TRƯỜNG CÔNG NGHỆ THÔNG TIN & TRUYỀN THÔNG
ĐẠI HỌC BÁCH KHOA HÀ NỘI

# BÀI GIẢNG HỌC PHẦN KỸ THUẬT LẬP TRÌNH

**Nhóm tác giả:**
**Vũ Đức Vượng**
**Bùi Thị Mai Anh**

# Lời nói đầu

Sau quá trình giảng dạy trên 10 năm, với những kinh nghiệm được tích lũy, cùng các ý kiến đống góp của đồng nghiệp và người học, chúng tôi tiến hành thực hiện biên tập cuốn sách này, nhằm cung cấp cho các đồng nghiệp khác và sinh viên có thêm tài liệu để giảng dạy và học tập có kết quả tốt hơn. Nội dung tài liệu dựa trên các slides bài giảng Kỹ thuật lập trình cùng các nội dung bổ sung, mở rộng nhằm giúp người đọc nắm vững hơn, sâu hơn và rộng hơn các kiến thức trên slides.

Hà Nội, ngày 20 tháng 09 năm 2023

**Vũ Đức Vượng**

# Mục lục

# Chương 1
# Tổng quan về Kỹ thuật lập trình



## 1.1. Giới thiệu môn học Kỹ thuật lập trình

Kỹ thuật lập trình (KTLT) là một môn học cốt lõi trong chương trình đào tạo ngành Công nghệ Thông tin. Môn học này trang bị cho sinh viên những kiến thức nền tảng và kỹ năng cần thiết để có thể viết, kiểm thử và duy trì các chương trình máy tính từ đơn giản đến phức tạp. Ngoài ra, KTLT còn giúp sinh viên hiểu rõ các phương pháp luận khi lập trình, cách tổ chức và thiết kế một chương trình sao cho tối ưu và dễ bảo trì. Kỹ năng lập trình không chỉ là việc viết mã lệnh; nó bao gồm việc phân tích vấn đề, thiết kế thuật toán, chọn lựa các cấu trúc dữ liệu phù hợp, và triển khai chúng bằng cách sử dụng ngôn ngữ lập trình. Môn học này cũng giới thiệu về các nguyên lý cơ bản của lập trình như nguyên lý DRY (Don't Repeat Yourself - Đừng lặp lại chính mình), nguyên lý KISS (Keep It Simple, Stupid - Giữ cho mọi thứ đơn giản), và nguyên lý SOLID trong thiết kế phần mềm.

## 1.2. Nội dung của môn học

Kỹ thuật lập trình bao gồm nhiều chủ đề khác nhau, trải dài từ những kiến thức cơ bản đến các kỹ thuật nâng cao. Dưới đây là mô tả chi tiết về các chương trong môn học:

- Chương 1: Tổng quan về Kỹ thuật lập trình (KTLT)

Chương đầu tiên giới thiệu các khái niệm cốt lõi của lập trình, như chương trình máy tính, ngôn ngữ lập trình, và phân loại các ngôn ngữ lập trình. Sinh viên sẽ hiểu được vai trò quan trọng của lập trình trong phát triển phần mềm, cùng với các yếu tố tạo nên một chương trình phần mềm chất lượng.

- Chương 2: Vài kiến thức nâng cao về C và C++

  Chương này tập trung vào việc sử dụng ngôn ngữ C và C++ trong lập trình. Sinh viên sẽ học cách quản lý bộ nhớ, sử dụng con trỏ, và tối ưu hóa mã nguồn. Ngoài ra, các khái niệm về lập trình hướng đối tượng (OOP) trong C++ cũng được giới thiệu, như các khái niệm về lớp, đối tượng, kế thừa, và đa hình.

- Chương 3: Các kỹ thuật viết code hiệu quả và phong cách lập trình

  Trong chương này, sinh viên sẽ học cách viết mã nguồn rõ ràng, dễ bảo trì, và tuân thủ các phong cách lập trình tốt. Điều này bao gồm việc sử dụng tên biến và hàm một cách có ý nghĩa, tổ chức mã nguồn theo các module hợp lý, và tránh các lỗi lập trình thường gặp.

- Chương 4: Một số cấu trúc dữ liệu và giải thuật căn bản

  Chương này giới thiệu các cấu trúc dữ liệu cơ bản như mảng, danh sách liên kết, ngăn xếp, hàng đợi, cây nhị phân, và đồ thị. Các giải thuật cơ bản liên quan đến việc sắp xếp, tìm kiếm, và duyệt cây cũng được đề cập. Sinh viên sẽ học cách chọn lựa cấu trúc dữ liệu và giải thuật phù hợp cho từng bài toán cụ thể.

- Chương 5: Các kỹ thuật bẫy lỗi và lập trình phòng ngừa

  Khả năng phát hiện và xử lý lỗi trong lập trình là rất quan trọng. Chương này hướng dẫn cách sử dụng các công cụ và kỹ thuật để phát hiện lỗi trong quá trình viết mã, cũng như cách thiết kế chương trình để phòng ngừa lỗi (error prevention). Điều này giúp tăng độ tin cậy của phần mềm và giảm thời gian sửa lỗi.

- Chương 6: Testing

  Kiểm thử phần mềm là một bước quan trọng trong quá trình phát triển phần mềm. Chương này trình bày các phương pháp kiểm thử như kiểm thử đơn vị (unit testing), kiểm thử tích hợp (integration testing), và kiểm thử hệ thống (system testing). Sinh viên sẽ học cách viết các trường hợp kiểm thử (test cases) và sử dụng các công cụ kiểm thử tự động.

- Chương 7: Code tuning và documentation

  Chương cuối cùng tập trung vào việc tối ưu hóa mã nguồn để cải thiện hiệu năng chương trình (code tuning) và viết tài liệu hướng dẫn sử dụng cũng như tài liệu cho lập trình viên (documentation). Điều này rất quan trọng trong việc bảo trì và phát triển phần mềm trong dài hạn.

## 1.3. Phương pháp học tập hiệu quả

Để thành công trong môn học Kỹ thuật lập trình, sinh viên cần phải tuân thủ một số phương pháp học tập sau:

- Tham gia đầy đủ và tích cực vào các buổi học:

  Việc tham gia đầy đủ các buổi học giúp sinh viên tiếp thu kiến thức một cách liên tục và có cơ hội đặt câu hỏi, thảo luận về các vấn đề khó hiểu. Sự tương tác giữa giảng viên và sinh viên trong các buổi học giúp củng cố kiến thức và kỹ năng.

- Hoàn thành bài tập đúng hạn:

  Bài tập là phương tiện để rèn luyện kỹ năng lập trình. Sinh viên nên tự làm bài tập và nộp đúng hạn để không bị mất điểm và cũng để nắm vững kiến thức qua việc thực hành.

- Tự học và nghiên cứu thêm:

  Ngoài tài liệu giảng dạy, sinh viên nên chủ động tìm kiếm thêm tài liệu và tham khảo các nguồn học liệu khác như sách, tài liệu trực tuyến, và các khóa học lập trình miễn phí hoặc có phí để mở rộng kiến thức.

- Thảo luận và làm việc nhóm:

  Việc thảo luận với bạn bè và làm việc nhóm giúp giải quyết các vấn đề phức tạp và học hỏi lẫn nhau. Đây cũng là cách tốt để rèn luyện kỹ năng làm việc nhóm và giao tiếp, rất cần thiết trong môi trường làm việc sau này.

## 1.4. Tổng quan về chương trình máy tính và ngôn ngữ lập trình

Chương trình máy tính là tập hợp các lệnh (instruction) được viết bằng ngôn ngữ lập trình, hướng dẫn máy tính thực hiện các nhiệm vụ cụ thể. Một chương trình có thể là một ứng dụng lớn như hệ điều hành, hoặc là các tiện ích nhỏ như một ứng dụng máy tính (calculator). Ví dụ:

- Một chương trình máy tính đơn giản có thể là một đoạn mã cộng hai số nguyên do người dùng nhập vào.
- Một chương trình phức tạp hơn có thể là một ứng dụng quản lý dữ liệu khách hàng của một công ty, cho phép lưu trữ, tìm kiếm và cập nhật thông tin khách hàng.

Ngôn ngữ lập trình là phương tiện để lập trình viên viết các chương trình máy tính. Ngôn ngữ lập trình cung cấp cú pháp (syntax) và ngữ nghĩa (semantics) cho phép người lập trình mô tả một cách chính xác các thao tác mà máy tính cần thực hiện. Ví dụ về ngôn ngữ lập trình:

- Python: Một ngôn ngữ lập trình phổ biến với cú pháp đơn giản, dễ học, và mạnh mẽ trong nhiều lĩnh vực như khoa học dữ liệu, phát triển web, và trí tuệ nhân tạo.
- Java: Một ngôn ngữ lập trình hướng đối tượng, mạnh mẽ, đa nền tảng, thường được sử dụng trong phát triển ứng dụng doanh nghiệp, ứng dụng di động, và hệ thống nhúng.

## 1.5. Phân loại ngôn ngữ lập trình

Ngôn ngữ lập trình có thể được phân loại theo nhiều tiêu chí khác nhau, nhưng phổ biến nhất là dựa trên cách tiếp cận lập trình (Programming Paradigms).

**Ngôn ngữ lập trình thủ tục (Procedural Programming):** Các ngôn ngữ như C, Pascal là các ngôn ngữ lập trình thủ tục, nơi chương trình được cấu trúc thành các hàm hoặc thủ tục nhỏ hơn. Mỗi thủ tục thực hiện một nhiệm vụ cụ thể và chương trình được thực thi theo một chuỗi các lệnh.

- Ưu điểm: Dễ hiểu, dễ quản lý với các chương trình nhỏ và vừa.
- Nhược điểm: Khó bảo trì và mở rộng khi chương trình trở nên lớn hơn.

**Ngôn ngữ lập trình hướng đối tượng (Object-Oriented Programming - OOP):** OOP là một mô hình lập trình dựa trên khái niệm về đối tượng (objects), mỗi đối tượng là một thực thể có dữ liệu và phương thức liên quan. Các ngôn ngữ phổ biến như C++, Java, và Python hỗ trợ OOP.

- Ưu điểm: Dễ mở rộng, dễ bảo trì, mã nguồn rõ ràng hơn nhờ vào các khái niệm như kế thừa (inheritance) và đóng gói (encapsulation).

- Nhược điểm: Đôi khi phức tạp hơn đối với các vấn đề đơn giản hoặc đối với người mới học lập trình.

**Ngôn ngữ lập trình hàm (Functional Programming):** Ngôn ngữ như Haskell, Lisp, và đôi khi là Python (có hỗ trợ lập trình hàm) khuyến khích viết các hàm thuần túy không có trạng thái. Đây là một phương pháp lập trình trong đó các hàm, không thay đổi trạng thái và không có tác dụng phụ, là đơn vị cơ bản.

- Ưu điểm: Thích hợp cho các tính toán phức tạp, dễ phân tích và kiểm thử.
- Nhược điểm: Khó hiểu đối với những người đã quen với lập trình thủ tục hoặc hướng đối tượng.

**Ngôn ngữ lập trình logic (Logic Programming):** Ngôn ngữ như Prolog sử dụng logic để mô tả các mối quan hệ giữa các đối tượng và quy tắc suy luận. Thay vì viết các thuật toán cụ thể, lập trình viên sẽ mô tả vấn đề cần giải quyết và ngôn ngữ sẽ tự động tìm kiếm lời giải.

- Ưu điểm: Phù hợp với các bài toán về trí tuệ nhân tạo và tự động hóa lý luận.
- Nhược điểm: Khó áp dụng trong các ứng dụng thực tế với yêu cầu phức tạp về hiệu suất. Mức độ gần gũi với phần cứng (Level of Abstraction):

**Ngôn ngữ bậc thấp (Low-Level Languages):** Ngôn ngữ như Assembly là ngôn ngữ bậc thấp, cho phép lập trình viên thao tác trực tiếp với phần cứng. Các lệnh trong Assembly tương ứng với các lệnh máy (machine code) của vi xử lý, do đó chúng rất nhanh nhưng khó viết và duy trì.

**Ngôn ngữ bậc cao (High-Level Languages):** Các ngôn ngữ như Python, Java, và C++ là ngôn ngữ bậc cao. Chúng cung cấp các cấu trúc ngôn ngữ trừu tượng hơn, giúp lập trình viên tập trung vào logic của chương trình thay vì chi tiết phần cứng.

- Ưu điểm: Dễ viết, dễ đọc, và dễ duy trì.
- Nhược điểm: Hiệu suất có thể thấp hơn so với ngôn ngữ bậc thấp do có nhiều lớp trừu tượng hơn.

## 1.6. Đặc điểm của các ngôn ngữ lập trình

Mỗi ngôn ngữ lập trình có những đặc điểm riêng, điều này ảnh hưởng đến việc chọn lựa ngôn ngữ phù hợp cho từng loại dự án. Một số đặc điểm nổi bật bao gồm:

- Tính dễ học (Ease of Learning): Ví dụ, Python được coi là một trong những ngôn ngữ dễ học nhất do cú pháp đơn giản và cộng đồng hỗ trợ mạnh mẽ. Điều này khiến nó trở thành lựa chọn lý tưởng cho người mới bắt đầu.
- Tính phổ biến (Popularity): Các ngôn ngữ như Java, Python và JavaScript được sử dụng rộng rãi trong nhiều lĩnh vực khác nhau, từ phát triển ứng dụng doanh nghiệp đến trí tuệ nhân tạo và phát triển web. Điều này đảm bảo rằng có nhiều tài liệu học tập, công cụ hỗ trợ, và cơ hội việc làm.
- Tính linh hoạt (Flexibility): Một ngôn ngữ linh hoạt cho phép lập trình viên làm việc trong nhiều lĩnh vực khác nhau mà không cần phải học nhiều ngôn ngữ khác nhau. Python là một ví dụ điển hình, nó được sử dụng trong khoa học dữ liệu, phát triển web, phát triển phần mềm nhúng, và nhiều lĩnh vực khác.
- Tính bảo mật (Security): Các ngôn ngữ như Java và Rust được thiết kế với các tính năng bảo mật mạnh mẽ, giúp giảm thiểu các lỗ hổng bảo mật thường gặp như lỗi tràn bộ nhớ đệm (buffer overflow). Điều này rất quan trọng trong phát triển các hệ thống nhúng, ứng dụng tài chính, và các hệ thống yêu cầu độ an toàn cao.

Ví dụ về các ngôn ngữ lập trình phổ biến

- C: Ngôn ngữ lập trình thủ tục, thường được sử dụng trong phát triển hệ điều hành và các ứng dụng hệ thống nhúng. C nổi bật với khả năng quản lý bộ nhớ thủ công và hiệu năng cao.
- Java: Ngôn ngữ lập trình hướng đối tượng, phổ biến trong phát triển ứng dụng doanh nghiệp và ứng dụng di động (Android). Java chạy trên máy ảo Java (JVM), giúp đảm bảo tính di động giữa các nền tảng.
- Python: Ngôn ngữ bậc cao với cú pháp đơn giản, được ưa chuộng trong lĩnh vực khoa học dữ liệu, trí tuệ nhân tạo, và phát triển web.
- JavaScript: Ngôn ngữ lập trình web chính, chạy trên trình duyệt, giúp tạo ra các ứng dụng web tương tác.

## 1.7. Ngôn ngữ lập trình thủ tục

Ngôn ngữ lập trình thủ tục (Procedural Programming Languages) là một trong những phương pháp lập trình truyền thống và phổ biến nhất. Trong mô hình này, chương trình được chia thành các thủ tục hoặc hàm, mỗi thủ tục đảm nhiệm một nhiệm vụ cụ thể. Điều này giúp tổ chức mã nguồn thành các phần nhỏ hơn, dễ quản lý và bảo trì hơn. Ví dụ: Trong ngôn ngữ C, chúng ta có thể tạo ra một hàm để tính tổng hai số và sau đó gọi hàm này bất cứ khi nào cần thực hiện phép cộng, thay vì phải viết lại mã cho phép cộng nhiều lần.

Các đặc điểm chính của ngôn ngữ lập trình thủ tục bao gồm:

- Dòng chảy tuyến tính (Linear Flow): Chương trình thường được thực thi theo một trình tự tuyến tính, từ trên xuống dưới, theo thứ tự của các câu lệnh. Điều này giúp lập trình viên dễ dàng theo dõi quá trình thực thi của chương trình.

- Tính mô-đun (Modularity): Khả năng chia nhỏ chương trình thành các thủ tục giúp cải thiện khả năng bảo trì và tái sử dụng mã nguồn. Ví dụ, nếu cần thay đổi cách tính tổng trong một chương trình, bạn chỉ cần thay đổi trong hàm tính tổng mà không ảnh hưởng đến phần còn lại của chương trình.

- Tính minh bạch của dữ liệu (Transparency of Data): Các ngôn ngữ lập trình thủ tục thường sử dụng các biến toàn cục (global variables) và biến cục bộ (local variables) để lưu trữ và truy xuất dữ liệu. Tuy nhiên, việc lạm dụng biến toàn cục có thể dẫn đến mã nguồn khó hiểu và khó bảo trì.

- Ví dụ về ngôn ngữ lập trình thủ tục: BASIC, COBOL, Pascal, C, và C++ đều là những ngôn ngữ lập trình thủ tục tiêu biểu.

Ngoài ra, trong lập trình thủ tục, chúng ta có thể sử dụng hai phương pháp chính để thực thi mã:

- Thông dịch (Interpreter):

  Trong phương pháp này, chương trình nguồn được dịch và thực hiện từng dòng lệnh một cách đồng thời. Điều này mang lại tính linh hoạt cao, nhưng hiệu năng có thể chậm hơn so với các chương trình được biên dịch. Ví dụ: Các ngôn ngữ như Python và Ruby sử dụng thông dịch viên để thực thi mã nguồn, cho phép lập trình viên thấy ngay kết quả sau khi viết mã.

- Trình dịch (Compiler):

Trình dịch biên dịch toàn bộ chương trình nguồn thành mã máy trước khi thực thi. Điều này giúp chương trình chạy nhanh hơn, nhưng quá trình biên dịch có thể mất thời gian. Ví dụ: Ngôn ngữ C và C++ sử dụng trình biên dịch để chuyển mã nguồn thành mã máy, giúp tăng hiệu suất khi chạy chương trình.

## 1.8. Ngôn ngữ lập trình hướng đối tượng

Ngôn ngữ lập trình hướng đối tượng (Object-Oriented Programming - OOP) là một phương pháp lập trình hiện đại, tập trung vào việc mô hình hóa các thực thể trong thế giới thực dưới dạng đối tượng. OOP giúp tổ chức mã nguồn theo cách trực quan hơn, phản ánh chính xác hơn các mối quan hệ và hành vi của các đối tượng trong một hệ thống. Các đặc điểm chính của OOP bao gồm:

- Kế thừa (Inheritance): Kế thừa cho phép một lớp con (subclass) kế thừa các thuộc tính và phương thức của lớp cha (superclass). Điều này giúp giảm thiểu sự trùng lặp mã nguồn và tái sử dụng các thành phần phần mềm một cách hiệu quả. Ví dụ: Trong Java, nếu bạn có một lớp "Động vật" (Animal) với các thuộc tính chung như "tuổi" và "cân nặng", các lớp con như "Chó" (Dog) và "Mèo" (Cat) có thể kế thừa từ lớp "Động vật" và thêm các thuộc tính hoặc hành vi riêng.

- Đa hình (Polymorphism): Đa hình cho phép một phương thức có thể có nhiều hình thái khác nhau. Điều này có nghĩa là các đối tượng thuộc các lớp khác nhau có thể được xử lý theo cùng một cách nếu chúng triển khai cùng một giao diện hoặc kế thừa từ cùng một lớp cha. Ví dụ: Một phương thức "nói" (speak) có thể được triển khai khác nhau trong các lớp "Chó" và "Mèo", nhưng cả hai đều có thể được gọi thông qua lớp "Động vật" mà không cần biết chính xác đối tượng đang là loại động vật nào.

- Đóng gói (Encapsulation): Đóng gói là kỹ thuật che giấu dữ liệu và chỉ cho phép truy cập thông qua các phương thức được cung cấp bởi lớp đó. Điều này giúp bảo vệ dữ liệu khỏi các thay đổi không mong muốn và đảm bảo rằng đối tượng được sử dụng theo cách mà nhà phát triển dự định. Ví dụ: Trong Java, bạn có thể sử dụng các từ khóa "private", "protected", và "public" để kiểm soát mức độ truy cập vào các thuộc tính và phương thức của một lớp.

- Trừu tượng hóa (Abstraction): Trừu tượng hóa là quá trình đơn giản hóa một đối tượng bằng cách chỉ giữ lại những thông tin cần thiết và ẩn đi các chi tiết phức tạp. Điều này giúp giảm sự phức tạp của mã nguồn và tập trung vào các

khía cạnh quan trọng của đối tượng. Ví dụ: Một lớp "Xe hơi" có thể có các phương thức như "lái" và "dừng", nhưng không cần chi tiết cách động cơ hoạt động bên trong.

Ngôn ngữ hướng đối tượng giúp việc phát triển phần mềm trở nên dễ dàng hơn khi cần mở rộng hoặc bảo trì mã nguồn. Một số ngôn ngữ lập trình OOP phổ biến bao gồm:

- Java: Ngôn ngữ lập trình OOP phổ biến nhất, được sử dụng rộng rãi trong phát triển ứng dụng doanh nghiệp và ứng dụng di động.
- C++: Một mở rộng của ngôn ngữ C với các tính năng OOP, thường được sử dụng trong phát triển phần mềm hệ thống, trò chơi điện tử, và các ứng dụng đòi hỏi hiệu năng cao.
- Python: Hỗ trợ lập trình hướng đối tượng với cú pháp đơn giản, thích hợp cho nhiều loại dự án từ phát triển web đến trí tuệ nhân tạo.

## 1.9. Kết luận chương

Kỹ thuật lập trình là một môn học nền tảng, cung cấp cho sinh viên những công cụ và phương pháp cần thiết để phát triển các phần mềm hiệu quả và chất lượng cao. Việc nắm vững các khái niệm và kỹ thuật trong lập trình sẽ giúp sinh viên trở thành những lập trình viên giỏi, có khả năng giải quyết các vấn đề phức tạp trong thực tế.

Để thành công trong môn học này, sinh viên cần phải chủ động học tập, thực hành lập trình thường xuyên, và luôn sẵn sàng nghiên cứu, khám phá những kiến thức mới.

# Chương 2
# C/C++ Mở rộng và Quản lý bộ nhớ



## 2.1. Thứ tự ưu tiên các phép toán trong C/C++

Khi làm việc với các phép toán phức tạp, việc hiểu rõ thứ tự ưu tiên của từng toán tử là rất quan trọng. Hình 2.1 là bảng chi tiết về thứ tự ưu tiên trong C/C++.

### 2.1.1. Các phép toán tăng/giảm ++ và −− trong C/C++

Phép toán tăng ++ và giảm −− là các toán tử đơn nguyên (unary operators) được sử dụng rộng rãi trong lập trình C/C++. Chúng có thể được áp dụng trước (prefix) hoặc sau (postfix) biến để tăng hoặc giảm giá trị của biến đó.

**Toán tử tăng** ++ Toán tử tăng trước (++variable) tăng giá trị của biến lên 1 trước khi sử dụng biến trong biểu thức. Ví dụ:

```
int a = 5;
int b = ++a;  // a = 6, b = 6
```

Ở đây, a được tăng lên 6 trước khi giá trị của nó được gán cho b. Toán tử tăng sau (variable++) sử dụng giá trị hiện tại của biến trong biểu thức trước, sau đó mới tăng giá trị của biến lên 1. Ví dụ:

```
int a = 5;
int b = a++;  // a = 6, b = 5
```

Ở đây, b nhận giá trị 5 (giá trị ban đầu của a), sau đó a mới được tăng lên 6.

| Mức | Các toán tử | Trật tự kết hợp |
|---|---|---|
| 1 | () [] . -> ++ (hậu tố) -- hậu tố | ------> |
| 2 | ! ~ ++ (tiền tố) -- (tiền tố) - * & sizeof | <------ |
| 3 | * / % | ------> |
| 4 | + - | ------> |
| 5 | << >> | ------> |
| 6 | < <= > >= | ------> |
| 7 | == != | ------> |
| 8 | & | ------> |
| 9 | ^ | ------> |
| 10 | \| | ------> |
| 11 | && | ------> |
| 12 | \|\| | ------> |
| 13 | ?: | <------ |
| 14 | = += -= | <------ |

Hình 2.1 Thứ tự thực hiện các phép toán

**Toán tử giảm** $--$ Toán tử giảm trước ($--$variable) giảm giá trị của biến xuống 1 trước khi sử dụng biến trong biểu thức. Ví dụ:

```
int a = 5;
int b = --a;  // a = 4, b = 4
```

Ở đây, a được giảm xuống 4 trước khi giá trị của nó được gán cho b. Toán tử giảm sau (variable$--$) sử dụng giá trị hiện tại của biến trong biểu thức trước, sau đó mới giảm giá trị của biến xuống 1. Ví dụ:

```
int a = 5;
int b = a--;  // a = 4, b = 5
```

Ở đây, b nhận giá trị 5 (giá trị ban đầu của a), sau đó a mới được giảm xuống 4.

**Sự khác biệt giữa $++$ và $--$**

- Toán tử tăng ($++$): Sử dụng để tăng giá trị của biến lên 1. Đây là thao tác rất phổ biến trong các vòng lặp.
- Toán tử giảm ($--$): Sử dụng để giảm giá trị của biến xuống 1. Cũng được sử dụng rộng rãi trong các vòng lặp hoặc để đếm ngược.

**Tác động đến hiệu suất** Mặc dù việc sử dụng ++ và −− là rất nhanh, nhưng vị trí của chúng (trước hoặc sau) có thể ảnh hưởng đến hiệu suất trong các trường hợp cụ thể là tăng trước (++a) thường hiệu quả hơn tăng sau (a++) khi làm việc với các đối tượng phức tạp (không phải kiểu dữ liệu nguyên thủy), vì tăng sau có thể yêu cầu lưu trữ tạm thời giá trị hiện tại để sử dụng trong biểu thức trước khi tăng.

**Lưu ý về an toàn khi sử dụng ++ và −−** Tránh sử dụng trong cùng một biểu thức: Khi ++ hoặc −− được sử dụng nhiều lần trên cùng một biến trong một biểu thức, thứ tự thực hiện của các phép toán có thể không được đảm bảo, dẫn đến hành vi không xác định. Ví dụ:

```
int i = 1;
int a = i++ + ++i;  // Hành vi không xác định
```

Do đó, tránh viết các biểu thức phức tạp kết hợp ++ hoặc −− với các toán tử khác để tránh lỗi khó phát hiện.

### 2.1.2. Các phép toán cơ bản

**Nhân - Chia - Phép dư** Thứ tự trung bình cao: *, /, %. Ví dụ:

```
int z = 10 * 3 % 4;  // z = 2, vì phép nhân được thực hiện trước rồi mới đến
```

**Cộng - Trừ** Toán tử cộng và trừ, trung bình: +, -. Ví dụ:

```
int result = 5 + 3 * 2;  // result = 11, vì phép nhân có độ ưu tiên cao hơn
```

**Toán tử dịch bit** Thấp hơn: «, ». Ví dụ:

```
int a = 5 << 2;  // a = 20, vì 5 được dịch trái 2 bit (tương đương với 5 * 2
```

**Toán tử quan hệ** Thấp: <, >, <=, >=. Ví dụ:

```
bool check = 10 > 5;  // check = true.
```

**Toán tử so sánh bằng** Thấp hơn nữa: ==, !=. Ví dụ:

```
bool equal = (10 + 5) == 15;  // equal = true.
```

| Với x = 10 | | | | devc |
|---|---|---|---|---|
| Expression | Visual++ | g++ | Turbo C++ | |
| b = x + x++ | 20 | 21 | 20 | 21 |
| b = x++ + x | 20 | 21 | 20 | 21 |
| b = 2*x + x++ | 30 | 30 | 30 | 30 |
| b = ++x + x++ | 22 | 23 | 22 | 22 |
| b = x++ + ++x | 22 | 22 | 22 | 22 |
| b = x++ | 10 | 10 | 10 | 10 |
| b = x++ + 3 | 13 | 13 | 13 | 13 |
| b = x++ + x++ | 20 | 21 | 20 | 21 |
| b = x++ + (++x + 2*x) | 44 | 46 | 44 | 46 |
| b = ++x + ++x | 24 | 24 | 24 | 24 |
| b = ++x + ++x + ++x | 39 | 37 | 39 | 37 |
| b = ++x + ++x * ++x | 182 | 182 | 182 | 182 |
| ++b = ++b + b++ | 7 | 7 | 7 | |
| ++b = ++b + ++b + ++b | 15 | 13 | 15 | |

Hình 2.2 Hành vi không xác định

**Toán tử logic bitwise** Rất thấp: &, ^, |. Ví dụ:

```
int bitwiseResult = 5 & 3;  // bitwiseResult = 1, vì phép AND giữa 0101 và 0
```

**Toán tử logic** Rất rất thấp: &&, ||. Ví dụ:

```
bool logic = (5 > 3) && (3 < 2);  // logic = false, vì một điều kiện là fals
```

**Toán tử gán** Thấp nhất: =, +=, -=, *=, /=, %= và các toán tử gán khác. Ví dụ:

```
int b = 10;
b *= 2 + 3;  // b = 50, vì phép cộng được thực hiện trước rồi mới đến gán.
```

### 2.1.3. Lưu ý thực tế

- Sử dụng dấu ngoặc để rõ ràng: Trong các biểu thức phức tạp, nên sử dụng dấu ngoặc để làm rõ thứ tự các phép toán, đảm bảo rằng chương trình thực hiện theo ý định của lập trình viên.
- Tránh nhầm lẫn với toán tử gán: Một lỗi phổ biến là nhầm lẫn giữa = (gán) và == (so sánh bằng), điều này có thể gây ra các lỗi khó phát hiện.

## 2.2. Hành vi không xác định UB

**Truy cập bộ nhớ ngoài phạm vi mảng**

```
int arr[3] = {1, 2, 3};
int x = arr[5];  // Hành vi không xác định, vì truy cập ngoài phạm vi mảng.
```

**Thực thi lệnh với thứ tự không xác định**

```
int i = 1;
int arr[5];
arr[i] = i++;  // Hành vi không xác định, vì thứ tự thực hiện phép toán không
```

**Truy cập biến sau khi giải phóng bộ nhớ**

```
int *ptr = (int*)malloc(sizeof(int));
*ptr = 5;
free(ptr);
int value = *ptr;  // Hành vi không xác định, vì truy cập bộ nhớ đã giải phó
```

Phòng tránh hành vi không xác định:

- Sử dụng công cụ phát hiện lỗi:
  - Valgrind: Công cụ phân tích và phát hiện các vấn đề về bộ nhớ.
  - AddressSanitizer: Một công cụ mạnh mẽ tích hợp trong GCC và Clang giúp phát hiện lỗi truy cập bộ nhớ.
  - UBSan (Undefined Behavior Sanitizer): Phát hiện hành vi không xác định trong mã nguồn.
- Kiểm tra và xử lý lỗi cẩn thận:
  - Luôn kiểm tra biên của mảng trước khi truy cập.
  - Tránh sử dụng các biến không được khởi tạo.
  - Khi dùng con trỏ, luôn kiểm tra con trỏ null trước khi dereference.

## 2.3. Mảng (Arrays)

Chi tiết và nâng cao về mảng:

- Mảng đa chiều: Mảng đa chiều trong C/C++ là một mảng chứa các mảng. Mảng hai chiều (ma trận) là loại phổ biến nhất.

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
int value = matrix[1][2];  // value = 6, truy cập phần tử ở hàng 2, cột 3.
```

- Tên mảng là một hằng địa chỉ, có giá trị bằng địa chỉ phần tử đầu tiên của mảng, cũng chính là địa chỉ mảng

```
int M[10];
M== &M[0] => *M == M[0]
M+i  == &M[i]  => *(M+i) == M[i]
```

Lưu ý : M là 1 hằng số, ++M => Error

```
int M[10],k;
for(k=1;k<15;k++)
```

```
    k[M]=k*5;
for(k=1;k<15;k++)
    printf(" M[%d] = %d ",k,M[k]);
```

Cho nhận xét về chương trình trên ! Tại sao chỉ số của mảng trong C/C++ bắt đầu từ 0 ?

- Mảng động: Mảng động có kích thước thay đổi trong thời gian chạy, sử dụng bộ nhớ động.

```
int size = 10;
int *dynamicArray = (int*)malloc(size * sizeof(int));
for (int i = 0; i < size; i++) {
    dynamicArray[i] = i * 2;
}
free(dynamicArray);  // Giải phóng bộ nhớ khi không sử dụng.
```

- Mảng con trỏ: Mảng con trỏ là mảng mà mỗi phần tử là một con trỏ, thường được sử dụng để quản lý danh sách các chuỗi.

```
const char *names[] = {"Alice", "Bob", "Charlie"};
printf("%s", names[1]);  // Output: Bob.
```

Những sai lầm phổ biến với mảng:

- Truy cập ngoài phạm vi: Đây là một trong những nguyên nhân phổ biến dẫn đến hành vi không xác định.
- Không giải phóng bộ nhớ: Với mảng động, việc không giải phóng bộ nhớ sau khi sử dụng sẽ gây ra rò rỉ bộ nhớ.

## 2.4. Con trỏ (Pointers)

### 2.4.1. Khái niệm về Biến và Vùng Nhớ trong C

Biến trong C là một khái niệm quan trọng, đại diện cho một vị trí bộ nhớ trong máy tính để lưu trữ dữ liệu. Mỗi biến có một tên, kiểu dữ liệu, và giá trị được lưu trữ tại một địa chỉ cụ thể trong bộ nhớ.

Khai báo biến và vùng nhớ:

```
int x = 10;  // Khai báo biến x kiểu int và gán giá trị 10.
```

Khi khai báo `int x = 10;`, trình biên dịch sẽ cấp phát một vùng nhớ để lưu giá trị của biến x. Giả sử int chiếm 4 byte, vùng nhớ sẽ được thiết lập để chứa giá trị 10.

Toán tử lấy địa chỉ & và lấy giá trị *:

- &x trả về địa chỉ bộ nhớ nơi giá trị của x được lưu trữ.
- *(&x) hoặc x trả về giá trị được lưu trong vùng nhớ của x.

Ví dụ:

```
int x = 10;
printf("Value of x is %d\n", x);  // Output: 10
printf("Address of x in Hex is %p\n", &x);  // Output: địa chỉ dạng Hex của x.
printf("Value at address of x is %d\n", *(&x));  // Output: 10
```

### 2.4.2. Khái niệm về Con trỏ (Pointers)

Con trỏ là một biến đặc biệt dùng để lưu trữ địa chỉ của biến khác. Con trỏ cho phép truy cập và thao tác với vùng nhớ một cách linh hoạt và mạnh mẽ.

Khai báo con trỏ:

```
int *ptr;  // Khai báo con trỏ ptr trỏ tới kiểu int.
```

Gán địa chỉ cho con trỏ:

```
int x = 10;
int *ptr = &x;  // ptr trỏ tới địa chỉ của biến x.
```

Truy cập giá trị thông qua con trỏ:

```
int value = *ptr;  // Lấy giá trị tại địa chỉ mà ptr trỏ tới.
```

Ví dụ :

```
int x = 10;
int *ptr = &x;
printf("Value of x is %d\n", x);  // Output: 10
printf("Address of x is %lu\n", (unsigned long int)&x);  // Output: địa chỉ của x
printf("Value of pointer ptr is %lu\n", (unsigned long int)ptr);  // Output: địa chỉ
printf("Ptr pointing value is %d\n", *ptr);  // Output: 10
```

### 2.4.3. Các phép toán trên con trỏ

Con trỏ không chỉ đơn thuần trỏ tới một vị trí bộ nhớ, mà còn có thể tham gia vào các phép toán như cộng, trừ với số nguyên, giúp điều chỉnh vị trí mà con trỏ trỏ tới.

**Cộng và trừ con trỏ với số nguyên:** Khi cộng hoặc trừ con trỏ với một số nguyên, địa chỉ mà con trỏ trỏ tới sẽ thay đổi tương ứng với kích thước của kiểu dữ liệu mà con trỏ trỏ tới.

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;  // ptr trỏ tới phần tử đầu tiên của mảng.
ptr++;  // ptr trỏ tới phần tử thứ hai.
```

**Trừ hai con trỏ:** Trừ hai con trỏ cùng kiểu sẽ cho kết quả là số phần tử giữa hai con trỏ.

```
int *p1 = &arr[1];
int *p2 = &arr[3];
int distance = p2 - p1;  // distance = 2
```

Ví dụ :

```
char *pchar;
short *pshort;
long *plong;

pchar += 1;    // pchar trỏ tới byte tiếp theo
pshort += 1;   // pshort trỏ tới địa chỉ cách địa chỉ hiện tại 2 byte
plong += 1;    // plong trỏ tới địa chỉ cách địa chỉ hiện tại 4 byte
```

### 2.4.4. Con trỏ void (void *)

Con trỏ void là con trỏ không định kiểu. Nó có thể trỏ tới bất kỳ kiểu dữ liệu nào, nhưng không thể dereference trực tiếp mà không ép kiểu. Ví dụ về con trỏ void :

```
float x;
int y;
void *p;  // Khai báo con trỏ void
```

```
p = &x;  // p trỏ tới x
*((float*)p) = 2.5;  // Sử dụng ép kiểu để dereference và gán giá trị
p = &y;  // p trỏ tới y
*((int*)p) = 2;  // Sử dụng ép kiểu để dereference và gán giá trị
```

### 2.4.5. Mảng và Con trỏ

Trong C, mảng và con trỏ có mối liên hệ chặt chẽ. Tên của mảng thực chất là một hằng con trỏ trỏ tới phần tử đầu tiên của mảng. Quan hệ giữa mảng và con trỏ:

- M là địa chỉ của phần tử đầu tiên trong mảng M.
- M + i là địa chỉ của phần tử thứ i trong mảng M.
- *(M + i) tương đương với M[i].

Ví dụ:

```
int M[10];
printf("%p\n", M);        // Địa chỉ của phần tử đầu tiên
printf("%p\n", &M[0]);    // Địa chỉ của phần tử đầu tiên (tương tự M)
```

### 2.4.6. Các khái niệm con trỏ nâng cao

#### Con trỏ hàm

Con trỏ hàm lưu trữ địa chỉ của một hàm và có thể được sử dụng để gọi hàm thông qua con trỏ.

```
int add(int a, int b) {
    return a + b;
}
int (*funcPtr)(int, int) = &add;  // Khai báo con trỏ hàm.
int result = funcPtr(3, 4);  // Gọi hàm qua con trỏ hàm, result = 7.
```

#### Con trỏ tới con trỏ

Con trỏ tới con trỏ là một con trỏ lưu trữ địa chỉ của một con trỏ khác, thường được sử dụng trong mảng động và trong quản lý bộ nhớ phức tạp.

```
int a = 10;
int *ptr = &a;
int **ptrPtr = &ptr;  // Con trỏ tới con trỏ.
```

Bản thân con trỏ cũng là 1 biến, vì vậy nó cũng có địa chỉ và có thể dùng 1 con trỏ khác để trỏ tới địa chỉ đó.

<Kiểu DL> **<Tên biến trỏ>;

Ví dụ :

```
int x = 12;
int *p1 = &x;
int **p2 = &p1;
```

Có thể mô tả 1 mảng 2 chiều qua con trỏ của con trỏ theo công thức :

$$M[i][k] = *(*(M + i) + k)$$

Với:

- M+i là địa chỉ nới chứa địa chỉ hàng có chỉ số là i của mảng
- *(M+i) cho nội dung địa chỉ hàng có chỉ số là i
- *(M+i)+k là địa chỉ phần tử [i][k]

**Khi nào buộc phải dùng *, khi nào dùng ** và khi nào ***?**

### 2.4.7. Các lỗi thường gặp với con trỏ

- Dereference con trỏ null: Một lỗi nghiêm trọng, thường gây ra sụp đổ chương trình.
- Sử dụng con trỏ chưa khởi tạo: Dẫn đến hành vi không xác định.
- Rò rỉ bộ nhớ: Khi sử dụng con trỏ để quản lý bộ nhớ động, việc quên giải phóng bộ nhớ sẽ dẫn đến rò rỉ bộ nhớ, đặc biệt là trong các ứng dụng chạy dài hạn.

## 2.5. Bộ nhớ Động - Dynamic Memory

Việc quản lý bộ nhớ động trong C/C++ bao gồm cấp phát và giải phóng bộ nhớ trong thời gian chạy của chương trình.

### 2.5.1. Cấp phát bộ nhớ động trong C

- `malloc`: Cấp phát bộ nhớ nhưng không khởi tạo.

- `calloc`: Cấp phát bộ nhớ và khởi tạo tất cả các phần tử với giá trị 0.
- `realloc`: Thay đổi kích thước của vùng nhớ đã cấp phát.
- `free`: Giải phóng vùng nhớ đã cấp phát.

