**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;

}

# Chương III. Phân tích thuật toán

## 3.1. Thuật toán là gì?

Thuật toán (algorithm) là một chuỗi các bước hoặc hướng dẫn cụ thể, rõ ràng và hữu hạn được thiết kế để giải quyết một vấn đề hoặc thực hiện một nhiệm vụ nhất định. Các thuật toán có thể được thực hiện bằng tay hoặc thông qua máy tính, và chúng được sử dụng rộng rãi trong nhiều lĩnh vực khác nhau như toán học, khoa học máy tính, kỹ thuật, và nhiều ngành khoa học khác.

Đặc điểm của thuật toán:

* Rõ ràng và xác định: Mỗi bước trong thuật toán phải được xác định một cách rõ ràng và không có sự mơ hồ.
* Tính hữu hạn: Thuật toán phải kết thúc sau một số bước hữu hạn.
* Tính đầu vào: Thuật toán có thể có các đầu vào từ tập dữ liệu đã cho.
* Tính đầu ra: Thuật toán phải có ít nhất một kết quả đầu ra.
* Tính hiệu quả: Thuật toán nên được thiết kế sao cho hiệu quả nhất có thể, tối ưu về thời gian và không gian bộ nhớ.

## 3.2. Các khía cạnh cần phân tích

### 3.2.1. Độ phức tạp không gian (Space Complexity)

Độ phức tạp không gian của một thuật toán là lượng bộ nhớ cần thiết để thực hiện thuật toán, tính theo kích thước của đầu vào. Nó đo lường tổng bộ nhớ mà thuật toán yêu cầu trong quá trình thực thi, bao gồm cả bộ nhớ cố định và bộ nhớ động.

Các thành phần của độ phức tạp không gian:

* Bộ nhớ cố định (Fixed part):
* Bộ nhớ cần thiết cho các biến cố định, hằng số, và mã lệnh.
* Đây là phần bộ nhớ không phụ thuộc vào kích thước của đầu vào. Ví dụ: các biến toàn cục, hằng số, hoặc mã lệnh.
* Thường được ký hiệu là O(1).
* Bộ nhớ động (Variable part):
  + Bộ nhớ cần thiết cho các cấu trúc dữ liệu thay đổi theo kích thước của đầu vào, chẳng hạn như mảng, danh sách liên kết, hoặc cây.
  + Phần này phụ thuộc vào kích thước của đầu vào và có thể thay đổi khi kích thước đầu vào thay đổi. Ví dụ: mảng có kích thước n sẽ cần O(n) bộ nhớ.
  + Bộ nhớ động thường được ký hiệu là O(n), O(n^2), tùy thuộc vào cách cấu trúc dữ liệu được sử dụng và cách chúng thay đổi theo kích thước đầu vào.

Cách tính độ phức tạp không gian:

* Bộ nhớ cố định: Các biến đơn giản như số nguyên, số thực, hoặc ký tự thường sử dụng O(1) bộ nhớ.
* Bộ nhớ động: Một mảng có kích thước n sẽ sử dụng O(n) bộ nhớ. Một danh sách liên kết có n phần tử cũng sẽ sử dụng O(n) bộ nhớ. Một cây có n nút sẽ sử dụng O(n) bộ nhớ.

Việc tính toán độ phức tạp không gian giúp xác định lượng bộ nhớ tối thiểu mà thuật toán yêu cầu để thực thi, từ đó giúp tối ưu hóa việc sử dụng bộ nhớ và tránh tình trạng thiếu bộ nhớ khi xử lý các tập dữ liệu lớn.

### 3.2.2. Độ phức tạp thời gian (Time Complexity)

Độ phức tạp thời gian của một thuật toán là số bước cần thiết để thực hiện thuật toán, tính theo kích thước của đầu vào. Nó đo lường tổng số lần thực hiện các thao tác cơ bản của thuật toán khi kích thước của đầu vào thay đổi.

Ký hiệu phổ biến:

* Big O (O-notation): Mô tả thời gian chạy trong trường hợp xấu nhất.
* Omega (Ω-notation): Mô tả thời gian chạy trong trường hợp tốt nhất.
* Theta (Θ-notation): Mô tả thời gian chạy trung bình, nghĩa là thuật toán luôn chạy trong thời gian này bất kể trường hợp nào.

