Mục lục

Giới thiệu

Lập trình là một trong những bước quan trọng nhất trong quy trình giải quyết một bài toán. Nhiều ngôn ngữ lập trình đã được ra đời nhằm giúp chúng ta giải quyết các bài toán một cách hiệu quả nhất. Mỗi ngôn ngữ lập trình có những thế mạnh và nhược điểm riêng. Các ngôn ngữ lập trình có thể chia ra thành hai loại chính là ngôn ngữ lập trình bậc thấp (gần gũi với ngôn ngữ máy), và ngôn ngữ lập trình bậc cao (ngần gũi với ngôn ngữ tự nhiên của con người).

Giáo trình này trang bị cho sinh viên những kiến thức cơ bản về lập trình. Ngôn ngữ lập trình C++ được sử dụng để minh họa cho việc lập trình. Việc lựa chọn C++ bởi vì nó là một ngôn ngữ lập trình hướng đối tượng chuyên nghiệp được sử dụng rộng rãi trên toàn thế giới để phát triển các chương trình từ đơn giản đến phức tạp. Hơn thế nữa, sự mềm dẻo của C++ cho phép chúng ta giải quyết những bài toán thực tế một cách nhanh chóng, ngoài ra cho phép chúng ta quản lý và tương tác trực tiếp đến hệ thống, bộ nhớ để nâng cao hiệu quả của chương trình.

Giáo trình được chia thành 8 chương, mỗi chương trình bày một vấn đề lý thuyết trong lập trình. Các vấn đề lý thuyết được mô tả bằng các ví dụ thực tế. Kết thúc mỗi chương là phần bài tập để sinh viên giải quyết và nắm rõ hơn về lý thuyết. Cấu trúc của giáo trình như sau:

- Chương 1: giới thiệu các khái niệm cơ bản về lập trình và quy trình giải quyết một bài toán. Sinh viên sẽ hiểu về ngôn ngữ lập trình bậc cao và ngôn ngữ lập trình bậc thấp. Cuối chương chúng tôi giới thiệu về môi trường lập trình cũng như ngôn ngữ lập trình C++.

- Chương 2: Chúng tôi giới thiệu các khái niệm cơ bản về biến số, hằng số, các kiểu dữ liệu cơ bản và các phép toán cơ bản. Sau khi học, sinh viên sẽ biết cách khai báo và sử dụng biến số, hằng số, và các phép toán trên biến và hằng số.

- Chương 3: Trương này giới thiệu về cấu trúc chương trình cũng như các cấu trúc điều khiển. Cụ thể là các cấu rẽ nhánh (if-else, switch), cấu trúc lặp (for-do, while-do, repeat) sẽ được giới thiệu.

- Chương 4: Chương trình con và hàm sẽ được giới thiệu để sinh viên hiểu được chiến lược lập trình “chia để trị”. Chúng tôi sẽ trình bày chi tiết về

cách khai báo và sử dụng hàm, cũng như cách truyền tham số, truyền giá trị cho hàm.

- Chương 5: Chương này trình bày cấu trúc dữ liệu kiểu mảng và xâu kí tự. Cách khai báo và sử dụng mảng một chiều cũng như mảng nhiều chiều và ví dụ liên quan được trình bày chi tiết ở chương này.
- Chương 6: Đây là một chương tương đối đặc thù cho C++, khi chúng tôi trình bày về bộ nhớ và kiểu dữ liệu con trỏ. Cấu trúc bộ nhớ, cách quản lý và xin cấp phép bộ nhớ động thông qua việc sử dụng biến con trỏ sẽ được trình bày.
- Chương 7: Trong chương này chúng tôi sẽ trình bày về cấu trúc dữ liệu trừu tượng (cụ thể là struct và class trong C++). Sinh viên sẽ hiểu và biết cách tạo ra những cấu trúc dữ liệu trừu tượng phù hợp với các kiểu đối tượng dữ liệu cần biểu diễn. Cuối chương, chúng tôi cũng giới thiệu về lập trình hướng đối tượng, một thế mạnh của ngôn ngữ lập trình C++.
- Chương 8: Chúng tôi giới thiệu về cách vào ra dữ liệu. Sinh viên sẽ được giới thiệu chi tiết về cách làm việc với các tệp dữ liệu.

Chương 1. Mở đầu

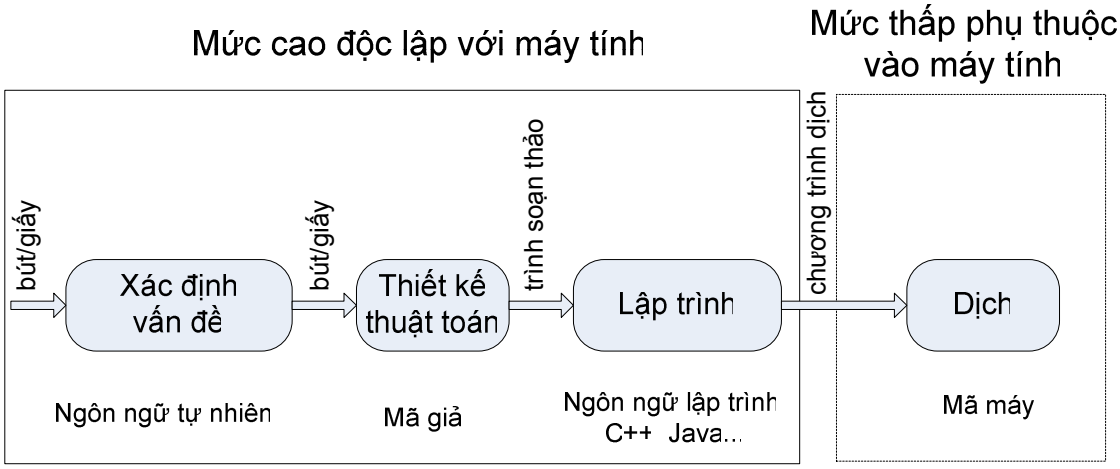
Trong chương này, chúng tôi sẽ giới thiệu qua một số khái niệm cơ bản về: chương trình, lập trình, ngôn ngữ lập trình.

1.1. Chương trình là gì?

Bạn chắc chắn đã dùng qua nhiều chương trình khác nhau, ví dụ như chương trình soạn thảo văn bản "Microsoft Word". Chương trình, hay phần mềm, được hiểu đơn giản là một tập các lệnh để máy tính thực hiện theo. Khi bạn đưa cho máy tính một chương trình và yêu cầu máy tính thực hiện theo các lệnh của chương trình, bạn đang chạy chương trình đó.

1.2. Lập trình là gì?

Lập trình là có thể hiểu đơn giản là quá trình viết ra các lệnh hướng dẫn máy tính thực hiện để giải quyết một bài toán cụ thể nào đó. Lập trình là một bước quan trọng trong quy trình giải quyết một bài toán như mô tả ở Hình 1.1.



Hình 1.1: Quy trình giải quyết một bài toán.

Quy trình trên có thể được chia ra thành hai mức: mức cao độc lập với máy tính (*machine independent*) và mức thấp phụ thuộc vào máy tính (*machine specific*).

1.2.1. Mức cao độc lập với máy tính

Mức cao độc lập với máy tính thường được chia thành ba bước chính là: xác định vấn đề, thiết kế thuật toán và lập trình.

Xác định vấn đề: Bước này định nghĩa bài toán, xác định dữ liệu đầu vào, các ràng buộc, yêu cầu cần giải quyết và kết quả đầu ra. Bước này thường sử dụng bút/giấy và ngôn ngữ tự nhiên như tiếng Anh, tiếng Việt để mô tả và xác định vấn đề cần giải quyết.

Thiết kế thuật toán: Một thuật toán là một bộ các chỉ dẫn nhằm giải quyết một bài toán. Các chỉ dẫn này cần được diễn đạt một cách hoàn chỉnh và chính xác sao cho mọi người có thể hiểu và tiến hành theo. Thuật toán thường được mô tả dưới dạng **mã giả** (*pseudocode*). Bước này có thể sử dụng giấy bút và thường không phụ thuộc vào ngôn ngữ lập trình. Ví dụ về thuật toán "tìm ước số chung lớn nhất (UCLN) của hai số x và y" viết bằng ngôn ngữ tự nhiên:

- Bước 1: Nếu x>y thì thay x bằng phần dư của phép chia x/y.
- Bước 2: Nếu không, thay y bằng phần dư của phép chia y/x.
- Bước 3: Nếu trong hai số x và y có một số bằng 0 thì kết luận UCLN là số còn lại.
- Bước 4: Nếu không, quay lại Bước 1.

hoặc bằng mã giả:

repeat

if $x > y$ **then** $x := x$ **mod** y

else $y := y$ **mod** x

until $x = 0$ **or** $y = 0$

if $x = 0$ **then** UCLN := y

else UCLN := x

Lập trình là bước chuyển đổi thuật toán sang một ngôn ngữ lập trình, phổ biến là các ngôn ngữ lập trình bậc cao, ví dụ như các ngôn ngữ C++, Java. Bước này, lập trình viên sử dụng một chương trình soạn thảo văn bản để viết chương trình. Trong và sau quá trình lập trình, người ta phải tiến hành kiểm thử và sửa lỗi chương trình. Có ba loại lỗi thường gặp: lỗi cú pháp, lỗi trong thời gian chạy, và lỗi lô-gic (xem chi tiết ở Mục 1.5).

1.2.2. Mức thấp phụ thuộc vào máy tính

Các ngôn ngữ lập trình bậc cao, ví dụ như C, C++, Java, Visual Basic, C#, được thiết kế để con người tương đối dễ hiểu và dễ sử dụng. Tuy nhiên, máy tính không hiểu được các ngôn ngữ bậc cao. Do đó, trước khi một chương trình viết bằng ngôn ngữ bậc cao có thể chạy được, nó phải được dịch sang **ngôn ngữ máy**, hay còn gọi là **mã máy**, mà máy tính có thể hiểu và thực hiện được. Việc dịch đó được thực hiện bởi một chương trình máy tính gọi là chương trình dịch.

1.3. Ngôn ngữ lập trình và chương trình dịch

Như chúng ta thấy, quá trình giải quyết một bài toán thông qua các bước khác nhau để chuyển đổi từ ngôn ngữ tự nhiên mà con người hiểu được sang ngôn ngữ máy mà máy tính có thể hiểu và thực hiện được. Ngôn ngữ lập trình thường được chia ra thành hai loại: ngôn ngữ lập trình bậc thấp và ngôn ngữ lập trình bậc cao.