Ví dụ:

```
int *arr = (int*)malloc(5 * sizeof(int));  // Cấp phát mảng động.
if (arr == NULL) {
    // Xử lý lỗi khi cấp phát bộ nhớ thất bại.
}
free(arr);  // Giải phóng bộ nhớ sau khi sử dụng.
```

### 2.5.2. Cấp phát bộ nhớ động trong C++

- `new`: Cấp phát bộ nhớ cho một đối tượng hoặc mảng.
- `delete`: Giải phóng bộ nhớ đã cấp phát.

Ví dụ:

```
int *arr = new int[5];  // Cấp phát mảng động với 5 phần tử.
delete[] arr;  // Giải phóng bộ nhớ.
```

### 2.5.3. Bộ nhớ động cho mảng 2 chiều

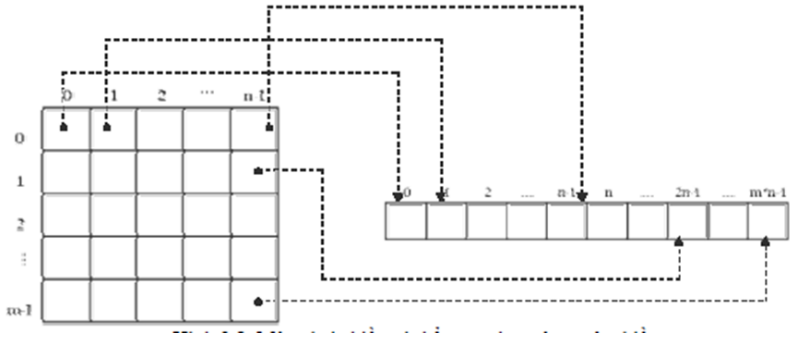
Khi làm việc với mảng động trong C/C++, ta cần cấp phát bộ nhớ cho từng chiều của mảng một cách thủ công. Điều này giúp tiết kiệm bộ nhớ và cho phép kiểm soát tốt hơn, nhưng cũng đòi hỏi sự cẩn thận để tránh rò rỉ bộ nhớ.

**Cấp phát bộ nhớ động cho mảng 2 chiều** Biểu diễn mảng 2 chiều thành mảng 1 chiều:

```
int *array = (int*)malloc(rows * cols * sizeof(int));
int value = array[i * cols + j];  // Truy cập phần tử [i][j].
```

Sử dụng con trỏ của con trỏ (pointer to pointer):

```
int **array = (int**)malloc(rows * sizeof(int*));
for (int i = 0; i < rows; i++) {
    array[i] = (int*)malloc(cols * sizeof(int));
}
```



Hình 2.3 Chuyển mảng 2 chiều thành mảng 1 chiều

Với C++, mảng 2 chiều có thể được coi như một mảng của các mảng 1 chiều. Dưới đây là các bước để cấp phát và giải phóng bộ nhớ cho mảng 2 chiều. Ta có thể coi một mảng 2 chiều là 1 mảng 1 chiều như trong Hình 2.3.

Gọi X là mảng hai chiều có kích thước R dòng và C cột, A là mảng một chiều tương ứng sẽ cs kích thước [R * C], và `X[i][j] = A[ i*n + j]`.

**Cấp phát bộ nhớ cho mảng 2 chiều dùng ** :** Cách thứ nhất:

```
int **mt;
 mt = new int *[R];
int *temp = new  int [R*C];
for (i=0; i< R; ++i) {
   mt[i] = temp;
   temp += C;
}  // Khai bao xong từ đây có thể sử dụng mt[i][j] như bình thường. cuối cùng
delete [] mt[0];   // xoá ? Tại sao?
delete [] mt;
```

Cách thứ hai:

```
// Nhap R,C;
float  ** M = new float *[R];
for(i=0; i<R;i++)
    M[i] = new float[C];
//dung M[i][j] nhu binh thuong

// giai phong
```

```
for(i=0; i<R;i++)
    delete []M[i];   // giai fong cac hang
delete []M;
```

Cấp phát bộ nhớ cho một mảng gồm 'rows' con trỏ đến mảng 1 chiều:

```
int **arr = (int**)malloc(rows * sizeof(int*));
```

Cấp phát bộ nhớ cho từng mảng 1 chiều với kích thước 'cols':

```
for (int i = 0; i < rows; i++) {
    arr[i] = (int*)malloc(cols * sizeof(int));
}
```

Giải phóng bộ nhớ của mảng 2 chiều:

```
// Giải phóng bộ nhớ cho từng mảng 1 chiều
for (int i = 0; i < rows; i++) {
    free(arr[i]);
}
// Giải phóng bộ nhớ cho mảng con trỏ
free(arr);
```

**Cấp phát bộ nhớ động cho mảng 3 chiều** Mảng 3 chiều có thể được coi như một mảng của các mảng 2 chiều. Quá trình cấp phát bộ nhớ diễn ra theo cách tương tự như mảng 2 chiều, nhưng thêm một bước cấp phát cho từng mảng 2 chiều. Cấp phát bộ nhớ cho mảng 3 chiều:

```
int x = 3, y = 4, z = 5;
// Cấp phát bộ nhớ cho mảng gồm 'x' con trỏ đến mảng 2 chiều
int ***arr = (int***)malloc(x * sizeof(int**));

// Cấp phát bộ nhớ cho từng mảng 2 chiều
for (int i = 0; i < x; i++) {
    arr[i] = (int**)malloc(y * sizeof(int*));

    // Cấp phát bộ nhớ cho từng mảng 1 chiều trong mỗi mảng 2 chiều
    for (int j = 0; j < y; j++) {
        arr[i][j] = (int*)malloc(z * sizeof(int));
    }
}
```

Giải phóng bộ nhớ của mảng 3 chiều:

```
// Giải phóng bộ nhớ cho từng mảng 1 chiều
for (int i = 0; i < x; i++) {
    for (int j = 0; j < y; j++) {
        free(arr[i][j]);
    }

    // Giải phóng bộ nhớ cho từng mảng 2 chiều
    free(arr[i]);
}

// Giải phóng bộ nhớ cho mảng con trỏ đến mảng 2 chiều
free(arr);
```

### 2.5.4. Tái cấp phát bộ nhớ cho mảng 2 chiều và 3 chiều

Tái cấp phát bộ nhớ cho mảng 2 chiều và 3 chiều có thể phức tạp hơn vì cần đảm bảo không làm mất dữ liệu trong quá trình tái cấp phát. Trong C, việc tái cấp phát chủ yếu được thực hiện bằng hàm realloc, nhưng realloc chỉ hoạt động với các mảng 1 chiều. Do đó, ta phải thực hiện tái cấp phát thủ công.

**Tái cấp phát bộ nhớ cho mảng 2 chiều:**

```
// Tăng số lượng hàng
int new_rows = 5;
arr = (int**)realloc(arr, new_rows * sizeof(int*));

// Cấp phát bộ nhớ cho các hàng mới
for (int i = rows; i < new_rows; i++) {
    arr[i] = (int*)malloc(cols * sizeof(int));
}
```

**Tái cấp phát bộ nhớ cho mảng 3 chiều:**

```
// Tăng kích thước cho chiều đầu tiên
int new_x = 5;
arr = (int***)realloc(arr, new_x * sizeof(int**));
```

```
// Cấp phát bộ nhớ cho các mảng 2 chiều mới
for (int i = x; i < new_x; i++) {
    arr[i] = (int**)malloc(y * sizeof(int*));

    // Cấp phát bộ nhớ cho từng mảng 1 chiều trong mỗi mảng 2 chiều mới
    for (int j = 0; j < y; j++) {
        arr[i][j] = (int*)malloc(z * sizeof(int));
    }
}
```

Việc cấp phát bộ nhớ động cho mảng nhiều chiều đòi hỏi sự cẩn thận và chính xác để tránh lỗi rò rỉ bộ nhớ và sử dụng con trỏ treo. Mặc dù tái cấp phát bộ nhớ có thể phức tạp hơn, nó cung cấp sự linh hoạt cho các ứng dụng cần quản lý bộ nhớ động một cách hiệu quả. Trong lập trình thực tế, việc sử dụng các cấu trúc dữ liệu cao cấp hoặc thư viện như std::vector trong C++ có thể giúp đơn giản hóa quá trình này.

### 2.5.5. Quản lý Bộ nhớ Stack và Heap

Stack: Quản lý tự động, sử dụng cho các biến cục bộ và các lời gọi hàm.

Heap: Quản lý thủ công, sử dụng cho cấp phát bộ nhớ động.

Mục này cung cấp cái nhìn tổng quan về quản lý bộ nhớ trong C/C++, bao gồm biến, con trỏ, các phép toán con trỏ, và cách quản lý bộ nhớ động. Việc nắm vững các khái niệm này giúp lập trình viên quản lý tài nguyên bộ nhớ một cách hiệu quả và tránh các lỗi phổ biến như rò rỉ bộ nhớ và truy cập bộ nhớ không hợp lệ.

### 2.5.6. Kỹ thuật quản lý bộ nhớ nâng cao

- Sử dụng realloc: realloc cho phép thay đổi kích thước vùng bộ nhớ đã cấp phát mà không mất dữ liệu hiện có. Tuy nhiên, bạn cần kiểm tra con trỏ trả về để đảm bảo việc cấp phát lại thành công.

  ```
  int *arr = (int*)malloc(5 * sizeof(int));
  arr = (int*)realloc(arr, 10 * sizeof(int));  // Mở rộng mảng lên 10 phần tử.
  ```

- Buffer Overflows: Là một trong những vấn đề bảo mật quan trọng nhất khi làm việc với bộ nhớ động. Cần đặc biệt cẩn thận khi làm việc với các chuỗi ký tự và các mảng.

- Smart Pointers (C++):

  - Unique Pointer (std::unique_ptr): Đảm bảo rằng chỉ có một con trỏ duy nhất quản lý một vùng bộ nhớ, giúp ngăn ngừa lỗi sử dụng con trỏ treo (dangling pointer).
  - Shared Pointer (std::shared_ptr): Cho phép nhiều con trỏ cùng quản lý một vùng bộ nhớ, sử dụng đếm số tham chiếu để tự động giải phóng bộ nhớ khi không còn tham chiếu nào.

Lưu ý quan trọng về bộ nhớ động:

- Luôn giải phóng bộ nhớ sau khi sử dụng: Quên giải phóng bộ nhớ dẫn đến rò rỉ bộ nhớ, gây ra sự cố với các ứng dụng dài hạn.
- Tránh delete cùng malloc hoặc free cùng new: Trong C++, không nên kết hợp malloc/free với new/delete vì chúng quản lý bộ nhớ theo các cách khác nhau.
- Sử dụng Smart Pointers khi có thể (trong C++): Giúp quản lý bộ nhớ tự động, giảm nguy cơ rò rỉ bộ nhớ và con trỏ treo.

## 2.6. Kết luận

Chương này đã mở rộng và đi sâu hơn vào các khái niệm nâng cao về C/C++, bao gồm thứ tự ưu tiên các phép toán, hành vi không xác định, mảng, con trỏ và quản lý bộ nhớ động. Việc nắm vững các kiến thức này là vô cùng cần thiết để viết mã nguồn hiệu quả, an toàn và tối ưu. Hy vọng những kiến thức này sẽ giúp bạn trở thành một lập trình viên giỏi hơn, có khả năng đối phó với các tình huống phức tạp và phát triển các phần mềm chất lượng cao.

# Chương 3
# Hàm trong C/C++



## 3.1. Giới thiệu

Chương này tập trung vào các khái niệm cơ bản về hàm trong C/C++, bao gồm:

- Khái niệm về hàm: Hàm là một nhóm các khai báo và câu lệnh được gán một tên gọi, giúp chương trình chia nhỏ thành các tác vụ con, dễ bảo trì và quản lý.
- Truyền tham trị và tham chiếu: Cách truyền tham số vào hàm. Trong C, truyền tham trị (pass by value) nghĩa là truyền một bản sao của giá trị. Trong C++, ngoài truyền tham trị còn có thể truyền tham chiếu (pass by reference), cho phép hàm thao tác trực tiếp trên biến gốc.
- Đa năng hóa hàm (function overloading): C++ cho phép định nghĩa nhiều hàm cùng tên với các tham số khác nhau. Trình biên dịch sẽ chọn phiên bản hàm phù hợp dựa trên tham số trong lời gọi hàm.
- Con trỏ hàm (function pointers): Con trỏ có thể trỏ đến địa chỉ của hàm, cho phép gọi hàm thông qua con trỏ.
- Khái quát hóa hàm (function templates): C++ hỗ trợ templates để định nghĩa các hàm tổng quát mà không phụ thuộc vào kiểu dữ liệu cụ thể. Điều này giúp tái sử dụng mã và giảm sự lặp lại.

- Biểu thức lambda và hàm nặc danh: Lambda là hàm nặc danh (không có tên) trong C++ có thể được sử dụng để thực hiện các tác vụ nhỏ một cách ngắn gọn.

## 3.2. Khái niệm về Hàm trong C/C++

Hàm là một khối mã được gán tên và có thể được gọi để thực thi nhiệm vụ cụ thể. Việc sử dụng hàm giúp tổ chức mã nguồn thành các phần logic nhỏ hơn, dễ quản lý và bảo trì hơn. Hàm cũng giúp tránh lặp lại mã nguồn (theo nguyên tắc DRY - Don't Repeat Yourself) và làm cho mã dễ đọc hơn.

Khi nào nên dùng hàm:

- Khi một đoạn mã cần được sử dụng lại nhiều lần.
- Khi một đoạn mã thực hiện một nhiệm vụ logic cụ thể, nên tách ra thành một hàm để cải thiện tính rõ ràng của mã nguồn.
- Khi cần chia nhỏ chương trình thành các đơn vị nhỏ hơn để dễ dàng quản lý và bảo trì.

Khi nào nên tránh tạo quá nhiều hàm:

- Khi hàm quá nhỏ hoặc đơn giản, việc chia nhỏ có thể làm mã nguồn trở nên phức tạp không cần thiết.
- Khi hiệu suất là yếu tố quan trọng (như trong lập trình hệ thống nhúng), việc gọi hàm có thể gây ra chi phí về thời gian do quá trình chuyển điều khiển.

## 3.3. Truyền tham trị và tham chiếu

Trong C++, có hai cách truyền tham số vào hàm: truyền tham trị (pass by value) và truyền tham chiếu (pass by reference).

**Truyền tham trị (Pass by Value):** Khi truyền tham trị, một bản sao của giá trị được truyền vào hàm. Các thay đổi trên bản sao này không ảnh hưởng đến biến gốc.

- Ưu điểm: An toàn, tránh thay đổi dữ liệu ngoài ý muốn của biến gốc.
- Nhược điểm: Tốn bộ nhớ và thời gian (với kiểu dữ liệu lớn), vì phải sao chép toàn bộ giá trị.

- Khi nên dùng: Khi không cần thay đổi giá trị của tham số bên ngoài hàm, hoặc khi làm việc với kiểu dữ liệu nhỏ (int, char, float, double).
- Khi nên tránh: Khi làm việc với kiểu dữ liệu lớn (như struct lớn hoặc class) hoặc khi cần hiệu suất cao.

**Truyền tham chiếu (Pass by Reference):** Truyền tham chiếu cho phép hàm thao tác trực tiếp trên biến gốc thông qua tham chiếu.

- Ưu điểm: Không tốn bộ nhớ để sao chép dữ liệu, hiệu suất cao hơn cho kiểu dữ liệu lớn.
- Nhược điểm: Có thể dẫn đến thay đổi dữ liệu ngoài ý muốn, khó kiểm soát lỗi hơn.
- Khi nên dùng: Khi cần thay đổi giá trị của biến gốc, hoặc khi làm việc với kiểu dữ liệu lớn mà không muốn sao chép.
- Khi nên tránh: Khi không muốn hàm thay đổi giá trị của biến gốc, hoặc khi cần bảo vệ tính toàn vẹn của dữ liệu.

Ví dụ so sánh:

```
// Truyền tham trị:

void increment(int x) {
    x = x + 1;
}

int main() {
    int a = 5;
    increment(a);
    cout << a << endl;  // Output: 5 (giá trị của a không thay đổi)
    return 0;
}

// Truyền tham chiếu:

void increment(int &x) {
    x = x + 1;
```

```
}

int main() {
    int a = 5;
    increment(a);
    cout << a << endl;  // Output: 6 (giá trị của a đã thay đổi)
    return 0;
}
```

### 3.4. Đa năng hóa hàm (Function Overloading)

Đa năng hóa hàm cho phép định nghĩa nhiều hàm cùng tên nhưng khác nhau về danh sách tham số (số lượng hoặc kiểu dữ liệu tham số). Trình biên dịch sẽ chọn phiên bản hàm phù hợp dựa trên các tham số trong lời gọi hàm.

Ưu điểm:

- Tăng tính linh hoạt cho mã nguồn.
- Giúp mã nguồn dễ đọc và dễ hiểu hơn, vì có thể sử dụng cùng tên hàm cho các chức năng tương tự nhưng với các kiểu dữ liệu khác nhau.

Nhược điểm:

- Có thể gây nhầm lẫn nếu quá lạm dụng, hoặc nếu các hàm khác nhau nhưng có tên và chức năng tương tự.
- Có thể làm tăng kích thước mã nguồn và gây khó khăn cho việc bảo trì.

Khi nên dùng:

- Khi muốn thực hiện cùng một hành động trên các kiểu dữ liệu khác nhau.
- Khi các hàm có chức năng tương tự và bạn muốn giữ tên hàm nhất quán.

Khi nên tránh:

- Khi các hàm thực hiện các hành động khác nhau (kể cả khi chúng có tham số giống nhau).
- Khi quá nhiều phiên bản hàm có thể gây nhầm lẫn cho người đọc mã.

Ví dụ về đa năng hóa hàm:

```cpp
#include <iostream>
using namespace std;

// Hàm tính tổng cho kiểu int
int add(int a, int b) {
    return a + b;
}
.
// Hàm tính tổng cho kiểu double
double add(double a, double b) {
    return a + b;
}

int main() {
    cout << add(3, 4) << endl;        // Output: 7
    cout << add(3.5, 4.5) << endl;    // Output: 8.0
    return 0;
}
```

## 3.5. Con trỏ hàm (Function Pointers)

### 3.5.1. Khái niệm

Con trỏ hàm là một loại con trỏ đặc biệt trong C++ được sử dụng để lưu trữ địa chỉ của một hàm. Con trỏ hàm cho phép chúng ta gọi một hàm thông qua địa chỉ của nó. Điều này đặc biệt hữu ích khi chúng ta muốn truyền hàm như một tham số cho một hàm khác hoặc khi chúng ta cần lựa chọn hàm để gọi tại runtime.

Con trỏ hàm là một biến lưu trữ địa chỉ của một hàm và có thể được sử dụng để gọi hàm đó. Con trỏ hàm cung cấp tính linh hoạt cao, cho phép lập trình viên thay đổi hàm được gọi tại runtime hoặc thực hiện các kỹ thuật lập trình bậc cao như callback và xử lý sự kiện.

Cú pháp của con trỏ hàm có dạng:

```
returnType (*pointerName)(parameterType1, parameterType2, ...);
```

Trong đó:

- returnType là kiểu trả về của hàm mà con trỏ sẽ trỏ tới.

- parameterType1, parameterType2, ... là danh sách kiểu dữ liệu của các tham số mà hàm nhận.
- pointerName là tên của con trỏ hàm.

### 3.5.2. Khai Báo và Sử Dụng Con Trỏ Hàm

Để khai báo một con trỏ hàm, chúng ta cần chỉ định kiểu trả về và danh sách tham số của hàm mà con trỏ sẽ trỏ tới. Ví dụ:

```
#include <iostream>
using namespace std;

// Hàm mẫu
int add(int a, int b) {
    return a + b;
}

int main() {
    // Khai báo con trỏ hàm 'funcPtr' trỏ tới một hàm trả về int và nhận hai
    int (*funcPtr)(int, int);

    // Gán địa chỉ của hàm 'add' cho con trỏ hàm 'funcPtr'
    funcPtr = &add;

    // Gọi hàm thông qua con trỏ hàm
    int result = funcPtr(5, 10);
    cout << "Result of add function: " << result << endl;

    return 0;
}
```

Gán Hàm Cho Con Trỏ Hàm, có thể gán địa chỉ của một hàm cho con trỏ hàm như sau:

```
funcPtr = &add;
```

Hoặc đơn giản hơn:

```
funcPtr = add; // & có thể được bỏ qua
```

Gọi Hàm Thông Qua Con Trỏ Hàm, sau khi con trỏ hàm đã được gán, có thể gọi hàm thông qua con trỏ hàm như sau:

```
int result = funcPtr(5, 10);
```

### 3.5.3. Ví Dụ Về Con Trỏ Hàm

**Sử dụng con trỏ hàm trong chương trình đơn giản.**

```
#include <iostream>
using namespace std;

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int main() {
    // Khai báo con trỏ hàm
    int (*operation)(int, int);

    // Gán hàm 'add' cho con trỏ hàm
    operation = add;
    cout << "Addition: " << operation(5, 3) << endl;  // Kết quả: 8

    // Gán hàm 'subtract' cho con trỏ hàm
    operation = subtract;
    cout << "Subtraction: " << operation(5, 3) << endl;  // Kết quả: 2

    return 0;
}
```

**Truyền con trỏ hàm làm tham số cho hàm khác.** Con trỏ hàm có thể được sử dụng làm tham số cho các hàm khác để tạo tính linh hoạt.

```
#include <iostream>
using namespace std;
```

```
// Định nghĩa hàm mẫu
int multiply(int a, int b) {
    return a * b;
}

// Hàm nhận con trỏ hàm làm tham số
void printResult(int (*operation)(int, int), int x, int y) {
    cout << "Result: " << operation(x, y) << endl;
}

int main() {
    // Truyền con trỏ hàm 'multiply' cho hàm 'printResult'
    printResult(multiply, 4, 5);  // Kết quả: 20

    return 0;
}
```

### 3.5.4. Ứng Dụng của Con Trỏ Hàm

Con trỏ hàm có nhiều ứng dụng quan trọng, bao gồm:

- Truyền hàm như tham số: Con trỏ hàm có thể được truyền vào các hàm khác để thay đổi hành vi của chúng mà không cần viết lại mã.
- Triển khai callback: Trong lập trình sự kiện hoặc lập trình GUI, con trỏ hàm thường được sử dụng để thực hiện các hàm callback.
- Tối ưu hóa hiệu năng: Cho phép chuyển đổi linh hoạt giữa các hàm mà không cần sử dụng các câu lệnh điều kiện như if hoặc switch, giúp tối ưu hóa hiệu năng.

### 3.5.5. Ưu và Nhược Điểm của Con Trỏ Hàm

Ưu điểm của con trỏ hàm là:

- Linh hoạt: Cho phép các hàm nhận hành vi động tại runtime.
- Giảm thiểu mã lặp: Cho phép tái sử dụng mã bằng cách sử dụng các hàm chung.

- Hiệu quả: Gọi hàm thông qua con trỏ hàm nhanh hơn so với việc sử dụng các cấu trúc điều khiển như if hoặc switch.

Nhược Điểm:

- Khó đọc và bảo trì mã: Mã sử dụng con trỏ hàm có thể trở nên khó đọc và bảo trì, đặc biệt đối với những người không quen thuộc với khái niệm này.
- Rủi ro bảo mật: Nếu không sử dụng cẩn thận, con trỏ hàm có thể dẫn đến lỗi runtime hoặc lỗi bảo mật như code injection.

### 3.5.6. Kết luận về con trỏ hàm

Con trỏ hàm là một công cụ mạnh mẽ trong C++ cho phép lập trình viên tạo ra mã linh hoạt và tái sử dụng cao. Chúng đặc biệt hữu ích trong các ứng dụng yêu cầu tính linh hoạt cao hoặc khi làm việc với các hàm callback và thiết kế theo mẫu. Tuy nhiên, việc sử dụng con trỏ hàm cũng đòi hỏi sự cẩn thận để tránh các lỗi phổ biến và đảm bảo mã dễ bảo trì.

- Ưu điểm:
  - Cho phép lựa chọn hàm tại runtime.
  - Rất hữu ích trong các mô hình xử lý sự kiện hoặc callback, nơi mà hàm cụ thể không thể xác định tại thời gian biên dịch.
  - Giảm sự phụ thuộc của mã nguồn vào các hàm cụ thể, tăng tính mở rộng.
- Nhược điểm:
  - Dễ gây lỗi nếu không kiểm tra kỹ càng con trỏ trước khi sử dụng (ví dụ: con trỏ null).
  - Làm tăng độ phức tạp của mã, khó đọc và bảo trì hơn, đặc biệt là đối với người mới học.
- Khi nào nên dùng:
  - Khi cần thực hiện callback (ví dụ: trong các thư viện hoặc các ứng dụng GUI).
  - Khi cần linh hoạt chọn hàm để gọi tại runtime, đặc biệt là trong lập trình hướng đối tượng.
- Khi nào nên tránh:

- Khi không cần sự linh hoạt của con trỏ hàm.
- Khi code readability và maintainability là quan trọng và bạn muốn giữ mã nguồn đơn giản.

Con trỏ hàm không chỉ hữu ích trong các mô hình xử lý sự kiện mà còn trong các tình huống lập trình cần sử dụng các hàm gọi lại (callbacks) hoặc triển khai các thuật toán có thể thay đổi linh hoạt tại runtime.

Ví dụ nâng cao: giả sử chúng ta có một mảng các số nguyên và muốn sắp xếp mảng này theo các tiêu chí khác nhau (tăng dần hoặc giảm dần). Chúng ta có thể sử dụng con trỏ hàm để truyền hàm so sánh mong muốn vào hàm sắp xếp.

```
#include <iostream>
using namespace std;

// Hàm so sánh tăng dần
bool ascending(int a, int b) {
    return a < b;
}

// Hàm so sánh giảm dần
bool descending(int a, int b) {
    return a > b;
}

// Hàm sắp xếp sử dụng con trỏ hàm để so sánh
void sort(int arr[], int n, bool (*compare)(int, int)) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (compare(arr[j + 1], arr[j])) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
.
int main() {
    int arr[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    cout << "Original array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << endl;

    // Sắp xếp mảng theo thứ tự tăng dần
    sort(arr, n, ascending);
    cout << "Sorted ascending: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << endl;

    // Sắp xếp mảng theo thứ tự giảm dần
    sort(arr, n, descending);
    cout << "Sorted descending: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

## 3.6. Khái quát hóa hàm (Function Templates)

### 3.6.1. Khái Niệm Hàm Template

Hàm template trong C++ là một khái niệm mạnh mẽ cho phép viết các hàm tổng quát, có thể hoạt động với nhiều kiểu dữ liệu khác nhau mà không cần viết lại mã cho từng kiểu dữ liệu cụ thể. Nó cho phép tạo ra các hàm mà kiểu dữ liệu của tham số và giá trị trả về được xác định khi hàm được gọi, dựa trên kiểu dữ liệu của các đối số thực tế.

Cú pháp cơ bản của một hàm template như sau:

```
template <typename T>
T functionName(T parameter) {
    // Function body
}
```

Trong đó:

- `template <typename T>`: Định nghĩa một template với một tham số kiểu T

- `T functionName(T parameter)`: Định nghĩa một hàm sử dụng kiểu T

### 3.6.2. Ví dụ về Hàm Template

Hãy xem xét một ví dụ đơn giản về hàm template để tìm giá trị lớn hơn trong hai giá trị:

```
#include <iostream>
using namespace std;

// Định nghĩa một hàm template để tìm giá trị lớn hơn
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max of 3 and 7 is: " << findMax(3, 7) << endl;   // Sử dụng templat
    cout << "Max of 3.5 and 7.2 is: " << findMax(3.5, 7.2) << endl; // Sử dụng t
    cout << "Max of 'A' and 'Z' is: " << findMax('A', 'Z') << endl; // Sử dụng te
    return 0;
}
```

### 3.6.3. Lợi Ích Của Việc Sử Dụng Hàm Template

- Tái Sử Dụng Mã Nguồn: Hàm template cho phép viết một lần, sử dụng cho nhiều kiểu dữ liệu khác nhau, từ đó giảm thiểu mã nguồn thừa và làm mã ngắn gọn hơn.
- Tăng Tính Linh Hoạt: Hàm template cho phép viết mã linh hoạt hơn, dễ dàng mở rộng và bảo trì hơn so với việc viết hàm riêng cho từng kiểu dữ liệu.
- Hiệu Năng Tốt: Khi sử dụng template, trình biên dịch sẽ tạo ra mã cụ thể cho mỗi kiểu dữ liệu tại thời điểm biên dịch (compile-time), từ đó tối ưu hóa hiệu năng.

### 3.6.4. Đặc Điểm và Hạn Chế của Hàm Template

- Ưu Điểm:
  - Tổng quát hóa: Một hàm template có thể hoạt động với nhiều kiểu dữ liệu khác nhau mà không cần viết lại mã.

  - Tối ưu hóa tại thời điểm biên dịch: Trình biên dịch sẽ tạo ra mã cụ thể cho từng kiểu dữ liệu mà hàm template được sử dụng, giúp tăng hiệu năng.

- Nhược Điểm:
  - Phức tạp trong gỡ lỗi: Do mã nguồn của template được tạo ra tại thời điểm biên dịch, việc gỡ lỗi có thể trở nên khó khăn hơn khi các lỗi xảy ra.
  - Mã bloat: Nếu sử dụng template với quá nhiều kiểu dữ liệu khác nhau, mã nguồn được tạo ra có thể trở nên quá lớn, dẫn đến mã bloat (béo phì mã nguồn).

### 3.6.5. Cách Sử Dụng Hàm Template Hiệu Quả

Sử dụng khi cần tổng quát hóa: Khi một hàm cần hoạt động với nhiều kiểu dữ liệu khác nhau, hàm template là một lựa chọn phù hợp. Hạn chế việc quá lạm dụng: Không nên lạm dụng hàm template cho các tình huống không cần thiết, đặc biệt khi một kiểu dữ liệu cụ thể đã đủ cho nhu cầu.

### 3.6.6. Các Lỗi Phổ Biến Khi Sử Dụng Hàm Template

Không xác định đúng kiểu dữ liệu: Khi gọi hàm template, cần đảm bảo rằng kiểu dữ liệu truyền vào phù hợp với định nghĩa của template. Sử dụng hàm template cho các kiểu dữ liệu không hỗ trợ toán tử: Một số kiểu dữ liệu có thể không hỗ trợ các toán tử như +, -, *, / cần thiết trong hàm template.

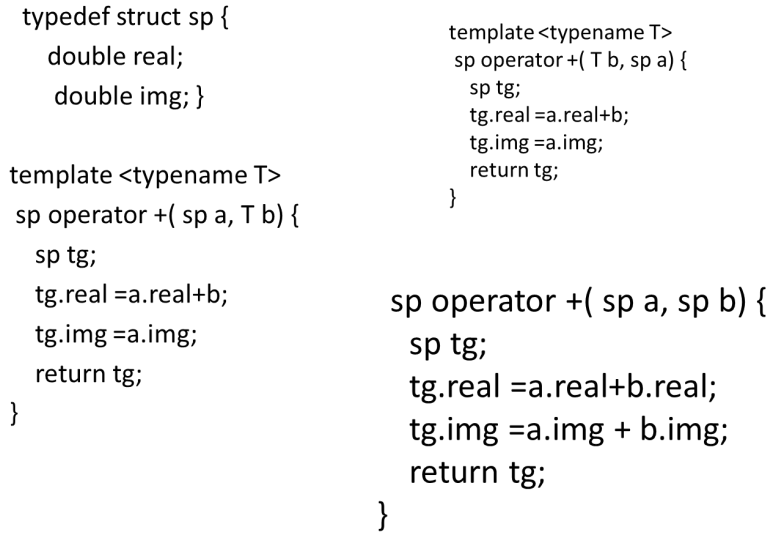
### 3.6.7. Ví Dụ Nâng Cao Về Hàm Template

Ví dụ về hàm template với nhiều tham số kiểu:

```
#include <iostream>
using namespace std;

// Hàm template với hai tham số kiểu
template <typename T1, typename T2>
void printPair(T1 first, T2 second) {
    cout << "First: " << first << ", Second: " << second << endl;
}

int main() {
```




Hình 3.1 Đa năng hóa toán tử + với số phức kết hợp template

```
    printPair(10, 20.5);  // int, double
    printPair("Hello", 'A');  // const char*, char
    return 0;
}
```

### 3.6.8. Tổng Kết

Hàm template là một công cụ mạnh mẽ trong C++ giúp tổng quát hóa mã nguồn và tăng tính linh hoạt. Tuy nhiên, cần sử dụng chúng một cách cẩn trọng để tránh các lỗi phổ biến và đảm bảo mã nguồn dễ bảo trì. Khi sử dụng hàm template, hãy cân nhắc các tình huống cụ thể để quyết định khi nào nên và không nên sử dụng chúng.

## 3.7. Biểu thức Lambda và Hàm Nặc Danh (Lambda Expressions and Anonymous Functions)

### 3.7.1. Giới thiệu về Hàm Lambda trong C++

Hàm Lambda, hay còn gọi là biểu thức lambda (lambda expressions), được giới thiệu lần đầu tiên trong tiêu chuẩn C++11. Lambda là những hàm vô danh (anonymous functions) cho phép định nghĩa các hàm ngắn gọn tại nơi chúng được sử dụng mà không cần đặt tên. Lambda được sử dụng rộng rãi trong lập trình hiện đại, đặc biệt là khi cần viết các hàm nhỏ, đơn giản để truyền vào các hàm khác như đối số,

hoặc khi cần xử lý các thao tác logic ngay tại chỗ mà không cần định nghĩa hàm riêng biệt.

Lambda rất hữu ích trong các tình huống mà bạn cần viết các hàm nhỏ, sử dụng một lần (one-off functions) hoặc khi bạn muốn mã nguồn của mình gọn gàng hơn. Lambda giúp tăng cường tính linh hoạt của mã nguồn và cho phép lập trình viên triển khai các mô hình lập trình hàm (functional programming) trong C++.

### 3.7.2. Cú pháp của Hàm Lambda trong C++

Cú pháp cơ bản của hàm lambda trong C++ là:

```
[capture](parameters) -> return_type { body }
```

Capture: Xác định cách lambda "bắt giữ" các biến từ phạm vi bên ngoài. Các biến có thể được bắt giữ bằng giá trị (value) hoặc tham chiếu (reference).

Parameters: Danh sách các tham số của lambda, giống như tham số của hàm thông thường.

Return Type: Kiểu trả về của lambda (có thể bỏ qua nếu trình biên dịch có thể suy diễn được từ body).

Body: Khối mã của lambda, chứa các biểu thức hoặc câu lệnh để thực hiện một hành động cụ thể.

Ví dụ cơ bản về hàm lambda:

```cpp
#include <iostream>
using namespace std;

int main() {
    // Lambda đơn giản không có tham số và không bắt giữ biến nào
    auto greet = []() {
        cout << "Hello, World!" << endl;
    };

    greet();  // Output: Hello, World!

    return 0;
}
```

Trong ví dụ trên, greet là một lambda không có tham số và không bắt giữ biến nào. Lambda này chỉ đơn giản in ra "Hello, World!" khi được gọi.

### 3.7.3. Các cách bắt giữ (capture) trong Lambda

Lambda có thể bắt giữ các biến từ phạm vi bao quanh (enclosing scope) của nó. Có ba cách bắt giữ chính:

**Bắt giữ bằng giá trị (Capture by Value):** Biến từ phạm vi bên ngoài được sao chép vào lambda. Bên trong lambda, những biến này là bản sao và không ảnh hưởng đến giá trị gốc bên ngoài lambda. Ví dụ bắt giữ bằng giá trị:

```
int a = 10;
auto captureByValue = [a]() {
    cout << "Inside lambda, a = " << a << endl;
};
a = 20;
captureByValue();  // Output: Inside lambda, a = 10
```

Trong ví dụ này, biến a được bắt giữ bằng giá trị, do đó khi a thay đổi sau khi lambda được tạo, giá trị bên trong lambda không thay đổi.

**Bắt giữ bằng tham chiếu (Capture by Reference):** Lambda có thể thao tác trực tiếp với biến gốc bên ngoài. Nếu giá trị của biến thay đổi bên trong lambda, giá trị gốc bên ngoài cũng sẽ thay đổi. Ví dụ bắt giữ bằng tham chiếu:

```
int a = 10;
auto captureByReference = [&a]() {
    a += 5;
    cout << "Inside lambda, a = " << a << endl;
};
captureByReference();  // Output: Inside lambda, a = 15
cout << "Outside lambda, a = " << a << endl;  // Output: Outside lambda, a = 15
```

Trong ví dụ này, biến a được bắt giữ bằng tham chiếu, vì vậy khi a được thay đổi bên trong lambda, giá trị bên ngoài của a cũng thay đổi.

**Bắt giữ hỗn hợp (Mixed Capture):** Lambda có thể bắt giữ một số biến bằng giá trị và một số biến khác bằng tham chiếu. Điều này giúp linh hoạt trong việc sử dụng các biến bên ngoài trong lambda. Ví dụ bắt giữ hỗn hợp:

```
int a = 10, b = 20;
auto mixedCapture = [a, &b]() {
    cout << "Inside lambda, a = " << a << ", b = " << b << endl;
    b += 10;
};
mixedCapture();  // Output: Inside lambda, a = 10, b = 20
cout << "Outside lambda, b = " << b << endl;  // Output: Outside lambda, b = 30
```

Trong ví dụ này, a được bắt giữ bằng giá trị và b được bắt giữ bằng tham chiếu. Bên trong lambda, b thay đổi thì giá trị bên ngoài của b cũng thay đổi, còn a không thay đổi.

### 3.7.4. Tính chất của Lambda

**Immutability:** Khi các biến được bắt giữ bằng giá trị, chúng sẽ là không thể thay đổi (immutable) bên trong lambda. Nếu muốn thay đổi chúng, cần khai báo lambda là mutable. Ví dụ về mutable lambda:

```
int a = 10;
auto mutableLambda = [a]() mutable {
    a += 5;
    cout << "Inside lambda, modified a = " << a << endl;
};
mutableLambda();  // Output: Inside lambda, modified a = 15
cout << "Outside lambda, a = " << a << endl;  // Output: Outside lambda, a = 10
```

Trong ví dụ này, a được sao chép vào lambda. Bằng cách khai báo lambda là mutable, a có thể được thay đổi bên trong lambda, nhưng giá trị của a bên ngoài không thay đổi.

**Generic Lambda:** Từ C++14, có thể sử dụng auto trong danh sách tham số của lambda để tạo lambda tổng quát (generic lambda), tương tự như hàm template. Ví dụ về generic lambda:

```
auto add = [](auto x, auto y) {
    return x + y;
};

cout << add(3, 4) << endl;      // Output: 7
cout << add(3.5, 4.5) << endl; // Output: 8.0
```

Lambda này có thể cộng hai số nguyên hoặc số thực, tùy thuộc vào kiểu dữ liệu của tham số đầu vào.

### 3.7.5. Ưu điểm và Nhược điểm của Lambda Expressions

Ưu điểm:

- Ngắn gọn và dễ đọc: Lambda giúp viết mã ngắn gọn hơn, đặc biệt là khi cần các hàm nhỏ, đơn giản. Thay vì phải định nghĩa một hàm riêng biệt, bạn có thể định nghĩa logic ngay tại nơi sử dụng.
- Tính linh hoạt cao: Lambda có thể được sử dụng làm đối số cho các hàm khác hoặc lưu trữ trong các cấu trúc dữ liệu, cung cấp tính linh hoạt cao trong lập trình hàm.
- Hiệu suất: Lambda được triển khai tại nơi sử dụng, không cần gọi hàm thông qua tên hàm, điều này có thể cải thiện hiệu suất so với các hàm gọi thông thường.
- Tích hợp tốt với STL và các thư viện khác: Lambda rất hữu ích khi sử dụng với các thuật toán STL như `std::for_each, std::find_if, std::sort`, v.v.

Nhược điểm:

- Có thể khó hiểu nếu lạm dụng: Mặc dù lambda giúp mã ngắn gọn hơn, nhưng nếu sử dụng quá nhiều hoặc quá phức tạp, mã nguồn có thể trở nên khó hiểu.
- Không thể tái sử dụng: Lambda thường được sử dụng tại chỗ, nên chúng không thể tái sử dụng ở các phần khác của chương trình nếu cần logic tương tự.
- Giới hạn về tính năng: Lambda không thể thay thế hoàn toàn các hàm đầy đủ vì chúng không thể có tên và không thể tái sử dụng dễ dàng trong toàn bộ chương trình.

### 3.7.6. Khi nào nên dùng và khi nào nên tránh dùng Lambda

Nên dùng Lambda khi:

- Khi cần một hàm nhỏ, đơn giản: Sử dụng lambda khi bạn cần một hàm chỉ sử dụng một lần hoặc chỉ thực hiện một thao tác đơn giản.
- Khi cần truyền hàm làm đối số: Lambda rất hữu ích khi truyền hàm làm đối số cho các hàm khác, đặc biệt là các thuật toán STL.

- Khi cần logic tại chỗ: Khi bạn muốn viết logic ngay tại nơi cần sử dụng, mà không muốn tạo một hàm riêng biệt.
- Khi làm việc với các hàm gọi lại (callbacks): Lambda rất tiện lợi khi sử dụng cho các hàm gọi lại trong lập trình GUI hoặc lập trình sự kiện.

Khi nên tránh dùng Lambda:

- Khi hàm phức tạp hoặc quá dài: Nếu lambda quá phức tạp hoặc dài, nên chuyển thành hàm riêng biệt để mã nguồn dễ đọc và bảo trì hơn.
- Khi cần tái sử dụng logic: Nếu bạn cần sử dụng lại cùng một logic ở nhiều nơi, nên sử dụng hàm riêng thay vì lambda.
- Khi ảnh hưởng đến hiệu suất: Mặc dù lambda có thể tối ưu hóa hiệu suất, nhưng việc lạm dụng lambda trong các tình huống yêu cầu tối ưu hóa cực cao có thể không phù hợp.

### 3.7.7. Ví dụ thực tế về Lambda Expressions

Lambda có thể được sử dụng để tạo ra các logic ngắn gọn và linh hoạt trong các ứng dụng thực tế. Dưới đây là một số ví dụ thực tế:

**Sử dụng Lambda để sắp xếp một vector:**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> numbers = {5, 2, 9, 1, 5, 6};

    // Sắp xếp vector theo thứ tự giảm dần
    sort(numbers.begin(), numbers.end(), [](int a, int b) {
        return a > b;
    });

    for (int num : numbers) {
        cout << num << " ";  // Output: 9 6 5 5 2 1
```

```
    }
    cout << endl;

    return 0;
}
```

**Sử dụng Lambda với std::for$_{e}ach$ :**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};

    // Sử dụng lambda để in các số trong vector
    for_each(numbers.begin(), numbers.end(), [](int n) {
        cout << n << " ";  // Output: 1 2 3 4 5
    });
    cout << endl;

    return 0;
}
```

**Sử dụng Lambda để lọc các phần tử:**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Lọc các số chẵn
    vector<int> evenNumbers;
    copy_if(numbers.begin(), numbers.end(), back_inserter(evenNumbers), [](int n)
```

```
        return n % 2 == 0;
    });

    for (int num : evenNumbers) {
        cout << num << " ";  // Output: 2 4 6 8 10
    }
    cout << endl;

    return 0;
}
```

### 3.7.8. Kết luận

Hàm lambda là một công cụ mạnh mẽ và linh hoạt trong C++ giúp viết mã ngắn gọn, dễ đọc và dễ duy trì hơn. Chúng đặc biệt hữu ích trong các tình huống cần sử dụng các hàm nhỏ hoặc logic ngắn gọn ngay tại chỗ. Tuy nhiên, cần cẩn trọng khi sử dụng lambda để tránh làm mã nguồn trở nên khó hiểu hoặc khó bảo trì. Việc hiểu rõ ưu điểm và nhược điểm của lambda sẽ giúp lập trình viên sử dụng chúng một cách hiệu quả trong các dự án thực tế.