Các mức độ phức tạp thời gian phổ biến:

* O(1): Thời gian chạy không phụ thuộc vào kích thước của đầu vào. Các thao tác truy cập phần tử trong mảng hoặc so sánh hai số nguyên đều có độ phức tạp thời gian O(1).
* O(log n): Thời gian chạy tỷ lệ thuận với logarit của kích thước đầu vào. Ví dụ, thuật toán tìm kiếm nhị phân có độ phức tạp thời gian O(log n).
* O(n): Thời gian chạy tỷ lệ thuận với kích thước đầu vào. Ví dụ, duyệt qua một mảng có n phần tử sẽ có độ phức tạp thời gian O(n).
* O(n log n): Thời gian chạy là sự kết hợp của O(n) và O(log n). Các thuật toán sắp xếp như merge sort và quicksort có độ phức tạp thời gian O(n log n).
* O(n2): Thời gian chạy tỷ lệ thuận với bình phương của kích thước đầu vào. Các thuật toán sắp xếp đơn giản như bubble sort, insertion sort có độ phức tạp thời gian O(n^2).
* O(2n): Thời gian chạy tăng theo hàm mũ với kích thước đầu vào, rất kém hiệu quả. Ví dụ, thuật toán quay lui (backtracking) trong một số trường hợp cụ thể có thể có độ phức tạp thời gian O(2n).
* O(n!): Thời gian chạy tăng theo giai thừa của kích thước đầu vào, cực kỳ kém hiệu quả. Một số bài toán tổ hợp cụ thể có thể có độ phức tạp thời gian O(n!).

Cách tính độ phức tạp thời gian:

* Dựa trên số lượng thao tác: Đếm số lần thực hiện các thao tác cơ bản như gán giá trị, so sánh, và các phép toán số học.
* Dựa trên vòng lặp: Xác định số lần lặp của các vòng lặp, bao gồm cả vòng lặp lồng nhau.
* Dựa trên đệ quy: Phân tích lời gọi đệ quy và xác định số lần gọi đệ quy theo chiều sâu của cây đệ quy.

Việc tính toán độ phức tạp thời gian giúp xác định tốc độ thực thi của thuật toán khi kích thước đầu vào thay đổi, từ đó giúp tối ưu hóa hiệu suất và lựa chọn thuật toán phù hợp với yêu cầu về thời gian.

## 3.3. Các phương pháp phân tích

### 3.3.1. Phương pháp thực nghiệm

Phương pháp thực nghiệm liên quan đến việc triển khai và chạy thuật toán trên các tập dữ liệu thực tế hoặc giả lập để quan sát và đo lường hiệu suất của thuật toán.

Quy trình thực hiện:

* Triển khai thuật toán: Viết mã nguồn của thuật toán trong ngôn ngữ lập trình mong muốn.
* Chuẩn bị tập dữ liệu: Lựa chọn hoặc tạo ra các tập dữ liệu đầu vào phù hợp, bao gồm các trường hợp tốt nhất, xấu nhất và trung bình.
* Chạy thuật toán: Thực hiện thuật toán với các tập dữ liệu đã chuẩn bị.
* Đo lường hiệu suất: Sử dụng các công cụ và phương pháp đo lường để ghi lại thời gian chạy, sử dụng bộ nhớ và các chỉ số khác.
* Phân tích kết quả: So sánh và phân tích kết quả thu được để đánh giá hiệu suất của thuật toán.

Ưu điểm:

* Chính xác: Cung cấp số liệu chính xác về hiệu suất của thuật toán trong thực tế.
* Trực quan: Dễ dàng quan sát và hiểu được cách thuật toán hoạt động với các tập dữ liệu cụ thể.
* Phát hiện lỗi: Giúp phát hiện và sửa lỗi trong quá trình triển khai thuật toán.

Nhược điểm:

* Tốn thời gian: Cần nhiều thời gian để triển khai, chuẩn bị dữ liệu và chạy thử.
* Phụ thuộc vào phần cứng: Hiệu suất có thể bị ảnh hưởng bởi cấu hình phần cứng và môi trường chạy thử.
* Không tổng quát: Kết quả có thể không phản ánh chính xác hiệu suất của thuật toán trên mọi tập dữ liệu và điều kiện khác nhau.

### 3.3.2. Phương pháp phân tích lý thuyết

Phân tích lý thuyết là phương pháp sử dụng các công cụ toán học để đánh giá hiệu suất của thuật toán mà không cần triển khai thực tế.