Ngôn ngữ lập trình bậc thấp như hợp ngữ (*assembly language*) hoặc mã máy là ngôn ngữ gần với ngôn ngữ máy mà máy tính có thể hiểu được. Đặc điểm chính của các ngôn ngữ này là chúng có liên quan chặt chẽ đến phần cứng của máy tính. Các họ máy tính khác nhau sử dụng các ngôn ngữ khác nhau. Chương trình viết bằng các ngôn ngữ này có thể chạy mà không cần qua chương trình dịch. Các ngôn ngữ bậc thấp có thể dùng để viết những chương trình cần tối ưu hóa về tốc độ. Tuy nhiên, chúng thường khó hiểu đối với con người và không thuận tiện cho việc lập trình.

Ngôn ngữ lập trình bậc cao như Pascal, Ada, C, C++, Java, Visual Basic, Python, … là các ngôn ngữ có mức độ trừu tượng hóa cao, gần với ngôn ngữ tự nhiên của con người hơn. Việc sử dụng các ngôn ngữ này cho việc lập trình do đó dễ dàng hơn và nhanh hơn rất nhiều so với ngôn ngữ lập trình bậc thấp. Khác với ngôn ngữ bậc thấp, chương trình viết bằng các ngôn ngữ bậc cao nói chung có thể sử dụng được trên nhiều loại máy tính khác nhau.

Các chương trình viết bằng một ngôn ngữ bậc cao muốn chạy được thì phải được dịch sang ngôn ngữ máy bằng cách sử dụng chương trình dịch. Chương trình dịch có thể chia ra thành hai loại là trình biên dịch và trình thông dịch.

Một số ngôn ngữ bậc cao như C, C++ yêu cầu loại chương trình dịch được gọi là **trình biên dịch** (*compiler*). Trình biên dịch dịch mã nguồn thành mã máy – dạng có thể thực thi được. Kết quả của việc dịch là một chương trình thực thi được và có thể chạy nhiều lần mà không cần dịch lại. Ví dụ, với ngôn ngữ C++

một trình biên dịch rất phổ biến là gcc/g++ trong bộ GNU Compiler Collection (GCC) chạy trong các môi trường Unix/Linux cũng như Windows. Ngoài ra, Microsoft Visual C++ là trình biên dịch C++ phổ biến nhất trong môi trường Windows. Một số ngôn ngữ bậc cao khác như Perl, Python yêu cầu loại chương trình dịch gọi là **trình thông dịch** (*interpreter*). Khác với trình biên dịch, thay vì dịch toàn bộ chương trình một lần, trình thông dịch vừa dịch vừa chạy chương trình, dịch đến đâu chạy chương trình đến đó.

Trong môn học này, C++ được chọn làm ngôn ngữ thể hiện. Đây là một trong những ngôn ngữ lập trình chuyên nghiệp được sử dụng rộng rãi nhất trên thế giới. Trong phạm vi nhập môn của môn học này, C++ chỉ được giới thiệu ở mức rất cơ bản, rất nhiều tính năng mạnh của C++ sẽ không được nói đến hoặc chỉ được giới thiệu sơ qua. Người học nên tiếp tục tìm hiểu về ngôn ngữ C++, vượt ra ngoài giới hạn của cuốn sách này.

1.4. Môi trường lập trình bậc cao

Để lập trình giải quyết một bài toán bằng ngôn ngữ lập trình bậc cao, bạn cần có công cụ chính là: chương trình soạn thảo, chương trình dịch dành cho ngôn ngữ sử dụng, và các thư viện chuẩn của ngôn ngữ sử dụng (*standard library*), và chương trình tìm lỗi (*debugger*).

Các bước cơ bản để xây dựng và thực hiện một chương trình:

1. **Soạn thảo**: Mã nguồn chương trình được viết bằng một phần mềm soạn thảo văn bản dạng text và lưu trên ổ đĩa. Ta có thể dùng những phần mềm soạn thảo văn bản đơn giản nhất như Notepad (trong môi trường Windows) hay vi (trong môi trường Unix/Linux), hoặc các công cụ soạn thảo trong môi trường tích hợp để viết mã nguồn chương trình. Mã nguồn C++ thường đặt trong các tệp với tên có phần mở rộng là `.cpp`, `cxx`, `.cc`, hoặc `.C` (viết hoa).

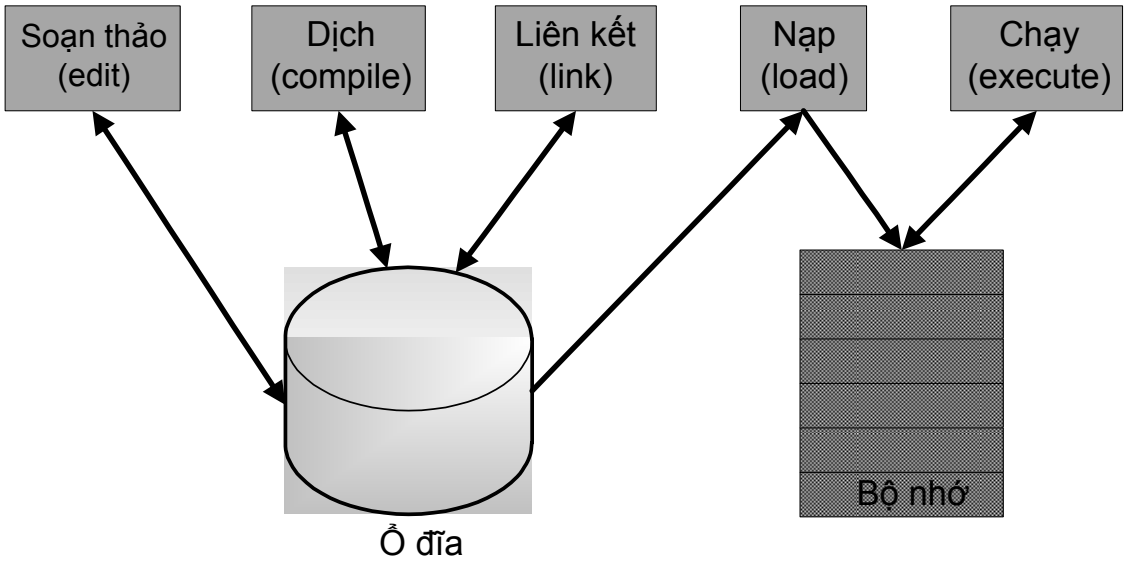
2. **Dịch**: Dùng trình biên dịch dịch mã nguồn chương trình ra thành các đoạn mã máy riêng lẻ (gọi là "object code") lưu trên ổ đĩa. Các trình biên dịch phổ biến cho C++ là vc.exe trong bộ Microsoft Visual Studio hay gcc trong bộ GNU Compiler với các tham số thích hợp để dịch và liên kết để tạo ra tệp chạy được. Với C++, ngay trước khi dịch còn có giai đoạn tiền xử lý (*preprocessing*) khi các định hướng tiền xử lý được thực thi để làm các thao tác như bổ sung các tệp văn bản cần dịch hay thay thế một số chuỗi văn bản. Một số định hướng tiền xử lý quan trọng sẽ được giới thiệu dần trong cuốn sách này.

3. **Liên kết**: Một tệp mã nguồn thường không chứa đầy đủ những phần cần thiết cho một chương trình hoàn chỉnh. Nó thường dùng đến dữ liệu hoặc hàm được định nghĩa trong các tệp khác hoặc trong thư viện chuẩn. Trình liên kết (*linker*) kết nối các đoạn mã máy riêng lẻ với nhau và với các thư viện có sẵn để tạo ra một chương trình mã máy hoàn chỉnh chạy được.

4. **Nạp**: Trình nạp (*loader*) sẽ nạp chương trình dưới dạng mã máy vào bộ nhớ. Các thành phần bổ sung từ thư viện cũng được nạp vào bộ nhớ.

5. **Chạy**: CPU nhận và thực hiện lần lượt các lệnh của chương trình, dữ liệu và kết quả thường được ghi ra màn hình hoặc ổ đĩa.

Thường thì không phải chương trình nào cũng chạy được và chạy đúng ngay ở lần chạy thử đầu tiên. Chương trình có thể có lỗi cú pháp nên không qua được bước dịch, hoặc chương trình dịch được nhưng gặp lỗi trong khi chạy. Trong những trường hợp đó, lập trình viên phải quay lại bước soạn thảo để sửa lỗi và thực hiện lại các bước sau đó.



Hình 1.2: Các bước cơ bản để xây dựng một chương trình.

Để thuận tiện cho việc lập trình, các công cụ soạn thảo, dịch, liên kết, chạy... nói trên được kết hợp lại trong một môi trường lập trình tích hợp (IDE – *integrated development environment*), trong đó, tất cả các công đoạn đối với người dùng chỉ còn là việc chạy các tính năng trong một phần mềm duy nhất. IDE rất hữu ích cho các lập trình viên. Tuy nhiên, đối với những người mới học lập trình, thời gian đầu nên tự thực hiện các bước dịch và chạy chương trình thay vì thông qua các chức năng của IDE. Như vậy, người học sẽ có thể nắm

được bản chất các bước của quá trình xây dựng chương trình, hiểu được bản chất và đặc điểm chung của các IDE, tránh tình trạng bị phụ thuộc vào một IDE cụ thể.

Ví dụ về các IDE phổ biến là Microsoft Visual Studio – môi trường lập trình thương mại cho môi trường Windows, và Eclipse – phần mềm miễn phí với các phiên bản cho cả môi trường Windows cũng như Unix/Linux, cả hai đều hỗ trợ nhiều ngôn ngữ lập trình.

Dành cho C++, một số môi trường lập trình tích hợp phổ biến là Microsoft Visual Studio, Dev-C++, Code::Blocks, KDevelop. Mỗi môi trường có thể hỗ trợ một hoặc nhiều trình biên dịch. Chẳng hạn Code::Blocks hỗ trợ cả GCC và MSVC Do C++ có các phiên bản khác nhau.

Có những bản cài đặt khác nhau của C++. Các bản ra đời trước chuẩn C++ 1998 (ISO/IEC 14882) có thể không hỗ trợ đầy đủ các tính năng được đặc tả trong chuẩn ANSI/ISO 1998. Bản C++ do Microsoft phát triển khác với bản C++ của GNU. Tuy nhiên, các trình biên dịch hiện đại hầu hết hỗ trợ C++ chuẩn, ta cũng nên chọn dùng các phần mềm này. Ngôn ngữ C++ được dùng trong cuốn sách này tuân theo chuẩn ISO/IEC 14882, còn gọi là "C++ thuần túy" (*pure C++*).