## 3.8. Từ khóa auto trong C++

### 3.8.1. Giới thiệu về từ khóa auto trong C++

Từ khóa auto được giới thiệu lần đầu tiên trong C++98, nhưng ý nghĩa của nó đã thay đổi đáng kể trong C++11. Trong C++11 và các tiêu chuẩn mới hơn, auto cho phép trình biên dịch tự động suy diễn kiểu dữ liệu của biến từ biểu thức khởi tạo. Điều này giúp lập trình viên tránh phải khai báo kiểu dữ liệu rõ ràng, đặc biệt hữu ích khi làm việc với các kiểu dữ liệu phức tạp hoặc dài dòng như iterator, con trỏ hàm, hoặc kiểu trả về của hàm.

### 3.8.2. Cách sử dụng từ khóa auto

Từ khóa auto được sử dụng để khai báo một biến mà không cần chỉ định rõ kiểu dữ liệu của nó. Thay vào đó, trình biên dịch sẽ suy diễn kiểu dữ liệu của biến dựa trên giá trị khởi tạo. Ví dụ cơ bản về auto:

```
auto x = 42;              // x có kiểu int
auto y = 3.14159;         // y có kiểu double
auto name = "John";       // name có kiểu const char*
auto ptr = &x;            // ptr có kiểu int*
```

Phân tích ví dụ:

- x được khởi tạo bằng 42, do đó trình biên dịch suy diễn x có kiểu int.
- y được khởi tạo bằng 3.14159, do đó trình biên dịch suy diễn y có kiểu double.
- name được khởi tạo bằng chuỗi ký tự "John", do đó trình biên dịch suy diễn name có kiểu const char*.
- ptr được khởi tạo bằng địa chỉ của x, do đó trình biên dịch suy diễn ptr có kiểu int*.

### 3.8.3. Lợi ích của việc sử dụng auto

**Giảm thiểu lỗi khai báo kiểu dữ liệu:** Khi làm việc với các kiểu dữ liệu phức tạp, chẳng hạn như iterator của STL hoặc con trỏ hàm, việc khai báo kiểu dữ liệu chính xác có thể dễ gây nhầm lẫn và sai sót. Sử dụng auto giúp trình biên dịch tự động suy diễn kiểu, giảm thiểu khả năng mắc lỗi.

Ví dụ với iterator:

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};

    // Sử dụng auto để khai báo iterator, tránh phải biết kiểu iterator cụ thể
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

Trong ví dụ trên, auto giúp tránh phải khai báo kiểu dữ liệu đầy đủ của iterator, làm cho mã nguồn ngắn gọn và dễ hiểu hơn.

**Tăng cường tính dễ đọc và dễ bảo trì của mã nguồn:** Khi khai báo kiểu dữ liệu cụ thể, mã nguồn có thể trở nên dài dòng và khó đọc, đặc biệt là với các kiểu dữ liệu phức tạp. auto làm cho mã nguồn ngắn gọn hơn, tập trung vào logic thay vì chi tiết triển khai.

Ví dụ với con trỏ hàm:

```
#include <iostream>
using namespace std;

// Hàm cộng hai số nguyên
int add(int a, int b) {
    return a + b;
}

// Hàm trừ hai số nguyên
int subtract(int a, int b) {
    return a - b;
}

int main() {
    // Sử dụng auto để khai báo con trỏ hàm, tránh phải biết kiểu con trỏ hàm cụ thể
    auto func = add;
    cout << "Sum: " << func(5, 3) << endl;  // Output: Sum: 8

    func = subtract;
    cout << "Difference: " << func(5, 3) << endl;  // Output: Difference: 2

    return 0;
}
```

Sử dụng auto để khai báo con trỏ hàm giúp mã nguồn trở nên ngắn gọn và dễ bảo trì hơn.

**Linh hoạt khi thay đổi kiểu dữ liệu:** Khi sử dụng auto, nếu biểu thức khởi tạo thay đổi kiểu dữ liệu, trình biên dịch sẽ tự động suy diễn kiểu dữ liệu mới. Điều này giúp mã nguồn dễ dàng thích ứng với thay đổi mà không cần cập nhật khai báo kiểu dữ liệu.

Ví dụ thay đổi kiểu dữ liệu:

```
auto value = 5;  // value có kiểu int
value = 3.14;    // Lỗi biên dịch vì value đã được suy diễn thành kiểu int

auto flexibleValue = 5.0;  // flexibleValue có kiểu double
flexibleValue = 3.14;      // Không có lỗi, vì flexibleValue là kiểu double
```

Trong ví dụ trên, value được khởi tạo là int và không thể thay đổi kiểu dữ liệu sau khi đã khởi tạo. auto giúp phát hiện lỗi tại thời điểm biên dịch, đảm bảo tính toàn vẹn của dữ liệu.

### 3.8.4. Hạn chế và lưu ý khi sử dụng auto

Mặc dù auto mang lại nhiều lợi ích, nhưng cũng có một số hạn chế và lưu ý khi sử dụng:

**Khi kiểu dữ liệu không rõ ràng:** Sử dụng auto có thể gây ra nhầm lẫn khi kiểu dữ liệu không rõ ràng hoặc khi giá trị khởi tạo có nhiều kiểu dữ liệu tiềm ẩn. Điều này có thể dẫn đến lỗi khó phát hiện trong quá trình phát triển.

Ví dụ về sự không rõ ràng:

```
auto result = 10 + 2.5;  // result được suy diễn thành kiểu double

int x = 10;
int y = 4;
auto division = x / y;   // division được suy diễn thành kiểu int (phép chia ngu
```

Trong ví dụ trên, division được suy diễn thành int vì cả x và y đều là int, nhưng điều này có thể không rõ ràng đối với lập trình viên.

**Khi cần tính nhất quán về kiểu dữ liệu:** Trong một số trường hợp, việc xác định rõ ràng kiểu dữ liệu có thể quan trọng để đảm bảo tính nhất quán trong mã nguồn, đặc biệt khi làm việc với các phép toán số học hoặc khi truyền tham số cho hàm.

Ví dụ về tính nhất quán:

```
int a = 10;
auto b = 2.5;        // b được suy diễn thành double
auto c = a / b;      // c được suy diễn thành double, nhưng phép chia có thể khô
```

Trong ví dụ trên, việc sử dụng auto cho biến c có thể gây nhầm lẫn vì kiểu của nó phụ thuộc vào kết quả của phép toán giữa a và b.

**Khi kiểu dữ liệu phức tạp cần được nhấn mạnh:** Nếu kiểu dữ liệu của biến rất quan trọng đối với logic của chương trình hoặc giúp người đọc hiểu rõ mã nguồn, việc sử dụng auto có thể làm giảm tính rõ ràng của mã nguồn. Ví dụ về kiểu dữ liệu cần nhấn mạnh:

```
std::vector<std::vector<int>> matrix(10, std::vector<int>(10));
auto it = matrix.begin();  // it có kiểu std::vector<std::vector<int>>::iterator
```

Trong ví dụ trên, sử dụng auto có thể làm người đọc khó nhận biết kiểu dữ liệu chính xác của it, đặc biệt khi kiểu dữ liệu này quan trọng đối với logic của chương trình.

### 3.8.5. Từ khóa auto với hàm Lambda

Lambda là một hàm nặc danh (không có tên) được giới thiệu trong C++11 và rất tiện lợi khi làm việc với các biểu thức ngắn hoặc các hàm đơn giản. Lambda có thể được sử dụng làm đối số cho các hàm như `for_each, transform`, hoặc làm hàm gọi lại (`callback`). Ví dụ với lambda và auto:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Sử dụng auto để khai báo lambda
    auto print = [](int value) {
        cout << value << " ";
    };

    for_each(vec.begin(), vec.end(), print);  // Output: 1 2 3 4 5

    return 0;
}
```

Lambda giúp mã nguồn ngắn gọn và linh hoạt hơn, đặc biệt khi cần thực hiện các tác vụ đơn giản.

### 3.8.6. Từ khóa auto với kiểu trả về của hàm

Trong C++11 và các tiêu chuẩn mới hơn (C++14, C++17, C++20), từ khóa auto không chỉ được sử dụng để suy diễn kiểu dữ liệu của biến mà còn có thể được sử dụng để suy diễn kiểu trả về của hàm. Điều này giúp lập trình viên linh hoạt hơn khi định nghĩa hàm, đặc biệt là khi kiểu trả về của hàm phức tạp hoặc không rõ ràng.

**Cách Sử Dụng auto với Kiểu Trả Về của Hàm:** Khi sử dụng auto để suy diễn kiểu trả về của hàm, trình biên dịch sẽ tự động suy luận kiểu dữ liệu từ biểu thức trả về của hàm. Điều này cho phép bạn viết các hàm mà không cần chỉ định kiểu trả về cụ thể, đặc biệt hữu ích khi kiểu trả về là kết quả của một biểu thức phức tạp hoặc sử dụng các hàm template.

Cú pháp cơ bản:

```
auto functionName(parameters) -> return_type {
    // function body
}
```

`auto`: Cho phép trình biên dịch tự động suy diễn kiểu trả về từ biểu thức trả về.

`-> return_type`: Định nghĩa rõ ràng kiểu trả về, có thể bỏ qua nếu trình biên dịch có thể suy diễn từ ngữ cảnh.

Ví dụ cơ bản về auto với kiểu trả về của hàm:

```
#include <iostream>
using namespace std;

// Sử dụng auto để suy diễn kiểu trả về từ biểu thức trả về
auto add(int a, int b) {
    return a + b;  // Trình biên dịch suy diễn kiểu trả về là int
}

int main() {
    cout << add(3, 4) << endl;  // Output: 7
    return 0;
}
```

Trong ví dụ trên, trình biên dịch tự động suy diễn kiểu trả về của hàm add là int dựa trên kiểu của biểu thức a + b.

**Lợi Ích của việc sử dụng auto với kiểu trả về của hàm:**

- Giảm bớt sự phức tạp khi khai báo kiểu trả về phức tạp: Trong các trường hợp mà kiểu trả về phức tạp, ví dụ như các iterator của STL hoặc khi làm việc với các hàm template, auto giúp giảm bớt sự phức tạp trong khai báo hàm.
- Dễ dàng duy trì mã nguồn: Khi sử dụng auto, nếu hàm thay đổi logic và do đó thay đổi kiểu trả về, trình biên dịch sẽ tự động cập nhật kiểu trả về mà không cần sửa đổi mã nguồn.
- Linh hoạt và tổng quát hơn: Sử dụng auto giúp hàm tổng quát và linh hoạt hơn, đặc biệt hữu ích khi kết hợp với các template, cho phép các hàm này hoạt động với nhiều kiểu dữ liệu khác nhau.

Ví dụ về kiểu trả về phức tạp với auto:

```
#include <vector>
#include <iostream>
using namespace std;

// Hàm trả về iterator của vector
auto findMax(vector<int>& vec) {
    return max_element(vec.begin(), vec.end());  // Trả về iterator tới phần tử lớn nhất
}

int main() {
    vector<int> numbers = {1, 3, 5, 7, 2, 8};
    auto maxIt = findMax(numbers);
    cout << "Max element: " << *maxIt << endl;  // Output: Max element: 8
    return 0;
}
```

Trong ví dụ này, hàm findMax trả về một iterator tới phần tử lớn nhất trong vector. Việc sử dụng auto giúp chúng ta không phải viết kiểu trả về phức tạp.

**Khi Nào Nên Sử Dụng auto với Kiểu Trả Về của Hàm:**

- Khi kiểu trả về phức tạp hoặc dài dòng: Sử dụng auto rất hữu ích khi kiểu trả về phức tạp như các iterator của STL, các con trỏ hàm, hoặc các kiểu dữ liệu của thư viện bên ngoài.
- Khi kiểu trả về phụ thuộc vào biểu thức: Nếu kiểu trả về của hàm phụ thuộc vào biểu thức trả về và biểu thức này có thể thay đổi, auto giúp linh hoạt hơn và giảm lỗi bảo trì.
- Khi làm việc với các hàm template: Sử dụng auto giúp các hàm template dễ đọc hơn và ít phức tạp hơn khi kiểu trả về phụ thuộc vào tham số template.

Ví dụ với hàm template và auto:

```
#include <iostream>
using namespace std;

// Hàm template sử dụng auto để suy diễn kiểu trả về
template <typename T, typename U>
auto multiply(T x, U y) {
    return x * y;  // Trình biên dịch suy diễn kiểu trả về từ x * y
}

int main() {
    cout << multiply(3, 4) << endl;       // Output: 12 (int)
    cout << multiply(3.5, 4) << endl;     // Output: 14.0 (double)
    return 0;
}
```

Trong ví dụ trên, hàm multiply sử dụng auto cho phép trình biên dịch tự động suy diễn kiểu trả về dựa trên các tham số x và y.

**Khi Nào Nên Tránh Sử Dụng auto với Kiểu Trả Về của Hàm:**

- Khi cần rõ ràng về kiểu trả về: Trong một số trường hợp, việc chỉ rõ kiểu trả về giúp mã nguồn rõ ràng hơn và dễ hiểu hơn đối với những người khác đọc mã hoặc khi bảo trì mã.
- Khi kiểu trả về không rõ ràng hoặc có thể gây nhầm lẫn: Nếu kiểu trả về của hàm không rõ ràng hoặc có thể gây nhầm lẫn (ví dụ: khi sử dụng các phép

toán số học mà kiểu trả về phụ thuộc vào các phép toán này), thì việc sử dụng auto có thể làm giảm tính rõ ràng.

Ví dụ về trường hợp nên tránh dùng auto:

```
#include <iostream>
using namespace std;

auto division(int a, int b) {
    return a / b;  // Kiểu trả về là int, có thể gây nhầm lẫn nếu mong muốn kết quả là số
}

int main() {
    cout << division(5, 2) << endl;  // Output: 2, không phải 2.5
    return 0;
}
```

Trong ví dụ trên, việc sử dụng auto có thể gây nhầm lẫn vì kiểu trả về thực tế là int do phép chia hai số nguyên. Nếu người đọc mã không chú ý, họ có thể hiểu sai kiểu trả về.

**decltype(auto) trong C++14:** Trong C++14, decltype(auto) được giới thiệu để bổ sung cho auto. Trong khi auto luôn loại bỏ phần tham chiếu và phần const của kiểu, decltype(auto) sẽ suy diễn kiểu chính xác từ biểu thức, bao gồm cả tham chiếu và const. Ví dụ về decltype(auto):

```
#include <iostream>
using namespace std;

int x = 5;

decltype(auto) getX() {
    return (x);  // Trả về x bằng tham chiếu
}

int main() {
    getX() = 10;  // Thay đổi giá trị của x thông qua tham chiếu
    cout << x << endl;  // Output: 10
```

```
    return 0;
}
```

Trong ví dụ này, decltype(auto) cho phép hàm getX trả về một tham chiếu tới x, do đó việc thay đổi giá trị của getX() cũng thay đổi giá trị của x. Sử dụng auto với kiểu trả về của hàm giúp mã nguồn ngắn gọn, dễ đọc và linh hoạt hơn, đặc biệt khi làm việc với các kiểu dữ liệu phức tạp hoặc khi cần thay đổi kiểu trả về mà không muốn cập nhật nhiều nơi trong mã nguồn. Tuy nhiên, cần sử dụng auto một cách cẩn trọng để tránh làm giảm tính rõ ràng của mã nguồn và gây nhầm lẫn cho người đọc.

Khi nên dùng auto với kiểu trả về của hàm:

- Khi kiểu trả về phức tạp và dài dòng.
- Khi làm việc với các hàm template hoặc hàm tổng quát.
- Khi cần linh hoạt trong việc thay đổi kiểu trả về.
- Khi nên tránh dùng auto với kiểu trả về của hàm:
- Khi cần rõ ràng về kiểu trả về của hàm.
- Khi kiểu trả về có thể gây nhầm lẫn hoặc không rõ ràng.

### 3.8.7. Tổng kết về auto

Từ khóa auto là một công cụ mạnh mẽ và linh hoạt trong C++ giúp giảm bớt sự phức tạp khi khai báo biến với kiểu dữ liệu phức tạp. Tuy nhiên, cần sử dụng auto một cách cẩn thận để tránh những nhầm lẫn không đáng có và đảm bảo mã nguồn dễ đọc, dễ hiểu. Khi làm việc với các biến có kiểu dữ liệu rõ ràng và cần nhấn mạnh, nên khai báo kiểu cụ thể thay vì sử dụng auto.

Khi nào nên sử dụng auto:

- Khi khai báo các biến với kiểu dữ liệu phức tạp (như iterator, con trỏ hàm).
- Khi muốn mã nguồn ngắn gọn và dễ đọc.
- Khi làm việc với các hàm tổng quát hoặc lambda.

Khi nên tránh sử dụng auto:

- Khi cần rõ ràng về kiểu dữ liệu.

- Khi làm việc với các kiểu dữ liệu phức tạp cần được nhấn mạnh.
- Khi kiểu dữ liệu của biến không rõ ràng hoặc dễ gây nhầm lẫn.

## 3.9. Tổng kết chương

Trong C/C++, có nhiều kỹ thuật và khái niệm liên quan đến hàm mà lập trình viên cần nắm vững để viết mã nguồn hiệu quả, rõ ràng, và dễ bảo trì. Việc lựa chọn kỹ thuật nào để sử dụng phụ thuộc vào yêu cầu cụ thể của từng tình huống:

- Sử dụng hàm đơn giản khi cần tách mã nguồn thành các khối nhỏ, dễ quản lý.
- Sử dụng con trỏ hàm khi cần sự linh hoạt cao và thay đổi hành vi của chương trình tại runtime.
- Sử dụng function templates để giảm sự lặp lại mã nguồn và tăng tính tái sử dụng.
- Sử dụng biểu thức lambda khi cần các hàm đơn giản hoặc cần hàm ngay tại chỗ mà không cần định nghĩa hàm riêng biệt.

## 3.10. Bổ sung - Một số ví dụ về Lambda

**Lambda Cơ Bản:** Hàm lambda có thể được sử dụng để định nghĩa một hàm ngắn gọn.

```
#include <iostream>

int main() {
    // Hàm lambda cơ bản không có tham số
    auto greet = []() {
        std::cout << "Hello, World!" << std::endl;
    };

    greet(); // Gọi hàm lambda

    return 0;
}
```

**Lambda với Tham Số:** Lambda có thể nhận các tham số và trả về một giá trị.

```
#include <iostream>

int main() {
    // Hàm lambda nhận hai tham số và trả về tổng của chúng
    auto add = [](int a, int b) -> int {
        return a + b;
    };

    int result = add(5, 3);
    std::cout << "Result of addition: " << result << std::endl; // Output: 8

    return 0;
}
```

**Lambda với Tham Chiếu Biến Bên Ngoài (Capture by Reference):** Có thể "bắt" các biến từ phạm vi bên ngoài (capture by reference) để sử dụng chúng trong lambda.

```
#include <iostream>

int main() {
    int a = 5;
    int b = 10;

    // Sử dụng [&, =] để "bắt" tất cả các biến bên ngoài theo tham chiếu
    auto modifyVariables = [&]() {
        a += 10; // Thay đổi giá trị của a
        b *= 2;  // Thay đổi giá trị của b
    };

    modifyVariables(); // Gọi hàm lambda

    std::cout << "Modified a: " << a << std::endl; // Output: 15
    std::cout << "Modified b: " << b << std::endl; // Output: 20

    return 0;
```

```
}
```

**Lambda với Giá Trị Trả Về Tự Động (Auto Return Type):** Nếu lambda chỉ có một biểu thức duy nhất, kiểu trả về có thể được suy diễn tự động mà không cần chỉ định kiểu trả về.

```
#include <iostream>

int main() {
    auto multiply = [](int x, int y) { return x * y; }; // Kiểu trả về được suy diễn tự đ

    std::cout << "Multiplication result: " << multiply(5, 3) << std::endl; // Output: 15

    return 0;
}
```

**Lambda với Capture by Value:** Có thể "bắt" các biến từ phạm vi bên ngoài theo giá trị để sử dụng trong lambda mà không ảnh hưởng đến biến bên ngoài.

```
#include <iostream>

int main() {
    int x = 10;

    // Sử dụng [=] để "bắt" các biến bên ngoài theo giá trị
    auto copyValue = [x]() mutable {
        x += 5; // Chỉ thay đổi giá trị trong lambda, không ảnh hưởng đến biến bên ngoài
        std::cout << "Inside lambda, x: " << x << std::endl; // Output: 15
    };

    copyValue();

    std::cout << "Outside lambda, x: " << x << std::endl; // Output: 10

    return 0;
}
```

**Lambda như Tham Số Hàm:** Lambda có thể được truyền như một tham số vào hàm khác, điều này rất hữu ích trong các thuật toán tổng quát.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Sử dụng std::count_if với hàm lambda để đếm các số chẵn
    int evenCount = std::count_if(numbers.begin(), numbers.end(), [](int x) {
        return x % 2 == 0; // Điều kiện số chẵn
    });

    std::cout << "Number of even numbers: " << evenCount << std::endl; // Output

    return 0;
}
```

**Lambda với std::function:** Lambda có thể được gán cho std::function, cho phép lambda được lưu trữ và sử dụng lại nhiều lần.

```
#include <iostream>
#include <functional>

int main() {
    std::function<int(int, int)> add = [](int a, int b) {
        return a + b;
    };

    int result = add(3, 4);
    std::cout << "Result using std::function: " << result << std::endl; // Output

    return 0;
}
```

**Lambda Đệ Quy:** Lambda trong C++14 và các phiên bản sau có thể sử dụng đệ quy nếu chúng ta sử dụng một mẹo nhỏ bằng cách sử dụng std::function.

```
#include <iostream>
#include <functional>

int main() {
    // Sử dụng std::function để cho phép lambda gọi chính nó
    std::function<int(int)> factorial = [&factorial](int n) -> int {
        return (n <= 1) ? 1 : n * factorial(n - 1);
    };

    std::cout << "Factorial of 5: " << factorial(5) << std::endl; // Output: 120

    return 0;
}
```

**Lambda với Trạng Thái Mutable:** Mặc định, các biến bắt bởi lambda là const, nhưng bạn có thể sử dụng từ khóa mutable để cho phép thay đổi giá trị của chúng.

```
#include <iostream>

int main() {
    int counter = 0;

    auto increment = [counter]() mutable {
        return ++counter; // cho phép thay đổi giá trị của counter
    };

    std::cout << "First call: " << increment() << std::endl; // Output: 1
    std::cout << "Second call: " << increment() << std::endl; // Output: 2
    std::cout << "Original counter: " << counter << std::endl; // Output: 0 (counter bên

    return 0;
}
```

**Lambda và Con trỏ hàm (Function Pointers):** Lambda có thể được chuyển đổi thành con trỏ hàm.

```cpp
#include <iostream>

int main() {
    // Lambda đơn giản
    auto add = [](int a, int b) { return a + b; };

    // Con trỏ hàm
    int (*functionPtr)(int, int) = add;

    std::cout << "Addition using function pointer: " << functionPtr(2, 3) << std:

    return 0;
}
```

### 3.10.1. Kết luận

Hàm lambda trong C++ mang lại sự linh hoạt cao trong việc lập trình và đặc biệt hữu ích trong các trường hợp như xử lý các thuật toán tổng quát, xử lý các sự kiện bất đồng bộ, và các thao tác xử lý dữ liệu phức tạp. Các ví dụ trên chỉ là một số trong rất nhiều cách sử dụng thú vị và sáng tạo mà hàm lambda có thể mang lại.

# Chương 4
# Viết code hiệu quả



## 4.1. Giới thiệu

Viết mã hiệu quả là một kỹ năng quan trọng trong lập trình. Mã hiệu quả không chỉ giúp chương trình chạy nhanh hơn mà còn sử dụng tài nguyên hệ thống một cách tối ưu và dễ bảo trì hơn. Viết mã hiệu quả là nghệ thuật cân bằng giữa tính dễ đọc, dễ bảo trì của mã nguồn và hiệu suất của chương trình. Một mã nguồn hiệu quả không chỉ chạy nhanh mà còn sử dụng tài nguyên hệ thống (như bộ nhớ, CPU) một cách tối ưu và dễ dàng bảo trì, mở rộng. Để đạt được điều này, chúng ta cần tuân thủ một số nguyên tắc cơ bản, kỹ thuật và quy tắc tối ưu hóa.

## 4.2. Nguyên lý cơ bản để viết code hiệu quả

### 4.2.1. Đơn giản hóa mã (Code Simplification)

**Nguyên lý**: Mã đơn giản không chỉ dễ đọc và dễ bảo trì mà còn thường nhanh hơn vì ít xử lý hơn và tránh được những thao tác không cần thiết. Việc giảm thiểu sự phức tạp của mã giúp tối ưu hóa quá trình dịch mã và thực thi.

Chi tiết hơn về cách áp dụng:

- Tránh sử dụng nhiều lệnh điều kiện lồng nhau: Thay vì sử dụng nhiều câu lệnh if-else lồng nhau, hãy sử dụng cấu trúc điều khiển như switch-case hoặc các câu lệnh if riêng biệt khi có thể.
- Sử dụng toán tử gán đơn giản: Ví dụ, sử dụng toán tử += thay vì a = a + b để làm cho mã ngắn gọn và rõ ràng hơn.
- Giảm thiểu số phép toán: Tối ưu hóa các phép toán số học và logic để giảm bớt khối lượng công việc cho CPU.

Ví dụ chi tiết:

```
// Trước tối ưu hóa
int calculateDiscount(int quantity) {
    int discount = 0;
    if (quantity > 100) {
        discount = 20;
    } else if (quantity > 50) {
        discount = 10;
    } else {
        discount = 0;
    }
    return discount;
}

// Sau tối ưu hóa
int calculateDiscount(int quantity) {
    if (quantity > 100) return 20;
    if (quantity > 50) return 10;
    return 0;
}
```

Ví dụ 2:

```
// Trước tối ưu hóa
for (int i = 0; i < n; i++) {
    if (x > y) {
        result[i] = x * i + y * i;
    } else {
        result[i] = y * i + x * i;
    }
}

// Sau tối ưu hóa
int sum = (x > y) ? x + y : y + x;
for (int i = 0; i < n; i++) {
    result[i] = sum * i; }
```

### 4.2.2. Tối ưu hóa vấn đề (Problem Simplification)

**Nguyên lý**: Thay vì giải quyết vấn đề một cách toàn diện, hãy tìm cách đơn giản hóa vấn đề. Điều này có thể bao gồm việc giảm kích thước của tập dữ liệu, sử dụng các phương pháp tính toán đơn giản hơn hoặc cải tiến thuật toán.

Chi tiết hơn về cách áp dụng:

- Tìm các công thức toán học thay thế: Như sử dụng công thức tổng dãy số để thay thế cho vòng lặp tính tổng.
- Sử dụng các kỹ thuật phân tích bài toán: Ví dụ, sử dụng Dynamic Programming để tránh việc tính toán lặp lại nhiều lần trong các bài toán tối ưu hóa.

Ví dụ chi tiết: Tối ưu hóa bài toán tìm số Fibonacci bằng cách sử dụng Dynamic Programming thay vì tính toán đệ quy đơn giản.

```
// Tính Fibonacci với Dynamic Programming
int fib(int n) {
    int fibArray[n + 2]; // Tạo mảng để lưu trữ giá trị Fibonacci
    fibArray[0] = 0;
    fibArray[1] = 1;

    for (int i = 2; i <= n; i++) {
        fibArray[i] = fibArray[i - 1] + fibArray[i - 2]; // Tính giá trị Fibonacci
```

```
    }

    return fibArray[n];
}
```

Ví dụ 2: Tính tổng của dãy số từ 1 đến n:

```
// Trước tối ưu hóa
int sum = 0;
for (int i = 1; i <= n; i++) {
    sum += i;
}

// Sau tối ưu hóa
int sum = n * (n + 1) / 2;
```

## 4.3. Các Quy Tắc Tối Ưu Hóa Mã

### 4.3.1. Quy tắc đổi không gian lấy thời gian (Space for Time Rules)

**Nguyên lý:** Thông tin dư thừa có thể làm giảm thời gian chạy của chương trình bằng cách tăng không gian bộ nhớ sử dụng. Các kỹ thuật mở rộng:

**Data Structure Augmentation:** Tăng cường cấu trúc dữ liệu với thông tin bổ sung để cải thiện thời gian truy cập hoặc tính toán. Ví dụ chi tiết: Sử dụng hashmap để đếm tần suất xuất hiện của các phần tử trong một mảng thay vì sử dụng vòng lặp lồng nhau.

```
#include <iostream>
#include <unordered_map>
using namespace std;

void countFrequencies(int arr[], int n) {
    unordered_map<int, int> freq;
    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
    }

    for (auto pair : freq) {
```

```
        cout << "Element " << pair.first << " occurs " << pair.second << " times." << endl
    }
}

int main() {
    int arr[] = {1, 2, 3, 2, 3, 1, 3, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    countFrequencies(arr, n);
    return 0;
}
```

**Store Precomputed Results (Memoization):** Lưu trữ kết quả đã tính toán để sử dụng lại. Ví dụ chi tiết: Tính toán giá trị factorial sử dụng kỹ thuật lưu trữ kết quả để tránh tính toán lại.

```
#include <iostream>
#include <unordered_map>
using namespace std;

unordered_map<int, long long> factorialMemo;

long long factorial(int n) {
    if (n <= 1) return 1;
    if (factorialMemo.find(n) != factorialMemo.end()) return factorialMemo[n];
    factorialMemo[n] = n * factorial(n - 1);
    return factorialMemo[n];
}

int main() {
    cout << "Factorial of 5: " << factorial(5) << endl; // Output: 120
    cout << "Factorial of 6: " << factorial(6) << endl; // Output: 720
    return 0;
}
```

Ví dụ về Caching:

```
using namespace std;
```

```
map<int, int> cache;

int fib(int n) {
    if (n <= 1) return n;
    if (cache.find(n) != cache.end()) return cache[n];
        // Kiểm tra xem đã tính toán trước chưa
    cache[n] = fib(n - 1) + fib(n - 2);
    return cache[n];
}

int main() {
    cout << "Fibonacci(10): " << fib(10) << endl;  // Output: 55
    return 0;
}
```

### 4.3.2. Quy tắc đổi thời gian lấy không gian (Time for Space Rules)

**Nguyên lý:** Giảm không gian bộ nhớ cần thiết bằng cách tái tính toán các giá trị khi cần, đổi lại chi phí là tăng thời gian thực thi.

Các kỹ thuật mở rộng:

**Packing Data:** Nén dữ liệu để giảm không gian lưu trữ, chấp nhận tăng thời gian xử lý khi truy xuất hoặc thao tác trên dữ liệu. Ví dụ chi tiết:

```
// Giả sử cần lưu trữ trạng thái của 8 lá cờ (flag)
bool flags[8];  // Trước tối ưu hóa: sử dụng 8 byte (trên hầu hết các hệ thống)
```

Thay vì lưu trữ trạng thái của 8 cờ boolean như mảng, ta có thể sử dụng một biến kiểu `unsigned char` để đại diện cho cả 8 cờ.

```
#include <iostream>
using namespace std;

void setFlag(unsigned char &flags, int position) {
    flags |= (1 << position);
}

bool isFlagSet(unsigned char flags, int position) {
    return flags & (1 << position);
```

```
}

int main() {
    unsigned char flags = 0;
    setFlag(flags, 3); // Bật cờ ở vị trí 3
    cout << "Is flag at position 3 set? " << (isFlagSet(flags, 3) ? "Yes" : "No") << endl;
    return 0;
}
```

## 4.4. Tối Ưu Hóa Vòng Lặp (Loop Optimization Rules)

**Nguyên lý:** Vòng lặp thường chiếm phần lớn thời gian chạy trong các chương trình, do đó tối ưu hóa vòng lặp là bước quan trọng để cải thiện hiệu suất tổng thể.

Các kỹ thuật mở rộng:

**Code Motion Out of Loops:** Di chuyển các phép tính không thay đổi ra khỏi vòng lặp để tránh tính toán lại không cần thiết. Ví dụ chi tiết: Tính tổng các phần tử mảng đã chia hết cho 2.

```
// Trước tối ưu hóa
int sum = 0;
for (int i = 0; i < n; i++) {
    if (arr[i] % 2 == 0) {
        sum += arr[i] / 2;
    }
}

// Sau tối ưu hóa
int sum = 0;
for (int i = 0; i < n; i++) {
    if (arr[i] % 2 == 0) {
        int half = arr[i] / 2;  // Tính toán trước
        sum += half;
    }
}
```

**Loop Unrolling:** Tăng kích thước vòng lặp và giảm số lần lặp để giảm chi phí chỉ số vòng lặp và cải thiện hiệu suất. Ví dụ chi tiết:

```
// Trước tối ưu hóa
for (int i = 0; i < n; i++) {
    arr[i] = 0;
}

// Sau tối ưu hóa (Loop Unrolling)
int i = 0;
for (; i < n - 4; i += 4) {
    arr[i] = 0;
    arr[i + 1] = 0;
    arr[i + 2] = 0;
    arr[i + 3] = 0;
}

for (; i < n; i++) {
    arr[i] = 0;
}
```

**Combining Tests:** Giảm số lượng kiểm tra điều kiện trong vòng lặp để giảm tải cho CPU. Ví dụ thêm về Code Motion Out of Loops:

```
// Trước tối ưu hóa
for (int i = 0; i < n; i++) {
    result[i] = 2 * 3.14 * radius;
}

// Sau tối ưu hóa
double circumference = 2 * 3.14 * radius;
for (int i = 0; i < n; i++) {
    result[i] = circumference;
}
```

## 4.5. Quy Tắc Tối Ưu Hóa Logic (Logic Optimization Rules)

- Exploit Algebraic Identities: Sử dụng các nhận dạng đại số để giảm chi phí tính toán.
- Short-Circuiting: Sử dụng các phép toán ngắn mạch để tránh các tính toán không cần thiết.

- Reordering Tests: Sắp xếp các kiểm tra logic sao cho kiểm tra rẻ nhất và thường thành công nhất thực hiện trước.

Ví dụ về Short-Circuiting:

```
// Trước tối ưu hóa
if (isExpensiveCheck() && isSimpleCheck())

// Sau tối ưu hóa
if (isSimpleCheck() && isExpensiveCheck()) // Đảm bảo kiểm tra rẻ nhất trước
```

Tối ưu hóa các biểu thức logic để giảm số phép tính và cải thiện tốc độ thực thi. Các kỹ thuật mở rộng:

**Reordering Tests:** Sắp xếp lại thứ tự các kiểm tra logic để giảm số lần kiểm tra. Ví dụ chi tiết: Đoạn mã kiểm tra xem một số có phải là số nguyên tố hay không.

```
// Trước tối ưu hóa
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i <= sqrt(n); i++) {  // Sử dụng hàm sqrt() trong mỗi vòng lặp
        if (n % i == 0) return false;
    }
    return true;
}

// Sau tối ưu hóa
bool isPrime(int n) {
    if (n <= 1) return false;
    int root = sqrt(n);  // Tính sqrt() một lần trước vòng lặp
    for (int i = 2; i <= root; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

## 4.6. Quy Tắc Tối Ưu Hóa Thủ Tục (Procedure Optimization Rules)

**Nguyên lý:** Tối ưu hóa việc sử dụng các thủ tục (hàm) để giảm chi phí gọi hàm và cải thiện hiệu suất tổng thể. Các kỹ thuật mở rộng:

**Inlining Procedures:** Thay vì gọi hàm, hãy chèn mã của hàm trực tiếp vào nơi cần sử dụng để giảm chi phí gọi hàm. Ví dụ chi tiết:

```
// Trước tối ưu hóa
inline int square(int x) {
    return x * x;
}

int main() {
    int a = square(5);  // Gọi hàm
    return 0;
}

// Sau tối ưu hóa (thực hiện inline)
int main() {
    int a = 5 * 5;  // Chèn mã trực tiếp
    return 0;
}
```

**Collapsing Procedure Hierarchies:** Sáp nhập các hệ thống thủ tục để giảm chi phí gọi hàm.

**Exploit Common Cases:** Tổ chức thủ tục để xử lý tất cả các trường hợp một cách chính xác và các trường hợp thông thường một cách hiệu quả.

**Transformations on Recursive Procedures:** Tối ưu hóa các thủ tục đệ quy bằng cách loại bỏ đệ quy đuôi, sử dụng vòng lặp thay vì gọi đệ quy khi có thể.

Ví dụ về Tail Recursion Elimination:

```
// Trước tối ưu hóa
int factorial(int n) {
    if (n <= 1) return 1;
```

```
    return n * factorial(n - 1);
}

// Sau tối ưu hóa với vòng lặp
int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

## 4.7. Quy Tắc Tối Ưu Hóa Biểu Thức (Expression Optimization Rules)

**Nguyên lý:** Sử dụng các kỹ thuật tối ưu hóa để giảm chi phí tính toán biểu thức. Các kỹ thuật mở rộng:

**Common Subexpression Elimination:** Nếu cùng một biểu thức được đánh giá nhiều lần mà không có biến nào trong đó thay đổi, hãy lưu kết quả để tránh tính toán lại. Ví dụ chi tiết:

```
// Trước tối ưu hóa
double area = length * width;
double perimeter = 2 * (length * width);

// Sau tối ưu hóa
double lw = length * width;  // Tính toán chung một lần
double area = lw;
double perimeter = 2 * lw;
```

## 4.8. Tối Ưu Hóa Phụ Thuộc Hệ Thống (System Dependent Optimization)

**Nguyên lý:** Tận dụng tối đa hiệu quả của hệ thống phần cứng và trình biên dịch để tối ưu hóa mã nguồn. Các kỹ thuật mở rộng:

**Register Allocation:** Lưu trữ các biến trong thanh ghi (register) thay vì trong bộ nhớ chính để tăng tốc độ truy cập. Ví dụ chi tiết:

```
register int counter;  // Khai báo biến lưu trữ trong thanh ghi
for (counter = 0; counter < 1000; counter++) {
    // Các thao tác sử dụng biến counter
}
```

**Instruction Caching:** Sử dụng bộ nhớ đệm lệnh để tối ưu hóa các vòng lặp chặt chẽ (tight loops) để tránh bị cache miss.