Quy trình thực hiện:

* Xác định các thao tác cơ bản: Chọn ra các thao tác cơ bản của thuật toán để làm cơ sở đánh giá.
* Xác định các trường hợp: Phân tích thuật toán trong các trường hợp tốt nhất, xấu nhất và trung bình.
* Tính toán độ phức tạp: Sử dụng các ký hiệu Big O, Ω, và Θ để mô tả độ phức tạp thời gian và không gian của thuật toán.
* So sánh và đánh giá: So sánh độ phức tạp của thuật toán với các thuật toán khác để đưa ra kết luận về hiệu suất tương đối.

Ưu điểm:

* Tổng quát: Cung cấp đánh giá tổng quát về hiệu suất của thuật toán mà không phụ thuộc vào môi trường cụ thể.
* Tiết kiệm thời gian: Không cần triển khai thực tế và chạy thử, giúp tiết kiệm thời gian và công sức.
* Phân tích mọi trường hợp: Cho phép đánh giá hiệu suất trong mọi trường hợp đầu vào, bao gồm trường hợp tốt nhất, xấu nhất và trung bình.

Nhược điểm:

* Trừu tượng: Có thể khó hiểu và không trực quan đối với những người không quen với toán học và lý thuyết tính toán.
* Không chính xác tuyệt đối: Mặc dù cung cấp một cái nhìn tổng quát, nhưng phân tích lý thuyết không thể thay thế hoàn toàn việc kiểm tra thực tế.
* Phức tạp: Yêu cầu kiến thức toán học sâu rộng và kỹ năng phân tích để thực hiện chính xác.

# Chương IV. Các cấu trúc dữ liệu cơ bản

## 4.1. Cấu trúc Vector

### 4.1.1. Cấu trúc tuyến tính, phi tuyến

Cấu trúc dữ liệu tuyến tính là một cấu trúc mà các phần tử dữ liệu được liên kết với nhau tuần tự. Một phần tử chỉ kết nối với một phần tử trước và một tử sau.

Bằng cách tổ chức như vậy, ta có thể duyệt qua các phần tử của nó trong một lần chạy.

Ví dụ: Danh sách (lists)

Vector, chuỗi (vectors sequences)

Danh sách kiểu ngăn xếp, danh sách kiểu hàng đợi (stack, queue)

Cấu trúc dữ liệu phi tuyến là một cấu trúc mà các phần tử dữ liệu được liên kết với nhau thứ tự tuyến tính. Một phần tử có thể kết nối với nhiều phần tử trước, sau.

Ví dụ: Cây (tree)

Đồ thị (graph)

Cấu trúc dữ liệu tuyến tính và phi tuyến đều có những đặc điểm và ứng dụng riêng. Việc lựa chọn cấu trúc dữ liệu phù hợp phụ thuộc vào bài toán cụ thể và yêu cầu về hiệu suất cũng như cách tổ chức dữ liệu. Hiểu rõ về từng loại cấu trúc dữ liệu và cách chúng hoạt động sẽ giúp lập trình viên tối ưu hóa chương trình và giải quyết các vấn đề phức tạp một cách hiệu quả.

### 4.1.2. Kiểu dữ liệu Vector

Vector là một loại cấu trúc dữ liệu mạnh mẽ và linh hoạt, được cung cấp bởi thư viện chuẩn (<vector>). Nó thuộc nhóm cấu trúc dữ liệu tuyến tính và là một trong những lớp dữ liệu quan trọng nhất trong lập trình C++ nhờ tính năng động và hiệu quả của nó.

Đặc điểm chính của Vector:

* Kích Thước Động: Vector có thể thay đổi kích thước trong thời gian thực. Bạn có thể thêm hoặc xóa phần tử mà không cần phải xác định kích thước trước.
* Truy Cập Ngẫu Nhiên (Random Access): Bạn có thể truy cập bất kỳ phần tử nào của vector trực tiếp thông qua chỉ số, tương tự như trong mảng.
* Quản Lý Bộ Nhớ: Vector tự động quản lý bộ nhớ. Khi kích thước của vector thay đổi, bộ nhớ được cấp phát hoặc giải phóng tự động.
* Hiệu Suất Tốt: Thao tác thêm phần tử vào cuối vector (sử dụng push\_back) là rất nhanh chóng. Tuy nhiên, việc chèn hoặc xóa phần tử ở giữa vector có thể tốn thời gian vì phải dịch chuyển các phần tử khác.