1.5. Lỗi và tìm lỗi

Trong và sau quá trình lập trình, chúng ta phải tiến hành kiểm thử và sửa lỗi chương trình. Có ba loại lỗi thường gặp: lỗi cú pháp, lỗi run-time và lỗi lô-gic.

Lỗi cú pháp là do lập trình viên viết sai với các quy tắc cú pháp của ngôn ngữ lập trình, chẳng hạn thiếu dấu chấm phảy ở cuối lệnh. Chương trình biên dịch sẽ phát hiện ra các lỗi cú pháp và cung cấp thông báo về vị trí mà nó cho là có lỗi. Nếu trình biên dịch nói rằng chương trình có lỗi cú pháp thì chắc chắn là có lỗi cú pháp trong chương trình. Tuy nhiên, lỗi là chỗ nào thì trình biên dịch chỉ có thể đoán, và nó có thể đoán sai.

Lỗi run-time là lỗi xuất hiện trong khi chương trình đang chạy. Lỗi dạng này sẽ gây ra thông báo lỗi và ngừng chương trình. Ví dụ là khi chương trình thực hiện phép chia cho 0.

Lỗi lô-gic có nguyên nhân là do thuật toán không đúng, hoặc do lập trình viên gặp sai sót khi thể hiện thuật toán bằng ngôn ngữ lập trình (ví dụ viết nhầm dấu cộng thành dấu trừ). Khi có lỗi lô-gic, chương trình của bạn có thể dịch và chạy bình thường, nhưng kết quả của chương trình đưa ra lại có trường hợp sai hoặc

hoạt động của chương trình không như mong đợi. Lỗi lô-gic là loại lỗi khó tìm ra nhất.

Nếu chương trình của bạn dịch và chạy không phát sinh thông báo lỗi, thậm chí chương trình cho ra kết quả có đúng với một vài bộ dữ liệu test, điều đó không có nghĩa chương trình của bạn hoàn toàn không có lỗi. Để có thể chắc chắn hơn về tính đúng đắn của chương trình, bạn cần chạy thử chương trình với nhiều bộ dữ liệu khác nhau và so sánh kết quả mà chương trình tạo ra với kết quả mong đợi.

1.6. Lịch sử C và C++

Ngôn ngữ lập trình C được tạo ra bởi Dennis Ritchie (phòng thí nghiệm Bell) và được sử dụng để phát triển hệ điều hành UNIX. Một trong những đặc điểm nổi bật của C là độc lập với phần cứng (portable), tức là chương trình có thể chạy trên các loại máy tính và các hệ điều hành khác nhau. Năm 1983, ngôn ngữ C đã được chuẩn hóa và được gọi là ANSI C bởi Viện chuẩn hóa quốc gia Hoa Kỳ (*American National Standards Institute*). Hiện nay ANSI C vẫn là ngôn ngữ lập trình chuyên nghiệp và được sử dụng rộng rãi để phát triển các hệ thống tính toán hiệu năng cao.

Ngôn ngữ lập trình C++ do Bjarne Stroustrup (thuộc phòng thí nghiệm Bell) phát triển trên nền là ngôn ngữ lập trình C và cảm hứng chính từ ngôn ngữ lập trình Simula67. So với C, C++ là ngôn ngữ an toàn hơn, khả năng diễn đạt cao hơn, và ít đòi hỏi các kỹ thuật bậc thấp. Ngoài những thế mạnh thừa kế từ C, C++ hỗ trợ trừu tượng hóa dữ liệu, lập trình hướng đối tượng và lập trình tổng quát, C++ giúp xây dựng dễ dàng hơn những hệ thống lớn và phức tạp.

Bắt đầu từ phiên bản đầu tiên năm 1979 với cái tên "C with Classes" (C kèm lớp đối tượng)[1] với các tính năng cơ bản của lập trình hướng đối tượng, C++ được phát triển dần theo thời gian. Năm 1983, cái tên "C++" chính thức ra đời, các tính năng như hàm ảo (*virtual function*), hàm trùng tên và định nghĩa lại toán tử (*overloading*), hằng ... được bổ sung. Năm 1989, C++ có thêm lớp trừu tượng, đa thừa kế, hàm thành viên tĩnh, hằng hàm, và thành viên kiểu protected. Các bổ sung cho C++ trong thập kỉ sau đó là khuôn mẫu (*template*), không gian tên (*namespace*), ngoại lệ (*exception*), các toán tử đổi kiểu dữ liệu mới, và kiểu dữ

[1] Theo lời kể của Bjarne Stroustrup tại trang cá nhân của ông tại trang web của phòng thí nghiệm AT&T http://www2.research.att.com/~bs/bs_faq.html#invention

liệu Boolean. Năm 1998, lần đầu tiên C++ được chính thức chuẩn hóa quốc tế bởi tổ chức ISO, kết quả là chuẩn ISO/IEC 14882[2].

Đi kèm với sự phát triển của ngôn ngữ là sự phát triển của thư viện chuẩn C++. Bên cạnh việc tích hợp thư viện chuẩn truyền thống của C với các sửa đổi nhỏ cho phù hợp với C++, thư viện chuẩn C++ còn có thêm thư viện stream I/O phục vụ việc vào ra dữ liệu dạng dòng. Chuẩn C++ năm 1998 tích hợp thêm phần lớn thư viện STL (*Standard Template Library – thư viện khuôn mẫu chuẩn*)[3]. Phần này cung cấp các cấu trúc dữ liệu rất hữu ích như vector, danh sách, và các thuật toán như sắp xếp và tìm kiếm.

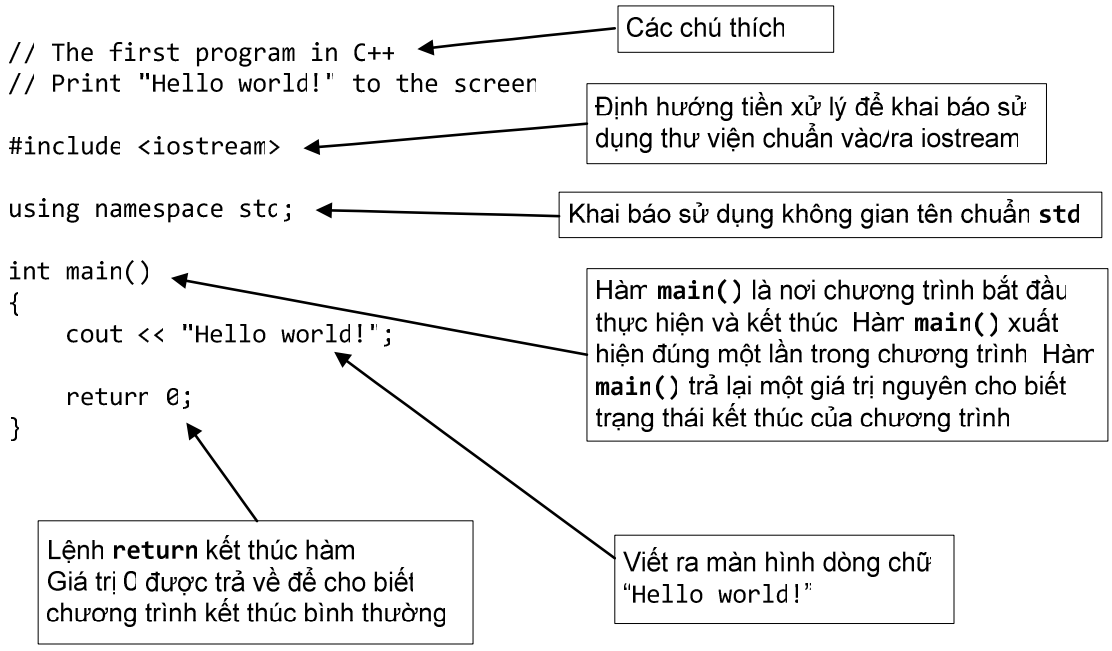
Hiện nay, C++ là một trong các ngôn ngữ lập trình chuyên nghiệp được sử dụng rộng rãi nhất.

1.7. Chương trình C++ đầu tiên

Chương trình đơn giản trong Hình 1.3 sẽ hiện ra màn hình dòng chữ "Hello world!". Trong chương trình có những đặc điểm quan trọng của C++. Ta sẽ xem xét từng dòng.

[2] Văn bản này (ISO/IEC 14882:1998) sau đó được phát hiện lỗi chỉnh sửa vào năm 2003, thành phiên bản ISO/IEC 14882:2003.

[3] STL vốn không nằm trong thư viện chuẩn mà là một thư viện riêng do HP và sau đó là SGI phát triển.



Hình 1.3: Chương trình C++ đầu tiên.

Hai dòng đầu tiên bắt đầu bằng chuỗi `//` là các dòng chú thích chương trình. Đó là kiểu chú thích dòng đơn. Các dòng chú thích không gây ra hoạt động gì của chương trình khi chạy, trình biên dịch bỏ qua các dòng này. Ngoài ra còn có dạng chú thích kiểu C dùng chuỗi /* và */ để đánh dấu điểm bắt đầu và kết thúc chú thích. Các lập trình viên dùng chú thích để giải thích và giới thiệu về nội dung chương trình.

Dòng thứ ba, `#include <iostream>` là một định hướng tiền xử lý (*preprocessor directive*) – chỉ dẫn về một công việc mà trình biên dịch cần thực hiện trước khi dịch chương trình. `#include` là khai báo về thư viện sẽ được sử dụng trong chương trình, trong trường hợp này là thư viện vào ra dữ liệu iostream trong thư viện chuẩn C++.

Tiếp theo là hàm `main`, phần không thể thiếu của mỗi chương trình C++. Nó bắt đầu từ dòng khai báo header của hàm:

```
int main()
```

Mỗi chương trình C++ thường bao gồm một hoặc nhiều hàm, trong đó có đúng một hàm có tên `main`, đây là nơi chương trình bắt đầu thực hiện và kết thúc. Bên trái từ `main` là từ khóa `int`, nó có nghĩa là hàm `main` sẽ trả về một giá trị là số nguyên. Từ khóa là những từ đặc biệt mà C++ dành riêng cho những mục

đích cụ thể. Chương 4 sẽ cung cấp thông tin chi tiết về khái niệm hàm và việc hàm trả về giá trị.