## 4.9. Tối Ưu Hóa Mã C/C++

### 4.9.1. Tối ưu hóa việc sử dụng biến và bộ nhớ

**Tối ưu hóa biến bằng cách sử dụng đúng kiểu dữ liệu:** Sử dụng kiểu dữ liệu phù hợp giúp tiết kiệm bộ nhớ và cải thiện hiệu suất. Ví dụ, nếu chỉ cần lưu trữ các số nguyên nhỏ (trong khoảng từ 0 đến 255), sử dụng kiểu unsigned char thay vì int sẽ tiết kiệm bộ nhớ.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
void example1() {
    int i;
    for (i = 0; i < 1000000; ++i) {
        // Mã xử lý...
    }
}

// Sau tối ưu hóa
void example2() {
    unsigned short i; // Sử dụng unsigned short thay vì int
    for (i = 0; i < 1000000; ++i) {
        // Mã xử lý...
    }
}

int main() {
```

```
    example2();
    return 0;
}
```

**Sử dụng biến register cho các biến được sử dụng thường xuyên:** Để tăng tốc độ truy cập biến, có thể yêu cầu trình biên dịch lưu trữ một biến cụ thể trong thanh ghi CPU bằng cách sử dụng từ khóa register. Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

void computeSum() {
    register int sum = 0; // Đề nghị trình biên dịch lưu trữ sum trong thanh ghi
    for (register int i = 0; i < 1000; ++i) {
        sum += i;
    }
    cout << "Sum is: " << sum << endl;
}

int main() {
    computeSum();
    return 0;
}
```

### 4.9.2. Tối ưu hóa vòng lặp (Loop Optimization)

**Di chuyển mã ra ngoài vòng lặp (Loop-Invariant Code Motion):** Di chuyển các phép toán hoặc biểu thức không thay đổi ra khỏi vòng lặp để tránh thực hiện các phép toán thừa trong mỗi lần lặp.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
void computeArray1(int arr[], int n, int factor) {
    for (int i = 0; i < n; ++i) {
        arr[i] = arr[i] * factor;
```

```
    }
}

// Sau tối ưu hóa
void computeArray2(int arr[], int n, int factor) {
    int mul = factor; // Di chuyển phép gán ra ngoài vòng lặp
    for (int i = 0; i < n; ++i) {
        arr[i] = arr[i] * mul;
    }
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    computeArray2(arr, 5, 2);
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";  // Output: 2 4 6 8 10
    }
    return 0;
}
```

**Gộp vòng lặp (Loop Fusion):** Nguyên lý: Nếu có hai hoặc nhiều vòng lặp lân cận hoạt động trên cùng một tập hợp dữ liệu, hãy gộp chúng thành một vòng lặp để giảm thiểu chi phí điều khiển vòng lặp.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
void separateLoops(int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        arr[i] = arr[i] * 2;
    }

    for (int i = 0; i < n; ++i) {
        arr[i] = arr[i] + 10;
    }
```

```
}

// Sau tối ưu hóa
void fusedLoops(int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        arr[i] = arr[i] * 2 + 10;  // Kết hợp hai vòng lặp thành một
    }
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    fusedLoops(arr, 5);
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";  // Output: 12 14 16 18 20
    }
    return 0;
}
```

**Loop Unrolling:** Mở rộng vòng lặp để giảm số lần kiểm tra điều kiện và cập nhật chỉ số vòng lặp. Điều này có thể giảm chi phí xử lý vòng lặp và cải thiện hiệu suất.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
void unrolledLoop1(int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        arr[i] = 0;
    }
}

// Sau tối ưu hóa
void unrolledLoop2(int arr[], int n) {
    int i = 0;
    for (; i < n - 4; i += 4) { // Mở rộng vòng lặp
```

```
        arr[i] = 0;
        arr[i + 1] = 0;
        arr[i + 2] = 0;
        arr[i + 3] = 0;
    }
    for (; i < n; ++i) { // Xử lý phần tử còn lại
        arr[i] = 0;
    }
}

int main() {
    int arr[10];
    unrolledLoop2(arr, 10);
    for (int i = 0; i < 10; i++) {
        cout << arr[i] << " ";  // Output: 0 0 0 0 0 0 0 0 0 0
    }
    return 0;
}
```

### 4.9.3. Tối ưu hóa thủ tục và hàm

**Sử dụng hàm inline:** Thay vì gọi hàm thông thường, hàm inline chèn trực tiếp mã của hàm tại nơi gọi. Điều này loại bỏ chi phí gọi hàm nhưng có thể làm tăng kích thước mã nếu sử dụng không cẩn thận.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
int square(int x) {
    return x * x;
}

// Sau tối ưu hóa
inline int squareInline(int x) {  // Định nghĩa hàm inline
    return x * x;
}
```

```
int main() {
    int num = 5;
    cout << "Square of 5 is: " << squareInline(num) << endl;  // Output: 25
    return 0;
}
```

**Sử dụng các biến cục bộ thay vì biến toàn cục:** Biến cục bộ có thời gian sống ngắn hơn và chiếm ít không gian bộ nhớ hơn so với biến toàn cục. Sử dụng biến cục bộ giúp giảm thiểu xung đột bộ nhớ và tăng hiệu suất truy cập.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa: sử dụng biến toàn cục
int globalSum = 0;

void addGlobal(int a, int b) {
    globalSum = a + b;
    cout << "Global Sum: " << globalSum << endl;
}

// Sau tối ưu hóa: sử dụng biến cục bộ
void addLocal(int a, int b) {
    int localSum = a + b;
    cout << "Local Sum: " << localSum << endl;
}

int main() {
    addLocal(3, 4);  // Output: Local Sum: 7
    return 0;
}
```

### 4.9.4. Tối ưu hóa biểu thức

**Khử biểu thức chung (Common Subexpression Elimination):** Nếu cùng một biểu thức được tính toán nhiều lần mà các biến trong biểu thức không thay đổi, lưu kết quả của biểu thức để sử dụng lại và tránh tính toán lại.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
void commonExpression1(int x, int y) {
    int a = (x + y) * 2;
    int b = (x + y) * 3;
    cout << "a: " << a << ", b: " << b << endl;
}

// Sau tối ưu hóa
void commonExpression2(int x, int y) {
    int sum = x + y;  // Khử biểu thức chung
    int a = sum * 2;
    int b = sum * 3;
    cout << "a: " << a << ", b: " << b << endl;
}

int main() {
    commonExpression2(2, 3);  // Output: a: 10, b: 15
    return 0;
}
```

**Khởi tạo tại thời điểm biên dịch (Compile-Time Initialization):** Khởi tạo càng nhiều biến càng tốt trước khi chương trình chạy để giảm chi phí khởi tạo trong runtime.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
const int size = 1000;
int arr[size];
```

```
// Sau tối ưu hóa
const int size = 1000;
int arr[size] = {0};  // Khởi tạo toàn bộ phần tử thành 0 tại thời điểm biên dịch

int main() {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";  // Output: 0 0 0 ... 0
    }
    return 0;
}
```

### 4.9.5. Tối ưu hóa dựa trên kiến trúc hệ thống

**Sử dụng bộ nhớ đệm (Caching):** Tận dụng bộ nhớ đệm của CPU để giảm thời gian truy cập dữ liệu. Dữ liệu được truy cập thường xuyên nên được lưu trữ trong bộ nhớ đệm để giảm thiểu việc truy cập bộ nhớ chính.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

void accessArrayCache(int arr[], int n) {
    for (int i = 0; i < n; i++) {  // Truy cập tuần tự
        arr[i] *= 2;
    }
}

int main() {
    const int size = 100000;
    int arr[size];
    accessArrayCache(arr, size);
    return 0;
}
```

Trong ví dụ trên, việc truy cập mảng arr theo thứ tự tuần tự giúp tối ưu hóa việc sử dụng bộ nhớ đệm, do CPU có thể tải trước các khối dữ liệu từ bộ nhớ chính vào bộ nhớ đệm, giảm thiểu thời gian chờ.

### 4.9.6. Tối Ưu Hóa Mã C/C++ Bằng Các Kỹ Thuật Bổ Sung

Để mở rộng hơn nữa về các kỹ thuật tối ưu hóa mã trong C/C++, dưới đây là các phương pháp bổ sung như đơn giản hóa biểu thức, tối ưu hóa các vòng lặp và các câu lệnh rẽ nhánh, sử dụng lính canh để giảm thiểu phép thử, cùng với các ví dụ minh họa cụ thể.

**Đơn giản hóa biểu thức (Expression Simplification):** Đơn giản hóa các biểu thức giúp giảm số lượng phép toán, từ đó giảm thời gian thực thi và tăng hiệu suất. Thay vì thực hiện các phép tính phức tạp, hãy cố gắng biểu diễn chúng dưới dạng các phép tính đơn giản hơn.

Ví dụ cụ thể: Tối ưu hóa các phép toán số học:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
double computeArea(double radius) {
    return 3.14159 * radius * radius; // Tính diện tích hình tròn
}

// Sau tối ưu hóa
double computeAreaOptimized(double radius) {
    const double pi = 3.14159; // Sử dụng hằng số
    return pi * (radius * radius); // Giảm số lần nhân
}

int main() {
    cout << computeAreaOptimized(5.0) << endl; // Output: 78.5398
    return 0;
}
```

Trong ví dụ trên, việc sử dụng hằng số pi giúp tránh việc tính toán lại số pi nhiều lần. Ngoài ra, việc tính radius * radius một lần giúp tối ưu hóa số lần phép nhân.

**Đưa vòng lặp ngắn ra ngoài (Short Loops First):** Nếu một vòng lặp ngắn được thực thi nhiều lần hơn một vòng lặp dài, hãy đưa vòng lặp ngắn ra ngoài để giảm thời gian khởi tạo và điều khiển vòng lặp.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa
void nestedLoops(int outer, int inner) {
    for (int i = 0; i < outer; i++) {
        for (int j = 0; j < inner; j++) {
            // Xử lý...
        }
    }
}

// Sau tối ưu hóa
void optimizedLoops(int outer, int inner) {
    for (int j = 0; j < inner; j++) {
        for (int i = 0; i < outer; i++) {
            // Xử lý...
        }
    }
}

int main() {
    optimizedLoops(1000, 10);
    return 0;
}
```

Trong ví dụ này, vòng lặp ngắn hơn (inner có giá trị nhỏ hơn) được đưa ra ngoài, điều này giúp giảm chi phí điều khiển vòng lặp vì vòng lặp ngoài chỉ cần khởi tạo và kiểm tra một số lần ít hơn.

**Đưa các trường hợp thường gặp lên trên trong các câu lệnh switch hoặc rẽ nhánh:** Trong các câu lệnh switch hoặc if-else, đặt các trường hợp thường gặp lên đầu để chúng được kiểm tra đầu tiên, từ đó giảm thiểu thời gian kiểm tra khi chương trình chạy.

Ví dụ cụ thể:

```cpp
#include <iostream>
using namespace std;

// Trước tối ưu hóa
void processChar(char c) {
    switch (c) {
        case 'z':
            cout << "Case z" << endl;
            break;
        case 'a':
            cout << "Case a" << endl;
            break;
        case 'e':
            cout << "Case e" << endl;
            break;
        default:
            cout << "Default case" << endl;
            break;
    }
}

// Sau tối ưu hóa
void optimizedProcessChar(char c) {
    switch (c) {
        case 'a': // Đưa trường hợp thường gặp lên trên
            cout << "Case a" << endl;
            break;
        case 'e':
            cout << "Case e" << endl;
            break;
        case 'z':
            cout << "Case z" << endl;
            break;
        default:
            cout << "Default case" << endl;
            break;
    }
```

```
}

int main() {
    optimizedProcessChar('a');  // Output: Case a
    return 0;
}
```

Trong ví dụ trên, ký tự 'a' được kiểm tra đầu tiên vì nó thường gặp nhất, giúp giảm thiểu thời gian xử lý khi chương trình gặp phải các trường hợp này.

**Sử dụng lính canh (Sentinels) để giảm bớt các phép thử:** Sử dụng lính canh (sentinel) là một kỹ thuật để giảm số lượng phép thử trong vòng lặp, đặc biệt là trong các thuật toán tìm kiếm hoặc xử lý dữ liệu. Lính canh là một giá trị đặc biệt được đặt tại vị trí cuối cùng của mảng hoặc cấu trúc dữ liệu để đánh dấu điểm kết thúc hoặc điều kiện dừng.

Ví dụ cụ thể: Tìm kiếm tuyến tính với lính canh:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa: tìm kiếm tuyến tính thông thường
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i;
    }
    return -1; // Không tìm thấy
}

// Sau tối ưu hóa: tìm kiếm tuyến tính với lính canh
int linearSearchWithSentinel(int arr[], int n, int key) {
    int last = arr[n - 1]; // Lưu lại giá trị cuối cùng
    arr[n - 1] = key; // Đặt lính canh

    int i = 0;
    while (arr[i] != key) {
        i++;
    }
```

```
    arr[n - 1] = last; // Khôi phục lại giá trị cuối cùng

    if (i < n - 1 || arr[n - 1] == key) return i;
    return -1; // Không tìm thấy
}

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 7;

    cout << "Found at index: " << linearSearchWithSentinel(arr, n, key) << endl;
    return 0;
}
```

Trong ví dụ trên, thay vì kiểm tra xem có vượt quá giới hạn mảng trong mỗi vòng lặp, chúng ta chỉ cần một phép kiểm tra điều kiện duy nhất (arr[i] != key) và dùng lính canh để đảm bảo thuật toán không vượt quá giới hạn mảng.

**Tối ưu hóa bộ nhớ (Memory Optimization):** Sử dụng các kỹ thuật tối ưu hóa bộ nhớ để giảm thiểu việc sử dụng bộ nhớ và tăng hiệu quả truy xuất dữ liệu.

Ví dụ cụ thể: Sử dụng mảng tĩnh thay vì mảng động nếu kích thước đã biết:

```
#include <iostream>
using namespace std;

// Trước tối ưu hóa: sử dụng mảng động
void dynamicArrayExample(int size) {
    int* arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = i * 2;
    }
    delete[] arr;
}

// Sau tối ưu hóa: sử dụng mảng tĩnh
```

```
void staticArrayExample() {
    const int size = 100;
    int arr[size];
    for (int i = 0; i < size; i++) {
        arr[i] = i * 2;
    }
}

int main() {
    staticArrayExample();
    return 0;
}
```

Trong ví dụ trên, sử dụng mảng tĩnh giúp tránh việc cấp phát và giải phóng bộ nhớ động, cải thiện hiệu suất và giảm nguy cơ rò rỉ bộ nhớ.

#### 4.9.7. Kết luận

Tối ưu hóa mã C/C++ là một quá trình phức tạp, đòi hỏi sự hiểu biết sâu sắc về ngôn ngữ, cấu trúc dữ liệu, thuật toán và kiến trúc phần cứng. Việc áp dụng đúng các kỹ thuật tối ưu hóa có thể giúp chương trình chạy nhanh hơn, sử dụng ít tài nguyên hơn và dễ bảo trì hơn. Các kỹ thuật tối ưu hóa mã C/C++ không chỉ cải thiện hiệu suất mà còn giúp mã nguồn trở nên rõ ràng và dễ duy trì hơn. Bằng cách đơn giản hóa biểu thức, tối ưu hóa vòng lặp và rẽ nhánh, sử dụng lính canh, và tối ưu hóa bộ nhớ, lập trình viên có thể viết mã hiệu quả và đáng tin cậy hơn. Tuy nhiên, việc tối ưu hóa luôn cần sự cân nhắc và đo lường để đảm bảo rằng nó thực sự mang lại lợi ích mà không làm giảm chất lượng mã nguồn.

### 4.10. Từ Khóa const Trong C++

#### 4.10.1. Giới thiệu về const

Từ khóa const trong C++ được sử dụng để chỉ định rằng một biến, tham số, hàm, hoặc thành viên của lớp không thể thay đổi sau khi được khởi tạo. Điều này giúp bảo vệ dữ liệu, ngăn ngừa các thay đổi không mong muốn, và có thể cải thiện hiệu suất do trình biên dịch có thể thực hiện các tối ưu hóa bổ sung.

#### 4.10.2. Các Ứng Dụng của const

**Biến const:** Biến được khai báo với từ khóa const sẽ trở thành bất biến, nghĩa là giá trị của nó không thể thay đổi sau khi đã được khởi tạo.

Cú pháp: `const int maxCount = 100;`

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

int main() {
    const int maxCount = 100; // Khai báo biến const
    cout << "Max count is: " << maxCount << endl;

    // maxCount = 200; // Lỗi biên dịch: Không thể thay đổi giá trị của biến cons

    return 0;
}
```

**Con trỏ const:** Con trỏ trong C++ có thể được kết hợp với từ khóa const theo nhiều cách khác nhau:

Con trỏ tới const (Pointer to const): Con trỏ trỏ tới một giá trị không thể thay đổi. Điều này nghĩa là bạn không thể sử dụng con trỏ này để thay đổi giá trị mà nó trỏ tới. Cú pháp: `const int* ptr;`

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

int main() {
    int value = 10;
    const int* ptr = &value; // Con trỏ tới const int

    cout << "Value: " << *ptr << endl;

    // *ptr = 20; // Lỗi biên dịch: Không thể thay đổi giá trị thông qua con trỏ

    value = 20; // Giá trị của biến gốc có thể thay đổi trực tiếp
    cout << "Updated value: " << *ptr << endl; // Output: Updated value: 20

    return 0;
}
```

Con trỏ const (const Pointer): Con trỏ không thể trỏ tới địa chỉ khác sau khi được khởi tạo, nhưng giá trị tại địa chỉ đó có thể thay đổi. Cú pháp: `int* const ptr;`

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

int main() {
    int value1 = 10;
    int value2 = 20;
    int* const ptr = &value1; // Con trỏ không thể trỏ tới địa chỉ khác

    cout << "Value pointed to by ptr: " << *ptr << endl;

    *ptr = 30; // Giá trị tại địa chỉ mà ptr trỏ tới có thể thay đổi
    cout << "Updated value1: " << *ptr << endl; // Output: Updated value1: 30

    // ptr = &value2; // Lỗi biên dịch: Không thể thay đổi địa chỉ mà con trỏ const trỏ t

    return 0;
}
```

Con trỏ const tới const (const Pointer to const): Con trỏ không thể trỏ tới địa chỉ khác, và giá trị tại địa chỉ đó cũng không thể thay đổi. Cú pháp: `const int* const ptr;`

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

int main() {
    int value = 10;
    const int* const ptr = &value; // Con trỏ const tới giá trị const

    cout << "Value: " << *ptr << endl;

    // *ptr = 20; // Lỗi biên dịch: Không thể thay đổi giá trị tại địa chỉ mà ptr trỏ tới
```

```
    // ptr = &value; // Lỗi biên dịch: Không thể thay đổi địa chỉ mà con trỏ trỏ

    return 0;
}
```

**Tham số hàm const:** Khi truyền tham số cho hàm bằng tham chiếu hoặc con trỏ, việc sử dụng const giúp ngăn chặn việc hàm thay đổi giá trị của đối số thực sự.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

void printValue(const int& value) { // Tham số là tham chiếu const
    cout << "Value: " << value << endl;
    // value = 100; // Lỗi biên dịch: Không thể thay đổi giá trị của tham chiếu c
}

int main() {
    int number = 42;
    printValue(number);
    return 0;
}
```

Trong ví dụ này, tham số value là một tham chiếu const, nghĩa là hàm printValue không thể thay đổi giá trị của number.

**Hàm thành viên const trong lớp:** Hàm thành viên const trong lớp không thể thay đổi bất kỳ thành viên dữ liệu nào của lớp và không thể gọi các hàm thành viên không phải const. Cú pháp: Đặt từ khóa const sau định nghĩa hàm.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

class Example {
private:
    int data;
```

```
public:
    Example(int d) : data(d) {}

    int getData() const { // Hàm thành viên const
        return data;
    }

    void setData(int d) {
        data = d;
    }
};

int main() {
    Example ex(10);
    cout << "Data: " << ex.getData() << endl; // Output: Data: 10

    // ex.getData() = 20; // Lỗi biên dịch: Hàm thành viên const không thể được dùng để t

    ex.setData(20); // Hàm thành viên không phải const có thể thay đổi giá trị
    cout << "Updated Data: " << ex.getData() << endl; // Output: Updated Data: 20

    return 0;
}
```

Trong ví dụ trên, getData() là một hàm thành viên const, vì vậy nó không thể thay đổi bất kỳ thành viên nào của lớp. Ngược lại, setData() không phải là hàm const và có thể thay đổi giá trị của data.

**Lợi ích của việc sử dụng const:**

- Tăng tính an toàn của mã nguồn: Việc sử dụng const giúp ngăn ngừa các thay đổi không mong muốn và lỗi logic trong chương trình.
- Cải thiện tối ưu hóa của trình biên dịch: Trình biên dịch có thể thực hiện các tối ưu hóa bổ sung khi biết rằng giá trị không thay đổi.
- Dễ dàng đọc và bảo trì mã nguồn: Việc sử dụng const giúp chỉ rõ ý định của lập trình viên, rằng giá trị sẽ không thay đổi, làm cho mã dễ hiểu và dễ bảo trì hơn.

**Khi nào nên sử dụng const?**

- Khi khai báo hằng số: Sử dụng const để định nghĩa các giá trị không thay đổi trong suốt quá trình thực thi chương trình.
- Khi truyền tham số cho hàm bằng tham chiếu hoặc con trỏ: Sử dụng const để bảo vệ dữ liệu đầu vào không bị thay đổi bởi hàm.
- Trong các hàm thành viên không cần thay đổi dữ liệu thành viên: Sử dụng const cho các hàm thành viên mà không cần thay đổi bất kỳ dữ liệu thành viên nào của đối tượng.
- Khi làm việc với con trỏ: Sử dụng const để bảo vệ giá trị được trỏ tới hoặc địa chỉ mà con trỏ trỏ tới.

**Kết luận** Từ khóa const là một công cụ mạnh mẽ và linh hoạt trong C++ để bảo vệ dữ liệu, cải thiện hiệu suất và tăng cường tính an toàn của mã nguồn. Sử dụng const đúng cách sẽ giúp mã nguồn rõ ràng, dễ bảo trì và tránh các lỗi tiềm ẩn. Tuy nhiên, cần lưu ý sử dụng const một cách thích hợp để không làm giảm tính linh hoạt của mã.

## 4.11. Kết luận

Viết mã hiệu quả là một phần quan trọng trong quá trình phát triển phần mềm. Bằng cách áp dụng các nguyên tắc và quy tắc tối ưu hóa một cách có hệ thống, lập trình viên có thể cải thiện hiệu suất của chương trình, giảm thiểu tiêu thụ tài nguyên, và tăng cường khả năng bảo trì mã. Đừng ngần ngại thử nghiệm và đo lường tác động của các thay đổi tối ưu hóa để đạt được kết quả tốt nhất. Tuy nhiên, điều quan trọng là phải luôn đo lường hiệu quả của mỗi tối ưu hóa để đảm bảo rằng nó thực sự mang lại lợi ích.

## 4.12. Bổ sung về Macro

Macro là một trong những tính năng mạnh mẽ và nguy hiểm trong C/C++, cho phép lập trình viên thực hiện nhiều tác vụ đa dạng, từ định nghĩa hằng số, viết hàm inline đến việc tạo ra mã nguồn linh hoạt và phức tạp. Tuy nhiên, việc sử dụng macro cũng tiềm ẩn nhiều vấn đề khó lường. Dưới đây là một nội dung bổ sung về kỹ thuật macro trong C/C++, những vấn đề thường gặp khi làm việc với macro, và khi nào nên hoặc không nên sử dụng macro.

### 4.12.1. Giới thiệu về Macro

Macro trong C là các định nghĩa được xử lý bởi bộ tiền xử lý (preprocessor) trước khi mã nguồn được biên dịch. Macro có thể được sử dụng để định nghĩa các hằng số, hàm đơn giản, hoặc thậm chí là các đoạn mã phức tạp. Cú pháp cơ bản của macro:

**Macro đơn giản (Hằng số):**

```
#define PI 3.14159
```

**Macro hàm:**

```
#define SQUARE(x) ((x) * (x))
```

### 4.12.2. Các loại Macro và Ví dụ

**Macro hằng số:** Thay thế một tên bằng một giá trị cố định. Macro hằng số thường được sử dụng để định nghĩa các giá trị không thay đổi.

Ví dụ cụ thể:

```
#include <stdio.h>

#define PI 3.14159
#define MAX_SIZE 100

int main() {
    printf("Value of PI: %f\n", PI);  // Output: Value of PI: 3.141590
    printf("Max size is: %d\n", MAX_SIZE);  // Output: Max size is: 100
    return 0;
}
```

**Macro hàm:** Macro hàm là một cách thay thế mã nguồn ngắn gọn cho các đoạn mã lặp lại. Nó cho phép truyền tham số giống như hàm, nhưng mã được "inline" thay vì gọi hàm thông thường, loại bỏ chi phí gọi hàm.

Ví dụ cụ thể:

```
#include <stdio.h>
```

```c
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    int num = 5;
    printf("Square of %d is %d\n", num, SQUARE(num));  // Output: Square of 5 is

    int a = 10, b = 20;
    printf("Max of %d and %d is %d\n", a, b, MAX(a, b));  // Output: Max of 10 a

    return 0;
}
```

### 4.12.3. Các Vấn Đề Thường Gặp Khi Làm Việc Với Macro:

Mặc dù macro cung cấp sự linh hoạt và tiết kiệm chi phí gọi hàm, chúng cũng có nhiều nhược điểm và có thể dẫn đến các lỗi khó phát hiện:

**Thiếu kiểm tra kiểu dữ liệu:** Macro không kiểm tra kiểu dữ liệu của các tham số truyền vào, dẫn đến các lỗi tiềm ẩn khi sử dụng với các kiểu dữ liệu không mong muốn.

Ví dụ vấn đề:

```c
#include <stdio.h>

#define SQUARE(x) ((x) * (x))

int main() {
    printf("Square of 2.5 is %d\n", SQUARE(2.5)); // Output không như mong muốn:
    return 0;
}
```

**Vấn đề với dấu ngoặc đơn và độ ưu tiên toán tử:** Macro có thể gặp vấn đề khi sử dụng với các biểu thức phức tạp do thiếu dấu ngoặc đơn bảo vệ hoặc sự thay đổi ưu tiên toán tử.

Ví dụ vấn đề:

```c
#include <stdio.h>
```

```c
#define SQUARE(x) (x * x)  // Thiếu dấu ngoặc kép xung quanh x * x

int main() {
    int result = SQUARE(1 + 2);  // Output: 1 + 2 * 1 + 2 = 1 + 2 + 2 = 5, thay vì 9 như
    printf("Result: %d\n", result); // Output không như mong muốn
    return 0;
}
```

Cách khắc phục: Sử dụng dấu ngoặc đơn xung quanh toàn bộ macro và các tham số truyền vào:

```c
#define SQUARE(x) ((x) * (x))
```

### 4.12.4. Vấn đề với các tác dụng phụ (side effects)

Nếu một tham số macro có tác dụng phụ (như tăng hoặc giảm giá trị), các tác dụng phụ này có thể xảy ra nhiều lần khi sử dụng trong macro.

Ví dụ vấn đề:

```c
#include <stdio.h>

#define INCREMENT(x) ((x)++)

int main() {
    int a = 5;
    int result = INCREMENT(a) * INCREMENT(a);  // a sẽ tăng hai lần thay vì một lần
    printf("Result: %d, a: %d\n", result, a); // Output không như mong muốn
    return 0;
}
```

Cách khắc phục: Sử dụng hàm thay vì macro để tránh tác dụng phụ:

```c
int increment(int x) {
    return x + 1;
}
```

### 4.12.5. Macro và gỡ lỗi (debugging)

Macro không tạo ra mã máy trực tiếp mà thay thế văn bản trước khi biên dịch. Điều này khiến việc gỡ lỗi macro trở nên khó khăn hơn vì không thể theo dõi mã nguồn gốc dễ dàng trong quá trình debug.

### 4.12.6. Khi nào nên dùng macro?

Macro có thể hữu ích trong một số trường hợp:

- Định nghĩa hằng số: Sử dụng macro để định nghĩa các hằng số giá trị không đổi trong suốt chương trình.
- Tạo hàm inline đơn giản: Khi cần một "hàm" nhỏ được inline để tối ưu hóa hiệu suất, ví dụ như các phép toán đơn giản hoặc các phép toán logic.
- Điều kiện biên dịch: Sử dụng macro để bao bọc mã chỉ chạy trên các nền tảng hoặc cấu hình nhất định.

```
#ifdef DEBUG
#define LOG(msg) printf("DEBUG: %s\n", msg)
#else
#define LOG(msg)
#endif
```

### 4.12.7. Khi nào nên tránh dùng macro?

Macro nên được tránh trong các trường hợp sau:

- Khi có thể sử dụng const, enum, hoặc inline: Những từ khóa này cung cấp tính an toàn kiểu và kiểm tra lỗi tốt hơn.
- Khi macro quá phức tạp hoặc có khả năng gây hiểu nhầm: Nếu macro quá phức tạp, nó sẽ khó đọc và khó bảo trì.
- Khi macro có thể gây ra tác dụng phụ không mong muốn: Tránh sử dụng macro khi có nguy cơ các tác dụng phụ không mong muốn xảy ra, đặc biệt là khi truyền vào các tham số có thể thay đổi.

### 4.12.8. Thay thế macro bằng các tính năng an toàn hơn trong C++

Trong C++, nhiều trường hợp sử dụng macro có thể được thay thế bằng các tính năng ngôn ngữ an toàn hơn, chẳng hạn như:

`const` và `constexpr`: Để định nghĩa các hằng số.

```
constexpr double PI = 3.14159;
inline function: Để thay thế macro hàm nhằm đảm bảo kiểm tra kiểu và tránh tác dụ
```

```
inline int square(int x) {
    return x * x;
}
```

### 4.12.9. Kết luận

Macro là một công cụ mạnh mẽ trong C/C++, nhưng cũng tiềm ẩn nhiều nguy hiểm nếu không sử dụng đúng cách. Chúng nên được sử dụng một cách cẩn thận, và nên tránh khi có thể thay thế bằng các tính năng ngôn ngữ an toàn hơn như const, inline, hoặc các hàm thông thường. Khi sử dụng macro, luôn cần đảm bảo rằng chúng rõ ràng, dễ hiểu, và không gây ra các tác dụng phụ không mong muốn.

## 4.13. Hành vi không xác định

Hành vi không xác định (Undefined Behavior - UB) là một trong những khái niệm quan trọng trong lập trình C/C++, thường dẫn đến những kết quả không mong muốn và lỗi khó tìm. Đây là một trong những lý do chính khiến các chương trình C/C++ dễ bị lỗi và hành xử không nhất quán trên các trình biên dịch khác nhau.

Dưới đây là một vài nội dung về hành vi không xác định trong C/C++, các ví dụ minh họa cụ thể, và lý do tại sao nó có thể dẫn đến những kết quả khác nhau trên mỗi trình biên dịch.

### 4.13.1. Giới thiệu về Hành Vi Không Xác Định

Hành vi không xác định trong C/C++ là kết quả của việc thực hiện các thao tác không được chuẩn ngôn ngữ định nghĩa. Khi một chương trình gặp phải UB, mọi thứ đều có thể xảy ra: chương trình có thể bị lỗi, treo, tạo ra kết quả không mong đợi, hoặc thậm chí hoạt động "bình thường" trên một số trình biên dịch nhưng không hoạt động trên những trình biên dịch khác.

Lý do hành vi không xác định tồn tại: C/C++ được thiết kế để cung cấp hiệu suất cao và khả năng truy cập phần cứng trực tiếp. Không xử lý các lỗi cụ thể giúp trình biên dịch tối ưu hóa mã tốt hơn, nhưng đổi lại là khả năng xảy ra UB nếu lập trình viên không cẩn thận.

### 4.13.2. Các Ví Dụ về Hành Vi Không Xác Định và Giải Thích

**Truy cập bộ nhớ không hợp lệ:** Ví dụ: Truy cập con trỏ null - Khi một con trỏ null được truy cập (ví dụ: dereference), chương trình sẽ gặp phải hành vi không xác định.

```cpp
#include <iostream>
using namespace std;

int main() {
    int *ptr = nullptr;  // Con trỏ null
    cout << *ptr << endl;  // UB: Dereference con trỏ null
    return 0;
}
```

Giải thích: Theo chuẩn C++, dereference một con trỏ null không được định nghĩa hành vi. Một số trình biên dịch có thể cảnh báo, nhưng nhiều trình biên dịch không. Trình biên dịch có thể tạo mã gây lỗi tại thời điểm chạy hoặc thậm chí không báo lỗi, dẫn đến crash hoặc kết quả không xác định.

**Phép toán trên biến chưa được khởi tạo:** Ví dụ: Sử dụng biến chưa khởi tạo - Khi một biến chưa được khởi tạo được sử dụng, giá trị của nó là không xác định, dẫn đến UB.

```cpp
#include <iostream>
using namespace std;

int main() {
    int a;  // Biến chưa khởi tạo
    cout << a << endl;  // UB: Sử dụng biến chưa khởi tạo
    return 0;
}
```

Giải thích: Chuẩn C++ không định nghĩa giá trị của biến chưa khởi tạo, và trình biên dịch không bắt buộc phải cung cấp một giá trị mặc định. Trình biên dịch khác nhau có thể gán giá trị khác nhau cho a, dẫn đến kết quả không thể dự đoán được.

**Tràn số nguyên (Integer Overflow):** Ví dụ: Tràn số nguyên khi cộng hai số lớn - Khi một phép tính số nguyên dẫn đến tràn (vượt quá giới hạn của kiểu dữ liệu), nó dẫn đến hành vi không xác định.

```cpp
#include <iostream>
using namespace std;
```

```cpp
int main() {
    int max = INT_MAX;  // Giá trị lớn nhất của int
    int overflow = max + 1;  // UB: Tràn số nguyên
    cout << "Overflow result: " << overflow << endl;  // Kết quả không xác định
    return 0;
}
```

Giải thích: Chuẩn C++ không định nghĩa hành vi khi số nguyên bị tràn. Một số trình biên dịch có thể quay vòng (wrap around), trong khi số khác có thể gây ra lỗi runtime. Kết quả của phép cộng max + 1 có thể là giá trị âm, số nhỏ, hoặc kết quả bất kỳ tùy thuộc vào trình biên dịch.

**Phép toán với các con trỏ không hợp lệ:** Ví dụ: Con trỏ vượt quá phạm vi mảng - Khi một con trỏ trỏ đến địa chỉ ngoài phạm vi của mảng, thao tác dereference sẽ dẫn đến UB.

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[3] = {1, 2, 3};
    int *ptr = arr + 5;  // Con trỏ vượt quá phạm vi mảng
    cout << *ptr << endl;  // UB: Dereference con trỏ không hợp lệ
    return 0;
}
```

Giải thích: Con trỏ ptr trỏ đến địa chỉ không hợp lệ, và dereference nó là hành vi không xác định. Trình biên dịch khác nhau có thể xử lý trường hợp này theo cách khác nhau: có thể in giá trị rác, treo chương trình, hoặc gây lỗi runtime.

**Biến đồng thời truy cập (Data Race):** Ví dụ: Biến chia sẻ trong các luồng mà không đồng bộ - Khi hai luồng truy cập cùng một biến chia sẻ mà không có cơ chế đồng bộ, kết quả sẽ không thể dự đoán được.

```cpp
#include <iostream>
#include <thread>
using namespace std;
```

```
int counter = 0;  // Biến chia sẻ

void increment() {
    for (int i = 0; i < 1000; i++) {
        counter++;  // UB: Data race
    }
}

int main() {
    thread t1(increment);
    thread t2(increment);
    t1.join();
    t2.join();
    cout << "Final counter value: " << counter << endl;  // Kết quả không xác đị
    return 0;
}
```

Giải thích: Hai luồng t1 và t2 cùng tăng giá trị của counter, dẫn đến data race vì không có đồng bộ. Giá trị cuối cùng của counter là không xác định và có thể khác nhau mỗi lần chạy.

**Chuuyển (Reinterpretation) kiểu dữ liệu không tương thích:** Ví dụ: Reinterpret cast không an toàn - Sử dụng reinterpret_cast để chuyển đổi kiểu dữ liệu không tương thích có thể dẫn đến UB.

```
#include <iostream>
using namespace std;

int main() {
    int x = 42;
    char* ptr = reinterpret_cast<char*>(&x);  // Chuyển đổi không an toàn
    cout << *ptr << endl;  // UB: Giá trị không xác định
    return 0;
}
```

Giải thích: reinterpret_cast cho phép chuyển đổi giữa các con trỏ kiểu khác nhau, nhưng kết quả là không xác định khi các kiểu này không tương thích. Trình biên dịch có thể tạo ra mã không hoạt động đúng hoặc dẫn đến lỗi runtime.

**Hành Vi Không Xác Định với các biểu thức có ++ và −−.** Các toán tử ++ và −− thường gây ra hành vi không xác định (Undefined Behavior - UB) trong C/C++ khi chúng được sử dụng trong các biểu thức phức tạp mà không tuân theo thứ tự đánh giá được định nghĩa rõ ràng. Điều này có thể dẫn đến các kết quả không nhất quán và khó dự đoán khi chương trình được biên dịch và chạy trên các trình biên dịch khác nhau. Ví dụ về Hành Vi Không Xác Định với Toán Tử ++ và −−

Ví dụ 1: Thứ tự đánh giá không xác định

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    int a = i++ + ++i; // UB: Thứ tự đánh giá của i++ và ++i không xác định
    cout << "Result: " << a << endl; // Kết quả không xác định
    return 0;
}
```

Giải thích: Trong biểu thức i++ + ++i, có hai toán tử ++ đang tác động lên i. Thứ tự đánh giá của các toán tử này không được chuẩn C++ xác định rõ. Một trình biên dịch có thể thực hiện i++ trước và sau đó ++i, trong khi trình biên dịch khác có thể làm ngược lại. Kết quả của biểu thức này hoàn toàn phụ thuộc vào cách trình biên dịch đánh giá, dẫn đến hành vi không xác định.

Ví dụ 2: Sử dụng ++ trong biểu thức gán

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    i = i++ + 1; // UB: Truy cập và thay đổi biến i trong cùng một biểu thức
    cout << "Result: " << i << endl; // Kết quả không xác định
    return 0;
}
```

Giải thích: Trong biểu thức i = i++ + 1, biến i vừa được truy cập vừa được thay đổi mà không có một thứ tự đánh giá rõ ràng. Chuẩn C++ không định nghĩa kết quả trong trường hợp này vì nó phụ thuộc vào việc i++ được thực hiện trước hay

sau khi cộng thêm 1. Kết quả có thể khác nhau trên các trình biên dịch khác nhau hoặc các phiên bản khác nhau của cùng một trình biên dịch.

Ví dụ 3: Truy cập cùng một biến nhiều lần với toán tử ++

```
#include <iostream>
using namespace std;

int main() {
    int i = 1;
    int arr[5] = {0};
    arr[i] = i++; // UB: Truy cập và thay đổi i trong cùng một biểu thức
    cout << "Result: " << arr[1] << endl; // Kết quả không xác định
    return 0;
}
```

Giải thích: Trong biểu thức arr[i] = i++, chỉ số mảng i được thay đổi bởi i++ trong khi đang được sử dụng để truy cập arr[i]. Thứ tự đánh giá của i và i++ không xác định, dẫn đến hành vi không xác định. Điều này có thể dẫn đến việc truy cập bộ nhớ không mong muốn hoặc lỗi runtime.

Ví dụ 4: Sử dụng toán tử – với nhiều tác động lên cùng một biến

```
#include <iostream>
using namespace std;

int main() {
    int j = 5;
    j = j-- - --j; // UB: Thứ tự đánh giá của j-- và --j không xác định
    cout << "Result: " << j << endl; // Kết quả không xác định
    return 0;
}
```

Giải thích: Trong biểu thức j = j−− - −−j, có hai thao tác giảm giá trị j (j−− và −−j) được sử dụng trong cùng một biểu thức. Việc thực hiện phép trừ và giảm giá trị không có thứ tự xác định, dẫn đến hành vi không xác định. Một số trình biên dịch có thể thực hiện j−− trước và sau đó −−j, trong khi trình biên dịch khác có thể làm ngược lại.