### 4.1.3. Cấu trúc Vector trong C++

Khai báo

* vector<T> vec\_name(n);
* vector<T> vec\_name(n, value);
* T arr = {a1, a2, .., an};
* vector<T> vec\_name(arr, arr + n);
* vector<T> vec\_name(arr, arr + sizeof(arr)/sizeof(T));

Các nhóm thao tác của Vector:

* Modifiers:
* assign(int size, int value);
* pop\_back();
* insert(position, value);
* erase(position);
* erase(start-position, end-position);
* emplace(position, element);
* emplace\_back();
* emplace\_back(value);
* swap(vector);
* clear();
* Iterators:
  + - vector<int>::iterator vec\_it;
      * begin()
      * end()
    - vector<int>::reverse\_iterator vec\_it;
    - rbegin()
    - rend()
* Capacity:
* size();
* max\_size();
* capacity();
* resize(n);
* resize(int n, int value);
* empty();
* shrink\_to\_fit();
* reserve(n);
* Element access
* at(position);
* data()
* front()
* back()
* []

### 4.1.4. Các ứng dụng của Vector

**Ứng dụng trực tiếp:**

Ứng dụng trực tiếp của vector đề cập đến việc sử dụng cấu trúc dữ liệu này để giải quyết các bài toán cụ thể hoặc thực hiện các chức năng nhất định. Những ứng dụng này làm nổi bật những lợi ích cơ bản của vector, chẳng hạn như khả năng mở rộng linh hoạt và thao tác dễ dàng.

vector có thể được sử dụng để lưu trữ và quản lý một tập hợp các đối tượng của cùng loại. Đây là một trong những ứng dụng cơ bản và phổ biến của vector.

**Ứng dụng gián tiếp:**

Ứng dụng gián tiếp của vector liên quan đến việc sử dụng vector như một phần của các cấu trúc dữ liệu hoặc thuật toán phức tạp hơn. Trong những trường hợp này, vector cung cấp một nền tảng vững chắc và hiệu quả cho việc triển khai các giải pháp phức tạp hơn.

vector có thể được sử dụng như một cấu trúc dữ liệu bổ trợ trong các thuật toán phức tạp. Ví dụ, bạn có thể sử dụng vector để lưu trữ các giá trị trung gian hoặc kết quả trong các thuật toán sắp xếp, tìm kiếm, hoặc phân tách.

vector có thể đóng vai trò là thành phần của các cấu trúc dữ liệu phức tạp hơn như ngăn xếp (stack), hàng đợi (queue), hoặc đồ thị (graph). Nhờ vào khả năng mở rộng và quản lý bộ nhớ tự động, vector thường được sử dụng để xây dựng các cấu trúc dữ liệu này.

vector cũng có thể được sử dụng trong các ứng dụng phức tạp như xử lý đồ họa, mô phỏng, hoặc phân tích dữ liệu.

### 4.1.5. Cài đặt Vector bằng mảng

Khởi tạo mảng:

* Một con trỏ để trỏ đến mảng động V. Mảng này sẽ được sử dụng để lưu trữ các phần tử của vector.
* Một biến để lưu kích thước hiện tại của mảng (size), và một biến để lưu sức chứa tối đa của mảng (capacity).

Cung cấp các phương thức cơ bản:

* Thêm phần tử (push\_back): Kiểm tra xem mảng hiện tại có đủ sức chứa để thêm phần tử mới không, nếu không đủ, tăng kích thước của mảng. Sau đó thêm phần tử vào mảng và cập nhật kích thước.
* Truy cập phần tử (operator[] ): Cung cấp phương thức để truy cập phần tử tại chỉ số nhất định, với kiểm tra lỗi nếu chỉ số nằm ngoài phạm vi của mảng.
* Truy cập phần tử đầu tiên và cuối cùng (front, back): Phương thức để truy cập phần tử đầu tiên và cuối cùng của vector. Đảm bảo kiểm tra nếu vector rỗng.
* Lấy số phần tử (size): Cung cấp phương thức để trả về số lượng phần tử hiện tại trong vector.
* Kiểm tra nếu vector rỗng (empty): Cung cấp phương thức để kiểm tra xem vector có chứa bất kỳ phần tử nào không.
* Xóa tất cả các phần tử (clear): Cung cấp phương thức để xóa tất cả các phần tử trong vector, thường bằng cách đặt kích thước về 0.

### 4.1.6. Phát triển mảng:

Khi thực hiện phép toán. Nếu mảng đầy sẽ dẫn đến xảy ra lỗi. Để có thể thêm phần tử đó vào ta phải mở rộng mảng.

Có hai chiến lược để có thể mở rộng mảng:

* Chiến lược phát triển theo hằng số: Tăng thêm kích thước mảng theo một hằng số c
* Chiến lược gấp đôi: Tăng gấp đôi số phần tử hiện có của mảng

**Chiến lược phát triển theo hằng số:**

Khi mảng cần mở rộng, kích thước của mảng mới được tăng thêm một số lượng cố định (c) phần tử.