Thân hàm `main` được bắt đầu và kết thúc bởi cặp ngoặc `{}`, bên trong đó là chuỗi các lệnh mà khi chương trình chạy chúng sẽ được thực hiện tuần tự từ lệnh đầu tiên cho đến lệnh cuối cùng. Hàm `main` trong ví dụ đang xét có chứa hai lệnh. Mỗi lệnh đều kết thúc bằng một dấu chấm phẩy, các định hướng tiền xử lý thì không.

Lệnh thứ nhất gồm cout, toán tử <<, xâu kí tự "Hello world!", và dấu chấm phẩy. Nó chỉ thị cho máy tính thực hiện một nhiệm vụ: in ra màn hình chuỗi kí tự nằm giữa hai dấu nháy kép – "Hello world!". Khi lệnh được thực thi, chuỗi kí tự `Hello world` sẽ được gửi cho `cout` – luồng dữ liệu ra chuẩn của C++, thường được nối với màn hình. Chi tiết về vào ra dữ liệu sẽ được nói đến trong Chương 8. Chuỗi kí tự nằm giữa hai dấu nháy kép được gọi là một **xâu kí tự** (*string*). Để ý dòng

```
using namespace std;
```

nằm ở gần đầu chương trình. Tất cả thành phần của thư viện chuẩn C++, trong đó có `cout` được dùng đến trong hàm `main`, được khai báo trong một không gian tên (*namespace*) có tên là `std`. Dòng trên thông báo với trình biên dịch rằng chương trình ví dụ của ta sẽ sử dụng đến một số thành phần nằm trong không gian tên `std`. Nếu không có khai báo trên, tiền tố `std::` sẽ phải đi kèm theo tên của tất cả các thành phần của thư viện chuẩn được dùng trong chương trình, chẳng hạn `cout` sẽ phải được viết thành `std::cout`. Chi tiết về không gian tên nằm ngoài phạm vi của cuốn sách này, người đọc có thể tìm hiểu tại các tài liệu [1] hoặc [2]. Nếu không có lưu ý đặc biệt thì tất cả các chương trình ví dụ trong cuốn sách này đều sử dụng khai báo sử dụng không gian tên `std` như ở trên.

Lệnh thứ hai nhảy ra khỏi hàm và trả về giá trị 0 làm kết quả của hàm. Đây là bước có tính chất quy trình do C++ quy định hàm `main` cần trả lại một giá trị là số nguyên cho biết trạng thái kết thúc của chương trình. Giá trị 0 được trả về ở cuối hàm `main` có nghĩa rằng hàm đã kết thúc thành công.

Để ý rằng tất các lệnh nằm bên trong cặp ngoặc {} của thân hàm đều được lùi đầu dòng một mức. Với C++, việc này không có ý nghĩa về cú pháp. Tuy nhiên, nó lại giúp cho cấu trúc chương trình dễ thấy hơn và chương trình dễ hiểu hơn đối với người lập trình. Đây là một trong các điểm quan trọng trong các quy ước về phong cách lập trình. Phụ lục A sẽ hướng dẫn chi tiết hơn về các quy ước này.

Đến đây ta có thể sửa chương trình trong Hình 1.3 để in ra lời chào "Hello world!" theo các cách khác nhau. Chẳng hạn, ta có thể in ra cùng một nội dung như cũ nhưng bằng hai lệnh gọi cout:

```
cout << "Hello "; cout << "world!";
```

hoặc in ra lời chào trên nhiều dòng bằng cách chèn vào giữa xâu kí tự các kí tự xuống dòng (kí tự đặc biệt được kí hiệu là \n):

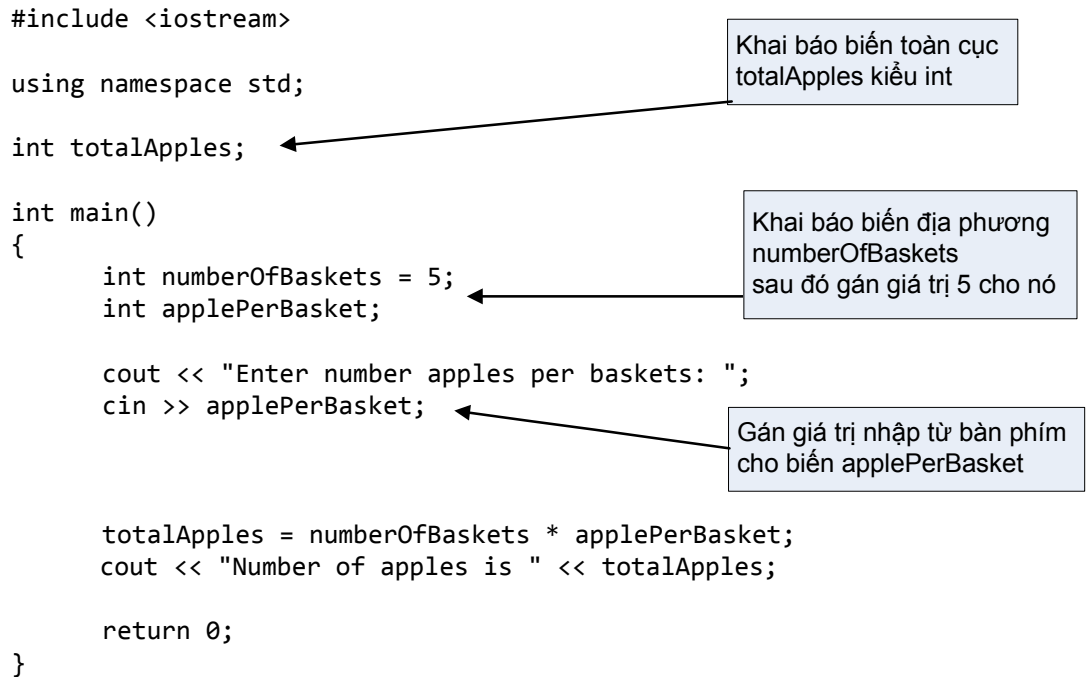
```
cout << "Hello \n world!\n";
```

Bài tập

1. Trình bày các bước chính để giải quyết một bài toán.
2. Tại sao cần phải có chương trình dịch, sự khác biệt giữa trình biên dịch và trình thông dịch?
3. Sự khác biệt giữa ngôn ngữ lập trình bậc cao và ngôn ngữ lập trình bậc thấp?
4. Liệt kê ra tất cả các ngôn ngữ lập trình bậc thấp, bậc cao mà bạn biết
5. Trình bày về môi trường lập trình bậc cao.
6. Làm quen với môi trường lập trình Dev-C.
7. Viết một chương trình C++ để hiện ra màn hình tên của bạn.
8. Trình bày sự khác biệt giữa C và C++.
9. Trình bày các loại lỗi thường gặp khi lập trình.

Chương 2. Biến, kiểu dữ liệu và các phép toán

Đa số chương trình không chỉ có những hoạt động đơn giản như là hiển thị một xâu kí tự ra màn hình mà còn phải thao tác với dữ liệu. Trong một chương trình, **biến** là tên của một vùng bộ nhớ được dùng để lưu dữ liệu trong khi chương trình chạy. Dữ liệu lưu trong một biến được gọi là giá trị của biến đó. Chúng ta có thể truy nhập, gán hay thay đổi giá trị của các biến, khi biến được gán một giá trị mới, giá trị cũ sẽ bị ghi đè lên.



Hình 2.1: Khai báo và sử dụng biến.

Hình 2.1 minh họa việc khai báo và sử dụng biến. Trong đó, các dòng

```
int totalApples;
int numberOfBaskets = 5;
int applePerBasket;
```

là các dòng **khai báo biến**. `totalApples`, `numberOfBaskets`, và `applePerBasket` là các **tên biến**. Các khai báo trên có nghĩa rằng `totalApples`, `numberOfBaskets`, và `applePerBasket` là dữ liệu thuộc kiểu `int`, nghĩa là các biến này sẽ giữ giá trị kiểu nguyên. Dòng khai báo `numberOfBaskets` có một điểm khác với hai dòng còn lại, đó là `numberOfBaskets` được khởi tạo với giá trị 5. C++ quy định rằng tất cả các biến đều phải được khai báo với một cái tên

và một kiểu dữ liệu trước khi biến đó được sử dụng. Các biến thuộc cùng một kiểu có thể được khai báo trên cùng một dòng, cách nhau bởi một dấu phảy. Chẳng hạn, có thể thay hai dòng khai báo cho `numberOfBaskets`, và `applePerBasket` bằng:

```
int numberOfBaskets = 5, applePerBasket;
```

Chương trình trong Hình 2.1 yêu cầu người dùng nhập số táo trong mỗi giỏ (`applePerBasket`), tính tổng số táo (`totalApples`) với dữ kiện đã biết là số giỏ táo (`numberOfBasket`), rồi in ra màn hình. Cụ thể, dòng

```
cout << "Enter number apples per baskets: ";
```

in ra màn hình xâu kí tự `Enter number apples per baskets:` . Đó là lời mời nhập dữ liệu, là hướng dẫn dành cho người sử dụng chương trình.

Dòng tiếp theo

```
cin >> applePerBasket;
```

đọc dữ liệu được người dùng nhập vào từ đầu vào chuẩn – thường là từ bàn phím. Khi chạy lệnh này, chương trình sẽ đợi người dùng nhập vào một giá trị cho biến `applePerBasket`. Người dùng đáp ứng bằng cách gõ vào một số nguyên dưới dạng chuỗi các chữ số rồi nhấn phím Enter để gửi các chữ số đó cho máy tính. Đến lượt nó, máy tính biến đổi chuỗi các chữ số nó nhận được thành một giá trị kiểu nguyên rồi chép giá trị này vào biến `applePerBasket`. Tương ứng với `cout` là đối tượng quản lý dòng dữ liệu ra chuẩn của thư viện C++, `cin` là đối tượng quản lý dòng dữ liệu vào chuẩn, thường là từ bàn phím.

Tiếp theo là lệnh gán

```
totalApples = numberOfBaskets * applePerBasket;
```

Lệnh này tính tích giá trị của hai biến `numberOfBaskets` và `applePerBasket` rồi gán kết quả cho biến `totalApples`, trong đó `*` là kí hiệu của phép nhân và `=` là kí hiệu của phép gán.

Lệnh in kết quả ra màn hình

```
cout << "Number of apples is " << totalApples;
```

hiển thị liên tiếp hai thành phần: xâu kí tự `"Number of apples is "` và giá trị của biến `totalApples`.