Ví dụ 5: Toán tử ++ sử dụng trên một mảng với chỉ số tăng dần

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[5] = {0, 1, 2, 3, 4};
    int i = 2;
    arr[i] = arr[++i]; // UB: Thứ tự đánh giá của ++i không xác định
    cout << "arr[2]: " << arr[2] << endl; // Kết quả không xác định
    return 0;
}
```

Giải thích: Trong biểu thức arr[i] = arr[++i];, chỉ số mảng i được thay đổi bởi ++i trong khi đang được sử dụng để truy cập arr[i]. Thứ tự đánh giá của ++i không xác định, dẫn đến việc truy cập bộ nhớ không mong muốn hoặc lỗi runtime. Kết quả có thể khác nhau giữa các trình biên dịch hoặc thậm chí giữa các lần chạy trên cùng một trình biên dịch.

Tại Sao Những Trường Hợp Này Dẫn Đến Hành Vi Không Xác Định?

- Thứ tự đánh giá không được chuẩn hóa: C++ không quy định thứ tự đánh giá các toán tử trong một biểu thức phức tạp, điều này dẫn đến UB khi một biến được thay đổi nhiều lần mà không có thứ tự rõ ràng.
- Trình biên dịch tự do tối ưu hóa: Trình biên dịch có quyền tối ưu hóa mã nguồn dựa trên giả định rằng UB sẽ không xảy ra, điều này có thể dẫn đến các hành vi không thể đoán trước được nếu UB xuất hiện.
- Truy cập đồng thời và thay đổi cùng một biến: Khi một biến được thay đổi và truy cập đồng thời mà không có thứ tự rõ ràng, kết quả của phép toán trở nên không xác định.
- C++ không quy định thứ tự đánh giá các toán tử trong một biểu thức phức tạp, điều này dẫn đến UB khi một biến được thay đổi nhiều lần mà không có thứ tự rõ ràng.
- Trình biên dịch tự do tối ưu hóa: Trình biên dịch có quyền tối ưu hóa mã nguồn dựa trên giả định rằng UB sẽ không xảy ra, điều này có thể dẫn đến các hành vi không thể đoán trước được nếu UB xuất hiện.
- Truy cập đồng thời và thay đổi cùng một biến: Khi một biến được thay đổi và truy cập đồng thời mà không có thứ tự rõ ràng, kết quả của phép toán trở nên không xác định.

- Thiết kế ngôn ngữ để tối ưu hóa hiệu suất: C++ được thiết kế để tối ưu hóa hiệu suất, và chuẩn C++ không yêu cầu xử lý hoặc báo lỗi cho hành vi không xác định để cho phép các tối ưu hóa tốt hơn.

Hành vi không xác định là một khía cạnh quan trọng nhưng khó quản lý của C/C++. Việc sử dụng các toán tử ++ và – trong các biểu thức phức tạp thường dẫn đến UB, gây ra kết quả không nhất quán và khó dự đoán. Hiểu rõ về UB và tuân thủ các quy tắc lập trình tốt có thể giúp tránh được những lỗi này và viết mã an toàn hơn.

### 4.13.3. Tại Sao Các Trình Biên Dịch Xử Lý UB Khác Nhau?

Lý do khiến các trình biên dịch xử lý hành vi không xác định khác nhau:

- Tối ưu hóa: Trình biên dịch có thể tối ưu hóa mã dựa trên giả định rằng hành vi không xác định sẽ không bao giờ xảy ra. Điều này có thể dẫn đến các kết quả không mong đợi nếu UB xảy ra.
- Kiến trúc phần cứng: Hành vi không xác định có thể phụ thuộc vào kiến trúc phần cứng, nơi mà chương trình đang chạy. Ví dụ: xử lý tràn số nguyên có thể khác nhau giữa các kiến trúc x86 và ARM.
- Chính sách xử lý lỗi: Một số trình biên dịch có thể cố gắng phát hiện UB và gây ra lỗi tại runtime, trong khi các trình biên dịch khác có thể bỏ qua hoặc không phát hiện được.
- Kiểm tra an toàn bộ nhớ: Một số trình biên dịch hỗ trợ các kiểm tra an toàn bộ nhớ (như AddressSanitizer trong GCC và Clang), giúp phát hiện UB liên quan đến truy cập bộ nhớ.

### 4.13.4. Làm Thế Nào Để Tránh Hành Vi Không Xác Định?

- Khởi tạo tất cả các biến: Đảm bảo tất cả các biến được khởi tạo trước khi sử dụng.
- Kiểm tra biên (bounds) của mảng: Luôn kiểm tra chỉ số trước khi truy cập phần tử mảng.
- Đồng bộ hóa truy cập biến chia sẻ: Sử dụng các cơ chế đồng bộ (mutex, atomic) khi truy cập các biến chia sẻ trong các chương trình đa luồng.
- Tránh sử dụng reinterpret_cast khi không cần thiết: Nếu cần chuyển đổi giữa các kiểu con trỏ, hãy sử dụng static_cast hoặc dynamic_cast khi phù hợp.

- Sử dụng các công cụ phát hiện lỗi: Sử dụng các công cụ như Valgrind, AddressSanitizer để phát hiện UB trong quá trình phát triển.

Hành vi không xác định là một phần nguy hiểm và khó quản lý của C/C++, nhưng hiểu rõ về nó sẽ giúp lập trình viên viết mã an toàn và đáng tin cậy hơn. Tránh UB bằng cách tuân thủ các quy tắc lập trình tốt và sử dụng các công cụ hỗ trợ phát hiện lỗi là rất quan trọng để đảm bảo chất lượng mã nguồn.

## 4.14. Hiệu ứng phụ - Side effect

Side effect (tác dụng phụ) trong lập trình là một khái niệm quan trọng mà lập trình viên cần hiểu rõ để viết mã sạch và tránh lỗi. Hiệu ứng phụ đề cập đến bất kỳ thay đổi trạng thái nào xảy ra bên ngoài phạm vi của hàm hoặc biểu thức, chẳng hạn như thay đổi giá trị của biến toàn cục, đọc/ghi dữ liệu vào tệp, hoặc thực hiện các phép tính có ảnh hưởng đến các phần khác của chương trình.

### 4.14.1. Giới thiệu về Tác Dụng Phụ.

Hiệu ứng phụ xảy ra khi một hàm hoặc biểu thức thay đổi trạng thái của chương trình hoặc hệ thống mà không phải là mục đích chính của hàm đó. Các tác dụng phụ có thể làm mã khó đọc và khó bảo trì hơn, vì chúng làm cho hành vi của chương trình không thể dự đoán và gây ra lỗi khó tìm.

### 4.14.2. Ví dụ về Hiệu ứng Phụ (Side Effect) trong C++

**Thay đổi giá trị biến toàn cục:** Ví dụ:

```cpp
#include <iostream>
using namespace std;

int globalCounter = 0; // Biến toàn cục

void incrementCounter() {
    globalCounter++; // Tác dụng phụ: thay đổi biến toàn cục
}

int main() {
    cout << "Before: " << globalCounter << endl; // Output: Before: 0
    incrementCounter();
    cout << "After: " << globalCounter << endl;  // Output: After: 1
```

```
    return 0;
}
```

Giải thích: Hàm incrementCounter() thay đổi giá trị của biến toàn cục globalCounter. Điều này có thể gây ra cácHiệu ứng phụ ngoài dự kiến khi hàm được gọi từ nhiều nơi khác nhau trong chương trình. Nếu nhiều hàm cùng thay đổi globalCounter, rất khó để theo dõi và kiểm soát trạng thái của nó, dẫn đến lỗi không mong muốn.

**Ghi vào tệp hoặc thiết bị đầu ra** Ví dụ:

```
#include <iostream>
#include <fstream>
using namespace std;

void writeToFile() {
    ofstream outFile("example.txt");
    outFile << "Hello, world!" << endl; // Hiệu ứng phụ: ghi dữ liệu vào tệp
    outFile.close();
}

int main() {
    writeToFile(); // Gọi hàm có Hiệu ứng phụ
    return 0;
}
```

Giải thích: Hàm writeToFile() có Hiệu ứng phụ là ghi dữ liệu vào tệp "example.txt". Hiệu ứng phụ này có thể gây ra lỗi nếu không kiểm tra điều kiện tệp hoặc xử lý ngoại lệ một cách chính xác.

**Thay đổi giá trị tham số truyền vào:** Ví dụ:

```
#include <iostream>
using namespace std;

void doubleValue(int& x) {
    x *= 2; // Hiệu ứng phụ: thay đổi giá trị tham số
}
```

```
int main() {
    int a = 5;
    cout << "Before: " << a << endl; // Output: Before: 5
    doubleValue(a); // Gọi hàm có tác dụng phụ
    cout << "After: " << a << endl;  // Output: After: 10
    return 0;
}
```

Giải thích: Hàm doubleValue(int& x) có tác dụng phụ là thay đổi giá trị của biến a được truyền vào theo tham chiếu. Mặc dù đôi khi việc thay đổi tham số là mục đích chính, nhưng nó cũng có thể dẫn đến lỗi nếu không được quản lý cẩn thận.

**Phép toán có Hiệu ứng phụ với các toán tử ++ và −−** Ví dụ:

```
#include <iostream>
using namespace std;

int main() {
    int a = 5;
    int b = ++a + a--; // Hiệu ứngphụ: thay đổi giá trị của a trong khi sử dụng nó
    cout << "Result: " << b << endl; // Kết quả không dự đoán được
    cout << "Final value of a: " << a << endl; // Kết quả cuối cùng của a không rõ ràng
    return 0;
}
```

Giải thích: Biểu thức ++a + a−− có Hiệu ứng phụ vì nó thay đổi giá trị của a trong khi đồng thời sử dụng giá trị của a. Điều này dẫn đến hành vi không xác định vì thứ tự thực hiện các phép toán này không được định nghĩa rõ ràng.

### 4.14.3. Vấn đề gặp phải với Hiệu ứng Phụ

**Gây ra lỗi không mong muốn:** Khó theo dõi và kiểm soát: Khi một hàm hoặc biểu thức thay đổi trạng thái toàn cục hoặc trạng thái bên ngoài phạm vi của nó, việc theo dõi và kiểm soát hành vi của chương trình trở nên khó khăn hơn. Khó bảo trì: Mã có Hiệu ứng phụ thường khó bảo trì vì các lập trình viên khác hoặc chính người viết mã không thể dự đoán đầy đủ hành vi của chương trình dựa trên mã nguồn.

**Làm phức tạp việc gỡ lỗi:** Gỡ lỗi phức tạp hơn: Các lỗi do tác dụng phụ thường khó gỡ lỗi hơn vì chúng không xuất hiện ngay lập tức và có thể phụ thuộc vào điều kiện cụ thể của chương trình. Lỗi không tái lập: Tác dụng phụ có thể dẫn đến lỗi không tái lập được, làm cho việc tái sản xuất lỗi và sửa lỗi trở nên khó khăn hơn.

**Ảnh hưởng đến tính toán đồng thời (Concurrency Issues):** Điều kiện tranh chấp (Race conditions): Trong các chương trình đa luồng, các tác dụng phụ có thể dẫn đến điều kiện tranh chấp, gây ra hành vi không dự đoán được. Tính không tương thích và lỗi thời gian: Việc sử dụng biến toàn cục hoặc tài nguyên chia sẻ mà không có đồng bộ hóa thích hợp có thể dẫn đến lỗi khó phát hiện và tái hiện.

### Khi Nào Nên Tránh Hiệu ứng Phụ?

- Khi viết hàm thuần (pure function): Hàm thuần không có Hiệu ứng phụ, tức là chúng không thay đổi trạng thái bên ngoài và luôn trả về cùng một kết quả với cùng một đầu vào. Hàm thuần dễ hiểu, dễ kiểm thử, và ít lỗi hơn.

  Ví dụ về hàm thuần:

  ```
  int add(int x, int y) {
      return x + y; // Không có Hiệu ứng phụ
  }
  ```

- Trong lập trình đa luồng: Tránh Hiệu ứng phụ khi làm việc với các tài nguyên chia sẻ trong các chương trình đa luồng. Sử dụng các cấu trúc đồng bộ như mutex, lock để bảo vệ tài nguyên chia sẻ.

- Khi mục đích không phải thay đổi trạng thái: Nếu mục đích của hàm chỉ là tính toán hoặc lấy dữ liệu mà không thay đổi trạng thái, tránh tác dụng phụ để làm cho hàm dễ hiểu và đáng tin cậy hơn.

### Khi Nào Hiệu ứng Phụ Có Thể Được Sử Dụng?

- Thao tác với I/O: Ghi vào tệp, in ra màn hình, hoặc đọc từ bàn phím là những Hiệu ứng phụ hợp lý vì chúng thay đổi trạng thái của hệ thống hoặc môi trường bên ngoài. Ví dụ:

  ```
  void logMessage(const string& message) {
      cout << message << endl; // Hiệu ứng phụ hợp lý
  }
  ```

- Cập nhật trạng thái chương trình: Khi cần thiết để cập nhật trạng thái của một đối tượng hoặc chương trình, các Hiệu ứng phụ có thể chấp nhận được. Ví dụ:

```
class Counter {
private:
    int count;
public:
    void increment() {
        count++; // Hiệu ứng phụ: thay đổi trạng thái của đối tượng
    }
};
```

### Cách Giảm Thiểu Hiệu ứng Phụ

- Sử dụng hàm thuần khi có thể: Viết hàm mà không thay đổi trạng thái bên ngoài để giảm thiểu lỗi và cải thiện khả năng kiểm thử.
- Sử dụng các công cụ quản lý bộ nhớ hiện đại: Như smart pointers (std::shared_ptr, std::unique_ptr) để quản lý tài nguyên và tránh rò rỉ bộ nhớ.
- Tách riêng các phần mã có tác dụng phụ: Đặt các phần mã có tác dụng phụ (như ghi vào tệp hoặc thay đổi biến toàn cục) vào một nơi riêng biệt hoặc trong các hàm cụ thể để làm cho mã dễ đọc hơn và dễ kiểm soát hơn.

**Ví dụ về Hiệu ứng Phụ với Vòng Lặp (Side Effect with Loops)** Ví dụ 1: Vòng lặp vô tận do Hiệu ứng phụ trong vòng lặp for

Mô tả: Trong ví dụ này, tác dụng phụ xảy ra khi biến được dùng làm điều kiện của vòng lặp bị thay đổi bên trong vòng lặp, gây ra vòng lặp vô tận.

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;

    // Vòng lặp với Hiệu ứng phụ trong điều kiện
    for (int j = 0; j < 5; j++) {
```

```
        cout << "i: " << i << ", j: " << j << endl;
        i++; // Tăng i trong vòng lặp

        // Hiệu ứng phụ: thay đổi giá trị j, làm điều kiện vòng lặp bị ảnh hưởng
        if (i == 2) {
            j = -1; // Đặt lại j = -1, vòng lặp quẩn
        }
    }

    return 0;
}
```

Giải thích: Vòng lặp for ban đầu dự kiến chạy từ j = 0 đến j < 5. Tuy nhiên, khi i == 2, câu lệnh j = -1 được thực thi, đặt lại giá trị của j. Điều này khiến vòng lặp trở lại trạng thái ban đầu (j = -1 sau đó tăng lên 0), dẫn đến vòng lặp vô tận.

Ví dụ 2: Vòng lặp vô tận do Hiệu ứng phụ với điều kiện thoát sai

Mô tả: Trong ví dụ này, vòng lặp while sử dụng một biến điều kiện thoát, nhưng biến này bị thay đổi bên trong vòng lặp một cách không kiểm soát, gây ra vòng lặp vô tận.

```
#include <iostream>
using namespace std;

int main() {
    int count = 0;
    bool condition = true;

    while (condition) {
        cout << "Count: " << count << endl;
        count++;

        // Hiệu ứng phụ: điều kiện thoát bị thay đổi nhưng không dừng vòng lặp
        if (count == 5) {
            condition = false;  // Tưởng rằng điều này sẽ thoát khỏi vòng lặp
        }

        // Hiệu ứng phụ không lường trước: điều kiện thoát lại bị đặt thành true
        if (count == 10) {
```

```
            condition = true;  // Sai lầm logic, vòng lặp sẽ tiếp tục vô tận
        }
    }

    return 0;
}
```

Giải thích: Vòng lặp while dự kiến dừng lại khi condition trở thành false. Tuy nhiên, sau khi count đạt giá trị 5, vòng lặp sẽ tiếp tục chạy và khi count đạt 10, condition lại được đặt thành true. Điều này dẫn đến vòng lặp vô tận vì điều kiện thoát liên tục bị tác động không chính xác bởi Hiệu ứng phụ.

Ví dụ 3: Vòng lặp vô tận do Hiệu ứng phụ trong biểu thức vòng lặp

Mô tả: Trong ví dụ này, vòng lặp for có một Hiệu ứng phụ trong biểu thức điều kiện khiến vòng lặp không bao giờ thoát.

```
#include <iostream>
using namespace std;

int main() {
    int x = 0;

    // Hiệu ứng phụ trong biểu thức điều kiện vòng lặp
    for (int i = 0; i < 10; ++i) {
        cout << "x: " << x << ", i: " << i << endl;
        x++;  // Tăng x trong vòng lặp

        // Hiệu ứng phụ: đặt lại i mỗi khi x = 3
        if (x == 3) {
            i = 0; // Reset i về 0, gây vòng lặp vô tận
        }
    }

    return 0;
}
```

Giải thích: Biểu thức $i < 10$ dự kiến sẽ giúp vòng lặp dừng lại khi i đạt giá trị 10. Tuy nhiên, khi $x == 3$, câu lệnh $i = 0$ được thực thi, đặt lại giá trị của i. Điều này

khiến vòng lặp không bao giờ đạt đến điều kiện thoát (i < 10), dẫn đến vòng lặp vô tận.

Ví dụ 4: Vòng lặp vô tận do thay đổi biến điều khiển bên trong vòng lặp

Mô tả: Trong ví dụ này, vòng lặp while có Hiệu ứng phụ thay đổi biến điều khiển bên trong vòng lặp, khiến vòng lặp không bao giờ thoát.

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;

    while (i < 5) {
        cout << "i: " << i << endl;
        i++;

        // Hiệu ứng phụ: giảm i xuống 0 khi i đạt giá trị 4
        if (i == 4) {
            i = 0; // Điều này làm cho vòng lặp trở nên vô tận
        }
    }

    return 0;
}
```

Giải thích: Vòng lặp while (i < 5) dự kiến sẽ dừng lại khi i đạt giá trị 5. Tuy nhiên, khi i == 4, câu lệnh i = 0 được thực thi, đặt lại giá trị của i về 0. Điều này khiến vòng lặp quay lại từ đầu và không bao giờ đạt đến điều kiện thoát (i < 5), dẫn đến vòng lặp vô tận.

Ví dụ 5: Vòng lặp vô tận do Hiệu ứng phụ kết hợp với toán tử ++

Mô tả: Trong ví dụ này, vòng lặp while có Hiệu ứng phụ khi sử dụng toán tử ++ kết hợp với biểu thức điều kiện, gây ra vòng lặp vô tận.

```
#include <iostream>
using namespace std;

int main() {
```

```
    int n = 5;
    int i = 0;

    while (i++ < n) {
        cout << "i: " << i << ", n: " << n << endl;

        // Hiệu ứng phụ: tăng n trong vòng lặp
        n++; // Mỗi lần tăng n, điều kiện thoát bị đẩy xa hơn
    }

    return 0;
}
```

Giải thích: Biểu thức i++ < n dự kiến sẽ giúp vòng lặp dừng lại khi i đạt giá trị n. Tuy nhiên, mỗi lần vòng lặp chạy, n lại tăng thêm 1 do câu lệnh n++. Điều này khiến điều kiện thoát bị đẩy xa hơn, và vòng lặp không bao giờ kết thúc.

Tại Sao Các Ví Dụ Này Dẫn Đến Vòng Lặp Vô Tận?

- Thay đổi biến điều khiển vòng lặp bên trong vòng lặp: Việc thay đổi biến điều khiển vòng lặp (như i, j, n, v.v.) bên trong vòng lặp mà không có kiểm soát hoặc điều kiện rõ ràng dẫn đến các vòng lặp không bao giờ đạt đến điều kiện thoát.

- Tác dụng phụ làm thay đổi điều kiện thoát: Khi biến hoặc biểu thức điều kiện thoát bị thay đổi không mong muốn hoặc không kiểm soát được bên trong vòng lặp, điều kiện thoát không bao giờ được thỏa mãn.

- Thiếu hiểu biết về thứ tự đánh giá và hiệu ứng phụ: Lập trình viên có thể vô tình thay đổi biến điều khiển vòng lặp hoặc biến điều kiện thoát trong khi biểu thức điều kiện vẫn phụ thuộc vào giá trị ban đầu, dẫn đến vòng lặp vô tận.

Cách Phòng Tránh Vòng Lặp Vô Tận Do Hiệu ứng Phụ:

- Tránh thay đổi biến điều khiển vòng lặp bên trong vòng lặp: Chỉ thay đổi biến điều khiển bên trong vòng lặp nếu thực sự cần thiết và có kiểm soát.

- Kiểm tra điều kiện thoát một cách rõ ràng: Đảm bảo rằng điều kiện thoát của vòng lặp không bị tác động bởi các tác dụng phụ không mong muốn.

- Sử dụng cấu trúc vòng lặp phù hợp: Sử dụng các cấu trúc vòng lặp (for, while, do-while) một cách thích hợp tùy theo mục đích sử dụng và điều kiện thoát.
- Dùng công cụ và phương pháp kiểm thử: Sử dụng các công cụ như Valgrind, AddressSanitizer để phát hiện lỗi và vòng lặp vô tận trong quá trình phát triển.

Hiệu ứng phụ trong vòng lặp có thể dẫn đến vòng lặp vô tận và gây ra các lỗi khó tìm. Hiểu rõ tác dụng phụ và cách quản lý chúng sẽ giúp lập trình viên viết mã an toàn, đáng tin cậy và dễ bảo trì hơn.

Hiệu ứng phụ (side effect) là một khái niệm quan trọng trong lập trình C++ mà lập trình viên cần phải hiểu rõ. Mặc dù Hiệu ứng phụ có thể không tránh khỏi trong một số trường hợp, nhưng hiểu rõ cách chúng hoạt động và ảnh hưởng đến chương trình sẽ giúp lập trình viên viết mã an toàn hơn, dễ hiểu hơn và dễ bảo trì hơn.

## 4.15. Tinh chỉnh mã nguồn - Code tuning

Code Tuning (Tinh chỉnh mã nguồn) là quá trình tối ưu hóa mã nguồn để cải thiện hiệu suất chương trình, giảm thiểu thời gian chạy, tiết kiệm bộ nhớ, và sử dụng tài nguyên một cách hiệu quả hơn. Trong lập trình C/C++, việc tinh chỉnh mã nguồn rất quan trọng, đặc biệt là khi phát triển các ứng dụng yêu cầu hiệu suất cao như hệ thống nhúng, trò chơi, và các ứng dụng tính toán khoa học. Dưới đây là một số kỹ thuật tinh chỉnh mã nguồn trong C/C++ cùng với các ví dụ minh họa.

### 4.15.1. Sử dụng các Cấu trúc Điều khiển Hiệu quả

**Tránh điều kiện lồng nhau (Nested Conditions):** Khi có nhiều điều kiện lồng nhau, chương trình có thể trở nên khó đọc và chậm hơn. Thay vào đó, nên sử dụng các cấu trúc điều kiện rõ ràng và ít lồng nhau hơn.

Ví dụ:

```
// Mã nguồn không tối ưu
if (a > 0) {
    if (b > 0) {
        if (c > 0) {
            // Thực hiện một số thao tác
        }
    }
}
```

```
// Mã nguồn tối ưu
if (a > 0 && b > 0 && c > 0) {
    // Thực hiện một số thao tác
}
```

Giải thích: Bằng cách kết hợp các điều kiện lại với nhau, mã nguồn sẽ trở nên rõ ràng hơn và chương trình sẽ thực hiện ít bước kiểm tra hơn.

**Sử dụng switch thay cho if-else khi cần:** Khi có nhiều điều kiện kiểm tra cùng một biến, switch có thể nhanh hơn và dễ đọc hơn so với nhiều câu lệnh if-else.

```
// Mã nguồn không tối ưu
if (option == 1) {
    // Thực hiện thao tác 1
} else if (option == 2) {
    // Thực hiện thao tác 2
} else if (option == 3) {
    // Thực hiện thao tác 3
} else {
    // Thực hiện thao tác mặc định
}

// Mã nguồn tối ưu
switch (option) {
    case 1:
        // Thực hiện thao tác 1
        break;
    case 2:
        // Thực hiện thao tác 2
        break;
    case 3:
        // Thực hiện thao tác 3
        break;
    default:
        // Thực hiện thao tác mặc định
        break;
}
```

Giải thích: switch giúp chương trình thực hiện các bước kiểm tra nhanh hơn và mã nguồn dễ đọc hơn so với nhiều câu lệnh if-else liên tiếp.

### 4.15.2. Tối ưu hóa Vòng Lặp (Loop Optimization)

**Giảm Số Lượng Phép Toán Trong Vòng Lặp:** Đặt các phép toán ngoài vòng lặp nếu chúng không cần phải tính lại trong mỗi lần lặp.

Ví dụ:

```
// Mã nguồn không tối ưu
for (int i = 0; i < 1000; i++) {
    int x = a * b; // Tính toán này được thực hiện 1000 lần không cần thiết
    array[i] = x * i;
}
.
// Mã nguồn tối ưu
int x = a * b; // Tính toán này chỉ cần thực hiện một lần
for (int i = 0; i < 1000; i++) {
    array[i] = x * i;
}
```

Giải thích: Tính toán a * b được đặt ngoài vòng lặp để không phải tính lại trong mỗi lần lặp, giúp tiết kiệm thời gian.

**Unrolling Loops (Mở rộng vòng lặp):** Unrolling loop là kỹ thuật xử lý nhiều phần tử trong một lần lặp thay vì xử lý từng phần tử trong mỗi lần lặp, giúp giảm số lần kiểm tra điều kiện.

Ví dụ:

```
// Mã nguồn không tối ưu
for (int i = 0; i < 1000; i++) {
    array[i] = 0;
}
.
// Mã nguồn tối ưu (unrolled loop)
for (int i = 0; i < 1000; i += 4) {
    array[i] = 0;
    array[i + 1] = 0;
```

```
    array[i + 2] = 0;
    array[i + 3] = 0;
}
```

Giải thích: Bằng cách mở rộng vòng lặp, chương trình giảm số lần kiểm tra điều kiện vòng lặp từ 1000 xuống còn 250 lần, cải thiện hiệu suất.

### 4.15.3. Tránh Sao Chép Không Cần Thiết (Avoid Unnecessary Copying)

Khi truyền các đối tượng lớn vào hàm, tránh truyền bằng giá trị. Thay vào đó, sử dụng tham chiếu hoặc con trỏ để tránh việc sao chép dữ liệu không cần thiết.

Ví dụ:

```
#include <vector>

// Mã nguồn không tối ưu
void processVector(std::vector<int> v) {
    // Xử lý vector
}

// Mã nguồn tối ưu
void processVector(const std::vector<int>& v) {
    // Xử lý vector
}
```

Giải thích: Truyền đối số bằng tham chiếu (const std::vector<int>) giúp tránh sao chép toàn bộ vector, tiết kiệm bộ nhớ và thời gian xử lý.

### 4.15.4. Sử Dụng Các Hàm Thư Viện Chuẩn Tối Ưu

Các hàm thư viện chuẩn thường được tối ưu hóa cao hơn so với mã nguồn tự viết. Sử dụng chúng khi có thể.

Ví dụ:

Thay vì viết hàm tìm giá trị lớn nhất:

```
// Mã nguồn tự viết (ít tối ưu hơn)
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

Sử dụng hàm std::max trong thư viện chuẩn:

```
#include <algorithm>

int result = std::max(a, b);
```

Giải thích: std::max được tối ưu hóa cao và đã được kiểm tra kỹ lưỡng trong các thư viện chuẩn.

### 4.15.5. Tối ưu hóa Bộ Nhớ

**Sử dụng Kích Thước Dữ Liệu Phù Hợp:** Chọn kiểu dữ liệu với kích thước phù hợp nhất để tiết kiệm bộ nhớ và tăng hiệu suất.

Ví dụ:

```
// Mã nguồn không tối ưu
long long int sum = 0; // Dùng kiểu dữ liệu quá lớn so với nhu cầu

// Mã nguồn tối ưu
int sum = 0; // Chọn kiểu dữ liệu nhỏ hơn, phù hợp hơn
```

Giải thích: Sử dụng kiểu dữ liệu lớn không cần thiết có thể làm chậm chương trình và tốn bộ nhớ.

**Cấp Phát và Giải Phóng Bộ Nhớ Hợp Lý** Sử dụng kỹ thuật cấp phát động bộ nhớ chỉ khi cần thiết và giải phóng bộ nhớ ngay sau khi sử dụng xong.

Ví dụ:

```
#include <iostream>

// Mã nguồn tối ưu
void createArray(int size) {
    int* arr = new int[size]; // Cấp phát bộ nhớ động
    // Sử dụng mảng
    delete[] arr; // Giải phóng bộ nhớ ngay sau khi sử dụng
}

int main() {
```

```
    createArray(100);
    return 0;
}
```

Giải thích: Giải phóng bộ nhớ ngay khi không còn cần sử dụng để tránh rò rỉ bộ nhớ và tối ưu hóa sử dụng tài nguyên.

### 4.15.6. Tối ưu hóa việc sử dụng con trỏ và tham chiếu

Sử dụng con trỏ và tham chiếu đúng cách có thể giúp tránh sao chép dữ liệu không cần thiết, giảm thiểu chi phí bộ nhớ và tăng tốc độ chương trình. Ví dụ:

```
#include <iostream>
#include <vector>

// Mã nguồn không tối ưu (truyền tham số theo giá trị)
void printVector(std::vector<int> vec) {
    for (int i : vec) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}

// Mã nguồn tối ưu (truyền tham số bằng tham chiếu)
void printVectorOptimized(const std::vector<int>& vec) {
    for (int i : vec) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> myVec = {1, 2, 3, 4, 5};

    printVector(myVec);             // Sao chép toàn bộ vector
    printVectorOptimized(myVec);    // Truyền tham chiếu, không sao chép

    return 0;
}
```

### 4.15.7. Tránh các phép tính phức tạp trong vòng lặp

Nếu một phép tính không cần phải thực hiện nhiều lần, hãy đưa nó ra khỏi vòng lặp để tránh thực hiện lặp lại.

Ví dụ:

```
#include <cmath>
#include <iostream>

// Mã nguồn không tối ưu
void calculateSin() {
    for (int i = 0; i < 1000; i++) {
        double result = sin(45); // Tính sin(45) nhiều lần không cần thiết
    }
}

// Mã nguồn tối ưu
void calculateSinOptimized() {
    double sine = sin(45); // Tính một lần duy nhất
    for (int i = 0; i < 1000; i++) {
        double result = sine;
    }
}

int main() {
    calculateSin();
    calculateSinOptimized();
    return 0;
}
```

### 4.15.8. Sử dụng Inline Function thay vì hàm thông thường

Hàm inline giúp giảm overhead của các lời gọi hàm, đặc biệt hữu ích với các hàm nhỏ được gọi thường xuyên. Ví dụ:

```
// Mã nguồn không tối ưu
int square(int x) {
    return x * x;
}
```

```
// Mã nguồn tối ưu (inline function)
inline int squareOptimized(int x) {
    return x * x;
}
```

Tinh chỉnh mã nguồn (Code tuning) là một phần quan trọng trong phát triển phần mềm để đảm bảo hiệu suất và hiệu quả của chương trình. Bằng cách áp dụng các kỹ thuật tối ưu hóa, lập trình viên có thể cải thiện đáng kể hiệu suất của ứng dụng. Tuy nhiên, cần nhớ rằng việc tối ưu hóa cần cân nhắc và chỉ nên thực hiện khi có lý do chính đáng, tránh làm mã trở nên khó hiểu và khó bảo trì.

## 4.16. Phụ lục - Lựa chọn giữa if-else if và switch

Việc lựa chọn giữa if-else if và switch trong C/C++ có thể ảnh hưởng đến hiệu suất của chương trình, nhưng không phải lúc nào if-else if cũng nhanh hơn switch. Để xác định khi nào if-else if nhanh hơn switch và ngược lại, chúng ta cần xem xét cách cả hai hoạt động và cách trình biên dịch xử lý chúng.

### 4.16.1. Cách hoạt động của if-else if và switch

**if-else if** Cách hoạt động: Mỗi câu lệnh if hoặc else if được đánh giá lần lượt từ trên xuống dưới cho đến khi tìm thấy điều kiện đúng hoặc cho đến cuối chuỗi câu lệnh. Tốc độ: Tốc độ của if-else if phụ thuộc vào vị trí của điều kiện đúng. Nếu điều kiện đúng là điều kiện đầu tiên, tốc độ sẽ nhanh hơn. Nếu điều kiện đúng là điều kiện cuối cùng hoặc không có điều kiện nào đúng, tất cả các điều kiện phải được kiểm tra tuần tự.

```
if (x == 1) {
    // do something
} else if (x == 2) {
    // do something else
} else if (x == 3) {
    // do another thing
} else {
    // do the default action
}
```

**switch** Cách hoạt động: switch thực hiện nhảy tới trường hợp phù hợp mà không kiểm tra tuần tự từng điều kiện. Nó sử dụng bảng tra cứu (lookup table) hoặc lệnh nhảy (jump table) để quyết định nhanh chóng trường hợp nào sẽ được thực thi. Tốc độ: switch thường nhanh hơn if-else if khi có nhiều nhánh (case), vì nó sử dụng bảng tra cứu để trực tiếp nhảy đến trường hợp phù hợp thay vì kiểm tra tuần tự từng điều kiện.

```
switch (x) {
    case 1:
        // do something
        break;
    case 2:
        // do something else
        break;
    case 3:
        // do another thing
        break;
    default:
        // do the default action
        break;
}
```

**Khi nào if-else if nhanh hơn switch?** Số lượng điều kiện ít: Khi chỉ có một hoặc hai điều kiện, if-else if có thể nhanh hơn do không cần tạo bảng tra cứu. Điều kiện phức tạp hoặc không thể dự đoán trước: Khi các điều kiện cần so sánh là phức tạp hoặc không phải là hằng số, if-else if là lựa chọn phù hợp vì switch chỉ hoạt động với các giá trị nguyên, ký tự, hoặc các biểu thức hằng số. Điều kiện đúng nằm ở đầu: Nếu điều kiện đúng nằm ở đầu, if-else if có thể thực thi nhanh hơn do ít phép so sánh hơn.

**Khi nào switch nhanh hơn if-else if?** Nhiều nhánh (case) và các giá trị là hằng số: Khi có nhiều nhánh và các giá trị là hằng số hoặc có thể tính toán tại thời gian biên dịch, switch sẽ tạo ra một bảng tra cứu giúp tăng tốc độ quyết định nhánh nào cần thực thi. Các điều kiện kiểm tra đơn giản: Khi các điều kiện là các phép kiểm tra đơn giản với các giá trị hằng số hoặc biểu thức có thể tối ưu hóa, switch có thể hiệu quả hơn.

**Thử nghiệm hiệu suất với if-else if và switch** Dưới đây là một thử nghiệm đơn giản để so sánh tốc độ giữa if-else if và switch.

```
#include <iostream>
#include <chrono>

void testIfElse(int x) {
    if (x == 1) {
        // do something
    } else if (x == 2) {
        // do something
    } else if (x == 3) {
        // do something
    } else if (x == 4) {
        // do something
    } else if (x == 5) {
        // do something
    } else if (x == 6) {
        // do something
    } else if (x == 7) {
        // do something
    } else if (x == 8) {
        // do something
    } else if (x == 9) {
        // do something
    } else {
        // do something
    }
}

void testSwitch(int x) {
    switch (x) {
        case 1:
            // do something
            break;
        case 2:
            // do something
            break;
```

```
        case 3:
            // do something
            break;
        case 4:
            // do something
            break;
        case 5:
            // do something
            break;
        case 6:
            // do something
            break;
        case 7:
            // do something
            break;
        case 8:
            // do something
            break;
        case 9:
            // do something
            break;
        default:
            // do something
            break;
    }
}

int main() {
    int iterations = 10000000;
    int x = 5; // Giá trị thử nghiệm

    // Kiểm tra hiệu suất với if-else if
    auto startIfElse = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < iterations; ++i) {
        testIfElse(x);
    }
    auto endIfElse = std::chrono::high_resolution_clock::now();
```

```
    auto durationIfElse = std::chrono::duration_cast<std::chrono::microseconds>(endIfElse
    std::cout << "Time taken by if-else if: " << durationIfElse << " microseconds." << sto

    // Kiểm tra hiệu suất với switch
    auto startSwitch = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < iterations; ++i) {
        testSwitch(x);
    }
    auto endSwitch = std::chrono::high_resolution_clock::now();
    auto durationSwitch = std::chrono::duration_cast<std::chrono::microseconds>(endSwitch
    std::cout << "Time taken by switch: " << durationSwitch << " microseconds." << std::en

    return 0;
}
```

**Kết quả và phân tích:** Kết quả sẽ phụ thuộc vào trình biên dịch, tối ưu hóa được áp dụng, và kiến trúc của hệ thống. Tuy nhiên, bạn sẽ thấy rằng switch thường nhanh hơn if-else if khi có nhiều nhánh và các giá trị là hằng số, nhờ vào việc sử dụng bảng tra cứu hoặc tối ưu hóa khác.

**Kết luận:** Không có câu trả lời rõ ràng nào cho việc if-else if nhanh hơn switch hay ngược lại. Điều này phụ thuộc vào số lượng và loại điều kiện, cách trình biên dịch tối ưu hóa mã, và kiến trúc của bộ xử lý. if-else if phù hợp cho các điều kiện đơn giản hoặc khi số lượng điều kiện ít. switch phù hợp khi có nhiều nhánh và các giá trị là hằng số, đặc biệt là khi trình biên dịch có thể tối ưu hóa nó bằng cách sử dụng bảng tra cứu. Để có kết quả chính xác hơn, lập trình viên nên thử nghiệm hiệu suất cụ thể trên mã nguồn của mình và trong môi trường của mình.

# Chương 5
# Phong cách lập trình



## 5.1. Giới thiệu về Phong Cách Lập Trình

Phong cách lập trình (programming style) là một phần quan trọng trong phát triển phần mềm, giúp mã nguồn dễ đọc, dễ hiểu, dễ bảo trì và ít lỗi hơn. Mặc dù phong cách lập trình có thể khác nhau giữa các nhóm và dự án, nhưng có một số nguyên tắc và quy tắc cơ bản được khuyến nghị áp dụng để đảm bảo mã nguồn chất lượng.

Phong cách lập trình đề cập đến các quy tắc và hướng dẫn mà lập trình viên tuân thủ để viết mã nguồn rõ ràng, dễ hiểu và dễ bảo trì. Một phong cách lập trình nhất quán giúp cải thiện khả năng đọc mã, giảm thiểu lỗi và nâng cao hiệu suất làm việc nhóm.

## 5.2. Các Nguyên Tắc Cơ Bản của Phong Cách Lập Trình

### 5.2.1. Quy tắc đặt tên (Naming Conventions)

Đặt tên là một phần quan trọng trong phong cách lập trình. Các tên rõ ràng và nhất quán giúp mã dễ đọc và giảm thiểu lỗi.

Nguyên tắc:

- Biến và hàm: Sử dụng camelCase hoặc snake$_{c}aseØØttnchoccbinvhm.Hngs$ : $Sdngchinhoavphncchbngdugchdi(UPPER_{S}NAKE_{C}ASE)ØØttnchocchngs.$

- Tên lớp: Sử dụng PascalCase để đặt tên cho các lớp và cấu trúc.

Ví dụ cụ thể:

Biến và hàm:

```
int employeeCount;  // Đúng: tên biến rõ ràng, theo camelCase
int employee_count; // Cũng đúng: tên biến rõ ràng, theo snake_case
int n;              // Sai: tên biến quá ngắn và không rõ nghĩa

double calculateArea(double width, double height); // Đúng: tên hàm rõ ràng và mô tả chức
double calc_area(double width, double height);     // Cũng đúng, rõ ràng
double ca(double w, double h);                     // Sai: viết tắt không cần thiết, khó
```

Hằng số:

```
const double PI = 3.14159;     // Đúng: rõ ràng và dễ hiểu
const double pi = 3.14159;     // Sai: không theo quy tắc hằng số (nên viết hoa)
const int MAX_CONNECTIONS = 10; // Đúng: sử dụng chữ in hoa và phân cách bằng dấu gạch dư
```

Tên lớp:

```
class EmployeeDetails {  // Đúng: tên lớp rõ ràng, theo PascalCase
public:
    string name;
    double salary;
};

class EmpDet {  // Sai: tên lớp viết tắt, không rõ ràng
public:
    string name;
    double salary;
};
```

### 5.2.2. Tổ chức mã nguồn (Code Organization)

Tổ chức mã nguồn tốt giúp mã dễ đọc, dễ bảo trì và dễ hiểu hơn.

Nguyên tắc:

- Hàm ngắn và có chức năng duy nhất: Mỗi hàm chỉ nên thực hiện một chức năng duy nhất. Tránh các hàm quá dài và phức tạp.

- Sử dụng khối mã rõ ràng: Mỗi khối mã nên có chức năng cụ thể và được phân cách bằng dòng trống để tăng khả năng đọc.
- Đặt hàm main() gọn gàng: Hàm main() nên gọn gàng và chỉ gọi các hàm khác để thực hiện các tác vụ cụ thể.

