**Cấu trúc dữ liệu & Giải thuật trong C++**

# Chương 1. Ngôn ngữ lập trình C++

## Giới thiệu ngôn ngữ lập trình C++

C++ là ngôn ngữ được phát triển bởi Bjarne Stroustrup vào năm 1979 tại Bell Labs, như là một mở rộng của ngôn ngữ C. Mục tiêu của Stroustrup là tạo ra một ngôn ngữ có thể cung cấp tính linh hoạt và hiệu suất cao của C, đồng thời bổ sung các tính năng hướng đối tượng.

Một số đặc điểm chính của C++:

* **Hướng đối tượng (OOP)**: C++ hỗ trợ các khái niệm OOP như lớp, đối tượng, thừa kế, đa hình và đóng gói.
* **Quản lý bộ nhớ**: C++ cho phép quản lý bộ nhớ bằng tay thông qua con trỏ và các toán tử new và delete.
* **Thư viện phong phú**: C++ cung cấp một số lượng lớn các thư viện chuẩn (STL) cho các cấu trúc dữ liệu và thuật toán.
* **Tính tương thích ngược**: C++ tương thích với C, tức là hầu hết mã C có thể được biên dịch và chạy trong môi trường C++.
* **Mẫu (Templates)**: C++ hỗ trợ lập trình tổng quát thông qua các mẫu, cho phép tạo ra mã linh hoạt và tái sử dụng.

## Nhập xuất dữ liệu trong C++

C++ sử dụng thư viện iostream để nhập xuất dữ liệu. Thư viện này cung cấp các đối tượng như cin, cout,…

* cin: Nhập dữ liệu từ bàn phím (chuẩn đầu vào).
* cout: Xuất dữ liệu ra màn hình (chuẩn đầu ra).

### Nhập dữ liệu

cin >> variable;

Trong đó variable là biến mà bạn muốn gán giá trị từ đầu vào. cin hỗ trợ nhiều kiểu dữ liệu khác nhau như int, float, double, char, và string.

Riêng với string ta có thể nhập toàn bộ dòng (bao gồm cả khoảng trắng) bằng cách sử dụng hàm getline. Cú pháp: getline(cin, variable);

Ví dụ: string fullName;

getline(cin, fullName);

Chú ý khi có kết hợp giữa cin và getline: Vì cin không loại bỏ ký tự newline (\n) khỏi bộ nhớ đệm, điều này có thể dẫn đến kết quả không mong muốn. Để tránh vấn đề này, bạn có thể sử dụng thêm một dòng cin.ignore() để loại bỏ ký tự newline.

Ví dụ: int age;

string fullName;

cin >> age;

cin.ignore(); // Loại bỏ ký tự newline khỏi bộ đệm

getline(cin, fullName);

### Xuất dữ liệu

cout << expression;

Trong đó expression là giá trị hoặc biến bạn muốn xuất ra màn hình.

Ví dụ: int age = 20;

cout << "Age: " << age << endl;

float height = 1.65;

double weight = 54;

cout << "Height: " << height << " m" << endl;

cout << "Weight: " << weight << " kg" << endl;

char grade = 'F';

cout << "Grade: " << grade << endl;

string name = "Ha Kien";

cout << "Name: " << name << endl;

Trong C++, ta còn có thư viện iomanip cung cấp các công cụ hữu ích để định dạng đầu ra khi sử dụng cout. Dưới đây là một số phương thức phổ biến trong iomanip:

* setw: được sử dụng để đặt độ rộng trường của đầu ra. Nếu giá trị cần xuất nhỏ hơn độ rộng đã đặt, nó sẽ được căn chỉnh và các khoảng trắng sẽ được thêm mặc định vào bên trái.

Ví dụ: int num = 123;

cout << "Number: " << setw(10) << num << endl;

//đầu ra: Number: 123

* setfill được sử dụng để đặt ký tự lấp đầy cho các khoảng trống được tạo ra bởi setw.