Ngoài việc in giá trị của một biến, C++ còn cho phép ta in kết quả của một biểu thức. Do đó, ta có một lựa chọn khác là gộp công việc của hai lệnh trên (tính tích hai biến và in tích ra màn hình) vào một lệnh:

```
cout << "Number of apples is " << numberOfBaskets *
applePerBasket;
```

Khi đó, biến `totalApples` vốn được dùng để lưu trữ kết quả của phép tính trở nên không còn cần thiết, ta có thể xóa bỏ dòng khai báo biến này. Để ý là lệnh trên dài và chiếm cả sang dòng thứ hai. C++ cho phép một lệnh nằm trên nhiều dòng, dấu chấm phảy cuối mỗi lệnh sẽ giúp trình biên dịch hiểu đâu là kết thúc của lệnh.

2.1. Kiểu dữ liệu

Mỗi biến phải được khai báo để lưu giữ giá trị thuộc một kiểu dữ liệu nào đó. Ngôn ngữ lập trình bậc cao thường có hai loại kiểu dữ liệu: các kiểu dữ liệu cơ bản và các kiểu dữ liệu dẫn xuất.

2.1.1. Kiểu dữ liệu cơ bản

Kiểu dữ liệu cơ bản là kiểu dữ liệu do ngôn ngữ lập trình định nghĩa sẵn. Ví dụ như các kiểu số nguyên – `char`, `int`, `long int`. Biến thuộc kiểu nguyên được dùng để lưu các số có giá trị nguyên. Đối với kiểu cơ bản chúng ta thường quan tâm đến kích thước bộ nhớ của kiểu dữ liệu, giới hạn giá trị mà kiểu dữ liệu đó có thể lưu giữ. Đối với các kiểu dấu chấm động (*floating-point*) để lưu các giá trị thuộc kiểu số thực, chúng ta còn quan tâm đến độ chính xác của kiểu dữ liệu đó. Tài liệu chuẩn C++ không quy định chính xác số byte cần dùng để lưu các biến thuộc các kiểu dữ liệu cơ bản trong bộ nhớ mà chỉ quy định yêu cầu về kích thước của kiểu dữ liệu này so với kiểu dữ liệu kia. Các kiểu nguyên có dấu, signed char, short int, int và long int, phải có kích thước tăng dần. Mỗi kiểu nguyên có dấu tương ứng với một kiểu nguyên không dấu với cùng kích thước. Các kiểu nguyên không dấu không thể biểu diễn giá trị âm nhưng có thể biểu diễn số giá trị dương nhiều gấp đôi kiểu có dấu tương ứng. Tương tự, các kiểu chấm động, `float`, `double` và `long double` cũng phải có kích thước tăng dần. Bảng 2.1 liệt kê một số kiểu dữ liệu cơ bản của C++ với kích thước được nhiều bản cài đặt C++ sử dụng.

Kiểu	Mô tả	Kích thước thông dụng (byte)	Phạm vi (tương ứng với kích thước)
char	ký tự / số nguyên nhỏ	1	các kí tự ASCII signed char: -128 → 127, hoặc unsighed char: 0 → 255
bool	giá trị Boolean	1	true hoặc false
short	số nguyên	2	signed short: -32767→ 32767 unsigned short: 0→ 65536
int	số nguyên lớn	4	signed int: 2147483648 → 2147483647 unsigned int: 0 → -4294967296
long	số nguyên rất lớn	4	signed long: 2147483648 → 2147483647 unsigned long: 0 → -4294967296
float	số thực	4	+/- $1.4023 \times 10^{-45} \rightarrow 3.4028 \times 10^{+38}$
double	số thực với độ chính xác cao	8	+/- $4.9406 \times 10^{-324} \rightarrow 1.7977 \times 10^{308}$
long double	số thực với độ chính xác rất cao	8	+/- $4.9406 \times 10^{-324} \rightarrow 1.7977 \times 10^{308}$

Bảng 2.1: Một số kiểu dữ liệu cơ bản trong C++.

Một số lưu ý:

- Kích thước và phạm vi của các kiểu dữ liệu cơ bản phụ thuộc vào hệ thống mà chương trình được biên dịch tại đó. Tuy nhiên, ở tất cả các hệ thống, kiểu `char` bao giờ cũng có kích thước là 1 byte; các kiểu dữ liệu `char`, `short`, `int`, `long` phải có kích thước tăng dần; còn các kiểu `float`, `double`, `long double` phải có độ chính xác cao dần.
- Kiểu `char` dùng để lưu các kí tự đơn (có mã nhỏ hơn 256), chẳng hạn như chữ cái La-tinh, chữ số, hay các kí hiệu. Trong C++, một kí tự đơn được đóng trong cặp nháy đơn, ví dụ `'A'`. Để lưu các kí tự có mã lớn hơn 255, ta có thể sử dụng kiểu `wchar_t`.

- Kiểu dữ liệu `bool` chỉ có hai giá trị `true` và `false`. Ta có thể dùng biến thuộc kiểu này để lưu câu trả lời của những câu hỏi đúng/sai chẳng hạn như "Có phải index lớn hơn 100?" hay lưu các trạng thái "Ta đã tìm thấy giá trị âm chưa?". Các giá trị `true` và `false` trong C++ thực ra chỉ là 0 và 1.

2.1.2. Kiểu dữ liệu dẫn xuất

Kiểu dữ liệu dẫn xuất là kiểu dữ liệu được xây dựng từ các kiểu dữ liệu cơ bản bằng các toán tử như * (con trỏ), & (tham chiếu), [] (mảng), () hàm, hoặc được định nghĩa bằng cơ chế `struct` hay `class`. Các kiểu dữ liệu được định nghĩa bằng cơ chế `struct` hay `class` còn được gọi là các kiểu dữ liệu có cấu trúc hoặc kiểu dữ liệu trừu tượng. Chi tiết về các kiểu dữ liệu dẫn xuất sẽ được trình bày dần dần trong các chương sau.

2.2. Khai báo và sử dụng biến

Để bắt đầu sử dụng một biến, chúng ta phải tiến hành hai bước: đặt cho biến một cái tên hợp lệ và khai báo biến.

2.2.1. Định danh và cách đặt tên biến

Định danh (*identifier*) là thuật ngữ trong ngôn ngữ lập trình khi nói đến tên (tên biến, tên hàm, tên lớp…). Định danh là một chuỗi kí tự (bao gồm các chữ cái a..z, A..Z, chữ số 0..9, dấu gạch chân '_') viết liền nhau. Định danh không được bắt đầu bằng chữ số và không được trùng với các **từ khóa** (những từ mang ý nghĩa đặc biệt) của ngôn ngữ lập trình. Lưu ý, C++ phân biệt chữ cái hoa và chữ cái thường.

Cách đặt tên biến tuân thủ theo cách đặt tên định danh. Ví dụ về các tên biến:

- `_sinhvien`, `sinhvien_01`, `sinhVien_01` là các tên biến hợp lệ khác nhau
- `01sinhvien`, `sinhviên`, `"sinhvien"` là các tên biến không hợp lệ

Tên biến nên dễ đọc, và gợi nhớ đến công dụng của biến hay kiểu dữ liệu mà biến sẽ lưu trữ. Ví dụ, nếu cần dùng một biến để lưu số lượng quả táo, ta có thể đặt tên là `totalApples`. Không nên sử dụng các tên biến chỉ gồm một kí tự và không có ý nghĩa như `a` hay `b`.

2.2.2. Khai báo biến

Các ngôn ngữ lập trình định kiểu mạnh, trong đó có C++, yêu cầu mỗi biến trước khi dùng phải được khai báo và biến phải thuộc về một kiểu dữ liệu nào đó. Có thể chia các biến thành hai loại:

- các biến được khai báo ở ngoài tất cả các chương trình con là **biến toàn cục,** có hiệu lực trên toàn bộ chương trình, chẳng hạn biến `totalApples` trong Hình 2.1.
- các biến được khai báo tại một chương trình con là **biến địa phương,** có hiệu lực ở bên trong chương trình con đó, chẳng hạn `numberOfBaskets` và `applePerBasket` trong Hình 2.1 là các biến địa phương của hàm `main` và chỉ có hiệu lực ở bên trong hàm `main`.

Chương 4 sẽ nói kĩ hơn về hai loại biến trên.

Trong C++, biến có thể được khai báo gần như bất cứ đâu trong chương trình, miễn là trước dòng đầu tiên sử dụng đến biến đó (xem Hình 2.1).

Một biến địa phương đã được khai báo nhưng chưa được gán một giá trị nào được gọi là biến chưa được khởi tạo. Giá trị của biến chưa được khởi tạo thường là không xác định. Để tránh tình trạng này, ta có thể khởi tạo giá trị của các biến bằng cách gán giá trị ngay tại lệnh khai báo biến.

2.3. Hằng

Hằng là một loại biến đặc biệt mà giá trị của nó được xác định tại thời điểm khai báo và không được thay đổi trong suốt chương trình. Để khai báo một hằng, ta thêm từ khóa `const` vào phía trước lệnh khai báo biến. Ví dụ:

```
const float PI = 3.1415926535;
const float SCREEN_WIDTH = 317.24;
```

Hằng được dùng để đặt tên cho các giá trị không thay đổi được dùng trong chương trình, chẳng hạn như độ rộng màn hình như trong ví dụ trên. Công dụng của hằng là mỗi khi cần thay đổi giá trị đó, ta chỉ cần sửa lệnh khai báo hằng thay vì tìm và sửa giá trị tương ứng tại tất cả các vị trí dùng đến nó. Ngoài ra, một cái tên có ý nghĩa đặt cho một giá trị được dùng đi dùng lại sẽ giúp cho chương trình dễ đọc và dễ hiểu hơn.

2.4. Các phép toán cơ bản

2.4.1. Phép gán

Phép gán là cách gắn một giá trị cho một biến hoặc thay đổi giá trị của một biến. Lệnh gán trong C++ là:

biến = biểu thức;

trong đó dấu bằng ("=") được gọi là dấu gán hay toán tử gán. Lưu ý, dấu bằng trong C++ không dùng để so sánh giá trị như trong một số ngôn ngữ lập trình khác. Ví dụ

```
symbol = 'A';
```

là phép gán giá trị 'A' cho biến symbol, không phải là biểu thức so sánh xem giá trị của symbol có phải là 'A' hay không.

Công việc của phép gán là tính giá trị của biểu thức bên phải dấu gán rồi lưu giá trị đó vào trong biến nằm bên trái dấu gán.