Ưu điểm:

* Dễ dàng cài đặt: Cài đặt chiến lược này thường đơn giản hơn vì chỉ cần thêm một số phần tử cố định.
* Dự đoán kích thước: Có thể dễ dàng dự đoán kích thước của mảng mới sau mỗi lần mở rộng.

Nhược điểm:

* Hiệu suất kém: Thao tác mở rộng mảng có thể xảy ra thường xuyên hơn, dẫn đến nhiều thao tác sao chép hơn. Ví dụ, nếu bạn mở rộng mảng mỗi lần thêm 10 phần tử, và bạn liên tục thêm nhiều phần tử, bạn sẽ phải sao chép mảng nhiều lần.
* Tài nguyên bộ nhớ: Khi mảng mở rộng, bạn có thể cấp phát nhiều bộ nhớ hơn cần thiết, dẫn đến lãng phí bộ nhớ.

**Chiến lược gấp đổi:**

Khi mảng cần mở rộng, kích thước của mảng mới được gấp đôi so với kích thước hiện tại.

Ưu điểm:

* Hiệu suất tốt hơn: Chiến lược này giảm số lần mở rộng mảng cần thiết. Vì mảng được gấp đôi kích thước mỗi lần mở rộng, số lần mở rộng sẽ ít hơn nhiều so với chiến lược phát triển theo hằng số.
* Tối ưu hóa tài nguyên: Mảng sẽ có kích thước lớn hơn cần thiết ít hơn, dẫn đến ít lãng phí bộ nhớ hơn so với việc mở rộng theo hằng số.

Nhược điểm:

* Cài đặt phức tạp hơn: Cần phải tính toán kích thước mới của mảng mỗi lần mở rộng, điều này có thể phức tạp hơn một chút so với việc mở rộng theo hằng số.
* Sự lãng phí đôi lúc tạo mảng: Mặc dù có thể tiết kiệm bộ nhớ tổng thể, nhưng mỗi lần mở rộng có thể dẫn đến việc cấp phát một lượng bộ nhớ lớn hơn cần thiết, gây lãng phí bộ nhớ tạm thời.



### 4.1.7. Iterator (Bộ lặp)

Iterator, hay còn gọi là bộ lặp, là một đối tượng trong lập trình cung cấp cách để duyệt qua các phần tử trong một cấu trúc dữ liệu, chẳng hạn như mảng, danh sách, hay vector, mà không cần biết chi tiết về cách dữ liệu được tổ chức bên trong.

Iterator cho phép người dùng truy cập và duyệt qua các phần tử của một cấu trúc dữ liệu theo cách tuần tự và nhất quán. Nó tách biệt việc duyệt qua các phần tử khỏi cấu trúc dữ liệu cụ thể, do đó hỗ trợ việc thay đổi cấu trúc dữ liệu mà không làm thay đổi mã duyệt qua.

Iterator thường có các thành phần sau:

* begin(): Trả về một iterator trỏ đến phần tử đầu tiên của cấu trúc dữ liệu.
* end(): Trả về một iterator trỏ đến vị trí ngay sau phần tử cuối cùng của cấu trúc dữ liệu (thường được sử dụng để xác định điểm dừng của vòng lặp).
* operator\*: Truy cập giá trị của phần tử mà iterator đang trỏ đến.
* operator++: Di chuyển iterator đến phần tử tiếp theo trong cấu trúc dữ liệu.
* operator!=: So sánh hai iterator để kiểm tra xem chúng có trỏ đến cùng một vị trí hay không.

Ví dụ: vector<int> V(7, 6);

vector<int>::iterator it;

for(it = V.begin(); it != V.end(); ++it){

cout << \*it << "\t";

}

Chèn thêm phần tử:

* iterator insert (const\_iterator position, const value\_type& val)
* Chúng ta cần tạo một ô mới có chỉ số r bằng cách đẩy n - r phần tử từ

V[r], ..., V[n - 1] về sau 1 vị trí

* Trong trường hợp xấu nhất (r = 0), phép toán thực hiện trong thời gian O(n)

Loại bỏ phần tử:

* Phép toán iterator erase (const\_iterator position) chúng ta cần đẩy n - r – 1 phần tử từ V[r + 1], ..., V[n - 1] về trước một vị trí
* Trong trường hợp xấu nhất (r = 0), phép toán thực hiện trong thời gian O(n)