước khi thân hàm tạo bắt đầu thực thi.
* **Quy ước đặt tên cho hàm lấy giá trị (Getter Functions)**:
  + Các hàm lấy giá trị (id() và name()) được đặt tên dựa trên tên của biến thành viên mà chúng trả về (id\_ và name\_). Quy ước này giúp mã nguồn trở nên dễ hiểu hơn, đặc biệt là khi làm việc với các lớp có nhiều biến thành viên.
* **Tránh sử dụng hàm thiết lập (Setters)**:
  + Không có hàm thiết lập (setter) được cung cấp cho id và name. Thay vào đó, các giá trị này được thiết lập một lần trong quá trình tạo đối tượng bằng các tham số của hàm tạo. Phương pháp này khuyến khích tính bất biến và đảm bảo rằng khi một đối tượng Student được tạo ra, id và name không thể được thay đổi từ bên ngoài, từ đó tăng tính toàn vẹn dữ liệu và giảm độ phức tạp của mã nguồn.

**DECLARATION AND DEFINITION**

**Khai Báo (Declaration)**

Khai báo là khi bạn thông báo với trình biên dịch về sự tồn tại của một biến, hàm, hoặc lớp mà không cung cấp chi tiết về cách nó được thực hiện. Điều này cho phép các phần khác của chương trình biết về sự tồn tại của nó và cách sử dụng nó.

Ví dụ về **khai báo lớp**:

**A black screen with white text

Description automatically generated**

 Dòng 2: Bắt đầu khai báo lớp SomeClass.

 Dòng 4: Khai báo hàm tạo SomeClass().

 Dòng 5: Khai báo hàm thành viên var() const. Từ khóa const sau danh sách tham số chỉ ra rằng hàm này không thay đổi bất kỳ thành viên nào của lớp.

 Dòng 7: Khai báo hàm thành viên riêng DoSmth().

 Dòng 8: Khai báo và khởi tạo biến thành viên var\_ với giá trị 0

**Định Nghĩa (Definition)**

Định nghĩa là khi bạn cung cấp chi tiết đầy đủ về cách một hàm hoặc một lớp hoạt động. Đối với lớp, định nghĩa bao gồm thân hàm (function body) thực sự của các phương thức.

Ví dụ về **định nghĩa các phương thức** của lớp SomeClass:

**A computer screen shot of a black background

Description automatically generated**

 Dòng 11: Định nghĩa hàm tạo SomeClass(). Đây là nơi thân hàm rỗng được cung cấp cho hàm tạo.

 Dòng 12: Định nghĩa hàm var() const, trả về giá trị của biến thành viên var\_.

 Dòng 13: Định nghĩa hàm DoSmth(), nơi bạn có thể thêm chức năng cụ thể của hàm này

**Tên Lớp là Một Phần của Tên Hàm**

Khi bạn định nghĩa một phương thức bên ngoài định nghĩa lớp, bạn cần sử dụng tên lớp cùng với toán tử phạm vi :: để chỉ rõ rằng phương thức này thuộc về lớp nào. Điều này tạo thành tên đầy đủ của phương thức.

Ví dụ

**A black and orange text

Description automatically generated**

**Quản lý mã nguồn dễ dàng hơn**: Bằng cách tách biệt khai báo và định nghĩa, bạn có thể giữ **các khai báo trong tập tin header** (.h) và **các định nghĩa trong tập tin nguồn (.cpp)**. Điều này giúp mã nguồn dễ quản lý và bảo trì hơn.

**Toán Tử Phạm Vi :: trong C++**

Toán tử phạm vi :: được sử dụng trong C++ để xác định phạm vi hoặc ngữ cảnh của các thành viên thuộc lớp, namespace, hoặc các kiểu liệt kê. Nó rất hữu ích khi bạn có các thành phần trùng tên trong các phạm vi khác nhau hoặc khi bạn muốn xác định rõ ràng thành viên nào bạn đang đề cập tới.