Biểu thức có thể là một số, một biến, hoặc một biểu thức phức tạp. Lưu ý rằng một biến có thể xuất hiện ở cả hai bên của dấu gán. Ví dụ lệnh sau tăng giá trị của biến apples thêm 2.

```
apples = apples + 2;
```

Điểm đặc biệt của C++ là bản thân phép gán cũng chính là một biểu thức với giá trị trả về là kết quả của phép gán. Ví dụ, phép gán x = 3 là một biểu thức có giá trị bằng 3.

2.4.2. Các phép toán số học

C++ hỗ trợ năm phép toán số học sau: `+` (cộng), `-` (trừ), `*` (nhân), `/` (chia), `%` (modulo – lấy phần dư của phép chia). Phép chia được thực hiện cho hai giá trị kiểu nguyên sẽ cho kết quả là thương nguyên. Ví dụ biểu thức 4 / 3 cho kết quả bằng 1, còn 3 / 5 cho kết quả bằng 0.

Một số phép gán kèm theo biểu thức xuất hiện nhiều lần trong một chương trình, vì vậy C++ cho phép viết các phép gán biểu thức đó một cách gắn ngọn hơn, sử dụng các phép gán phức hợp (`+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, `|=`).

Cách sử dụng phép gán phức hợp `+=` như sau:

biến += biểu thức; tương đương *biến = biến + biểu thức;*

Ví dụ:

`apples += 2;` tương đương `apples = apples + 2;`

Các phép gán phức hợp khác được sử dụng tương tự.

C++ còn cung cấp các phép toán `++` (hay `--`) để tăng (giảm) giá trị của biến lên một đơn vị. Ví dụ:

`apples++` hay `++apple` có tác dụng tăng `apples` thêm 1 đơn vị

`apples--` hay `--apple` có tác dụng giảm `apples` đi 1 đơn vị

Khác biệt giữa việc viết phép tăng/giảm ở trước biến (tăng/giảm trước) và viết phép tăng/giảm ở sau biến (tăng/giảm sau) là thời điểm thực hiện phép tăng/giảm, thể hiện ở giá trị của biểu thức. Phép tăng/giảm trước được thực hiện trước khi biểu thức được tính giá trị, còn phép tăng/giảm sau được thực hiện sau khi biểu thức được tính giá trị. Ví dụ, nếu `apples` vốn có giá trị 1 thì các biểu thức `++apples` hay `apples++` đều có hiệu ứng là `apples` được tăng từ 1 lên 2. Tuy nhiên, `++apples` là biểu thức có giá trị bằng 2 (tăng `apples` trước tính giá trị), trong khi `apples++` là biểu thức có giá trị bằng 1 (tăng `apples` sau khi tính giá trị biểu thức). Nếu ta chỉ quan tâm đến hiệu ứng tăng hay giảm của các phép ++ hay – thì việc phép toán được đặt trước hay đặt sau không quan trọng. Đó cũng là cách dùng phổ biến nhất của các phép toán này.

Lưu ý, cần hết sức cẩn thận khi sử dụng các phép tăng và giảm trong các biểu thức. Việc này không được khuyến khích vì tuy nó có thể tiết kiệm được một hai dòng lệnh nhưng lại làm giảm tính trong sáng của chương trình.

2.4.3. Các phép toán quan hệ

Các phép toán quan hệ được sử dụng để so sánh giá trị hai biểu thức. Các phép toán này cho kết quả bằng 0 nếu đúng và khác 0 nếu sai. Ta sử dụng giá trị của một biểu thức quan hệ như là một giá trị thuộc kiểu `bool`. Ví dụ:

```
bool enoughApples = (totalApples > 10);
```

Các phép toán quan hệ trong ngôn ngữ C++ được liệt kê trong Bảng 2.2.

Ký hiệu toán học	Toán tử của C++	Ví dụ	Ý nghĩa
>	>	x > y	x lớn hơn y
<	<	x < y	x nhỏ hơn y
≥	>=	x >= y	x lớn hơn hoặc bằng y
≤	<=	x <= y	x nhỏ hơn hoặc bằng y
=	==	x == y	x bằng y
≠	!=	x != y	x khác y

Bảng 2.2: Các phép toán quan hệ.

Chú ý tránh nhầm lẫn giữa phép gán giá trị "=" và phép so sánh bằng "==". Những nhầm lẫn giữa các phép toán kiểu này thường dẫn đến những lỗi lôgic rất khó phát hiện.

2.4.4. Các phép toán lô-gic

Toán tử C++	Ý nghĩa	Ví dụ	Ý nghĩa của ví dụ
&	and	x && y	Cho giá trị đúng khi cả x và y đúng, ngược lại cho giá trị sai.
||	or	x || y	Cho giá trị đúng khi hoặc x đúng hoặc y đúng, ngược lại cho giá trị sai
!	not	!x	Phủ định của x. Cho giá trị đúng khi x sai; cho giá trị sai khi x đúng

Bảng 2.3: Các phép toán lô-gic.

Các phép toán lô-gic dành cho các toán hạng là các biểu thức quan hệ hoặc các giá trị bool. Kết quả của biểu thức lôgic là giá trị bool.

Ví dụ:

```
bool enoughApples = (apples > 3) && (apples < 10);
```

có kết quả là biến `enoughApples` nhận giá trị là câu trả lời của câu hỏi "biến `apples` có giá trị lớn hơn 3 và nhỏ hơn 10 hay không?".

2.4.5. Độ ưu tiên của các phép toán

Dưới đây là mức độ ưu tiên của một số phép toán thường gặp. Thứ tự của chúng như sau:

Các phép toán nằm trong cặp dấu ngoặc ( ) có độ ưu tiên lớn nhất. Ví dụ:

2 * (1 + 3) cho kết quả bằng 8

Các phép toán *, /, +, -. Trong đó *, / có độ ưu tiên như nhau và cao hơn +, -. Ví dụ:

2 * 1 + 3 cho kết quả là 5

Các phép toán so sánh <, >, <=, >=. Ví dụ:

3 + 4 < 2 + 6 cho kết quả đúng

Các phép toán lô-gic có thứ thứ tự ưu tiên như sau: !, &&, ||. Ví dụ:

1 || 0 && 0 tương đương với 1 || (0 && 0) và cho kết quả 1

2.4.6. Tương thích giữa các kiểu

Về cơ bản, giá trị gán cho một biến nên cùng kiểu với biến đó. Khi một biến được gán một giá trị không đúng với kiểu dữ liệu của biến đó, thì giá trị đó sẽ được chuyển đổi sang kiểu của biến (*type conversion*). Một vài trường hợp thường gặp trong việc chuyển kiểu là:

Chuyển đổi giữa số thực và số nguyên

```
int  x = 2.5;    // x nhận giá trị 2
double y = 3.5;   // y nhận giá trị 3,5
x = y;            // x nhận giá trị 3
```

Phép chia của số nguyên

```
int divisor = 4;
int dividend = 6;
int quo = dividend/divisor;  // quo nhận giá trị 1.
```

Lưu ý: Phép chia một số nguyên cho một số nguyên sẽ cho kết quả là một số nguyên. Muốn kết quả là một số thực thì ít nhất một trong hai số phải là số thực.

Bài tập

1. Hãy viết chương trình tính diện tích của một hình tròn. Bán kính là một số thực và được nhập vào từ bàn phím. Diện tích hình tròn được hiện ra màn hình.

2. Nhập từ bàn phím hai số nguyên là chiều cao của Peter và Essen. Hãy tính xem Peter cao gấp bao nhiêu lần Essen. Ví dụ, nếu chiều cao của Peter là 180, chiều cao của Essen là 150, thì hiện ra màn hình dòng chữ "Peter is 1.2 times as tall as Essen".

3. Nhập từ bàn phím một số nguyên là nhiệt độ dưới dạng độ F (Fahrenheit), hãy hiện ra màn hình nhiệt độ dưới dạng độ C (Celsius). Sinh viên tự tìm hiểu công thức chuyển đổi.

4. Viết một chương trình trong đó có hai hằng số `WIDTH` với giá trị bằng 3.17654, và hằng số `LENGTH` với giá trị bằng 10.03212. Tính và hiện ra màn hình diện tích của hình chữ nhật với hai cạnh là `WIDTH` và `LENGTH`.

5. Nhập từ bàn phím bốn số nguyên a, b, c và d. Hãy tính và hiện ra màn hình giá trị của biểu thức:

```
(a + b) > (c + d) || (a - b) > (c - d)
```

6. Trình bày sự khác biệt giữa biến địa phương và biến toàn cục.

7. Viết một chương trình có chứa hai biến: biến địa phương `bonus` và biến toàn cục `score`. Nhập giá trị hai biến từ bàn phím, tính và hiện ra màn hình tổng của `score` và `bonus`.

8. Hãy tính giá trị của biểu thức: `1 + 3 < 2 * 4 - 1 && 1`

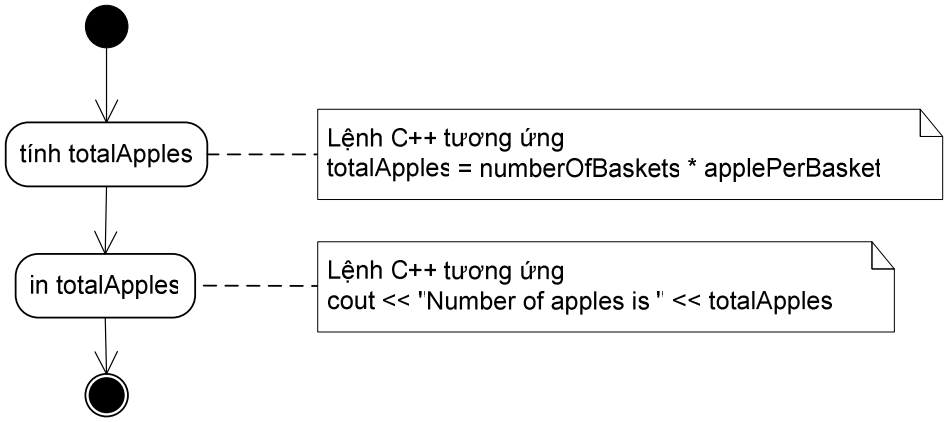
Chương 3. Các cấu trúc điều khiển

Như đã nói ở phần trên, chương trình máy tính mà chúng ta lập trình tạo ra thực chất là một tập các lệnh. Các chương trình được viết bằng ngôn ngữ bậc cao thường chứa một hàm chính (trong C++ là hàm `main`), nơi chương trình bắt đầu thực hiện các lệnh.