Ví dụ cụ thể:

```
#include <iostream>
#include <vector>
using namespace std;

// Đúng: hàm ngắn và có chức năng duy nhất
void fetchEmployeeData() {
    // Lấy dữ liệu nhân viên từ cơ sở dữ liệu...
}

void calculateSalaries() {
    // Tính toán lương cho nhân viên...
}

void calculateBonuses() {
    // Tính toán tiền thưởng cho nhân viên...
}

void saveToDatabase() {
    // Lưu dữ liệu nhân viên vào cơ sở dữ liệu...
}

// Sai: hàm quá dài và phức tạp, thực hiện nhiều chức năng
void processEmployeeData() {
    // Lấy dữ liệu nhân viên từ cơ sở dữ liệu...
    // Tính toán lương...
    // Tính toán tiền thưởng...
    // Lưu dữ liệu vào cơ sở dữ liệu...
}

int main() {
```

```
    // Đúng: hàm main gọn gàng và rõ ràng
    fetchEmployeeData();
    calculateSalaries();
    calculateBonuses();
    saveToDatabase();
    return 0;
}
```

### 5.2.3. Định dạng mã (Code Formatting)

Định dạng mã nguồn tốt giúp mã dễ đọc hơn và dễ hiểu hơn.

Nguyên tắc:

- Thụt đầu dòng nhất quán: Sử dụng thụt đầu dòng để chỉ rõ cấu trúc của mã (vòng lặp, điều kiện, hàm, v.v.).
- Tránh dòng mã quá dài: Dòng mã nên được giới hạn ở khoảng 80-100 ký tự để dễ đọc trên mọi thiết bị.
- Sử dụng khoảng trắng hợp lý: Sử dụng khoảng trắng để phân cách các toán tử, từ khóa và đối số hàm.

Ví dụ cụ thể:

```
#include <iostream>
using namespace std;

// Đúng: sử dụng thụt đầu dòng hợp lý và khoảng trắng hợp lý
if (condition) {
    doSomething();
} else {
    doSomethingElse();
}

// Sai: thụt đầu dòng không hợp lý, dòng mã dài
if(condition){doSomething();}else{doSomethingElse();}
```

### 5.2.4. Bình luận (Commenting)

Bình luận giúp người đọc hiểu mã nhanh hơn và dễ dàng bảo trì mã hơn.

Nguyên tắc:

- Sử dụng bình luận có ích: Chỉ sử dụng bình luận khi cần thiết để giải thích các đoạn mã phức tạp hoặc cung cấp ngữ cảnh.

- Tránh bình luận thừa thãi: Bình luận không nên mô tả lại những gì mã đã làm rõ.

- Sử dụng bình luận kiểu block và inline một cách hợp lý:

- Inline comments: Đặt trên cùng một dòng với mã mà nó giải thích.

- Block comments: Được sử dụng cho các đoạn mã dài hơn hoặc giải thích cấp độ cao hơn.

Ví dụ cụ thể:

```
#include <iostream>
#include <vector>
using namespace std;

// Đúng: bình luận có ích, giải thích rõ ràng ý định của mã
// Tính tổng số tiền lương của tất cả nhân viên
double calculateTotalSalary(const vector<double>& salaries) {
    double totalSalary = 0.0;
    for (const auto& salary : salaries) {
        totalSalary += salary;
    }
    return totalSalary;
}

// Sai: bình luận không cần thiết, thừa thãi
double calculateTotalSalary(const vector<double>& salaries) {
    double totalSalary = 0.0; // Khởi tạo biến totalSalary
    for (const auto& salary : salaries) { // Lặp qua từng phần tử trong vector s
        totalSalary += salary; // Cộng lương vào tổng lương
    }
    return totalSalary; // Trả về tổng lương
}
```

## 5.3. Các Thực Tiễn Tốt Trong Phong Cách Lập Trình

Nguyên tắc: Mỗi nhóm hoặc dự án thường có quy tắc phong cách lập trình riêng, như quy tắc đặt tên, cách thụt đầu dòng, khoảng trắng, và các tiêu chuẩn kiểm thử. Tuân thủ các quy tắc này để đảm bảo tính nhất quán và dễ bảo trì.

### 5.3.1. Tránh sử dụng các "magic numbers"

Nguyên tắc: Không sử dụng các giá trị số hoặc hằng số cố định trong mã mà không có ý nghĩa rõ ràng. Thay vào đó, sử dụng các hằng số có tên hoặc các biến const để tăng tính rõ ràng và dễ hiểu của mã. Ví dụ cụ thể:

```
// Sai: sử dụng magic number
double calculateDiscount(double amount) {
    return amount * 0.05; // 0.05 là magic number, không rõ nghĩa
}

// Đúng: sử dụng hằng số có tên
const double DISCOUNT_RATE = 0.05;
double calculateDiscount(double amount) {
    return amount * DISCOUNT_RATE; // Rõ ràng và dễ hiểu hơn
}
```

### 5.3.2. Sử dụng các cấu trúc điều khiển rõ ràng

Nguyên tắc: Tránh sử dụng các cấu trúc điều khiển phức tạp hoặc lồng nhau sâu. Sử dụng các cấu trúc điều khiển rõ ràng, dễ hiểu để mô tả luồng logic của chương trình. Ví dụ cụ thể:

```
// Sai: cấu trúc điều khiển lồng nhau sâu
if (condition1) {
    if (condition2) {
        if (condition3) {
            // Xử lý...
        }
    }
}

// Đúng: sử dụng các cấu trúc điều khiển rõ ràng
if (condition1 && condition2 && condition3) {
```

```
    // Xử lý...
}
```

### 5.3.3. Tối ưu hóa và đơn giản hóa mã

Nguyên tắc: Tránh các tối ưu hóa mã trước khi cần thiết. Tập trung vào việc viết mã rõ ràng và dễ hiểu trước, sau đó tối ưu hóa khi cần. Đơn giản hóa mã bằng cách loại bỏ các đoạn mã thừa và tập trung vào việc giải quyết vấn đề cụ thể. Ví dụ cụ thể:

```
// Sai: tối ưu hóa mã không cần thiết, làm cho mã khó đọc
void complexFunction() {
    // Các đoạn mã tối ưu hóa phức tạp...
}

// Đúng: đơn giản hóa mã, tập trung vào chức năng chính
void simpleFunction() {
    // Mã rõ ràng và tập trung vào chức năng chính...
}
```

## 5.4. Khi nào cần sử dụng phong cách lập trình tốt?

- Phong cách lập trình tốt nên được sử dụng mọi lúc để đảm bảo mã nguồn dễ hiểu, dễ bảo trì và ít lỗi hơn. Đặc biệt quan trọng khi:
- Làm việc trong nhóm hoặc dự án lớn: Giúp các thành viên nhóm hiểu mã của nhau và dễ dàng hợp tác.
- Phát triển phần mềm dài hạn: Đảm bảo mã nguồn dễ bảo trì và mở rộng trong tương lai.
- Viết mã nguồn mở hoặc chia sẻ mã: Đảm bảo rằng mã của bạn dễ đọc và dễ hiểu cho những người khác.

## 5.5. Lợi ích của việc tuân thủ phong cách lập trình tốt

- Cải thiện khả năng đọc mã: Mã dễ đọc và dễ hiểu giúp các lập trình viên khác dễ dàng tham gia và đóng góp vào dự án.
- Giảm thiểu lỗi: Mã rõ ràng và có tổ chức tốt thường ít lỗi hơn vì dễ kiểm tra và xác minh tính đúng đắn.

- Dễ dàng bảo trì và mở rộng: Mã được viết theo phong cách tốt dễ bảo trì hơn, cho phép thêm tính năng mới hoặc sửa lỗi dễ dàng hơn.
- Nâng cao hiệu suất làm việc nhóm: Giảm thiểu sự hiểu lầm và sai sót giữa các thành viên nhóm khi làm việc trên cùng một mã nguồn.

## 5.6. Kết luận

Phong cách lập trình tốt là một phần quan trọng của quá trình phát triển phần mềm, giúp mã nguồn dễ đọc, dễ bảo trì và ít lỗi hơn. Bằng cách tuân thủ các nguyên tắc và quy tắc cơ bản của phong cách lập trình, lập trình viên có thể viết mã chất lượng cao và đảm bảo sự thành công lâu dài của dự án.

# Chương 6
# Đệ quy - Recursion



## 6.1. Khái niệm Đệ Quy

Đệ quy là một phương pháp lập trình trong đó một hàm tự gọi lại chính nó. Đệ quy thường được sử dụng để giải quyết các bài toán có tính chất lặp lại hoặc phân chia bài toán lớn thành các bài toán con nhỏ hơn giống với bài toán ban đầu.

Định nghĩa đệ quy: Một đối tượng hoặc hàm được gọi là đệ quy nếu nó được định nghĩa (hoặc hoạt động) thông qua chính nó. Ví dụ:

- Tập hợp số tự nhiên: Được mô tả đệ quy như sau:
- Số 1 là số tự nhiên.
- Một số tự nhiên có thể được tạo ra bằng cách cộng 1 vào một số tự nhiên khác.
- Cấu trúc danh sách liên kết (list) kiểu T:
- Danh sách rỗng là một danh sách kiểu T.
- Ghép nối một phần tử kiểu T với một danh sách kiểu T khác sẽ tạo ra một danh sách kiểu T

## 6.2. Các Thành Phần Của Đệ Quy

Mỗi định nghĩa hoặc hàm đệ quy bao gồm hai phần quan trọng:

- Phần cơ sở (Base Case): Đây là điều kiện dừng của đệ quy, ngăn chặn đệ quy vô hạn. Đây thường là trường hợp đơn giản nhất của bài toán mà có thể giải quyết trực tiếp mà không cần gọi đệ quy.
- Phần đệ quy (Recursive Case): Đây là phần mà hàm gọi lại chính nó với một đầu vào khác (nhỏ hơn hoặc đơn giản hơn) để tiến dần đến trường hợp cơ sở.

Ví dụ: Tính giai thừa của một số nguyên dương

```
int factorial(int n) {
    if (n == 0) // Trường hợp cơ sở
        return 1;
    else
        return n * factorial(n - 1); // Trường hợp đệ quy
}
```

## 6.3. Các Dạng Đệ Quy Phổ Biến

### 6.3.1. Đệ Quy Tuyến Tính (Linear Recursion)

Định nghĩa: Một hàm đệ quy gọi lại chính nó chỉ một lần trong mỗi bước thực hiện. Ví dụ: Hàm tính giai thừa đã nêu ở trên là một ví dụ của đệ quy tuyến tính. Ví dụ khác: Tính tổng các số từ 1 đến n

```
int sum(int n) {
    if (n == 1) // Trường hợp cơ sở
        return 1;
    else
        return n + sum(n - 1); // Trường hợp đệ quy
}
```

### 6.3.2. Đệ Quy Nhị Phân (Binary Recursion)

Định nghĩa: Một hàm đệ quy gọi lại chính nó hai lần trong mỗi bước thực hiện. Ví dụ: Hàm tính số Fibonacci. Mã C++ cho hàm tính số Fibonacci:

```
int fibonacci(int n) {
    if (n <= 1) // Trường hợp cơ sở
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2); // Trường hợp đệ quy nhị phâ
}
```

### 6.3.3. Đệ Quy Phi Tuyến (Non-linear Recursion)

Định nghĩa: Đây là dạng đệ quy trong đó lời gọi đệ quy được thực hiện bên trong một vòng lặp.

Ví dụ: Hàm tính giá trị của dãy số An theo công thức truy hồi.

```
int A(int n) {
    if (n == 0) return 1; // Trường hợp cơ sở
    else {
        int tg = 0;
        for (int i = 0; i < n; i++) // Vòng lặp bên trong hàm đệ quy
            tg += (n - i) * A(i);
        return tg;
    }
}
```

### 6.3.4. Đệ Quy Gián Tiếp (Indirect Recursion)

Định nghĩa: Đệ quy gián tiếp xảy ra khi một hàm gọi một hàm khác, hàm này lại gọi hàm ban đầu.

Ví dụ:

```
void functionA(int n);
void functionB(int n);

void functionA(int n) {
    if (n > 0) {
        cout << "Function A: " << n << endl;
        functionB(n - 1); // Gọi functionB
    }
}
```

```
void functionB(int n) {
    if (n > 1) {
        cout << "Function B: " << n << endl;
        functionA(n / 2); // Gọi functionA
    }
}
```

## 6.4. Ứng Dụng Của Đệ Quy

Giải bài toán phân chia và trị (Divide and Conquer): Như thuật toán Merge Sort, Quick Sort. Giải các bài toán có tính chất lặp lại hoặc phân cấp: Như bài toán Tháp Hà Nội. Duyệt cây và đồ thị: Các thuật toán tìm kiếm như DFS (Depth-First Search).

Ví dụ: Bài toán Tháp Hà Nội

```
void hanoi(int n, char source, char target, char auxiliary) {
    if (n == 1) {
        cout << "Move disk 1 from " << source << " to " << target << endl;
        return;
    }
    hanoi(n - 1, source, auxiliary, target); // Chuyển n-1 đĩa từ source sang auxiliary
    cout << "Move disk " << n << " from " << source << " to " << target << endl;
    hanoi(n - 1, auxiliary, target, source); // Chuyển n-1 đĩa từ auxiliary sang target
}
```

## 6.5. Các Bài Toán Giải Bằng Đệ Quy

Bài toán Tháp Hà Nội: Chuyển n đĩa từ cột A sang cột C với sự trợ giúp của cột B sao cho không có đĩa lớn nào đặt trên đĩa nhỏ hơn. Bài toán chia phần thưởng: Chia một số lượng phần thưởng cố định cho một số học sinh đã được sắp xếp thứ tự. Bài toán tính số Fibonacci: Tính số Fibonacci thứ n dựa trên công thức truy hồi.

## 6.6. Khử Đệ Quy (Eliminating Recursion)

Khử đệ quy là quá trình chuyển đổi một thuật toán đệ quy thành một thuật toán không đệ quy (thường sử dụng vòng lặp hoặc cấu trúc dữ liệu như ngăn xếp). Mặc dù đệ quy là một kỹ thuật lập trình mạnh mẽ và thường đơn giản hóa việc giải

quyết các bài toán đệ quy, nhưng đôi khi nó có thể dẫn đến các vấn đề về hiệu suất và sử dụng bộ nhớ. Khử đệ quy giúp cải thiện hiệu suất và tránh lỗi tràn ngăn xếp (stack overflow).

### 6.6.1. Giới thiệu về Khử Đệ Quy

Khử đệ quy là việc chuyển đổi một hàm đệ quy thành một hàm không đệ quy, sử dụng các cấu trúc lặp (for, while) hoặc các cấu trúc dữ liệu như ngăn xếp (stack). Mục tiêu của việc khử đệ quy là:

- Tối ưu hóa hiệu suất: Hàm không đệ quy thường chạy nhanh hơn vì không cần thực hiện các thao tác lưu trữ và phục hồi trạng thái trong ngăn xếp khi gọi hàm.
- Giảm sử dụng bộ nhớ: Tránh lỗi tràn ngăn xếp khi số lượng đệ quy quá lớn.
- Dễ dàng quản lý và kiểm soát luồng thực thi: Vòng lặp cho phép kiểm soát tốt hơn luồng thực thi so với đệ quy.

### 6.6.2. Các Kỹ Thuật Khử Đệ Quy

**Khử Đệ Quy Tuyến Tính (Linear Recursion Elimination):** Đệ quy tuyến tính xảy ra khi một hàm gọi lại chính nó một lần trong quá trình thực thi. Đây là loại đệ quy dễ khử nhất, thường bằng cách sử dụng vòng lặp while hoặc for.

Ví dụ 1: Tính giai thừa của một số nguyên

```
//Hàm đệ quy tính giai thừa:

long factorial(int n) {
    if (n == 0) return 1; // Trường hợp cơ sở
    return n * factorial(n - 1); // Trường hợp đệ quy
}
//Hàm không đệ quy sử dụng vòng lặp for:

long factorial(int n) {
    long result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i; // Tính giai thừa bằng cách nhân liên tiếp
    }
    return result;
}
```

Ví dụ 2: Tính số Fibonacci

```
//Hàm đệ quy tính Fibonacci:

int fibonacci(int n) {
    if (n <= 1) return n; // Trường hợp cơ sở
    return fibonacci(n - 1) + fibonacci(n - 2); // Trường hợp đệ quy
}
//Hàm không đệ quy sử dụng vòng lặp for:

int fibonacci(int n) {
    if (n <= 1) return n;

    int a = 0, b = 1, fib = 0;
    for (int i = 2; i <= n; ++i) {
        fib = a + b; // Tính số Fibonacci tiếp theo
        a = b; // Cập nhật giá trị
        b = fib; // Cập nhật giá trị
    }
    return fib;
}
```

**Khử Đệ Quy Đuôi (Tail Recursion Elimination)** Đệ quy đuôi là một dạng đặc biệt của đệ quy, nơi lời gọi đệ quy là hành động cuối cùng được thực hiện bởi hàm. Khử đệ quy đuôi có thể dễ dàng thực hiện bằng cách chuyển đổi đệ quy thành một vòng lặp while.

Ví dụ: Tính tổng các số từ 1 đến n

```
//Hàm đệ quy đuôi:
int sum(int n, int acc = 0) { // acc là tham số tích lũy
    if (n == 0) return acc; // Trường hợp cơ sở
    return sum(n - 1, acc + n); // Trường hợp đệ quy đuôi
}

//Hàm không đệ quy sử dụng vòng lặp while:
int sum(int n) {
    int acc = 0; // Giá trị tích lũy
```

```
    while (n > 0) {
        acc += n; // Cộng dồn
        n--; // Giảm n
    }
    return acc;
}
```

**Khử Đệ Quy Nhị Phân (Binary Recursion Elimination)** Đệ quy nhị phân là khi một hàm gọi lại chính nó hai lần (như trong trường hợp tính số Fibonacci hoặc tìm kiếm nhị phân). Để khử đệ quy nhị phân, chúng ta có thể sử dụng cấu trúc dữ liệu như ngăn xếp (stack) hoặc hàng đợi (queue).

Ví dụ: Tìm kiếm nhị phân

```
//Hàm đệ quy tìm kiếm nhị phân:

int binarySearch(int arr[], int left, int right, int x) {
    if (right >= left) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == x) return mid; // Trường hợp tìm thấy
        if (arr[mid] > x) return binarySearch(arr, left, mid - 1, x); // Tìm tro
        return binarySearch(arr, mid + 1, right, x); // Tìm trong nửa phải
    }
    return -1; // Không tìm thấy
}

//Hàm không đệ quy sử dụng vòng lặp while:

int binarySearch(int arr[], int left, int right, int x) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == x) return mid; // Trường hợp tìm thấy
        if (arr[mid] > x) right = mid - 1; // Tìm trong nửa trái
        else left = mid + 1; // Tìm trong nửa phải
    }
    return -1; // Không tìm thấy
}
```

**Khử Đệ Quy Với Ngăn Xếp (Stack-based Recursion Elimination)** Khi khử đệ quy cho các thuật toán phức tạp hơn như duyệt cây (tree traversal), chúng ta có thể sử dụng một ngăn xếp để mô phỏng lại ngăn xếp hàm.

Ví dụ: Duyệt cây theo thứ tự trước (Pre-order Traversal)

Hàm đệ quy duyệt cây theo thứ tự trước:

```
struct Node {
    int data;
    Node* left;
    Node* right;
};

void preOrder(Node* root) {
    if (root == nullptr) return;
    cout << root->data << " "; // Thăm gốc
    preOrder(root->left); // Thăm cây con trái
    preOrder(root->right); // Thăm cây con phải
}
```

Hàm không đệ quy sử dụng ngăn xếp:

```
#include <stack>
void preOrder(Node* root) {
    if (root == nullptr) return;

    stack<Node*> s;
    s.push(root);

    while (!s.empty()) {
        Node* current = s.top();
        s.pop();
        cout << current->data << " "; // Thăm gốc

        // Thăm cây con phải trước, sau đó cây con trái
        if (current->right) s.push(current->right);
        if (current->left) s.push(current->left);
    }
}
```

### 6.6.3. Lợi Ích và Hạn Chế của Khử Đệ Quy

**Lợi ích:** Tối ưu hóa hiệu suất: Giảm chi phí gọi hàm và tránh lỗi tràn ngăn xếp. Kiểm soát tốt hơn: Cho phép lập trình viên kiểm soát tốt hơn luồng thực thi của chương trình. Tương thích hơn với môi trường hạn chế: Khử đệ quy hữu ích trong các hệ thống nhúng hoặc các môi trường hạn chế tài nguyên.

**Hạn chế:** Phức tạp hơn: Việc khử đệ quy đôi khi làm cho mã nguồn phức tạp hơn, khó đọc hơn. Tăng kích thước mã: Mã không đệ quy có thể dài và khó hiểu hơn, đặc biệt khi sử dụng các cấu trúc dữ liệu như ngăn xếp hoặc hàng đợi để mô phỏng đệ quy.

### 6.6.4. Khi Nào Nên Khử Đệ Quy?

Khi hiệu suất là ưu tiên hàng đầu: Đệ quy có thể dẫn đến chi phí gọi hàm lớn trong một số trường hợp, và khử đệ quy có thể giúp tối ưu hóa mã. Khi lo ngại về lỗi tràn ngăn xếp: Trong các bài toán với độ sâu đệ quy lớn, đệ quy có thể dẫn đến lỗi tràn ngăn xếp. Trong môi trường hạn chế tài nguyên: Hệ thống nhúng hoặc hệ thống với bộ nhớ giới hạn có thể không thích hợp cho đệ quy sâu.

**Kết Luận:** Khử đệ quy là một kỹ thuật quan trọng trong lập trình để tối ưu hóa hiệu suất và tránh lỗi trong các ứng dụng thực tế. Mặc dù đệ quy là một công cụ mạnh mẽ, nhưng nó không phải lúc nào cũng phù hợp cho mọi bài toán. Việc hiểu rõ khi nào nên sử dụng đệ quy và khi nào nên khử đệ quy sẽ giúp lập trình viên viết mã hiệu quả hơn và tránh được các vấn đề về hiệu suất và bộ nhớ.

## 6.7. Khử đệ quy nâng cao - Dùng Vòng Lặp và Biến Trạng Thái

Khi khử đệ quy, việc sử dụng vòng lặp kết hợp với các biến lưu trữ trạng thái trung gian có thể cải thiện hiệu năng so với việc sử dụng ngăn xếp (stack) hoặc hàng đợi (queue). Đặc biệt, đối với các bài toán đệ quy tương hỗ (mutual recursion), kỹ thuật này có thể giúp tối ưu hóa cả về tốc độ và sử dụng bộ nhớ.

### 6.7.1. Đệ Quy Tương Hỗ (Mutual Recursion) Là Gì?

Đệ quy tương hỗ xảy ra khi hai hoặc nhiều hàm gọi lại nhau. Đây là một dạng đệ quy phức tạp hơn đệ quy trực tiếp, vì nó liên quan đến nhiều hàm và thường có thể dẫn đến luồng kiểm soát khó theo dõi.

Ví dụ: Hàm isEven(int n) và isOdd(int n) gọi lại lẫn nhau để kiểm tra một số nguyên có phải là số chẵn hay không.

```
bool isEven(int n);
bool isOdd(int n);

bool isEven(int n) {
    if (n == 0) return true;  // Trường hợp cơ sở
    return isOdd(n - 1);      // Gọi hàm isOdd
}

bool isOdd(int n) {
    if (n == 0) return false; // Trường hợp cơ sở
    return isEven(n - 1);     // Gọi hàm isEven
}
```

### 6.7.2. Tại Sao Nên Khử Đệ Quy Tương Hỗ Bằng Vòng Lặp và Biến Trạng Thái Trung Gian?

Hiệu suất tốt hơn: Vòng lặp không yêu cầu lưu trữ trạng thái hàm trong ngăn xếp, do đó giảm chi phí liên quan đến gọi hàm (overhead). Tránh lỗi tràn ngăn xếp (stack overflow): Các bài toán đệ quy sâu hoặc có nhiều tầng đệ quy có thể gây ra lỗi tràn ngăn xếp. Sử dụng vòng lặp với các biến trạng thái giúp tránh vấn đề này. Dễ kiểm soát và dễ hiểu hơn: Sử dụng vòng lặp với biến trạng thái có thể làm mã rõ ràng hơn và dễ theo dõi luồng thực thi hơn.

### 6.7.3. Kỹ Thuật Khử Đệ Quy Tương Hỗ Bằng Vòng Lặp và Biến Trạng Thái

Thay vì sử dụng hàm đệ quy để gọi lại lẫn nhau, chúng ta có thể sử dụng một vòng lặp và một số biến để lưu trữ trạng thái của mỗi bước, cho phép tính toán các bước tiếp theo một cách rõ ràng và tuần tự.

Ví dụ: Khử đệ quy tương hỗ cho hàm isEven và isOdd

```
#include <iostream>
using namespace std;

// Khử đệ quy tương hỗ với vòng lặp và biến trạng thái trung gian
bool isEven(int n) {
    bool currentIsEven = true; // Khởi tạo trạng thái hiện tại
    for (int i = 0; i < n; i++) {
        currentIsEven = !currentIsEven; // Lật trạng thái ở mỗi bước
```

```
    }
    return currentIsEven;
}

int main() {
    int num = 4;
    if (isEven(num))
        cout << num << " là số chẵn." << endl; // Output: 4 là số chẵn.
    else
        cout << num << " là số lẻ." << endl;

    return 0;
}
```

Giải thích: Sử dụng biến currentIsEven để lưu trữ trạng thái hiện tại. Vòng lặp for được sử dụng để thực hiện lật trạng thái (true trở thành false và ngược lại) cho mỗi bước từ 0 đến n. Không cần sử dụng đệ quy hoặc ngăn xếp, giúp tiết kiệm bộ nhớ và cải thiện hiệu suất.

Ví Dụ Khác: Tính Fibonacci Không Đệ Quy Với Biến Trạng Thái Trung Gian.

Bài toán tính số Fibonacci thường sử dụng đệ quy nhị phân, nhưng có thể chuyển sang sử dụng vòng lặp với các biến trạng thái để tối ưu hóa.

Hàm đệ quy tính Fibonacci:

```
int fibonacci(int n) {
    if (n <= 1) return n; // Trường hợp cơ sở
    return fibonacci(n - 1) + fibonacci(n - 2); // Trường hợp đệ quy nhị phân
}
```

Hàm không đệ quy sử dụng vòng lặp và biến trạng thái:

```
int fibonacci(int n) {
    if (n <= 1) return n; // Trường hợp cơ sở

    int a = 0, b = 1; // Trạng thái ban đầu
    int fib = 0;

    for (int i = 2; i <= n; i++) {
```

```
        fib = a + b; // Tính Fibonacci hiện tại
        a = b;       // Cập nhật trạng thái cho lần tính tiếp theo
        b = fib;     // Cập nhật trạng thái cho lần tính tiếp theo
    }

    return fib;
}
```

Giải thích: Sử dụng hai biến a và b để lưu trữ hai giá trị Fibonacci trước đó. Sử dụng vòng lặp để tính giá trị Fibonacci hiện tại và cập nhật trạng thái. Hàm không đệ quy này không chỉ tối ưu hơn về mặt bộ nhớ mà còn tránh được lỗi tràn ngăn xếp do đệ quy sâu.

### 6.7.4. Lợi Ích Của Khử Đệ Quy Bằng Vòng Lặp và Biến Trạng Thái

Tiết kiệm bộ nhớ: Không cần lưu trữ trạng thái gọi hàm trong ngăn xếp. Cải thiện hiệu suất: Giảm chi phí liên quan đến việc gọi hàm và thao tác ngăn xếp. Tránh lỗi do đệ quy sâu: Đặc biệt hữu ích cho các bài toán có nhiều tầng đệ quy hoặc cần tính toán số lượng lớn. Dễ đọc và dễ kiểm soát: Vòng lặp và biến trạng thái giúp mã dễ đọc và kiểm soát hơn so với đệ quy phức tạp.

### 6.7.5. Khi Nào Nên Dùng Vòng Lặp và Biến Trạng Thái Thay Vì Đệ Quy?

Khi bài toán có thể biểu diễn một cách rõ ràng bằng vòng lặp: Các bài toán mà số bước có thể xác định trước hoặc có thể diễn tả bằng vòng lặp hữu hạn. Khi lo ngại về hiệu suất hoặc bộ nhớ: Đối với các bài toán lớn hoặc đệ quy sâu, vòng lặp thường hiệu quả hơn về mặt sử dụng tài nguyên. Khi muốn tránh lỗi tràn ngăn xếp: Đặc biệt quan trọng trong các hệ thống hạn chế bộ nhớ hoặc môi trường yêu cầu tính ổn định cao.

Tóm lại : Việc khử đệ quy sử dụng vòng lặp và biến trạng thái trung gian là một kỹ thuật hiệu quả giúp cải thiện hiệu suất và tránh các vấn đề liên quan đến sử dụng đệ quy như lỗi tràn ngăn xếp. Đặc biệt trong các bài toán đệ quy tương hỗ hoặc các bài toán có thể dễ dàng biểu diễn bằng vòng lặp, việc sử dụng vòng lặp và biến trạng thái trung gian là một lựa chọn tối ưu hơn cả.

## 6.8. Khử đệ quy nâng cao - Dùng Vòng Lặp và Mảng

Việc khử đệ quy bằng cách sử dụng vòng lặp và mảng (hoặc các cấu trúc dữ liệu tuyến tính khác) giúp cải thiện hiệu suất và tránh lỗi tràn ngăn xếp. Dưới đây là

một số ví dụ cụ thể hơn để minh họa cho các tình huống khác nhau mà kỹ thuật này có thể được áp dụng hiệu quả.

### 6.8.1. Bài Toán In Danh Sách Liên Kết Đơn Theo Thứ Tự Ngược

Đề bài: Bạn có một danh sách liên kết đơn (singly linked list) và bạn muốn in các phần tử của nó theo thứ tự ngược lại. Việc sử dụng đệ quy hoặc ngăn xếp có thể thực hiện được nhưng không tối ưu về hiệu suất khi danh sách quá lớn. Thay vào đó, bạn có thể sử dụng một mảng để lưu trữ các phần tử trong quá trình duyệt danh sách và sau đó in chúng từ cuối về đầu.

Ví dụ 1: In Danh Sách Liên Kết Đơn Theo Thứ Tự Ngược Bằng Vòng Lặp và Mảng

Cấu trúc danh sách liên kết đơn:

```
#include <iostream>
#include <vector> // Sử dụng vector để lưu trữ tạm thời các phần tử
using namespace std;

struct Node {
    int data;
    Node* next;
};

// Hàm in danh sách liên kết đơn theo thứ tự ngược sử dụng mảng
void printReverse(Node* head) {
    vector<int> elements;  // Mảng (vector) để lưu trữ các phần tử của danh sách

    // Bước 1: Duyệt qua danh sách và lưu các phần tử vào mảng
    Node* current = head;
    while (current != nullptr) {
        elements.push_back(current->data);  // Lưu trữ dữ liệu của node vào vecto
        current = current->next;  // Tiến đến phần tử tiếp theo
    }

    // Bước 2: In các phần tử từ cuối mảng về đầu
    for (int i = elements.size() - 1; i >= 0; i--) {
        cout << elements[i] << " ";  // In giá trị phần tử
    }
```

```
}

int main() {
    // Tạo danh sách liên kết đơn
    Node* head = new Node{1, nullptr};
    head->next = new Node{2, nullptr};
    head->next->next = new Node{3, nullptr};
    head->next->next->next = new Node{4, nullptr};

    // In danh sách theo thứ tự ngược
    printReverse(head);  // Output: 4 3 2 1

    // Giải phóng bộ nhớ
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

Giải thích: Sử dụng vector<int> elements để lưu trữ các phần tử: Chúng ta sử dụng một vector để lưu trữ tạm thời các phần tử của danh sách liên kết đơn trong quá trình duyệt từ đầu đến cuối. In các phần tử từ cuối về đầu: Sau khi lưu tất cả các phần tử vào mảng, chúng ta chỉ cần sử dụng một vòng lặp for để duyệt ngược từ cuối mảng về đầu và in ra các giá trị. Lợi ích của cách tiếp cận này:

Tránh lỗi tràn ngăn xếp: Không có lời gọi hàm đệ quy nào nên không có rủi ro tràn ngăn xếp khi danh sách quá lớn. Hiệu suất tốt hơn: Vòng lặp đơn giản hơn và ít tốn kém hơn về mặt chi phí xử lý so với việc sử dụng đệ quy.

### 6.8.2. Bài Toán Duyệt Cây Theo Chiều Sâu (Depth-First Traversal) Không Dùng Đệ Quy

Trong các bài toán duyệt cây nhị phân, chúng ta thường sử dụng đệ quy để thực hiện duyệt cây theo chiều sâu (DFS). Tuy nhiên, cách này có thể được thay thế bằng một vòng lặp và một mảng hoặc ngăn xếp thủ công.

Ví dụ 2: Duyệt Cây Nhị Phân Theo Chiều Sâu Bằng Vòng Lặp và Mảng Cấu

trúc cây nhị phân:

```cpp
#include <iostream>
#include <vector>
using namespace std;

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};

// Hàm duyệt cây theo chiều sâu (DFS) không dùng đệ quy
void dfs(TreeNode* root) {
    if (root == nullptr) return; // Trường hợp cơ sở: cây rỗng

    vector<TreeNode*> stack;  // Sử dụng vector như ngăn xếp
    stack.push_back(root);    // Đẩy gốc của cây vào ngăn xếp

    while (!stack.empty()) {
        TreeNode* current = stack.back(); // Lấy phần tử trên đỉnh ngăn xếp
        stack.pop_back(); // Loại bỏ phần tử trên đỉnh

        cout << current->data << " "; // Thăm node hiện tại

        // Đẩy node con phải vào ngăn xếp trước, rồi đến node con trái
        if (current->right != nullptr) stack.push_back(current->right);
        if (current->left != nullptr) stack.push_back(current->left);
    }
}

int main() {
    // Tạo cây nhị phân
    TreeNode* root = new TreeNode{1, nullptr, nullptr};
    root->left = new TreeNode{2, nullptr, nullptr};
    root->right = new TreeNode{3, nullptr, nullptr};
    root->left->left = new TreeNode{4, nullptr, nullptr};
    root->left->right = new TreeNode{5, nullptr, nullptr};
```

```
    // Duyệt cây theo chiều sâu
    dfs(root);  // Output: 1 2 4 5 3

    // Giải phóng bộ nhớ
    delete root->left->left;
    delete root->left->right;
    delete root->left;
    delete root->right;
    delete root;

    return 0;
}
```

Giải thích: Sử dụng vector<TreeNode*> stack để thay thế cho ngăn xếp đệ quy: Chúng ta sử dụng một mảng vector để lưu trữ các node cần thăm. Đẩy các node vào ngăn xếp theo thứ tự: Đẩy node con phải vào ngăn xếp trước để đảm bảo rằng node con trái sẽ được thăm trước (do tính chất LIFO của ngăn xếp).

Lợi ích: Kiểm soát luồng thực thi tốt hơn: Chúng ta có thể kiểm soát rõ ràng thứ tự các node được thăm mà không cần dựa vào ngăn xếp gọi hàm của hệ thống. Tránh lỗi tràn ngăn xếp: Kỹ thuật này tránh được rủi ro tràn ngăn xếp khi cây quá sâu.

### 6.8.3. Bài Toán Tìm Đường Đi Ngắn Nhất Trong Đồ Thị Bằng Duyệt Chiều Sâu

Trong bài toán tìm đường đi ngắn nhất trong một đồ thị không trọng số, chúng ta thường sử dụng BFS (Breadth-First Search). Tuy nhiên, nếu ta sử dụng DFS (Depth-First Search), ta có thể khử đệ quy bằng vòng lặp và một mảng để lưu trữ các đỉnh đã thăm.

Ví dụ 3: Tìm Đường Đi Ngắn Nhất Trong Đồ Thị Bằng Vòng Lặp và Mảng. Giả định: Đồ thị được biểu diễn dưới dạng danh sách kề.

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
```

```cpp
// Hàm BFS tìm đường đi ngắn nhất từ node nguồn s đến node đích d
bool bfs(vector<vector<int>>& graph, int s, int d) {
    int n = graph.size(); // Số đỉnh của đồ thị
    vector<bool> visited(n, false); // Mảng đánh dấu các đỉnh đã thăm

    queue<int> q; // Hàng đợi để duyệt các đỉnh
    q.push(s); // Đẩy đỉnh nguồn vào hàng đợi
    visited[s] = true; // Đánh dấu đỉnh nguồn đã thăm

    while (!q.empty()) {
        int node = q.front();
        q.pop();

        // Nếu tìm thấy đỉnh đích
        if (node == d) return true;

        // Duyệt qua các đỉnh kề với node
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                q.push(neighbor); // Đẩy đỉnh kề vào hàng đợi
                visited[neighbor] = true; // Đánh dấu đỉnh đã thăm
            }
        }
    }

    return false; // Không tìm thấy đường đi
}

int main() {
    int n = 6; // Số đỉnh
    vector<vector<int>> graph(n); // Danh sách kề

    // Định nghĩa đồ thị
    graph[0] = {1, 2};
    graph[1] = {0, 3};
    graph[2] = {0, 4};
    graph[3] = {1, 5};
```

```
    graph[4] = {2};
    graph[5] = {3};

    int source = 0, destination = 5;
    if (bfs(graph, source, destination))
        cout << "Có đường đi từ " << source << " đến " << destination << endl;
    else
        cout << "Không có đường đi từ " << source << " đến " << destination << endl;

    return 0;
}
```

Giải thích: Sử dụng queue<int> q để quản lý các đỉnh cần thăm: Hàng đợi giúp duyệt đồ thị theo chiều rộng, tuy nhiên việc này hoàn toàn có thể dùng vector và for để đạt được việc duyệt toàn bộ đỉnh của đồ thị. Kiểm soát tốt hơn các đỉnh đã thăm: Việc sử dụng mảng visited giúp chúng ta tránh các vòng lặp vô hạn và tối ưu hóa hiệu suất của thuật toán. Kết Luận và Phân Tích Chi Tiết Tối ưu hóa bằng vòng lặp và mảng giúp tránh được các chi phí không cần thiết liên quan đến đệ quy và ngăn xếp gọi hàm, đặc biệt khi làm việc với các cấu trúc dữ liệu lớn hoặc sâu như cây hoặc đồ thị. Kiểm soát tốt hơn luồng thực thi: Việc sử dụng vòng lặp và các biến trạng thái trung gian cho phép lập trình viên kiểm soát rõ ràng hơn luồng thực thi của chương trình, làm cho mã dễ đọc và dễ bảo trì hơn. Tránh lỗi tràn ngăn xếp: Sử dụng mảng hoặc vector để lưu trữ các phần tử thay vì đệ quy giúp tránh được lỗi tràn ngăn xếp, đặc biệt quan trọng trong các ứng dụng yêu cầu tính ổn định cao và hiệu suất tối ưu.

## 6.9. Kết Luận

Đệ quy là một kỹ thuật quan trọng trong lập trình, giúp giải quyết nhiều bài toán phức tạp một cách dễ dàng. Tuy nhiên, đệ quy cần được sử dụng một cách thận trọng để tránh gây ra lỗi tràn ngăn xếp (stack overflow) hoặc các vấn đề về hiệu suất.

## 6.10. Bài tập

### 6.10.1. Đệ quy Tuyến tính

**Bài tập 1: Tính tổng các số từ 1 đến n** Mô tả: Viết hàm đệ quy để tính tổng của các số từ 1 đến n. Gợi ý: sum(n) = n + sum(n-1) và sum(0) = 0.

**Bài tập 2: Tính dãy số Fibonacci** Mô tả: Viết hàm đệ quy để tính số Fibonacci thứ n. Gợi ý: fib(n) = fib(n-1) + fib(n-2), với fib(0) = 0 và fib(1) = 1.

**Bài tập 3: Tính ước chung lớn nhất (GCD) của hai số nguyên** Mô tả: Viết hàm đệ quy để tính ước chung lớn nhất của hai số nguyên a và b sử dụng thuật toán Euclid. Gợi ý: gcd(a, b) = gcd(b, a MOD b), với gcd(a, 0) = a.

**Bài tập 4:Tính tổng các chữ số của một số nguyên** Mô tả: Viết hàm đệ quy để tính tổng các chữ số của một số nguyên n. Gợi ý: sumDigits(n) = n

### 6.10.2. Đệ quy Phi tuyến (Non-linear Recursion)

**Bài tập 5: Tìm phần tử lớn nhất trong mảng** Mô tả: Viết hàm đệ quy để tìm phần tử lớn nhất trong một mảng số nguyên arr có kích thước n. Gợi ý: maxArray(arr, n) = max(arr[n-1], maxArray(arr, n-1)).

**Bài tập 6: In tất cả các hoán vị của một chuỗi** Mô tả: Viết hàm đệ quy để in tất cả các hoán vị của một chuỗi str. Gợi ý: Đổi chỗ các ký tự và gọi đệ quy cho phần còn lại của chuỗi.

**Bài tập 7: Tìm tất cả các tập con của một tập hợp** Mô tả: Viết hàm đệ quy để tìm tất cả các tập con của một tập hợp. Gợi ý: Sử dụng backtracking để tạo các tập con.