Ví dụ: int num = 123;

cout << "Number: " << setfill('\*') << setw(10) << num << endl;

//đầu ra: Number: \*\*\*\*\*\*\*123

* setprecision được sử dụng để đặt độ chính xác của số thập phân (số chữ số sau dấu chấm thập phân).

Ví dụ: double pi = 3.14159265359;

cout << "Pi: " << setprecision(5) << pi << endl;

//Đầu ra: Pi: 3.1416

* fixed được sử dụng để đặt định dạng số thập phân cố định.

Ví dụ: double pi = 3.14159265359;

cout << "Fixed: " << fixed << setprecision(5) << pi << endl;

//Đầu ra: Fixed: 3.14159

## Hàm trong C++

Khi xây dựng hàm, ngoài các kiểu hàm như trong C thì C++ còn cho phép xây dựng các hàm sau đây:

* Hàm đối tham chiếu
* Hàm đối mặc định
* Nạp chồng hàm (Function Overloading)
* Hàm mẫu (Template Functions)

### Hàm đối tham chiếu

Khai báo hàm: DataType Func\_Name(DataType & Arg\_Nam,..);

Hàm đối tham chiếu là hàm mà các tham số được truyền bằng tham chiếu thay vì truyền bằng giá trị. Điều này cho phép hàm thay đổi trực tiếp giá trị của biến được truyền vào.

Ví dụ: void swap(int &a, int &b) {

int temp = a;

a = b;

b = temp;

}

int main() {

int x = 10, y = 20;

cout << "Before swap: x = " << x << ", y = " << y << endl;

swap(x, y);

cout << "After swap: x = " << x << ", y = " << y << endl;

return 0;

}

//Đầu ra: Before swap: x = 10, y = 20

After swap: x = 20, y = 10

### Hàm đối mặc định

Khai báo hàm:

DataType Func\_Name(DataType Arg\_Nam1, DataType Arg\_Nam2 = value2, ...);

Hàm đối mặc định là hàm có thể có các tham số mặc định. Điều này cho phép bạn gọi hàm mà không cần truyền tất cả các tham số, và các tham số không được truyền sẽ sử dụng giá trị mặc định đã được định nghĩa.

Ví dụ: void display(int a = 10, int b = 20) {

cout << "a = " << a << ", b = " << b << endl;

}

int main() {

display();//Sử dụng giá trị mặc định cho cả hai tham số

display(30);//Sử dụng giá trị mặc định cho tham số thứ hai

display(40, 50);//Sử dụng giá trị được truyền vào cho cả hai tham số

return 0;

}

### Nạp chồng hàm (Function Overloading)

Nạp chồng hàm là khả năng định nghĩa nhiều hàm có cùng tên nhưng khác nhau về số lượng hoặc kiểu tham số. Trình biên dịch sẽ quyết định hàm nào được gọi dựa trên danh sách tham số.

Ví dụ: void print(int i) {

cout << "Integer: " << i << endl;

}

void print(double f) {

cout << "Float: " << f << endl;

}

void print(string s) {

cout << "String: " << s << endl;

}

int main() {

print(10); // Gọi hàm với tham số kiểu int

print(3.14); // Gọi hàm với tham số kiểu double

print("Hello"); // Gọi hàm với tham số kiểu string

return 0;

}

### Hàm mẫu (Template Functions)

Hàm mẫu cho phép bạn viết một hàm duy nhất có thể làm việc với nhiều kiểu dữ liệu khác nhau. Điều này rất hữu ích cho việc viết các hàm tổng quát.

Ví dụ: template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

cout << "Int: " << add(10, 20) << endl;

//Hàm mẫu cho kiểu int

cout << "Double: " << add(3.14, 2.71) << endl;

//Hàm mẫu cho kiểu double

cout << "String: " << add(string("Hello "), string("World")) << endl;

// Hàm mẫu cho kiểu string

}

# Chương II. Cơ bản về lập trình hướng đối tượng

## 2.1. Lập trình hướng đối tượng là gì?

Lập trình hướng đối tượng (Object-Oriented Programming - OOP) là một phương pháp lập trình dựa trên khái niệm "đối tượng", trong đó đối tượng là một thực thể bao gồm dữ liệu (thuộc tính) và các phương thức (hành vi) để thao tác với dữ liệu đó.