**Các Tình Huống Sử Dụng**

1. **Định nghĩa hàm thành viên bên ngoài lớp**: Khi bạn khai báo hàm thành viên trong một lớp nhưng định nghĩa nó bên ngoài lớp, bạn cần sử dụng toán tử phạm vi để cho trình biên dịch biết rằng hàm này thuộc về lớp nào.

class SomeClass {

public:

void DoSomething();

};

// Định nghĩa hàm DoSomething bên ngoài lớp

void SomeClass::DoSomething() {

// Thân hàm

}

1. **Sử dụng các thành viên lớp**: Khi sử dụng các thành viên của lớp mà đã được định nghĩa trước đó.

class SomeClass {

public:

static int value;

};

// Khởi tạo thành viên static bên ngoài lớp

int SomeClass::value = 0;

1. **Phạm vi của namespace**: Khi sử dụng các thành phần trong namespace để tránh xung đột tên.

namespace MyNamespace {

void Function() {

// Thân hàm

}

}

// Gọi hàm Function trong MyNamespace

MyNamespace::Function();

1. **Các kiểu liệt kê (enum)**: Khi truy cập các giá trị của kiểu liệt kê.

enum Color { RED, GREEN, BLUE };

Color myColor = Color::RED;

**STATIC VARIABLE**

#ifndef COUNTED\_HPP

#define COUNTED\_HPP

class Counted {

public:

// Tăng giá trị đếm mỗi khi tạo một đối tượng mới của lớp Counted

Counted() { Counted::count++; }

// Giảm giá trị đếm mỗi khi hủy một đối tượng của lớp Counted

~Counted() { Counted::count--; }

// Biến tĩnh đếm số lượng đối tượng được tạo ra

static int count;

};

#endif

* **Biến tĩnh count**: Được sử dụng để đếm số lượng đối tượng của lớp Counted đang tồn tại.
* **Constructor**: Tăng giá trị của count mỗi khi tạo một đối tượng mới.
* **Destructor**: Giảm giá trị của count mỗi khi hủy một đối tượng.
* **Định nghĩa biến tĩnh**: Phải được định nghĩa trong file nguồn.

**Sử dụng biến tĩnh count**: Có thể truy cập biến tĩnh count thông qua toán tử phạm vi (::) và tên lớp, ví dụ: Counted::count

#include <iostream>

using std::cout;

using std::endl;

// Include the Counted class declaration and

// Initialize the static member of the class only once.

// This could be any value

#include "Counted.hpp"

int Counted::count = 0;

int main() {

Counted a, b;

cout << "Count: " << Counted::count << endl;

Counted c;

cout << "Count: " << Counted::count << endl;

return 0;

}

**Constructor Mặc định (MyClass()):**

cpp

Sao chép mã

MyClass() : names\_(100, "name") {}

* **Ý nghĩa**: Đây là constructor mặc định của lớp MyClass. Nó khởi tạo một đối tượng MyClass mới với các giá trị mặc định.
* **Hoạt động**:
  + names\_(100, "name"): Khởi tạo vector names\_ với 100 phần tử, mỗi phần tử là chuỗi "name". Điều này có nghĩa là vector names\_ sẽ bao gồm 100 chuỗi "name" ngay khi đối tượng MyClass được khởi tạo.

**Constructor Sao chép (MyClass(const MyClass& other)):**

cpp

Sao chép mã

// Copy Constructor

MyClass(const MyClass& other) {

id\_ = other.id\_;

names\_ = other.names\_;

}

* **Ý nghĩa**: Đây là constructor sao chép của lớp MyClass. Nó được gọi khi một đối tượng MyClass được sao chép từ một đối tượng MyClass khác (other).
* **Hoạt động**:
  + id\_ = other.id\_;: Sao chép giá trị của trường dữ liệu id\_ từ đối tượng other sang đối tượng hiện tại.
  + names\_ = other.names\_;: Sao chép toàn bộ vector names\_ từ đối tượng other sang đối tượng hiện tại. Điều này dẫn đến việc tạo ra một bản sao sâu của vector names\_, bao gồm việc sao chép tất cả các phần tử từ vector names\_ của other sang vector names\_ của đối tượng hiện tại.

**Constructor Di chuyển (MyClass(MyClass&& other) noexcept):**

cpp

Sao chép mã

// Move Constructor

MyClass(MyClass&& other) noexcept {

id\_ = other.id\_;

names\_ = std::move(other.names\_);

}

* **Ý nghĩa**: Đây là constructor di chuyển của lớp MyClass. Nó được gọi khi một đối tượng MyClass được khởi tạo từ một đối tượng MyClass khác bằng cách di chuyển (move).
* **Hoạt động**:
  + id\_ = other.id\_;: Sao chép giá trị của trường dữ liệu id\_ từ đối tượng other sang đối tượng hiện tại. Điều này tương tự như constructor sao chép.
  + names\_ = std::move(other.names\_);: Sử dụng std::move để di chuyển (move) toàn bộ vector names\_ từ đối tượng other sang đối tượng hiện tại. Việc di chuyển như vậy cho phép chuyển quyền sở hữu các phần tử trong vector names\_ từ other sang names\_ của đối tượng hiện tại một cách hiệu quả hơn, thay vì sao chép lại từng phần tử.