### 6.10.3. Đệ quy Nhị phân (Binary Recursion)

**Bài tập 8: Tìm kiếm nhị phân (Binary Search)** Mô tả: Viết hàm đệ quy để tìm kiếm nhị phân một phần tử x trong mảng đã được sắp xếp arr có kích thước n. Gợi ý: So sánh x với phần tử ở giữa và gọi đệ quy cho nửa còn lại.

**Bài tập 9: Tính số lũy thừa** Mô tả: Viết hàm đệ quy để tính x mũ n . Gợi ý: Sử dụng đệ quy để tính x mũ (n/2) và nhân kết quả với chính nó hoặc thêm một lần x nữa nếu n là lẻ.

### 6.10.4. Đệ quy Tương hỗ (Mutual Recursion)

**Bài tập 10: Chuỗi gọi đệ quy tương hỗ** Mô tả: Viết hai hàm đệ quy tương hỗ isEven(n) và isOdd(n) để kiểm tra xem một số nguyên n là chẵn hay lẻ. Gợi ý: isEven(n) = n == 0 ? true : isOdd(n-1) và isOdd(n) = n == 0 ? false : isEven(n-1).

**Bài tập 11: Hàm Ackermann** Mô tả: Viết hàm đệ quy để tính giá trị của hàm Ackermann, một hàm đệ quy tương hỗ thường được sử dụng trong lý thuyết tính toán. Gợi ý: A(m, n) = n + 1 nếu m == 0, A(m-1, 1) nếu m > 0 và n == 0, A(m-1, A(m, n-1)) nếu m > 0 và n > 0.

### 6.10.5. Đệ quy Backtracking

**Bài tập 12: Giải bài toán N-queens** Mô tả: Viết hàm đệ quy để giải bài toán N-queens, tức là đặt N quân hậu trên bàn cờ N x N sao cho không quân nào ăn được nhau. Gợi ý: Sử dụng backtracking để thử đặt quân hậu trên từng hàng và kiểm tra xem vị trí đó có an toàn không.

**Bài tập 13: Giải bài toán mê cung** Mô tả: Viết hàm đệ quy để tìm đường đi từ điểm bắt đầu đến điểm kết thúc trong một mê cung 2D. Gợi ý: Sử dụng backtracking để thử di chuyển theo các hướng khác nhau và đánh dấu đường đi.

### 6.10.6. Đệ quy Tổng quát và Kết hợp (Generalized Recursion and Combination)

**Bài tập 14: Tính số tổ hợp C(n, k)** Mô tả: Viết hàm đệ quy để tính số tổ hợp C(n, k) (số cách chọn k phần tử từ n phần tử). Gợi ý: C(n, k) = C(n-1, k-1) + C(n-1, k) với C(n, 0) = C(n, n) = 1.

# Chương 7
# Thư viện STL



## 7.1. Giới thiệu

STL là viết tắt của Standard Template Library trong C++. Đây là một thư viện chuẩn mạnh mẽ bao gồm các cấu trúc dữ liệu và thuật toán có thể sử dụng lại (generic). STL được thiết kế để giúp lập trình viên thao tác với các tập hợp dữ liệu và thực hiện các thuật toán trên chúng một cách hiệu quả và dễ dàng hơn.

Ý nghĩa: STL cung cấp một tập hợp các công cụ mạnh mẽ cho phép lập trình viên C++ xử lý các cấu trúc dữ liệu và thuật toán một cách trừu tượng và linh hoạt. Thay vì viết lại mã nguồn cho các cấu trúc dữ liệu cơ bản như danh sách liên kết (linked list), ngăn xếp (stack), hàng đợi (queue), hay vector, lập trình viên có thể sử dụng các thành phần của STL.

Mục đích sử dụng STL:

- Tái sử dụng mã nguồn: Cung cấp các cấu trúc dữ liệu và thuật toán chung, cho phép lập trình viên tái sử dụng mã nguồn đã được kiểm thử và tối ưu hóa.
- Giảm thiểu lỗi: Việc sử dụng các thành phần chuẩn giúp giảm thiểu lỗi do lập trình viên tự viết lại mã nguồn cho các cấu trúc và thuật toán phổ biến.
- Tăng hiệu quả lập trình: Giúp lập trình viên tập trung vào logic ứng dụng thay vì viết mã cho các cấu trúc dữ liệu và thuật toán cơ bản.

- Tăng tính linh hoạt: STL sử dụng các template cho phép làm việc với nhiều kiểu dữ liệu khác nhau mà không cần viết lại mã nguồn.

## 7.2. Các Thành Phần Chính của STL

STL bao gồm ba thành phần chính:

- Containers (Bộ chứa): Các cấu trúc dữ liệu khác nhau để lưu trữ và quản lý dữ liệu. Containers là các cấu trúc dữ liệu cho phép lưu trữ và quản lý các phần tử. STL cung cấp nhiều loại containers, mỗi loại được thiết kế để đáp ứng các yêu cầu khác nhau về hiệu suất và tính năng.
- Algorithms (Thuật toán): Các thuật toán tiêu chuẩn để thao tác với dữ liệu trong containers.
- Iterators (Bộ lặp): Các công cụ trỏ và duyệt qua các phần tử trong containers.

Các loại containers phổ biến trong STL:

- Sequence Containers (Bộ chứa tuần tự): Lưu trữ các phần tử theo một thứ tự cụ thể.
- vector: Mảng động, cho phép truy cập ngẫu nhiên nhanh.
- list: Danh sách liên kết đôi, cho phép chèn và xóa phần tử nhanh ở bất kỳ vị trí nào.
- deque: Mảng động hai đầu, cho phép chèn/xóa phần tử ở cả đầu và cuối nhanh chóng.
- array: Mảng có kích thước cố định.
- Associative Containers (Bộ chứa kết hợp): Lưu trữ các phần tử được sắp xếp hoặc không sắp xếp và truy xuất dựa trên khóa.
- set: Tập hợp các phần tử không trùng lặp được sắp xếp.
- map: Cặp khóa-giá trị được sắp xếp.
- multiset: Tập hợp các phần tử có thể trùng lặp được sắp xếp.
- multimap: Cặp khóa-giá trị có thể trùng lặp được sắp xếp.

- Unordered Containers (Bộ chứa không sắp xếp): Lưu trữ các phần tử không sắp xếp, nhưng có thể truy cập nhanh thông qua hàm băm.
- unordered_set: Tập hợp các phần tử không trùng lặp không sắp xếp.
- unordered_map: Cặp khóa-giá trị không sắp xếp.
- unordered_multiset: Tập hợp các phần tử có thể trùng lặp không sắp xếp.
- unordered_multimap: Cặp khóa-giá trị có thể trùng lặp không sắp xếp.

Ví dụ về container vector:

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> numbers; // Khai báo một vector lưu trữ số nguyên

    // Thêm phần tử vào vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Truy cập phần tử trong vector
    cout << "Phần tử đầu tiên: " << numbers[0] << endl;

    // Duyệt qua các phần tử trong vector
    for (int i = 0; i < numbers.size(); i++) {
        cout << numbers[i] << " ";
    }

    return 0;
}
```

### 7.2.1. Algorithms (Thuật toán)

Thuật toán là các hàm mẫu (template functions) thực hiện các phép toán chung trên các phần tử của container như sắp xếp, tìm kiếm, sao chép, và thay đổi dữ liệu.

Các thuật toán trong STL được thiết kế để làm việc với các container thông qua các bộ lặp (iterators).

Các thuật toán phổ biến trong STL:

**Tìm kiếm và sắp xếp:** find: Tìm kiếm một phần tử trong một dải phần tử.

sort: Sắp xếp một dải phần tử.

binary_search: Tìm kiếm nhị phân trong một dải phần tử đã sắp xếp.

**Sao chép và chuyển đổi:** copy: Sao chép một dải phần tử từ vị trí này đến vị trí khác.

transform: Chuyển đổi các phần tử bằng một hàm nhất định.

Thuật toán cho container đặc biệt:

unique: Loại bỏ các phần tử trùng lặp liên tiếp trong một dải phần tử đã sắp xếp.

Ví dụ về thuật toán sort và find:

```
#include <iostream>
#include <vector>
#include <algorithm> // Thư viện cho các thuật toán STL
using namespace std;

int main() {
    vector<int> numbers = {10, 20, 5, 15, 25};

    // Sắp xếp các phần tử trong vector
    sort(numbers.begin(), numbers.end());

    // Tìm kiếm phần tử 15 trong vector
    if (find(numbers.begin(), numbers.end(), 15) != numbers.end()) {
        cout << "Phần tử 15 được tìm thấy." << endl;
    } else {
        cout << "Phần tử 15 không tồn tại." << endl;
    }

    // In ra các phần tử đã sắp xếp
    for (int num : numbers) {
```

```
        cout << num << " ";
    }

    return 0;
}
```

### 7.2.2. Iterators (Bộ lặp)

Iterators là các đối tượng đóng vai trò như con trỏ để duyệt qua các phần tử trong containers. Chúng cung cấp một giao diện chung để thao tác với các phần tử trong container mà không cần biết cụ thể về kiểu dữ liệu của container đó.

Các loại iterators phổ biến:

- Input Iterator: Duyệt qua các phần tử theo một hướng (từ đầu đến cuối).
- Output Iterator: Duyệt qua các phần tử để ghi giá trị.
- Forward Iterator: Duyệt qua các phần tử theo một hướng và hỗ trợ đọc/ghi.
- Bidirectional Iterator: Duyệt qua các phần tử theo cả hai hướng (từ đầu đến cuối và ngược lại).
- Random Access Iterator: Truy cập ngẫu nhiên các phần tử, như con trỏ.

Ví dụ về sử dụng iterator với vector:

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> numbers = {10, 20, 30, 40, 50};

    // Sử dụng iterator để duyệt qua các phần tử
    vector<int>::iterator it;
    for (it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " "; // Dereference iterator để lấy giá trị phần tử
    }

    return 0;
}
```

## 7.3. Cách Sử Dụng STL Hiệu Quả

### 7.3.1. Lựa chọn container phù hợp

Khi cần truy cập ngẫu nhiên nhanh chóng: Sử dụng vector hoặc deque. Khi cần chèn/xóa phần tử thường xuyên: Sử dụng list hoặc deque. Khi cần tìm kiếm nhanh và duy trì thứ tự: Sử dụng set hoặc map. Khi không cần duy trì thứ tự và cần truy cập nhanh: Sử dụng unordered_set hoặc unordered_map.

### 7.3.2. Sử dụng thuật toán STL

Sử dụng các thuật toán STL có sẵn thay vì tự viết lại các thuật toán cơ bản như sắp xếp, tìm kiếm. Kết hợp các thuật toán với các container thông qua iterators để thực hiện các thao tác phức tạp.

### 7.3.3. Sử dụng Iterators một cách linh hoạt

Begin/End Iterators: Sử dụng begin() và end() để duyệt qua toàn bộ container. Reverse Iterators: Sử dụng rbegin() và rend() để duyệt qua container theo thứ tự ngược. Const Iterators: Sử dụng cbegin() và cend() khi không cần thay đổi các phần tử của container.

## 7.4. Associative Containers

Associative Containers là một phần của thư viện chuẩn C++ (STL - Standard Template Library), được sử dụng để lưu trữ các phần tử theo cách mà mỗi phần tử có thể nhanh chóng được truy cập thông qua một khóa (key). Các Associative Containers cung cấp các phương thức để chèn, xóa và tìm kiếm phần tử hiệu quả.

Các loại Associative Containers:

- std::set và std::multiset
- std::set: Lưu trữ các phần tử không trùng lặp và tự động sắp xếp các phần tử theo một thứ tự cụ thể (thường là thứ tự tăng dần).
- std::multiset: Tương tự như std::set, nhưng cho phép lưu trữ nhiều phần tử có giá trị giống nhau.
- std::map và std::multimap
- std::map: Lưu trữ các cặp khóa-giá trị duy nhất. Các phần tử được sắp xếp theo thứ tự của khóa.

- std::multimap: Tương tự như std::map, nhưng cho phép nhiều phần tử có cùng khóa.
- std::unordered_set và std::unordered_multiset
- std::unordered_set: Lưu trữ các phần tử không trùng lặp nhưng không đảm bảo thứ tự của các phần tử.
- std::unordered_multiset: Tương tự như std::unordered_set, nhưng cho phép nhiều phần tử có giá trị giống nhau.
- std::unordered_map và std::unordered_multimap
- std::unordered_map: Lưu trữ các cặp khóa-giá trị duy nhất mà không cần sắp xếp các khóa.
- std::unordered_multimap: Tương tự như std::unordered_map, nhưng cho phép nhiều phần tử có cùng khóa.

Sự khác biệt giữa các loại Associative Containers:

- set và map: Được triển khai dưới dạng cây nhị phân tìm kiếm cân bằng (thường là Red-Black Tree), có tính năng sắp xếp tự động các phần tử. Thời gian truy cập, chèn và xóa phần tử là O(log n).
- unordered_set và unordered_map: Được triển khai dưới dạng bảng băm (hash table), không đảm bảo thứ tự các phần tử, nhưng truy cập, chèn và xóa phần tử trung bình là O(1).

Khi nào nên sử dụng Associative Containers:

- Sử dụng set hoặc unordered_set khi:
  - Cần lưu trữ các phần tử duy nhất.
  - Thao tác chèn, xóa và tìm kiếm cần thực hiện nhanh chóng.
  - Không quan tâm đến thứ tự phần tử (sử dụng unordered_set) hoặc cần thứ tự tự động (sử dụng set).
- Sử dụng map hoặc unordered_map khi:
  - Cần lưu trữ các cặp khóa-giá trị, trong đó mỗi khóa là duy nhất.
  - Cần thực hiện nhanh chóng các thao tác truy cập, chèn và xóa phần tử dựa trên khóa.

  - Không cần thứ tự của các khóa (sử dụng unordered_map) hoặc cần thứ tự tự động (sử dụng map).

- Sử dụng multiset hoặc unordered_multiset khi:
  - Cần lưu trữ các phần tử có thể trùng lặp.
  - Cần thực hiện các thao tác chèn, xóa và tìm kiếm nhanh chóng.
  - Không cần sắp xếp phần tử (sử dụng unordered_multiset) hoặc cần sắp xếp tự động (sử dụng multiset).

- Sử dụng multimap hoặc unordered_multimap khi:
  - Cần lưu trữ các cặp khóa-giá trị mà khóa có thể trùng lặp.
  - Các thao tác chèn, xóa và tìm kiếm dựa trên khóa cần được thực hiện nhanh chóng.
  - Không quan tâm thứ tự của các khóa (sử dụng unordered_multimap) hoặc cần thứ tự tự động (sử dụng multimap).

Ví dụ về sử dụng Associative Containers:

Ví dụ về std::set:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> numbers = {1, 2, 3, 4, 5};

    // In ra tất cả các phần tử
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Chèn một phần tử
    numbers.insert(6);

    // Xóa một phần tử
    numbers.erase(3);
```

```
    // Kiểm tra một phần tử có trong set hay không
    if (numbers.find(2) != numbers.end()) {
        std::cout << "Phan tu 2 co trong set." << std::endl;
    }

    return 0;
}
```

Ví dụ về std::map:

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> phonebook;
    phonebook["Alice"] = 123456;
    phonebook["Bob"] = 987654;

    // Duyệt qua map và in ra các cặp khóa-giá trị
    for (const auto& [key, value] : phonebook) {
        std::cout << key << ": " << value << std::endl;
    }

    // Tìm một khóa trong map
    if (phonebook.find("Alice") != phonebook.end()) {
        std::cout << "Alice's number is: " << phonebook["Alice"] << std::endl;
    }

    return 0;
}
```

Ví dụ về std::unordered_map:

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<char, int> letter_count;
```

```
    std::string word = "hello";

    for (char c : word) {
        letter_count[c]++;
    }

    // In ra số lần xuất hiện của từng ký tự
    for (const auto& [key, value] : letter_count) {
        std::cout << key << ": " << value << std::endl;
    }

    return 0;
}
```

Associative Containers là một công cụ mạnh mẽ trong C++ cho phép xử lý dữ liệu hiệu quả với các thao tác như chèn, xóa và tìm kiếm. Tùy vào từng tình huống cụ thể và yêu cầu về hiệu suất mà lựa chọn loại container phù hợp để tối ưu hóa chương trình.

## 7.5. Unordered Associative Containers

Unordered Associative Containers là một nhóm các container trong Thư viện Chuẩn C++ (STL) sử dụng bảng băm (hash table) để quản lý dữ liệu. Các container này không duy trì thứ tự của các phần tử mà thay vào đó cho phép truy cập và thao tác nhanh chóng trên các phần tử thông qua khóa.

Các loại Unordered Associative Containers:

- std::unordered_set
  - Lưu trữ các phần tử duy nhất không có thứ tự cụ thể.
  - Cung cấp các thao tác chèn, xóa, và tìm kiếm phần tử với thời gian trung bình O(1).
  - Không cho phép các phần tử trùng lặp.
- std::unordered_multiset
  - Tương tự như std::unordered_set, nhưng cho phép các phần tử trùng lặp.
  - Thời gian trung bình cho các thao tác chèn, xóa, và tìm kiếm là O(1).

- std::unordered_map
  - Lưu trữ các cặp khóa-giá trị không có thứ tự cụ thể.
  - Cung cấp các thao tác chèn, xóa, và tìm kiếm dựa trên khóa với thời gian trung bình O(1).
  - Mỗi khóa là duy nhất, nhưng giá trị có thể trùng lặp.
- std::unordered_multimap
  - Tương tự như std::unordered_map, nhưng cho phép nhiều cặp có cùng khóa.
  - Thời gian trung bình cho các thao tác chèn, xóa, và tìm kiếm là O(1).

Đặc điểm của Unordered Associative Containers:

- Không sắp xếp: Các phần tử không được sắp xếp theo thứ tự cụ thể nào. Thứ tự phụ thuộc vào hàm băm được sử dụng và trạng thái hiện tại của bảng băm.
- Tốc độ truy cập nhanh: Các thao tác như chèn, xóa và tìm kiếm phần tử có thời gian trung bình là O(1), vì các phần tử được đặt trong bảng băm dựa trên giá trị băm của khóa.
- Dễ dàng mở rộng: Khi số lượng phần tử vượt quá một ngưỡng nhất định, bảng băm sẽ tự động mở rộng để giữ cho tốc độ truy cập trung bình vẫn là O(1).

Ưu điểm và Nhược điểm của Unordered Associative Containers:
Ưu điểm:

- Hiệu suất cao: Các thao tác truy vấn như tìm kiếm, chèn và xóa thường rất nhanh (O(1) trung bình).
- Tiện dụng cho các ứng dụng yêu cầu truy cập nhanh với khóa: Đặc biệt hữu ích khi bạn chỉ cần kiểm tra sự tồn tại của phần tử hoặc thao tác với dữ liệu không cần sắp xếp.

Nhược điểm:

- Không duy trì thứ tự phần tử: Nếu bạn cần duy trì thứ tự, nên sử dụng các container có sắp xếp như std::set hoặc std::map.
- Chi phí bộ nhớ cao hơn: Sử dụng bảng băm có thể tốn nhiều bộ nhớ hơn, đặc biệt khi cần xử lý xung đột (collisions).

- Hiệu suất có thể giảm khi xảy ra xung đột nhiều: Nếu hàm băm không hiệu quả hoặc có quá nhiều xung đột, hiệu suất có thể giảm xuống O(n).

Ví dụ về Unordered Associative Containers:

Ví dụ về std::unordered_set:

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> mySet = {1, 2, 3, 4, 5};

    // Thêm phần tử vào unordered_set
    mySet.insert(6);

    // Xóa phần tử khỏi unordered_set
    mySet.erase(3);

    // Kiểm tra sự tồn tại của phần tử
    if (mySet.find(2) != mySet.end()) {
        std::cout << "Phan tu 2 ton tai trong unordered_set." << std::endl;
    }

    // Duyệt qua các phần tử trong unordered_set
    for (const int &elem : mySet) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Ví dụ về std::unordered_map:

```
#include <iostream>
#include <unordered_map>

int main() {
```

```
    std::unordered_map<std::string, int> myMap;
    myMap["apple"] = 2;
    myMap["banana"] = 3;
    myMap["orange"] = 5;

    // Truy cập giá trị thông qua khóa
    std::cout << "So luong apple: " << myMap["apple"] << std::endl;

    // Kiểm tra sự tồn tại của khóa
    if (myMap.find("banana") != myMap.end()) {
        std::cout << "Banana ton tai trong unordered_map." << std::endl;
    }

    // Duyệt qua các phần tử trong unordered_map
    for (const auto &[key, value] : myMap) {
        std::cout << key << ": " << value << std::endl;
    }

    return 0;
}
```

Ví dụ về std::unordered_multiset:

```
#include <iostream>
#include <unordered_multiset>

int main() {
    std::unordered_multiset<int> myMultiSet = {1, 2, 2, 3, 3, 3};

    // Thêm phần tử vào unordered_multiset
    myMultiSet.insert(4);

    // Đếm số lần xuất hiện của một phần tử
    std::cout << "So lan xuat hien cua phan tu 2: " << myMultiSet.count(2) << std

    // Duyệt qua các phần tử trong unordered_multiset
    for (const int &elem : myMultiSet) {
        std::cout << elem << " ";
```

```
    }
    std::cout << std::endl;

    return 0;
}
```

Khi nào nên sử dụng Unordered Associative Containers?

- Khi cần truy cập nhanh đến các phần tử bằng khóa: Khi bạn cần thực hiện thao tác tìm kiếm, chèn, hoặc xóa phần tử một cách nhanh chóng mà không quan tâm đến thứ tự các phần tử.
- Khi số lượng phần tử lớn: Với số lượng phần tử lớn và yêu cầu truy cập nhanh, unordered containers thường cung cấp hiệu suất tốt hơn do khả năng truy cập trung bình O(1).
- Khi thứ tự của các phần tử không quan trọng: Nếu thứ tự lưu trữ không quan trọng, việc sử dụng unordered containers có thể mang lại hiệu suất tốt hơn.

Unordered Associative Containers là công cụ mạnh mẽ trong C++ để xử lý các tập hợp phần tử và cặp khóa-giá trị không sắp xếp. Chúng cung cấp các thao tác truy vấn nhanh chóng với thời gian trung bình O(1), làm cho chúng trở nên hữu ích trong nhiều ứng dụng yêu cầu hiệu suất cao. Tuy nhiên, cần cẩn thận với việc sử dụng bộ nhớ và hiệu suất trong trường hợp xảy ra xung đột hàm băm.

## 7.6. Lựa chọn container phù hợp

Để sử dụng hiệu quả các container trong C++ Standard Template Library (STL), bạn cần chọn container phù hợp với yêu cầu của bài toán. Mỗi container có đặc điểm riêng về hiệu suất, cú pháp, và tính năng, vì vậy việc hiểu rõ những đặc điểm này sẽ giúp bạn tối ưu hóa chương trình của mình. Dưới đây là một số gợi ý về cách chọn và tối ưu sử dụng các container trong những tình huống cụ thể:

### 7.6.1. std::vector

Đặc điểm:

- Bộ nhớ liên tục (contiguous memory allocation).
- Cho phép truy cập ngẫu nhiên với thời gian O(1).
- Thêm/xóa phần tử ở cuối với thời gian O(1) trung bình (amortized time).

- Thêm/xóa phần tử ở đầu hoặc giữa với thời gian O(n).

Khi nào nên dùng:

- Khi bạn cần một mảng động, kích thước có thể thay đổi linh hoạt.
- Khi cần truy cập ngẫu nhiên nhanh chóng tới các phần tử.
- Khi bạn chủ yếu thêm hoặc xóa phần tử ở cuối.

Tối ưu hóa:

- Sử dụng reserve: Nếu biết trước số lượng phần tử, sử dụng reserve để giảm thiểu số lần cấp phát lại bộ nhớ.
- Tránh sử dụng với dữ liệu lớn nếu cần thêm/xóa thường xuyên ở đầu hoặc giữa vector do chi phí di chuyển phần tử.

Ví dụ:

```
std::vector<int> data;
data.reserve(1000); // Tối ưu hóa cấp phát bộ nhớ cho 1000 phần tử
for (int i = 0; i < 1000; ++i) {
    data.push_back(i);
}
```

### 7.6.2. std::deque

Đặc điểm:

- Hỗ trợ thêm/xóa phần tử ở cả hai đầu với thời gian O(1).
- Không đảm bảo bộ nhớ liên tục.

Khi nào nên dùng:

- Khi cần thêm/xóa phần tử nhanh chóng ở cả đầu và cuối.
- Khi không yêu cầu bộ nhớ liên tục.

Tối ưu hóa:

- Sử dụng std::deque khi cần queue hoặc stack với yêu cầu thêm/xóa cả hai đầu.

- Tránh dùng khi cần truy cập ngẫu nhiên thường xuyên.

Ví dụ:

```
std::deque<int> d;
d.push_front(1);
d.push_back(2);
d.pop_front();
d.pop_back();
```

### 7.6.3. std::list và std::forward_list

Đặc điểm:

- std::list là danh sách liên kết đôi, std::forward_list là danh sách liên kết đơn.
- Thêm/xóa phần tử tại vị trí bất kỳ với thời gian O(1) nếu đã có iterator.
- Truy cập ngẫu nhiên có thời gian O(n).

Khi nào nên dùng:

- Khi cần thêm/xóa phần tử tại các vị trí khác nhau thường xuyên mà không cần truy cập ngẫu nhiên.
- std::forward_list phù hợp khi chỉ cần truy cập theo một chiều và yêu cầu bộ nhớ tối ưu hơn std::list.

Tối ưu hóa: Sử dụng khi cần thao tác chèn/xóa trong danh sách với chi phí thấp. Tránh sử dụng nếu yêu cầu truy cập ngẫu nhiên nhiều. Ví dụ:

```
std::list<int> my_list;
my_list.push_back(1);
my_list.push_front(2);
auto it = my_list.begin();
my_list.insert(it, 3);
```

### 7.6.4. std::set và std::multiset

Đặc điểm:

- Lưu trữ các phần tử đã được sắp xếp.

- Không cho phép phần tử trùng lặp (std::set), cho phép trùng lặp (std::multiset).
- Thêm, xóa và tìm kiếm có thời gian O(log n).

Khi nào nên dùng:

- Khi cần lưu trữ các phần tử duy nhất và đã được sắp xếp.
- Khi cần thao tác tìm kiếm và xóa nhanh chóng mà không cần truy cập ngẫu nhiên.

Tối ưu hóa:

- Sử dụng khi cần lưu trữ các phần tử không trùng lặp với thứ tự sắp xếp.
- Tránh dùng khi không cần bảo đảm sự duy nhất và thứ tự của phần tử.

Ví dụ:

```
std::set<int> my_set;
my_set.insert(1);
my_set.insert(2);
my_set.insert(1); // Không có tác dụng vì 1 đã tồn tại
```

### 7.6.5. std::unordered_set và std::unordered_multiset

Đặc điểm:

- Lưu trữ các phần tử không theo thứ tự.
- Không cho phép phần tử trùng lặp (std::unordered_set), cho phép trùng lặp (std::unordered_multiset).
- Thêm, xóa và tìm kiếm có thời gian O(1) trung bình.

Khi nào nên dùng:

- Khi cần lưu trữ các phần tử duy nhất mà không quan tâm đến thứ tự.
- Khi cần thao tác thêm, xóa và tìm kiếm nhanh chóng.

Tối ưu hóa:

- Sử dụng khi cần thao tác trên tập hợp các phần tử không trùng lặp và không cần thứ tự.
- Tránh dùng khi cần sắp xếp phần tử.

Ví dụ:

```
std::unordered_set<int> my_unordered_set;
my_unordered_set.insert(1);
my_unordered_set.insert(2);
my_unordered_set.insert(1); // Không có tác dụng vì 1 đã tồn tại.
```

### 7.6.6. std::map và std::multimap

Đặc điểm:

- Lưu trữ các cặp khóa-giá trị đã được sắp xếp.
- Không cho phép các khóa trùng lặp (std::map), cho phép trùng lặp (std::multimap).
- Thêm, xóa và tìm kiếm theo khóa có thời gian O(log n).

Khi nào nên dùng:

- Khi cần lưu trữ dữ liệu dưới dạng cặp khóa-giá trị và đảm bảo các khóa duy nhất (std::map).
- Khi cần lưu trữ dữ liệu với nhiều giá trị cho cùng một khóa (std::multimap).

Tối ưu hóa:

- Sử dụng khi cần duy trì thứ tự của các khóa và cần thao tác tìm kiếm, chèn, xóa nhanh chóng.
- Tránh dùng khi không cần bảo đảm sự duy nhất hoặc thứ tự của khóa.

Ví dụ:

```
std::map<std::string, int> my_map;
my_map["apple"] = 1;
my_map["banana"] = 2;
my_map["apple"] = 3; // Cập nhật giá trị của "apple".
```

### 7.6.7. std::unordered_map và std::unordered_multimap

Đặc điểm:

- Lưu trữ các cặp khóa-giá trị không theo thứ tự.
- Không cho phép các khóa trùng lặp (std::unordered_map), cho phép trùng lặp (std::unordered_multimap).
- Thêm, xóa và tìm kiếm theo khóa có thời gian O(1) trung bình.

Khi nào nên dùng:

- Khi cần lưu trữ dữ liệu dưới dạng cặp khóa-giá trị mà không cần thứ tự và khóa duy nhất.
- Khi cần thao tác thêm, xóa và tìm kiếm nhanh chóng.

Tối ưu hóa:

- Sử dụng khi không quan tâm đến thứ tự của các khóa và cần hiệu suất cao cho thao tác tìm kiếm, chèn, xóa.
- Tránh dùng khi cần duy trì thứ tự hoặc cần thao tác duyệt qua các khóa theo thứ tự.

Ví dụ:

```
std::unordered_map<std::string, int> my_unordered_map;
my_unordered_map["apple"] = 1;
my_unordered_map["banana"] = 2;
```

- Chọn container phù hợp dựa trên yêu cầu cụ thể của bài toán
- Nếu cần thao tác thêm/xóa/truy cập nhanh và không cần thứ tự: std::unordered_set hoặc std::unordered_map.
- Nếu cần thứ tự sắp xếp tự nhiên và thao tác thêm/xóa/tìm kiếm nhanh: std::set, std::map.
- Nếu cần lưu trữ danh sách động và truy cập ngẫu nhiên nhanh: std::vector.
- Nếu cần thao tác thêm/xóa ở đầu hoặc cuối: std::deque.

- Nếu cần thêm/xóa ở các vị trí tùy ý mà không cần truy cập ngẫu nhiên: std::list.

- Hiểu rõ yêu cầu bài toán và lựa chọn container phù hợp sẽ giúp tối ưu hóa hiệu suất của chương trình, giảm thiểu thời gian và bộ nhớ sử dụng.

## 7.7. Kết Luận

STL là một phần quan trọng của C++ giúp tăng năng suất lập trình và giảm thiểu lỗi bằng cách cung cấp các cấu trúc dữ liệu và thuật toán tiêu chuẩn. Việc sử dụng STL không chỉ giúp đơn giản hóa mã nguồn mà còn đảm bảo hiệu suất cao và độ tin cậy trong các ứng dụng C++.

## 7.8. Bài tập và Ví dụ Thực hành

### 7.8.1. Bài tập 1

Sử dụng STL để tạo một danh sách liên kết lưu trữ tên của 5 thành phố và in chúng ra theo thứ tự ngược.

### 7.8.2. Bài tập 2

Sử dụng set để lưu trữ các số nguyên duy nhất từ một mảng và in ra các số đã được sắp xếp.

### 7.8.3. Bài tập 3

Sử dụng unordered_map để đếm số lần xuất hiện của mỗi từ trong một đoạn văn.

Dưới đây là một số bài tập liên quan đến các loại containers trong C++ (cả Associative và Unordered Associative Containers) nhằm giúp bạn luyện tập và hiểu rõ hơn về cách sử dụng chúng.

### 7.8.4. Bài tập với Associative Containers

**Bài tập 1**

Sử dụng std::map để đếm số lần xuất hiện của mỗi từ trong một đoạn văn bản.

**Mô tả:** Viết chương trình đọc một đoạn văn bản từ người dùng, sau đó sử dụng std::map để đếm và hiển thị số lần xuất hiện của mỗi từ.

**Gợi ý:** Bạn có thể sử dụng std::istringstream để tách từ trong đoạn văn bản và lưu chúng vào std::map<std::string, int>.

**Bài tập 2**

Quản lý danh sách sinh viên và điểm số bằng std::map.

**Mô tả:** Viết chương trình quản lý danh sách sinh viên với tên và điểm số tương ứng. Cho phép người dùng thêm sinh viên mới, cập nhật điểm số và hiển thị danh sách sinh viên.

**Gợi ý:** Sử dụng std::map<std::string, int> để lưu tên sinh viên và điểm số.

**Bài tập 3**

Tìm kiếm khóa gần nhất trong std::set.

**Mô tả:** Viết chương trình chèn một số giá trị vào std::set<int> và sau đó yêu cầu người dùng nhập một giá trị. Tìm khóa lớn nhất nhỏ hơn hoặc bằng giá trị đã nhập (nếu có).

**Gợi ý:** Sử dụng phương thức std::set::lower_bound() để tìm khóa gần nhất.

**Bài tập 4**

Sử dụng std::multimap để lưu trữ một từ điển từ đồng nghĩa.

**Mô tả:** Viết chương trình sử dụng std::multimap để lưu trữ một từ điển từ đồng nghĩa. Cho phép người dùng thêm các cặp từ đồng nghĩa và tìm kiếm từ đồng nghĩa dựa trên từ đã cho. **Gợi ý:** Sử dụng std::multimap<std::string, std::string> để lưu trữ từ điển.

### 7.8.5. Bài tập với Unordered Associative Containers

1. Kiểm tra tính duy nhất của từ trong một câu sử dụng std::unordered_set.
    - Mô tả: Viết chương trình kiểm tra xem tất cả các từ trong một câu do người dùng nhập vào có là duy nhất hay không.
    - Gợi ý: Sử dụng std::unordered_set<std::string> để lưu trữ các từ và kiểm tra tính duy nhất.
2. Lưu trữ thông tin sản phẩm và giá bằng std::unordered_map.
    - Mô tả: Viết chương trình lưu trữ thông tin sản phẩm (tên sản phẩm và giá) trong một cửa hàng. Cho phép người dùng thêm sản phẩm mới, cập nhật giá sản phẩm và tìm kiếm giá của sản phẩm theo tên.

- Gợi ý: Sử dụng std::unordered_map<std::string, double> để lưu trữ thông tin sản phẩm và giá.

3. Tạo danh sách bạn bè không trùng lặp bằng std::unordered_multiset.

   - Mô tả: Viết chương trình cho phép người dùng nhập danh sách bạn bè, trong đó một số người bạn có thể xuất hiện nhiều lần. In ra danh sách bạn bè không trùng lặp và số lần xuất hiện của mỗi người bạn.
   - Gợi ý: Sử dụng std::unordered_multiset<std::string> để lưu trữ tên bạn bè và đếm số lần xuất hiện.

4. Sắp xếp từ theo độ dài sử dụng std::unordered_map.

   - Mô tả: Viết chương trình nhận vào một câu và in ra các từ trong câu được sắp xếp theo độ dài, sử dụng std::unordered_map.
   - Gợi ý: Sử dụng std::unordered_map<int, std::vector<std::string» để lưu trữ các từ theo độ dài của chúng.

### 7.8.6. Bài tập kết hợp nhiều loại Containers

1. Lưu trữ dữ liệu điểm thi của học sinh sử dụng kết hợp std::map và std::unordered_set.

   - Mô tả: Viết chương trình để lưu trữ dữ liệu điểm thi của học sinh trong một kỳ thi. Tên học sinh và điểm của họ được lưu trong std::map, trong khi các môn thi duy nhất được lưu trong std::unordered_set.
   - Gợi ý: Sử dụng std::map<std::string, int> cho dữ liệu điểm và std::unordered_set<std::string> cho danh sách môn thi.

2. Xây dựng một hệ thống quản lý sách trong thư viện sử dụng std::unordered_map và std::set.

   - Mô tả: Viết chương trình để xây dựng một hệ thống quản lý sách trong thư viện. Mỗi sách có một mã ISBN duy nhất và danh sách các tác giả. Dữ liệu sách được lưu trữ trong std::unordered_map, trong khi tên tác giả được lưu trữ trong std::set.
   - Gợi ý: Sử dụng std::unordered_map<std::string, std::set<std::string» để lưu trữ mã ISBN và danh sách tác giả.

Lưu ý:

- Khi làm bài tập, hãy đảm bảo bạn hiểu cách sử dụng các phương thức và thuộc tính của từng container.
- Cố gắng thực hành nhiều với các container khác nhau để làm quen với cú pháp và đặc tính hiệu suất của chúng.
- Đọc kỹ tài liệu và tham khảo các ví dụ từ trang chủ của C++ hoặc các nguồn tài liệu uy tín để hiểu rõ hơn về mỗi container và cách tối ưu sử dụng chúng trong các tình huống cụ thể.
- Chúc bạn học tốt và hiểu sâu hơn về các loại container trong C++!

# Chương 8
# Kiểm thử trong C/C++



## 8.1. Giới thiệu

Kiểm thử (Testing) là một phần quan trọng trong phát triển phần mềm, giúp phát hiện và sửa lỗi trước khi phần mềm được phát hành. Trong lập trình C/C++, việc kiểm thử bao gồm nhiều kỹ thuật và phương pháp khác nhau để đảm bảo chất lượng mã nguồn.

## 8.2. Kiểm thử Đơn vị (Unit Testing)

Kiểm thử đơn vị tập trung vào kiểm tra từng đơn vị nhỏ nhất của chương trình, chẳng hạn như hàm hoặc lớp, để đảm bảo chúng hoạt động như mong đợi. Mỗi bài kiểm thử đơn vị thường bao gồm một hoặc nhiều trường hợp kiểm thử.

Ví dụ: Kiểm thử Đơn vị với assert trong C

Hàm assert được sử dụng để kiểm tra các điều kiện mà chúng ta mong muốn. Nếu điều kiện sai, chương trình sẽ dừng lại và thông báo lỗi.

```c
#include <assert.h>
#include <stdio.h>
```

```c
// Hàm cộng hai số nguyên
int add(int a, int b) {
    return a + b;
}

// Hàm kiểm thử đơn vị cho hàm add
void test_add() {
    assert(add(2, 3) == 5);
    assert(add(-1, 1) == 0);
    assert(add(0, 0) == 0);
    printf("All tests passed for add function!\n");
}

int main() {
    test_add();
    return 0;
}
```

Trong ví dụ này, nếu một trong các assert thất bại, chương trình sẽ dừng lại và in ra thông báo lỗi. Ngược lại, nếu tất cả các kiểm thử đều thành công, chương trình sẽ in ra thông báo "All tests passed for add function!".

Ví dụ: Kiểm thử Đơn vị với Google Test trong C++

Google Test (gtest) là một thư viện phổ biến để kiểm thử đơn vị trong C++.

```c
#include <gtest/gtest.h>

// Hàm cộng hai số nguyên
int add(int a, int b) {
    return a + b;
}

// Kiểm thử đơn vị cho hàm add
TEST(AdditionTest, HandlesPositiveInput) {
    EXPECT_EQ(add(1, 2), 3);
    EXPECT_EQ(add(5, 5), 10);
}

TEST(AdditionTest, HandlesNegativeInput) {
```

```
    EXPECT_EQ(add(-1, -1), -2);
    EXPECT_EQ(add(-5, 5), 0);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Trong ví dụ trên:

TEST là macro được cung cấp bởi Google Test để xác định một trường hợp kiểm thử. EXPECT_EQ kiểm tra hai giá trị có bằng nhau hay không. Nếu không, nó sẽ ghi lại thông tin lỗi và tiếp tục với kiểm thử tiếp theo.

## 8.3. Kiểm thử Tích hợp (Integration Testing)

Kiểm thử tích hợp tập trung vào việc kiểm tra sự tương tác giữa các đơn vị (hàm, lớp, module) để đảm bảo rằng chúng hoạt động đúng khi được kết hợp với nhau.