## 2.2. Khái niệm về Lớp (Class) và Đối tượng (Object)

Lớp là một bản thiết kế hoặc khuôn mẫu cho các đối tượng. Nó định nghĩa các thuộc tính (dữ liệu) và phương thức (hành vi) mà một đối tượng có thể có.

Đối tượng là một thể hiện cụ thể của một lớp. Mỗi đối tượng có các giá trị thuộc tính riêng và có thể thực thi các phương thức của lớp.

Ví dụ: class Triangle {

private:

double a, b, c; // Cạnh a, b, c

public:

// Phương thức để tính chu vi của tam giác

double perimeter(){

return a + b + c;

}

// Phương thức để tính diện tích của tam giác bằng định lý Heron

double area(){

double p = perimeter() / 2;

return sqrt(p \* (p - a) \* (p - b) \* (p - c));

}

// Phương thức để hiển thị thông tin về tam giác

void displayInfo(){

cout << "Side a: " << a << endl;

cout << "Side b: " << b << endl;

cout << "Side c: " << c << endl;

cout << "Perimeter: " << perimeter() << endl;

cout << "Area: " << area() << endl;

}

};

int main() {

// Tạo một đối tượng của lớp Triangle

Triangle myTriangle;

// Thiết lập giá trị cho các thuộc tính của đối tượng

myTriangle.a = 3.0; myTriangle.b = 4.0; myTriangle.c = 5.0;

// Hiển thị thông tin về tam giác

myTriangle.displayInfo();

}

* **Thuộc tính**:
  + a, b, c: Các cạnh của tam giác.
* **Phương thức**:
  + double perimeter(): Tính chu vi của tam giác.
  + double area(): Tính diện tích của tam giác bằng định lý Heron.
  + void displayInfo(): Hiển thị thông tin về các cạnh, chu vi và diện tích của tam giác.

## 2.3. Cài đặt phương thức

Việc cài đặt phương thức của lớp có thể được thực hiện theo hai cách: bên trong lớp và bên ngoài lớp.

Trong lớp: class ClassName {

public:

// Khai báo và cài đặt phương thức trong lớp

ReturnType methodName(ParameterType parameter) {

}

};

Ngoài lớp: class ClassName {

public:

ReturnType methodName(ParameterType parameter);

};

// Cài đặt phương thức ngoài lớp

ClassName::methodName(ParameterType parameter) {

// Cài đặt phương thức

}

## 2.4. Truy cập đến các phần tử trong lớp

Mức độ Truy cập:

* **Public (Công khai):**
  + Các thành phần được khai báo public có thể được truy cập từ bất kỳ đâu trong chương trình, bao gồm cả từ bên ngoài lớp.
  + Sử dụng khi bạn muốn cho phép các thành phần của lớp được truy cập và sử dụng một cách tự do.
* **Protected (Bảo vệ):**
  + Các thành phần được khai báo protected chỉ có thể được truy cập từ các phương thức trong lớp đó và các lớp kế thừa (lớp con).
  + Sử dụng khi bạn muốn bảo vệ các thành phần khỏi việc truy cập trực tiếp từ bên ngoài lớp nhưng vẫn cho phép các lớp kế thừa có thể truy cập.
* **Private (Riêng tư):**
  + Các thành phần được khai báo private chỉ có thể được truy cập từ các phương thức trong lớp đó. Không thể truy cập trực tiếp từ bên ngoài lớp hoặc từ các lớp kế thừa.
  + Sử dụng để ẩn thông tin và bảo vệ tính toàn vẹn của dữ liệu.

Cách truy cập:

Trong C++, bạn sử dụng toán tử dấu chấm (.) để truy cập các thành phần (thuộc tính và phương thức) của đối tượng. Khi làm việc với con trỏ đến đối tượng, bạn sử dụng toán tử mũi tên (->).