Trong chương này chúng ta sẽ tìm hiểu thứ tự thực hiện các lệnh trong một chương trình.

3.1. Luồng điều khiển

Luồng điều khiển của chương trình là thứ tự các lệnh (hành động) mà chương trình thực hiện. Cho đến chương này, chúng ta mới gặp thứ tự đơn giản: thứ tự tuần tự, nghĩa là các hành động được thực hiện đúng theo thứ tự mà chúng được viết trong chương trình. Ví dụ, Hình 3.1 là sơ đồ minh họa một cấu trúc tuần tự điển hình, trong đó hai hành động và các mũi tên biểu diễn thứ tự thực hiện các hành động.



Hình 3.1: Sơ đồ chuyển trạng thái của một đoạn lệnh có thứ tự thực thi tuần tự.

Trong chương này, ta sẽ làm quen với các luồng điều khiển phức tạp hơn. Hầu hết các ngôn ngữ lập trình, trong đó có C++, cung cấp hai loại lệnh để kiểm soát luồng điều khiển:

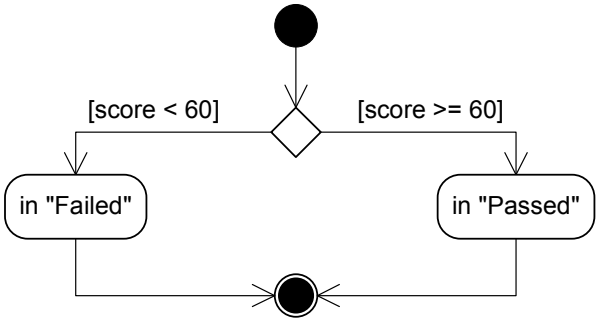
- **lệnh rẽ nhánh** (*branching*) chọn một hành động từ danh sách gồm nhiều hành động.
- **lệnh lặp** (*loop*) thực hiện lặp đi lặp lại một hành động cho đến khi một điều kiện dừng nào đó được thỏa mãn.

Hai loại lệnh đó tạo thành các **cấu trúc điều khiển** (*control structure*) bên trong chương trình.

3.2. Các cấu trúc rẽ nhánh

3.2.1. Lệnh `if-else`

Lệnh `if-else` (hay gọi tắt là lệnh `if`) cho phép rẽ nhánh bằng cách lựa chọn thực hiện một trong hai hành động. Ví dụ, trong một chương trình xếp loại điểm thi, nếu điểm của sinh viên nhỏ hơn 60, sinh viên đó được coi là trượt, nếu không thì được coi là đỗ.



Hình 3.2: Sơ đồ một lệnh if.

Thuật toán nhỏ này được thể hiện bằng sơ đồ trong Hình 3.2. Trong đó hình quả trám biểu diễn điểm rẽ nhánh, nơi chương trình đưa ra quyết định xem nên rẽ theo hướng nào. Từ điểm rẽ nhánh có các mũi tên đi ra, mỗi mũi tên kèm theo một điều kiện. Luồng điều khiển sẽ đi theo mũi tên nào mà điều kiện của nó được thỏa mãn.

Cũng có thể trình bày thuật toán bằng mã giả như sau:

```
If student's score is less than 60
        print "Failed"
else
        print "Passed"
```

Thể hiện thuật toán trên bằng một lệnh `if-else` của C++, ta có đoạn mã:

```
if (score < 60)
    cout << "Failed";
else
    cout << "Passed";
```

Khi chương trình chạy một lệnh `if-else`, đầu tiên nó kiểm tra biểu thức điều kiện nằm trong cặp ngoặc đơn sau từ khóa `if`. Nếu biểu thức có giá trị bằng `true` thì lệnh nằm sau từ khóa `if` sẽ được thực hiện. Ngược lại, lệnh nằm sau `else` sẽ được thực hiện. Chú ý là biểu thức điều kiện phải được đặt trong một cặp ngoặc đơn.

```
#include <iostream>

using namespace std;

int main()
{
    int score;
    cout << "Enter your score: ";
    cin >> score; //get the score from the keyboard

    if (score < 60)
        cout << "Sorry. You've failed the course.\n";
    else
        cout << "Congratulations! You've passed the course.\n";

    return 0;
}
```

Kết quả chạy chương trình

```
Enter your score: 20
Sorry. You've failed the course.

Enter your score: 70
Congratulations! You've passed the course.
```

Hình 3.3: Ví dụ về cấu trúc `if-else`.

Chương trình ví dụ trong Hình 3.3 yêu cầu người dùng nhập điểm rồi in ra các thông báo khác nhau tùy theo điểm số đủ đỗ hoặc trượt. Để ý rằng các lệnh nằm bên trong khối `if-else` cần được lùi đầu dòng một mức để chương trình trông sáng sủa dễ đọc.

Trong cấu trúc rẽ nhánh `if-else`, ta có thể bỏ phần `else` nếu không muốn chương trình thực hiện hành động nào nếu điều kiện không thỏa mãn. Chẳng hạn, nếu muốn thêm một lời khen đặc biệt cho điểm số xuất sắc từ 90 trở lên, ta có thể thêm lệnh `if` sau vào trong chương trình tại Hình 3.3.

```
if (score >= 90)
    cout << "Excellent!";
```

Ta có thể dùng các cấu trúc `if-else` lồng nhau để tạo ra điều kiện rẽ nhánh phức tạp. Hình 3.3 là một ví dụ đơn giản với chỉ hai trường hợp và một lệnh `if`. Xét một ví dụ phức tạp hơn: cho trước điểm số (lưu tại biến `score` kiểu `int`), xác định xếp loại học lực A, B, C, D, F tùy theo điểm đó. Quy tắc xếp loại là: nếu điểm từ 90 trở lên thì đạt loại A, điểm từ 80 tới dưới 90 đạt loại B, v.v.. Tại đoạn mã xét các trường hợp của xếp loại điểm, ta có thể dùng cấu trúc `if-else` lồng nhau như sau:

```
if (score >= 90)
    grade = 'A';
else
    if (score >= 80)
        grade = 'B';
    else
        if (score >= 70)
            grade = 'C';
        else
            if (score >= 60)
                grade = 'D';
            else
                grade = 'F';
```

Để ý rằng mỗi khối lệnh `if-else` sau nằm trong phần `else` của khối lệnh `if-else` liền trước. Nếu `score` không dưới 90, điều kiện của cả bốn lệnh `if-else` đều cho kết quả `true`. Tuy nhiên, do điều kiện của lệnh `if-else` ngoài cùng thỏa mãn, chỉ có lệnh gán đầu tiên `grade = 'A'` được thực thi, còn toàn bộ phần `else` của lệnh `if-else` ngoài cùng, bao gồm ba lệnh `if-else` còn lại, bị bỏ qua.

Với các khối lệnh `if-else` lồng nhau như ở trên, khi phải lùi đầu dòng quá xa về bên phải, dòng còn lại quá ngắn khiến mỗi lệnh có thể phải trải trên vài dòng, dẫn đến việc chương trình khó đọc. Do đó, đa số lập trình viên C++ thường

dùng kiểu lùi đầu dòng như dưới đây. (Hai cách lùi đầu dòng này là như nhau đối với trình biên dịch vì nó bỏ qua các kí tự trắng.)

```
if (score >= 90)
    grade = 'A';
else if (score >= 80)
    grade = 'B';
else if (score >= 70)
    grade = 'C';
else if (score >= 60)
    grade = 'D';
else
    grade = 'F';
```

Hình 3.4 là chương trình hoàn chỉnh thực hiện nhiệm vụ yêu cầu người dùng nhập điểm số sau đó tính và in ra xếp loại học lực tương ứng.

```
#include <iostream>

using namespace std;

int main()
{
    int score;
    char grade;

    cout << "Enter your score: ";
    cin >> score;

    if (score >= 90)
        grade = 'A';
    else if (score >= 80)
        grade = 'B';
    else if (score >= 70)
        grade = 'C';
    else if (score >= 60)
        grade = 'D';
    else
        grade = 'F';

    cout << "Grade = " << grade << endl;

    return 0;
}
```

Hình 3.4: Cấu trúc `if-else` lồng nhau.

Một điều cần đặc biệt lưu ý là nếu muốn thực hiện nhiều hơn một lệnh trong mỗi trường hợp của lệnh `if-else`, ta cần dùng cặp ngoặc {} bọc tập lệnh đó thành một **khối chương trình**. Ví dụ, phiên bản phức tạp hơn của lệnh `if` trong Hình 3.3:

```
if (score < 60) {
    cout << "Failed" << endl;
    cout << "You have to take this course again" << endl;
}
else {
    cout << "Congratulations!!!" << endl;
    cout << "You passed this course. ";
}
```

Để tránh lỗi quên cặp ngoặc, có một lời khuyên là tập thói quen dùng cặp ngoặc ngay cả khi khối lệnh chỉ bao gồm đúng một lệnh. Chẳng hạn, xét khối lệnh lồng nhau dưới đây:

```
if (mathGrade == 'A')
    if (artGrade == 'A')
        cout << "You have got two A's!" << endl;
else
    cout << "You don't have grade A for math." << endl;
```

Thông báo trong phần `else` đáng ra cần được thực thi khi xếp hạng của môn Toán (`mathGrade`) không đạt A. Tuy nhiên, trong thực tế, trình biên dịch lại hiểu rằng phần `else` đó cùng cặp với lệnh `if` thứ hai – lệnh `if` gần nhất, nghĩa là:

```
if (mathGrade == 'A')
    if (artGrade == 'A')
        cout << "You have got two A's!" << endl;
    else
        cout << "You don't have grade A for math." << endl;
```

Các lỗi lôgic dạng này thường khó phát hiện. Đối với C++, việc lùi đầu dòng chỉ nhằm mục đích để lập trình viên dễ đọc chương trình chứ không có ý nghĩa với trình biên dịch. Cách giải quyết là sử dụng cặp ngoặc {} để chỉ rõ cho trình biên dịch rằng lệnh `if` thứ hai nằm trong lệnh `if` ngoài cùng và rằng phần `else` cùng cặp với lệnh `if` ngoài cùng đó:

```
if (mathGrade == 'A') {
    if (artGrade == 'A')
        cout << "You have got two A's!" << endl;
}
else
    cout << "You don't have grade A for math." << endl;
```