Ví dụ: Kiểm thử Tích hợp Đơn giản trong C++

Giả sử chúng ta có hai hàm trong các module khác nhau: một hàm để đọc dữ liệu từ tệp và một hàm để xử lý dữ liệu đó.

```
#include <gtest/gtest.h>
#include <fstream>
#include <string>

// Hàm đọc dữ liệu từ tệp
std::string readDataFromFile(const std::string& filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        throw std::runtime_error("Cannot open file");
    }
    std::string data((std::istreambuf_iterator<char>(file)), std::istreambuf_iterator<char
    file.close();
    return data;
}
```

```
// Hàm xử lý dữ liệu
std::string processData(const std::string& data) {
    // Giả sử xử lý là chuyển toàn bộ ký tự thành chữ hoa
    std::string result = data;
    for (auto& c : result) {
        c = toupper(c);
    }
    return result;
}

// Kiểm thử tích hợp hai hàm
TEST(IntegrationTest, ReadAndProcessData) {
    std::string filename = "testfile.txt";
    std::ofstream outfile(filename);
    outfile << "hello world";
    outfile.close();

    std::string data = readDataFromFile(filename);
    std::string processedData = processData(data);

    EXPECT_EQ(processedData, "HELLO WORLD");
    remove(filename.c_str()); // Xóa file sau khi kiểm thử
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Trong ví dụ này:

Chúng ta kiểm thử sự tương tác giữa hàm readDataFromFile và processData. Kiểm thử này tạo một tệp tin, viết một số dữ liệu vào đó, đọc dữ liệu từ tệp tin và xử lý nó. Cuối cùng, chúng ta so sánh kết quả xử lý với giá trị mong đợi.

## 8.4. Kiểm thử Hồi quy (Regression Testing)

Kiểm thử hồi quy được thực hiện để đảm bảo rằng các thay đổi mới (sửa lỗi, cải tiến tính năng) không ảnh hưởng đến chức năng hiện tại của hệ thống.

## 8.5. Kiểm thử Hệ thống (System Testing)

Kiểm thử hệ thống kiểm tra toàn bộ hệ thống phần mềm để đảm bảo rằng tất cả các phần tương tác với nhau đúng cách và đáp ứng được yêu cầu.

## 8.6. Kiểm thử Hiệu năng (Performance Testing)

Kiểm thử hiệu năng kiểm tra tốc độ, khả năng mở rộng, và khả năng đáp ứng của hệ thống dưới tải cao.

Ví dụ: Kiểm thử Hiệu năng trong C++

Dưới đây là ví dụ kiểm thử hiệu năng cho một hàm tính giai thừa:

```
#include <chrono>
#include <iostream>

// Hàm tính giai thừa đệ quy
unsigned long long factorial(int n) {
    return (n == 0 || n == 1) ? 1 : n * factorial(n - 1);
}

// Kiểm thử hiệu năng
int main() {
    int n = 20; // Số để tính giai thừa
    auto start = std::chrono::high_resolution_clock::now();

    unsigned long long result = factorial(n);

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;

    std::cout << "Factorial of " << n << " is " << result << std::endl;
    std::cout << "Time taken: " << duration.count() << " seconds" << std::endl;

    return 0;
}
```

Trong ví dụ này, chúng ta sử dụng std::chrono để đo thời gian thực thi của hàm factorial.

## 8.7. Công cụ và Thư viện Kiểm thử C/C++ phổ biến

Google Test (gtest): Thư viện kiểm thử đơn vị và kiểm thử tích hợp cho C++. Catch2: Một framework kiểm thử đơn vị C++ đơn giản và mạnh mẽ. CppUnit: Thư viện kiểm thử đơn vị cho C++, tương tự như JUnit của Java. Valgrind: Công cụ phát hiện rò rỉ bộ nhớ và các lỗi bộ nhớ trong C/C++. AddressSanitizer (ASan): Công cụ phát hiện lỗi bộ nhớ ở runtime. CMocka: Thư viện kiểm thử đơn vị cho C, tập trung vào đơn giản và nhẹ.

## 8.8. Độ đo kiểm thử

Độ đo kiểm thử (Testing Metrics) là các chỉ số được sử dụng để đánh giá hiệu quả và chất lượng của các hoạt động kiểm thử phần mềm. Chúng giúp các nhà phát triển và kiểm thử viên hiểu rõ hơn về mức độ bao phủ kiểm thử, chất lượng mã nguồn, và các khía cạnh khác của quá trình kiểm thử. Việc sử dụng các độ đo kiểm thử phù hợp giúp cải thiện quy trình phát triển và kiểm thử phần mềm bằng cách cung cấp thông tin có giá trị để ra quyết định.

### 8.8.1. Độ Đo Liên Quan đến Quy Trình Kiểm Thử

Test Coverage (Độ phủ kiểm thử): Đo lường mức độ mà các đoạn mã nguồn đã được kiểm thử. Đây là một chỉ số quan trọng để đảm bảo rằng tất cả các phần của chương trình đều được kiểm tra.

Có nhiều loại độ phủ kiểm thử:

**Statement Coverage (Độ Phủ Câu Lệnh)** **Định nghĩa:** Phần trăm các câu lệnh trong mã nguồn đã được kiểm thử ít nhất một lần.

**Mục đích:** Đảm bảo rằng tất cả các câu lệnh trong mã nguồn đều đã được thực thi trong quá trình kiểm thử.

Ví dụ:

```
void exampleFunction(int x) {
    if (x > 0) {
        std::cout << "Positive";
    } else {
        std::cout << "Non-positive";
    }
    std::cout << "End of function";
}
```

Các trường hợp kiểm thử:

- exampleFunction(1); sẽ in "Positive" và "End of function".
- exampleFunction(-1); sẽ in "Non-positive" và "End of function".
- Statement Coverage: 100% vì mọi câu lệnh đều đã được thực thi.

**Branch Coverage (Độ Phủ Nhánh)** **Định nghĩa:** Phần trăm các nhánh điều kiện (như if, else, switch/case) đã được kiểm thử.

**Mục đích:** Đảm bảo rằng tất cả các đường nhánh trong mã nguồn đã được kiểm thử ít nhất một lần.

Ví dụ:

```
void exampleFunction(int x) {
    if (x > 0) {
        std::cout << "Positive";
    } else {
        std::cout << "Non-positive";
    }
}
```

Các trường hợp kiểm thử:

```
exampleFunction(1); (bao phủ nhánh if).
exampleFunction(-1); (bao phủ nhánh else).
Branch Coverage: 100\% vì cả hai nhánh if và else đều đã được kiểm thử.
```

**Function Coverage (Độ Phủ Hàm)** **Định nghĩa:** Phần trăm các hàm đã được gọi trong quá trình kiểm thử.

**Mục đích:** Đảm bảo rằng tất cả các hàm đã được gọi và kiểm thử ít nhất một lần.

Ví dụ:

```
void exampleFunction(int x);
void anotherFunction();

int main() {
```

```
    exampleFunction(1);
    return 0;
}
```

Function Coverage: 50% (chỉ có exampleFunction được gọi, anotherFunction không được gọi).

### 8.8.2. Defect Density (Mật Độ Lỗi)

**Định nghĩa:** Số lỗi tìm thấy trên mỗi đơn vị mã (ví dụ: trên 1000 dòng mã - KLOC).

**Mục đích:** Đo lường mức độ chất lượng mã nguồn; mật độ lỗi càng thấp thì chất lượng mã càng cao.

Ví dụ:

Dự án phần mềm có 20.000 dòng mã và đội ngũ kiểm thử phát hiện 40 lỗi.

Defect Density = 40 lỗi / 20 KLOC = 2 lỗi/KLOC.

### 8.8.3. Defect Removal Efficiency (Hiệu Quả Loại Bỏ Lỗi)

**Định nghĩa:** Tỷ lệ phần trăm lỗi được phát hiện và sửa chữa trước khi phần mềm được phát hành.

**Mục đích:** Đo lường hiệu quả của quy trình phát hiện và sửa lỗi trước khi phát hành.

Ví dụ:

Số lỗi tìm thấy trước khi phát hành: 80.

Số lỗi tìm thấy sau khi phát hành: 20.

Defect Removal Efficiency = (80 / (80 + 20)) * 100 = 80

### 8.8.4. Mean Time Between Failures (MTBF)

**Định nghĩa:** Thời gian trung bình giữa các lần thất bại của hệ thống.

**Mục đích:** Đo lường độ tin cậy của hệ thống.

Ví dụ:

Nếu hệ thống gặp lỗi sau mỗi 200 giờ hoạt động, thì MTBF = 200 giờ.

### 8.8.5. Mean Time to Repair (MTTR)

**Định nghĩa:** Thời gian trung bình để sửa chữa một lỗi sau khi nó được phát hiện.

**Mục đích:** Đo lường tính bảo trì của hệ thống.

Ví dụ:

Nếu trung bình mất 2 giờ để sửa chữa một lỗi sau khi nó được phát hiện, thì MTTR = 2 giờ.

### 8.8.6. Test Execution Time (Thời Gian Thực Thi Kiểm Thử)

**Định nghĩa:** Tổng thời gian cần thiết để chạy toàn bộ bộ kiểm thử.

**Mục đích:** Đo lường hiệu suất của quá trình kiểm thử.

Ví dụ:

Nếu bộ kiểm thử cần 5 giờ để hoàn thành, thì Test Execution Time = 5 giờ.

### 8.8.7. Test Cost (Chi Phí Kiểm Thử)

**Định nghĩa:** Chi phí trực tiếp liên quan đến hoạt động kiểm thử (nhân lực, công cụ, môi trường, v.v.).

**Mục đích:** Quản lý và tối ưu hóa chi phí liên quan đến kiểm thử.

Ví dụ:

Nếu tổng chi phí cho nhân lực và công cụ kiểm thử là $10,000, thì Test Cost = $10,000.

### 8.8.8. Test Case Effectiveness (Hiệu Quả của Trường Hợp Kiểm Thử)

**Định nghĩa:** Đo lường hiệu quả của các trường hợp kiểm thử trong việc phát hiện lỗi.

**Mục đích:** Đảm bảo các trường hợp kiểm thử có khả năng tìm thấy lỗi với số lượng kiểm thử ít.

Ví dụ:

Nếu có 100 trường hợp kiểm thử và chúng tìm thấy 50 lỗi, thì Test Case Effectiveness = 50%.

### 8.8.9. Tầm Quan Trọng của Việc Sử Dụng Độ Đo Kiểm Thử

- Cải thiện chất lượng phần mềm: Các độ đo kiểm thử giúp phát hiện các lỗi và khuyết điểm trong mã nguồn, từ đó cải thiện chất lượng phần mềm.

- Tối ưu hóa quy trình kiểm thử: Bằng cách phân tích các độ đo, đội ngũ kiểm thử có thể xác định các khu vực cần cải thiện và tối ưu hóa quy trình kiểm thử.
- Quản lý và lập kế hoạch tốt hơn: Các độ đo kiểm thử cung cấp dữ liệu quan trọng giúp quản lý dự án, lập kế hoạch kiểm thử, và phân bổ tài nguyên hiệu quả.

Độ đo kiểm thử đóng vai trò quan trọng trong việc đánh giá hiệu quả và chất lượng của quy trình kiểm thử phần mềm. Việc sử dụng các độ đo phù hợp giúp cải thiện chất lượng phần mềm, tối ưu hóa quy trình kiểm thử và đảm bảo rằng phần mềm đáp ứng được các yêu cầu chất lượng trước khi phát hành.

## 8.9. Tổng kết

Kiểm thử trong lập trình C/C++ là một phần không thể thiếu trong quá trình phát triển phần mềm. Nó giúp phát hiện lỗi sớm, cải thiện chất lượng phần mềm và đảm bảo rằng mã nguồn hoạt động đúng như mong đợi. Việc chọn đúng công cụ và phương pháp kiểm thử sẽ giúp nâng cao hiệu quả và chất lượng sản phẩm phần mềm.

## 8.10. Bài Tập Thực Hành

1. Tính Toán Test Coverage:
    - Viết một chương trình C++ đơn giản với các cấu trúc điều kiện và vòng lặp.
    - Tính toán Statement Coverage và Branch Coverage cho chương trình đó.
    - Viết các trường hợp kiểm thử để đạt được độ phủ 100%.
2. Xác Định Defect Density:
    - Cho một đoạn mã C++ dài 500 dòng.
    - Đếm số lỗi có thể xảy ra trong mã và tính toán Defect Density.
3. Thực Hành MTBF và MTTR:
    - Cho một ứng dụng chạy liên tục trong 1000 giờ và gặp 5 lần lỗi, mỗi lần lỗi mất trung bình 3 giờ để sửa.
    - Tính toán MTBF và MTTR.

4. Đánh Giá Test Case Effectiveness:

    - Cho một tập hợp các trường hợp kiểm thử (20 trường hợp) cho một đoạn mã.
    - Nếu 10 trường hợp kiểm thử tìm thấy lỗi, tính toán Test Case Effectiveness.

**Lưu ý :** Sử dụng Testing Metrics giúp cải thiện chất lượng phần mềm, tối ưu hóa quy trình kiểm thử, và đảm bảo rằng phần mềm đáp ứng được các yêu cầu chất lượng. Các độ đo như Test Coverage, Defect Density, MTBF, MTTR, và Test Case Effectiveness cung cấp thông tin có giá trị giúp cải thiện chất lượng kiểm thử và phát triển phần mềm.

Thực hành các bài tập trên sẽ giúp bạn hiểu rõ hơn về cách sử dụng các Testing Metrics trong các tình huống thực tế.

# Chương 9
# Lập trình phòng thủ, bẫy lỗi và gỡ rối chương trình



## 9.1. Giới thiệu

Lập trình phòng thủ (Defensive Programming) và bẫy lỗi (Error Handling) là hai phương pháp quan trọng trong lập trình nhằm đảm bảo tính ổn định và độ tin cậy của phần mềm. Gỡ rối (debugging) là quá trình tìm kiếm và sửa chữa lỗi trong mã nguồn để đảm bảo chương trình chạy đúng và hiệu quả. Trong lập trình C/C++, gỡ rối có thể phức tạp do tính chất của ngôn ngữ (ví dụ: truy cập bộ nhớ trực tiếp, sử dụng con trỏ, lỗi hành vi không xác định). Việc hiểu rõ các kỹ thuật gỡ rối khác nhau là rất quan trọng đối với lập trình viên C/C++. Dưới đây là phần trình bày chi tiết về các kỹ thuật lập trình phòng thủ, bẫy lỗi và phương pháp gỡ rối trong lập trình C/C++ cùng các ví dụ minh họa.

## 9.2. Lập trình phòng thủ (Defensive Programming)

Lập trình phòng thủ là một phương pháp thiết kế phần mềm nhằm dự đoán và chuẩn bị cho những tình huống bất thường hoặc lỗi có thể xảy ra trong quá trình thực thi. Mục tiêu của lập trình phòng thủ là giảm thiểu lỗi, tránh các sự cố ngoài ý muốn và đảm bảo phần mềm hoạt động ổn định trong mọi hoàn cảnh.

Nguyên tắc chính của lập trình phòng thủ:

- Kiểm tra đầu vào: Luôn kiểm tra và xác thực dữ liệu đầu vào từ người dùng hoặc các nguồn bên ngoài để đảm bảo rằng chúng hợp lệ và không chứa các giá trị không mong muốn.
- Xử lý các điều kiện bất thường: Luôn xử lý các tình huống bất thường hoặc các giá trị không hợp lệ. Điều này giúp tránh các lỗi có thể xảy ra trong quá trình thực thi chương trình.
- Giả định tối thiểu về môi trường: Không giả định quá nhiều về môi trường thực thi chương trình (như giả định rằng một tệp luôn tồn tại hoặc mạng luôn kết nối).
- Sử dụng các ràng buộc và kiểm tra trạng thái: Đảm bảo trạng thái nội bộ của chương trình luôn hợp lệ bằng cách sử dụng các ràng buộc và kiểm tra trước/sau khi thực hiện các phép toán.
- Giảm thiểu phụ thuộc giữa các thành phần: Các thành phần của phần mềm nên ít phụ thuộc lẫn nhau để tránh lỗi lan truyền.

Ví dụ về lập trình phòng thủ.

Ví dụ 1: Kiểm tra đầu vào - Hàm dưới đây kiểm tra đầu vào trước khi tính toán giai thừa của một số nguyên:

```
#include <iostream>
#include <stdexcept>

// Tính giai thừa của một số nguyên không âm
int factorial(int n) {
    if (n < 0) {
        throw std::invalid_argument("Negative input not allowed");
    }

    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

```
int main() {
    try {
        int number = -5; // Giá trị không hợp lệ
        std::cout << "Factorial: " << factorial(number) << std::endl;
    } catch (const std::invalid_argument& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

**Giải thích: Hàm factorial kiểm tra nếu đầu vào n là một số âm, nó sẽ ném ra một ngoại lệ (std::invalid_argument). Điều này giúp đảm bảo rằng hàm không bao giờ tính toán giai thừa của một số âm, tránh các lỗi logic tiềm ẩn.**

**Ví dụ 2: Giảm thiểu phụ thuộc giữa các thành phần**

```
class Sensor {
public:
    bool isOperational() {
        // Kiểm tra trạng thái của cảm biến
        return true; // giả định cảm biến luôn hoạt động tốt
    }
};

class DataProcessor {
public:
    void processData() {
        if (sensor.isOperational()) {
            // Xử lý dữ liệu khi cảm biến hoạt động tốt
            std::cout << "Processing data..." << std::endl;
        } else {
            std::cerr << "Sensor is not operational." << std::endl;
        }
    }

private:
    Sensor sensor;
```

```
};
```

**Giải thích: Ở đây, DataProcessor không phụ thuộc quá nhiều vào trạng thái của Sensor. Nếu Sensor không hoạt động, DataProcessor vẫn có thể xử lý lỗi một cách an toàn mà không gặp sự cố.**

## 9.3. Bẫy lỗi (Error Handling)

**Bẫy lỗi là quá trình phát hiện, xử lý và khắc phục lỗi trong quá trình thực thi chương trình. Bẫy lỗi bao gồm việc viết mã để xử lý các tình huống ngoại lệ, đảm bảo chương trình không bị sập và có thể khôi phục hoặc thông báo lỗi thích hợp cho người dùng.**

**Phương pháp bẫy lỗi.**

- **Sử dụng ngoại lệ (Exceptions): Đây là phương pháp phổ biến trong các ngôn ngữ như C++ để bẫy và xử lý lỗi. Ngoại lệ cho phép tách rời mã xử lý lỗi khỏi mã logic chính.**
- **Mã lỗi và kiểm tra điều kiện: Trong C, lập trình viên thường trả về mã lỗi từ các hàm và kiểm tra các mã lỗi này để xác định hành động tiếp theo.**
- **Sử dụng các hàm như assert: assert được sử dụng để kiểm tra các điều kiện mà lập trình viên cho là không bao giờ sai. Nếu điều kiện sai, chương trình sẽ bị dừng lại với thông báo lỗi.**

**Ví dụ: Sử dụng ngoại lệ trong C++**

```
#include <iostream>
#include <stdexcept>

// Hàm chia 2 số và xử lý chia cho 0
double divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero error");
    }
    return static_cast<double>(a) / b;
}
```

```
int main() {
    try {
        int x = 10, y = 0;
        double result = divide(x, y);
        std::cout << "Result: " << result << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

**Giải thích: Hàm divide ném ra một ngoại lệ nếu b bằng 0. Điều này giúp ngăn chặn lỗi chia cho 0, và mã xử lý lỗi được tách ra khỏi logic chính của chương trình.**

**Ví dụ: Sử dụng mã lỗi trong C.**

```
#include <stdio.h>

#define SUCCESS 0
#define ERROR_DIVISION_BY_ZERO 1

int divide(int a, int b, double* result) {
    if (b == 0) {
        return ERROR_DIVISION_BY_ZERO;
    }
    *result = (double)a / b;
    return SUCCESS;
}

int main() {
    int x = 10, y = 0;
    double result;

    int status = divide(x, y, &result);
    if (status == ERROR_DIVISION_BY_ZERO) {
        printf("Error: Division by zero.\n");
    } else {
```

```
        printf("Result: %.2f\n", result);
    }

    return 0;
}
```

**Giải thích: Hàm divide trả về một mã lỗi nếu xảy ra lỗi chia cho 0. Chương trình sau đó kiểm tra mã lỗi và xử lý tương ứng.**

#### 9.3.1. So sánh và Ứng dụng

- **Lập trình phòng thủ và bẫy lỗi có thể được sử dụng đồng thời để làm cho mã nguồn an toàn và mạnh mẽ hơn. Trong khi lập trình phòng thủ tập trung vào việc dự đoán và ngăn chặn các lỗi trước khi chúng xảy ra, bẫy lỗi xử lý các lỗi xảy ra trong quá trình thực thi chương trình.**
- **Các kỹ thuật này rất hữu ích trong các hệ thống yêu cầu độ ổn định cao như phần mềm điều khiển máy móc, phần mềm tài chính, và các ứng dụng an toàn.**

### 9.4. Gỡ rối - Debugging

#### 9.4.1. Các loại lỗi thường gặp trong C/C++

**Syntax Errors (Lỗi Cú Pháp)**

- **Lỗi cú pháp là lỗi do vi phạm quy tắc cú pháp của ngôn ngữ.**
- **Ví dụ: Thiếu dấu chấm phẩy, dùng sai dấu ngoặc, sử dụng từ khóa không hợp lệ.**
- **Cách khắc phục: Sử dụng trình biên dịch để kiểm tra mã nguồn và sửa lỗi cú pháp.**

**Semantic Errors (Lỗi Ngữ Nghĩa)**

- **Lỗi ngữ nghĩa xảy ra khi mã có cú pháp đúng nhưng không thực hiện đúng như ý định của lập trình viên.**
- **Ví dụ: Sử dụng biến chưa được khởi tạo, gọi hàm với sai kiểu tham số.**

- **Cách khắc phục: Xem lại logic của chương trình và đảm bảo mã thực hiện đúng như mong đợi.**

**Runtime Errors (Lỗi Thời Gian Chạy)**

- **Lỗi thời gian chạy xảy ra trong khi chương trình đang chạy, thường do các vấn đề như tràn bộ nhớ, chia cho 0, hoặc truy cập bộ nhớ không hợp lệ.**
- **Ví dụ: Segmentation Fault, Bus Error.**
- **Cách khắc phục: Sử dụng các công cụ gỡ rối hoặc trình quản lý bộ nhớ để theo dõi và sửa chữa lỗi.**

**Logical Errors (Lỗi Logic)**

- **Lỗi logic là lỗi mà chương trình chạy mà không có lỗi thời gian chạy nhưng lại cho kết quả sai.**
- **Ví dụ: Sai công thức tính toán, sai điều kiện vòng lặp.**
- **Cách khắc phục: Xem xét lại thuật toán và logic để đảm bảo chương trình hoạt động đúng như mong đợi.**

**Undefined Behavior (Hành Vi Không Xác Định)**

- **Hành vi không xác định xảy ra khi mã nguồn vi phạm chuẩn ngôn ngữ, dẫn đến kết quả không thể đoán trước.**
- **Ví dụ: Sử dụng con trỏ null, truy cập vùng nhớ không hợp lệ.**
- **Cách khắc phục: Sử dụng các chuẩn ngôn ngữ mới và các công cụ phân tích tĩnh để phát hiện hành vi không xác định.**

### 9.4.2. Các phương pháp gỡ rối (Debugging Techniques)

**Debugging bằng printf và cout**

- **Mô tả: Thêm các câu lệnh printf (C) hoặc std::cout (C++) vào mã nguồn để theo dõi giá trị biến, dòng thực thi, và trạng thái chương trình.**

- **Ưu điểm: Đơn giản, không cần công cụ bên ngoài, phù hợp với các chương trình nhỏ.**
- **Nhược điểm: Khó sử dụng cho chương trình lớn, dễ gây lộn xộn mã nguồn, mất thời gian.**

**Ví dụ:**

```
#include <iostream>

int main() {
    int x = 0;
    int y = 5 / x;  // Lỗi chia cho 0
    std::cout << "Value of y: " << y << std::endl;
    return 0;
}
```

**Gỡ rối: Thêm std::cout để kiểm tra giá trị của x trước khi thực hiện phép chia.**

**Sử dụng Trình Debugger (GDB, LLDB, Visual Studio Debugger)**

- **Mô tả: Các trình gỡ rối như GDB (GNU Debugger), LLDB, và Visual Studio Debugger cho phép kiểm tra chương trình ở mức chi tiết, thiết lập breakpoint, theo dõi giá trị biến, và thực hiện bước qua từng lệnh.**
- **Ưu điểm: Rất mạnh mẽ, hỗ trợ đa nền tảng, cung cấp nhiều tính năng nâng cao (watch, backtrace, step-into, step-over).**
- **Nhược điểm: Yêu cầu thời gian học tập và làm quen.**

**Ví dụ với GDB:**

1. **Compile chương trình với cờ debug -g:**

   ```
   g++ -g -o my_program my_program.cpp
   ```

2. **Chạy GDB với chương trình:**

   ```
   gdb ./my_program
   ```

3. **Thiết lập breakpoint và chạy chương trình:**

   ```
   (gdb) break main
   (gdb) run
   ```

4. **Theo dõi giá trị biến và kiểm tra lỗi:**

   ```
   (gdb) print x
   ```

**Static Analysis Tools (Công Cụ Phân Tích Tĩnh)**

- **Mô tả: Các công cụ như cppcheck, Clang Static Analyzer, và PVS-Studio phân tích mã nguồn để tìm ra lỗi mà không cần chạy chương trình.**
- **Ưu điểm: Phát hiện lỗi tiềm ẩn, lỗi bảo mật, và lỗi hành vi không xác định; cải thiện chất lượng mã nguồn trước khi chạy.**
- **Nhược điểm: Có thể phát sinh cảnh báo giả, yêu cầu tích hợp vào quy trình CI/CD.**

**Ví dụ:** `cppcheck my_program.cpp`

**Sanitizers (Công Cụ Kiểm Tra Bộ Nhớ)**

- **Mô tả: Công cụ như AddressSanitizer (ASan), UndefinedBehaviorSanitizer (UBSan), và MemorySanitizer (MSan) giúp phát hiện các vấn đề liên quan đến bộ nhớ, hành vi không xác định.**
- **Ưu điểm: Phát hiện nhanh chóng các lỗi bộ nhớ, tràn bộ nhớ, và các lỗi liên quan đến bộ nhớ khác.**
- **Nhược điểm: Làm chậm chương trình, không phải lúc nào cũng có thể phát hiện tất cả các lỗi.**

**Ví dụ với** `AddressSanitizer`**:**

1. **Compile chương trình với** `AddressSanitizer`**:**

   ```
   g++ -fsanitize=address -g -o my_program my_program.cpp
   ```

2. **Chạy chương trình:**

   ```
   ./my_program
   ```

**Memory Debugging Tools (Công Cụ Gỡ Rối Bộ Nhớ) - Valgrind**

- **Mô tả: Valgrind là một công cụ mạnh mẽ giúp phát hiện các lỗi bộ nhớ như tràn bộ nhớ, truy cập vùng nhớ không hợp lệ, và rò rỉ bộ nhớ.**
- **Ưu điểm: Phát hiện lỗi bộ nhớ và rò rỉ bộ nhớ hiệu quả, cung cấp báo cáo chi tiết.**
- **Nhược điểm: Làm chậm chương trình, cần thời gian để hiểu báo cáo.**

**Ví dụ với Valgrind:**

```
valgrind -leak-check=full ./my_program
```

**Unit Testing (Kiểm Thử Đơn Vị) và Assertions**

- **Mô tả: Viết các bài kiểm thử đơn vị để kiểm tra từng phần của mã nguồn một cách độc lập. Sử dụng các khẳng định (assertions) để kiểm tra điều kiện trong mã nguồn.**
- **Ưu điểm: Giúp phát hiện lỗi sớm, kiểm tra mã nguồn một cách tự động.**
- **Nhược điểm: Yêu cầu thời gian viết bài kiểm thử, không phù hợp với tất cả loại lỗi.**

**Ví dụ:**

```
#include <cassert>

void test_addition() {
    int result = 2 + 2;
    assert(result == 4);  // Sẽ không báo lỗi nếu kết quả đúng
}

int main() {
    test_addition();
    return 0;
}
```

**Logging (Ghi Nhật Ký) và Tạo Log Files**

- **Mô tả: Sử dụng thư viện ghi nhật ký (logging libraries) để ghi lại các sự kiện, giá trị biến, và trạng thái chương trình vào tệp nhật ký.**
- **Ưu điểm: Theo dõi chương trình một cách chi tiết, dễ dàng xem lại trạng thái sau khi chương trình chạy xong.**
- **Nhược điểm: Có thể tạo ra các tệp nhật ký lớn, không dễ dàng để tìm kiếm lỗi trong các chương trình phức tạp.**

**Ví dụ:**

```
#include <fstream>
#include <iostream>

int main() {
    std::ofstream logFile("debug.log");
    int x = 5;
    logFile << "x = " << x << std::endl;
    // Các dòng lệnh khác
    logFile.close();
    return 0;
}
```

## 9.5. Kết Luận

**Gỡ rối là một phần không thể thiếu của quá trình phát triển phần mềm. Việc nắm vững các phương pháp gỡ rối và biết cách áp dụng chúng trong các tình huống khác nhau sẽ giúp lập trình viên nhanh chóng phát hiện và sửa chữa lỗi, đảm bảo chương trình chạy đúng và hiệu quả. Các công cụ như debugger, static analysis tools, sanitizers, và Valgrind cung cấp các phương tiện mạnh mẽ để kiểm tra và cải thiện chất lượng mã nguồn. Học cách sử dụng chúng một cách hiệu quả là một kỹ năng quan trọng đối với mọi lập trình viên C/C++.**

**Lập trình phòng thủ và bẫy lỗi là hai chiến lược quan trọng giúp nâng cao chất lượng phần mềm. Bằng cách hiểu và áp dụng chúng đúng cách, lập trình viên có thể giảm thiểu lỗi, bảo vệ phần mềm khỏi các tình huống không lường trước, và cải thiện trải nghiệm người dùng.**

**Mỗi phương pháp gỡ rối đều có ưu và nhược điểm riêng, và cách tốt nhất là kết hợp chúng tùy vào từng hoàn cảnh cụ thể. Việc hiểu và áp dụng các công cụ này sẽ giúp lập trình viên phát hiện và sửa lỗi nhanh chóng, nâng cao hiệu suất và độ tin cậy của phần mềm.**

## 9.6. Ví dụ minh họa

**Để hiểu rõ hơn về các phương pháp gỡ rối (debugging) trong C/C++, dưới đây là một số ví dụ chi tiết minh họa cho từng phương pháp được đề cập:**

### 9.6.1. Debugging bằng printf và cout

**Phương pháp này sử dụng các câu lệnh in như printf trong C hoặc std::cout trong C++ để theo dõi giá trị của các biến và trạng thái chương trình.**

**Ví dụ 1 - Chúng ta có một hàm tính tổng các số từ 1 đến n:**

```
#include <iostream>

int sum(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        total += i;
    }
    return total;
}

int main() {
    int n = 10;
    int result = sum(n);
    std::cout << "Sum of 1 to " << n << " is " << result << std::endl;
    return 0;
}
```

**Trong ví dụ trên, nếu chúng ta muốn kiểm tra giá trị của total ở mỗi bước trong vòng lặp, chúng ta có thể thêm một câu lệnh std::cout như sau:**

```cpp
#include <iostream>

int sum(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        total += i;
        std::cout << "After adding " << i << ", total = " << total << std::endl;
    }
    return total;
}

int main() {
    int n = 10;
    int result = sum(n);
    std::cout << "Sum of 1 to " << n << " is " << result << std::endl;
    return 0;
}
```

### 9.6.2. Debugging với GDB (GNU Debugger)

**GDB cho phép chúng ta theo dõi chương trình ở mức chi tiết, thiết lập breakpoints, và kiểm tra giá trị biến tại runtime.**

**Ví dụ 2 - Đoạn mã dưới đây có lỗi chia cho 0:**

```cpp
#include <iostream>

int divide(int a, int b) {
    return a / b; // Lỗi tiềm ẩn nếu b = 0
}

int main() {
    int x = 10, y = 0;
    int result = divide(x, y);
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

**Gỡ lỗi bằng GDB:**

1. **Biên dịch mã nguồn với thông tin debug:**

   ```
   g++ -g -o debug_example example.cpp
   ```

2. **Chạy GDB với chương trình:**

   ```
   gdb ./debug_example
   ```

3. **Thiết lập breakpoint và chạy chương trình:**

   ```
   (gdb) break divide
   (gdb) run
   ```

4. **Kiểm tra giá trị biến trước khi thực hiện phép chia:**

   ```
   (gdb) print b
   ```

5. **Sửa lỗi và thử lại**

### 9.6.3. Static Analysis Tools (cppcheck)

**Static analysis tools giúp phát hiện lỗi trong mã nguồn mà không cần phải chạy chương trình.**

**Ví dụ 3 - Giả sử chúng ta có đoạn mã sau:**

```cpp
#include <iostream>

void printArray(int arr[], int size) {
    for (int i = 0; i <= size; i++) { // Lỗi: Truy cập ngoài giới hạn mảng
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printArray(arr, 5);
    return 0;
}
```

**Sử dụng cppcheck để phát hiện lỗi:**

```
cppcheck example.cpp
```

**Kết quả:**

```
example.cpp: In function 'void printArray(int*, int)':

example.cpp:4:32: warning: Array 'arr[5]' accessed at index 5, which is out of bo
    for (int i = 0; i <= size; i++) {
```

### 9.6.4. AddressSanitizer (ASan)

**AddressSanitizer là một công cụ phát hiện lỗi bộ nhớ như tràn bộ nhớ hoặc truy cập vùng nhớ không hợp lệ.**

**Ví dụ 4:**

```
#include <iostream>

int main() {
    int *arr = new int[5];
    for (int i = 0; i <= 5; i++) { // Lỗi: Truy cập ngoài giới hạn mảng
        arr[i] = i;
    }
    delete[] arr;
    return 0;
}
```

**Compile với AddressSanitizer:**

```
g++ -fsanitize=address -g -o asan_example example.cpp
```

**Chạy chương trình:**

```
./asan_example
```

**Output từ AddressSanitizer:**

```
arduino
=================================================================
==14272==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000018
WRITE of size 4 at 0x602000000018 thread T0
    #0 0x40070c in main asan_example.cpp:6
    ...
```

### 9.6.5. Valgrind (Memory Debugging Tool)

**Valgrind giúp phát hiện lỗi bộ nhớ và rò rỉ bộ nhớ trong chương trình.**

**Ví dụ 5:**

```cpp
#include <iostream>

int main() {
    int *arr = new int[5];
    arr[2] = 10; // Không xóa mảng sau khi sử dụng
    return 0;
}
```

**Sử dụng Valgrind để kiểm tra rò rỉ bộ nhớ:**

```
valgrind -leak-check=full ./valgrind_example
```

**Output từ Valgrind:**

```
==15007== HEAP SUMMARY:
==15007==     in use at exit: 20 bytes in 1 blocks
==15007==   total heap usage: 2 allocs, 1 frees, 73,704 bytes allocated
...
==15007== LEAK SUMMARY:
==15007==    definitely lost: 20 bytes in 1 blocks
==15007==    indirectly lost: 0 bytes in 0 blocks
...
```

### 9.6.6. Unit Testing với Frameworks như Google Test

**Viết các bài kiểm thử đơn vị để kiểm tra tính đúng đắn của mã nguồn.**

**Ví dụ 6 - Sử dụng Google Test để kiểm thử một hàm tính giai thừa:**

```cpp
// factorial.cpp
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

// factorial_test.cpp
#include "gtest/gtest.h"
#include "factorial.cpp"

TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(factorial(0), 1);
```

```
}

TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(factorial(5), 120);
}
```

**Chạy các bài kiểm thử:**

```
./factorial_test
```

**Output:**

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from FactorialTest
[ RUN      ] FactorialTest.HandlesZeroInput
[       OK ] FactorialTest.HandlesZeroInput (0 ms)
[ RUN      ] FactorialTest.HandlesPositiveInput
[       OK ] FactorialTest.HandlesPositiveInput (0 ms)
[----------] 2 tests from FactorialTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```

### 9.6.7. Logging và Tạo Log Files

**Ghi lại các sự kiện và thông tin trạng thái vào tệp nhật ký để dễ dàng xem lại sau.**

**Ví dụ 7:**

```
#include <fstream>
#include <iostream>

void debug_log(const std::string &message) {
    std::ofstream logFile("debug.log", std::ios_base::app);
    logFile << message << std::endl;
}

int main() {
```

```
    int x = 10;
    debug_log("Program started");
    debug_log("Value of x: " + std::to_string(x));
    // Các thao tác khác
    debug_log("Program ended");
    return 0;
}
```

**Kiểm tra tệp debug.log:**

```
Program started
Value of x: 10
Program ended
```

## 9.7. Bài tập

### 9.7.1. Bài tập về Gỡ Rối (Debugging)

**Bài tập 1: Gỡ rối với printf/cout**

**Mô tả: Bạn có một đoạn mã C++ sau đây, nhưng chương trình luôn trả về kết quả sai. Hãy sử dụng cout để tìm ra vấn đề.**

```
#include <iostream>
int main() {
    int x = 10;
    int y = 20;
    int z = x + y * 2;
    std::cout << "The result is: " << z << std::endl;
    return 0;
}
```

**Yêu cầu: Xác định lỗi trong phép tính và sửa mã nguồn để trả về kết quả đúng.**

**Bài tập 2: Sử dụng GDB để gỡ rối**

**Mô tả: Bạn có một đoạn mã C bị lỗi Segmentation Fault khi chạy. Hãy sử dụng GDB để tìm ra nguyên nhân và sửa lỗi.**

```
#include <stdio.h>
```

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i <= 5; i++) { // Lỗi tiềm ẩn
        printf("%d ", arr[i]);
    }
    return 0;
}
```

**Yêu cầu: Sử dụng GDB để tìm ra lỗi, giải thích nguyên nhân và sửa mã nguồn.**

**Bài tập 3: Sử dụng Valgrind để tìm rò rỉ bộ nhớ**

**Mô tả: Đoạn mã sau có vấn đề rò rỉ bộ nhớ. Hãy sử dụng Valgrind để phát hiện rò rỉ bộ nhớ và sửa lỗi.**

```
#include <iostream>

void createArray() {
    int* arr = new int[100];
    arr[0] = 10;
    // Lỗi: Không giải phóng bộ nhớ
}

int main() {
    createArray();
    return 0;
}
```

**Yêu cầu: Sử dụng Valgrind để tìm rò rỉ bộ nhớ và sửa mã nguồn để giải phóng bộ nhớ đúng cách.**

### 9.7.2. Bài tập về Lập Trình Phòng Thủ (Defensive Programming)

**Bài tập 4: Kiểm tra đầu vào hợp lệ**

**Mô tả: Viết một chương trình nhận vào một số nguyên từ người dùng và tính giai thừa của nó. Đảm bảo rằng chương trình kiểm tra đầu vào để không chấp nhận số âm. Yêu cầu: Sử dụng lập trình phòng thủ để đảm bảo rằng chương trình chỉ tính giai thừa cho các số không âm.**

**Bài tập 5: Kiểm tra chỉ số mảng**

**Mô tả: Viết một hàm nhận vào một mảng và chỉ số, sau đó trả về giá trị của phần tử tại chỉ số đó. Đảm bảo rằng chỉ số nằm trong phạm vi hợp lệ của mảng. Yêu cầu: Sử dụng lập trình phòng thủ để kiểm tra tính hợp lệ của chỉ số trước khi truy cập mảng.**

### 9.7.3. Bài tập về Bẫy Lỗi (Error Handling)

**Bài tập 6: Xử lý lỗi chia cho 0**

**Mô tả: Viết một hàm chia hai số nguyên. Hàm này phải phát hiện và xử lý lỗi chia cho 0. Yêu cầu: Sử dụng bẫy lỗi để xử lý trường hợp chia cho 0 và trả về thông báo lỗi hợp lý cho người dùng.**

**Bài tập 7: Xử lý lỗi đọc tệp tin**

**Mô tả: Viết một chương trình đọc dữ liệu từ một tệp tin. Nếu tệp tin không tồn tại hoặc không thể mở, chương trình phải xử lý lỗi và thông báo cho người dùng. Yêu cầu: Sử dụng bẫy lỗi để xử lý các tình huống ngoại lệ liên quan đến tệp tin.**

**Bài tập 8: Xử lý ngoại lệ khi phân bổ bộ nhớ Mô tả: Viết một chương trình để cấp phát bộ nhớ động cho một mảng. Chương trình cần xử lý ngoại lệ nếu không đủ bộ nhớ để cấp phát. Yêu cầu: Sử dụng std::bad_alloc để bẫy lỗi khi không thể cấp phát bộ nhớ.**

**Bài tập 9: Xử lý lỗi trong các hàm đệ quy  Mô tả: Viết một hàm đệ quy để tính số Fibonacci thứ n. Hãy xử lý lỗi nếu giá trị n là một số âm. Yêu cầu: Sử dụng lập trình phòng thủ và bẫy lỗi để xử lý các trường hợp đầu vào không hợp lệ.**

# Tài liệu tham khảo


[1] **Trần Đan Thư,** *Kỹ thuật lập trình*, **NXB Khoa học và kỹ thuật, 2014.**

[2] **Mcconnell and Steve,** *Code Complete: A Practical Handbook of Software Construction*, **2d Ed. Redmond, Wa.: Microsoft Press, 2004.**

[3] **Kernighan and Plauger,** *The elements of programming style*, **Addison-Wesley; 1st Edition,1999.**

[4] **Brian W. Kernighan and Rob Pike,** *The Practice of Programming*, **Addison-Wesley; 1st Edition,1999.**

[5] **Nicolai M. Josuttis,** *The C++ Standard Library: A Tutorial and Reference* **, Addison-Wesley; 2st Edition,2012.**