Ví dụ: class Person {

private:

string name; // Thuộc tính private

int age; // Thuộc tính private

public:

void setInfo(const string& n, int a) {

name = n;

age = a;

}

void displayInfo(){

cout << "Name: " << name << endl;

cout << "Age: " << age << endl;

}

};

int main() {

Person person;

person.setInfo("Alice", 30);

person.displayInfo();

//person.name = "Bob"; // Lỗi biên dịch

//person.age = 25; // Lỗi biên dịch

Person\* ptrPerson = new Person;

ptrPerson->setInfo("Bob", 25);

ptrPerson->displayInfo();

return 0;

}

## 2.5. Cấu tử (Constructor) và Hủy tử (Destructor)

### 2.5.1. Cấu tử (Constructor)

Constructor (hàm khởi tạo) là một phương thức đặc biệt được gọi tự động khi một đối tượng của lớp được tạo ra. Constructor thường được sử dụng để khởi tạo các thuộc tính của đối tượng và thực hiện bất kỳ thao tác chuẩn bị cần thiết nào.

* Khởi Tạo: Constructor được gọi ngay khi đối tượng được tạo ra, đảm bảo rằng tất cả các thuộc tính của đối tượng đều được khởi tạo đúng cách.
* Nhiều Dạng: Có thể có nhiều constructor (constructor overload) với các tham số khác nhau để khởi tạo đối tượng theo nhiều cách khác nhau.
* Không Trả Về Giá Trị: Constructor không có kiểu trả về, ngay cả void.

Ví dụ: class Rectangle {

private:

double width, height;

public:

// Constructor không tham số

Rectangle(){

this.width = this.height = 0;

}

// Constructor với tham số

Rectangle(double w, double h){

this.width = w;

this.height = h;

}

};

### 2.5.2. Hủy tử (Destructor)

Destructor (hàm hủy) là một phương thức đặc biệt được gọi tự động khi đối tượng bị hủy hoặc khi ra khỏi phạm vi của nó. Destructor thường được sử dụng để giải phóng tài nguyên mà đối tượng đã chiếm dụng, chẳng hạn như bộ nhớ động hoặc các tài nguyên hệ thống khác.

* Dọn Dẹp: Destructor được gọi khi đối tượng bị hủy, điều này giúp dọn dẹp các tài nguyên mà đối tượng chiếm dụng.
* Một Dạng: Mỗi lớp chỉ có một destructor duy nhất. Destructor không nhận tham số và không trả về giá trị.
* Tự Động Gọi: Destructor được gọi tự động khi đối tượng ra khỏi phạm vi hoặc bị hủy. Bạn không cần phải gọi destructor trực tiếp.

Ví dụ: class Rectangle {

private:

double width, height;

public:

//Constructor

Rectangle(double w, double h){

this.width = w;

this.height = h;

}

// Destructor

~Rectangle() {

}

};

### 2.6. Lớp mẫu (Template class)

Lớp mẫu là một lớp có thể hoạt động với các kiểu dữ liệu khác nhau được chỉ định khi lớp được sử dụng. Thay vì định nghĩa một lớp cho mỗi kiểu dữ liệu cụ thể, bạn chỉ cần định nghĩa một lớp mẫu duy nhất.

Cú pháp: template <typename T>

class ClassName {

public:

T data; // Thành viên của lớp với kiểu dữ liệu T

// Constructor ClassName(T value){

data = value

}

// Phương thức để trả về giá trị data

T getData(){

return data;

}

// Phương thức để thiết lập giá trị data

void setData(T value) {

data = value;

}

};

Ví dụ: template <typename T>

class Box {

private:

T content;

public:

Box(T value){

content = value;

}

T getContent(){

return content;

}

void setContent(T value) {

content = value;

}

};

int main() {

int Box<int> intBox(123);

cout << "Content of intBox: " << intBox.getContent() << endl;

Box<double> doubleBox(45.67);

cout << "Content of doubleBox: " << doubleBox.getContent() << endl;

Box<string> stringBox("Hello, Templates!");

cout << "Content of stringBox: " << stringBox.getContent() << endl;

return 0;

}