**Tóm lại:**

* Constructor mặc định khởi tạo một đối tượng MyClass với các giá trị mặc định, trong đó vector names\_ được khởi tạo với 100 chuỗi "name".
* Constructor sao chép sao chép các giá trị từ một đối tượng MyClass khác.
* Constructor di chuyển di chuyển các tài nguyên từ một đối tượng MyClass khác, giúp tối ưu hóa hiệu suất bằng cách tránh sao chép dữ liệu không cần thiết.

**Hàm copy\_swap**

cpp

Sao chép mã

void copy\_swap(MyClass& obj1, MyClass& obj2) {

MyClass tmp = obj1; // copy obj1 to tmp

obj1 = obj2; // copy obj2 to obj1

obj2 = tmp; // copy tmp to obj2

}

* **Ý nghĩa**: Hàm copy\_swap thực hiện việc trao đổi nội dung giữa hai đối tượng MyClass bằng cách sử dụng phép gán sao chép.
* **Hoạt động**:
  1. MyClass tmp = obj1;: Tạo một bản sao của obj1 và lưu vào biến tmp. Đây là bước sao chép.
  2. obj1 = obj2;: Gán giá trị của obj2 cho obj1. Đây cũng là một bước sao chép.
  3. obj2 = tmp;: Gán giá trị của tmp (là bản sao của obj1) cho obj2. Đây là bước sao chép nữa.

**Hàm move\_swap**

cpp

Sao chép mã

void move\_swap(MyClass& obj1, MyClass& obj2) {

MyClass tmp = std::move(obj1); // move obj1 to tmp

obj1 = std::move(obj2); // move obj2 to obj1

obj2 = std::move(tmp); // move tmp to obj2

}

* **Ý nghĩa**: Hàm move\_swap thực hiện việc trao đổi nội dung giữa hai đối tượng MyClass bằng cách sử dụng phép gán di chuyển (std::move).
* **Hoạt động**:
  1. MyClass tmp = std::move(obj1);: Di chuyển (move) nội dung của obj1 vào tmp. Điều này có nghĩa là obj1 không giữ nữa các tài nguyên của nó và tmp sẽ thừa kế chúng.
  2. obj1 = std::move(obj2);: Di chuyển (move) nội dung của obj2 vào obj1. Tương tự, obj2 không còn giữ nữa các tài nguyên của nó.
  3. obj2 = std::move(tmp);: Di chuyển (move) nội dung của tmp vào obj2. Cuối cùng, obj2 thừa kế các tài nguyên từ tmp.

**Benchmarking (StringCopySwap và StringMoveSwap)**

cpp

Sao chép mã

static void StringCopySwap(benchmark::State& state) {

MyClass obj1;

MyClass obj2;

for (auto \_ : state) {

copy\_swap(obj1, obj2);

}

}

BENCHMARK(StringCopySwap);

static void StringMoveSwap(benchmark::State& state) {

MyClass obj1;

MyClass obj2;

for (auto \_ : state) {

move\_swap(obj1, obj2);

}

}

BENCHMARK(StringMoveSwap);

* **Ý nghĩa**: Hai hàm StringCopySwap và StringMoveSwap được sử dụng để đo lường hiệu suất của hàm copy\_swap và move\_swap tương ứng.
* **Hoạt động**:
  + Trong mỗi hàm, đối tượng MyClass được khởi tạo (obj1 và obj2).
  + Vòng lặp for chạy theo state (được điều khiển bởi benchmarking framework) để thực hiện gọi copy\_swap hoặc move\_swap.
  + Hiệu suất của từng hàm được đo lường bằng cách tính thời gian hoàn thành vòng lặp và các thống kê khác như số lần gọi lại, thời gian trung bình mỗi lần gọi, v.v.

**BENCHMARK\_MAIN()**

cpp

Sao chép mã

BENCHMARK\_MAIN();

* **Ý nghĩa**: Đây là macro của Google Benchmark, dùng để tạo hàm main chạy benchmark. Nó tạo ra các cấu trúc dữ liệu và hàm để quản lý và chạy các bài benchmark đã được định nghĩa.