3.2.2. Lệnh `switch`

Khi chúng ta muốn viết một cấu trúc rẽ nhánh có nhiều lựa chọn, ta có thể sử dụng nhiều lệnh `if-else` lồng nhau. Tuy nhiên, trong trường hợp việc lựa chọn rẽ nhánh phụ thuộc vào giá trị (kiểu số nguyên hoặc kí tự) của một biến hay biểu thức, ta có thể sử dụng cấu trúc `switch` để chương trình dễ hiểu hơn. Lệnh `switch` điển hình có dạng như sau:

```
switch (biểu_thức) {
    case hằng_1:
        tập_lệnh_1; break;
    case hằng_2:
        tập_lệnh_2; break;
    ...
    default:
        tập_lệnh_mặc_định;
}
```

Khi lệnh `switch` được chạy, *biểu_thức* được tính giá trị và so sánh với *hằng_1*. Nếu bằng nhau, chuỗi lệnh kể từ *tập_lệnh_1* được thực thi cho đến khi gặp lệnh `break` đầu tiên, đến đây chương trình sẽ nhảy tới điểm kết thúc cấu trúc `switch`. Nếu *biểu_thức* không có giá trị bằng *hằng_1*, nó sẽ được so sánh với *hằng_2*, nếu bằng nhau, chương trình sẽ thực thi chuỗi lệnh kể từ *tập_lệnh_2* tới khi gặp lệnh `break` đầu tiên thì nhảy tới cuối cấu trúc `switch`. Quy trình cứ tiếp diễn như vậy. Cuối cùng, nếu *biểu_thức* có giá trị khác với tất cả các giá trị đã được liệt kê (*hằng_1*, *hằng_2*, ...), chương trình sẽ thực thi *tập_lệnh_mặc_định* nằm sau nhãn `default:` nếu như có nhãn này (không bắt buộc).

Ví dụ, lệnh sau so sánh giá trị của biến `grade` với các hằng kí tự 'A', 'B', 'C' và in ra các thông báo khác nhau cho từng trường hợp.

```
switch (grade) {
    case 'A':
        cout << "Grade = A"; break;
    case 'B':
        cout << "Grade = B"; break;
    case 'C':
        cout << "Grade = C"; break;
    default:
        cout << "Grade is not A, B, or C.";
}
```

Nó tương đương với khối lệnh `if-else` lồng nhau sau:

```
if (grade == 'A')
    cout << "Grade = A";
else if (grade == 'B')
    cout << "Grade = B";
else if (grade == 'C')
    cout << "Grade = C";
else
    cout << "Grade is not A, B, or C.";
}
```

Lưu ý, các nhãn `case` trong cấu trúc `switch` phải là hằng chứ không thể là biến hay biểu thức. Nếu cần so sánh với biến hay biểu thức, ta nên dùng khối lệnh `if-else` lồng nhau.

Vấn đề đặc biệt của cấu trúc `switch` là các lệnh `break`. Nếu ta không tự gắn một lệnh `break` vào cuối chuỗi lệnh cần thực hiện cho mỗi trường hợp, chương trình sẽ chạy tiếp chuỗi lệnh của trường hợp sau chứ không tự động nhảy tới cuối cấu trúc `switch`. Ví dụ, đoạn chương trình sau sẽ chạy lệnh in thứ nhất nếu grade nhận một trong ba giá trị 'A', 'B', 'C' và chạy lệnh in thứ hai trong trường hợp còn lại:

```
switch (grade) {
    case 'A':
    case 'B':
    case 'C':
        cout << "Grade is A, B or C."; break;
    default:
        cout << "Grade is not A, B or C.";
}
```

Chương trình trong Hình 3.5 là một ví dụ hoàn chỉnh sử dụng cấu trúc `switch` để in ra các thông báo khác nhau tùy theo xếp loại học lực (`grade`) mà người dùng nhập từ bàn phím. Trong đó, `case 'A'` kết thúc với `break` sau chỉ một lệnh, còn `case 'B'` chạy tiếp qua `case 'C'`, `'D'` rồi mới gặp `break` và thoát khỏi lệnh `switch`. Nhãn `default` được dùng để xử lý trường hợp biến grade giữ giá trị không hợp lệ đối với xếp loại học lực. Trong nhiều chương trình, phần `default` thường được dùng để xử lý các trường hợp không mong đợi, chẳng hạn như để bắt lỗi các kí hiệu học lực không hợp lệ mà người dùng có thể nhập sai trong Hình 3.5.

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

int main()
{
    char grade;

    cout << "Enter your grade: ";
    cin >> grade;

    switch (grade)
    {
        case 'A':
            cout << "Excellent!\n"; break;
        case 'B':
            cout << "Great!\n";
        case 'C':
        case 'D':
            cout << "Well done!\n"; break;
        case 'F':
            cout << "Sorry, you failed.\n"; break;
        default:
            cout << "Error! Invalid grade.";
    }

    return 0;
}
```

Kết quả chạy chương trình

```
Enter your grade: A
Excellent!

Enter your grade: B
Great!
Well done!

Enter your grade: D
Well done!

Enter your grade: F
Sorry, you failed.
```

Hình 3.5: Ví dụ sử dụng cấu trúc **`switch`**.

3.3. Các cấu trúc lặp

Các chương trình thường cần phải lặp đi lặp lại một hoạt động nào đó. Ví dụ, một chương trình xếp loại học lực sẽ chứa các lệnh rẽ nhánh để gán xếp loại A, B, C… cho một sinh viên tùy theo điểm số của sinh viên này. Để xếp loại cho cả một lớp, chương trình sẽ phải lặp lại thao tác đó cho từng sinh viên trong lớp. Phần chương trình lặp đi lặp lại một lệnh hoặc một khối lệnh được gọi là một **vòng lặp**. Lệnh hoặc khối lệnh được lặp đi lặp lại được gọi là thân của vòng lặp. Cấu trúc lặp cho phép lập trình viên chỉ thị cho chương trình lặp đi lặp lại một hoạt động trong khi một điều kiện nào đó vẫn được thỏa mãn.

Khi thiết kế một vòng lặp, ta cần xác định thân vòng lặp thực hiện hành động gì. Ngoài ra, ta còn cần một cơ chế để quyết định khi nào vòng lặp sẽ kết thúc.

Mục này sẽ giới thiệu về các lệnh lặp mà C++ cung cấp.

3.3.1. Vòng `while`

Vòng `while` lặp đi lặp lại chuỗi hành động, gọi là thân vòng lặp, nếu như điều kiện lặp vẫn còn được thỏa mãn. Cú pháp của vòng lặp `while` như sau:

```
while (điều_kiện_lặp)
    thân_vòng_lặp
```

Cấu trúc này bắt đầu bằng từ khóa `while`, tiếp theo là điều kiện lặp đặt trong một cặp ngoặc đơn, cuối cùng là thân vòng lặp. Thân vòng lặp hay chứa nhiều hơn một lệnh và khi đó thì phải được gói trong một cặp ngoặc {}.

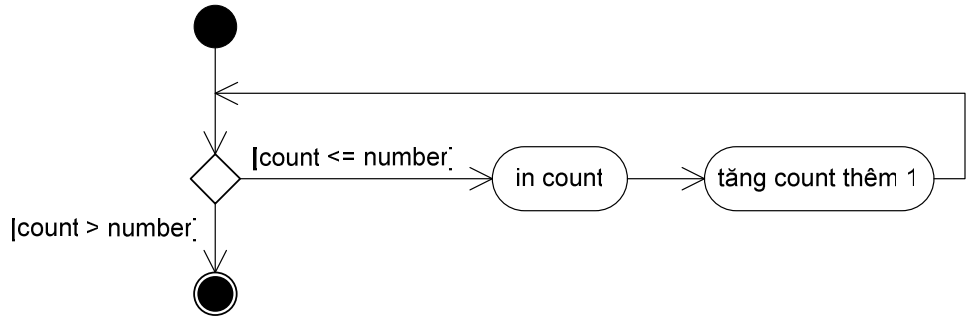
Khi thực thi một cấu trúc `while`, đầu tiên chương trình kiểm tra giá trị của biểu thức điều kiện, nếu biểu thức cho giá trị `false` (thực chất là bằng 0) thì nhảy đến điểm kết thúc lệnh `while`, còn nếu điều kiện lặp có giá trị `true` (thực chất là một giá trị nào đó khác 0) thì tiến hành thực hiện tập lệnh trong thân vòng lặp rồi quay trở lại kiểm tra điều kiện lặp, nếu không thỏa mãn thì kết thúc, nếu thỏa mãn thì lại thực thi thân vòng lặp rồi quay lại... Tập lệnh ở thân vòng lặp có thể làm thay đổi giá trị của biểu thức điều kiện từ `true` sang `false` (thực chất là từ khác 0 sang bằng 0) để dừng vòng lặp.

Ví dụ, xét một chương trình có nhiệm vụ đếm từ 1 đến một ngưỡng `number` cho trước. Đoạn mã đếm từ 1 đến `number` có thể được viết như sau:

```
count = 1;
while (count <= number)
{
    cout << count << ", ";
    count++;
}
```

Giả sử biến `number` có giá trị bằng 2, đoạn mã trên hoạt động như sau: Đầu tiên, biến `count` được khởi tạo bằng 1. Vòng while bắt đầu bằng việc kiểm tra điều kiện (`count <= number`), nghĩa là $1 \leq 2$, điều kiện thỏa mãn. Thân vòng lặp được thực thi lần thứ nhất: giá trị 1 của `count` được in ra màn hình kèm theo dấu phẩy, sau đó `count` được tăng lên 2. Vòng lặp quay về điểm xuất phát: kiểm tra điều kiện lặp, giờ là $2 \leq 2$, vẫn thỏa mãn. Thân vòng lặp được chạy lần thứ hai (in giá trị 2 của `count` và tăng `count` lên 3) trước khi quay lại điểm xuất phát của vòng lặp. Tại lần kiểm tra điều kiện lặp này, biểu thức $3 \leq 2$ cho giá trị `false`, vòng lặp kết thúc do điều kiện lặp không còn được thỏa mãn, chương trình chạy tiếp ở lệnh nằm sau cấu trúc `while` đang xét.

Cấu trúc `while` trong đoạn mã trên có thể được biểu diễn bằng sơ đồ trong Hình 3.6. Trong sơ đồ, mũi tên sau chuỗi hành động in biến `count` và tăng biến `count` quay ngược trở lại điểm kiểm tra điều kiện rẽ nhánh, chuỗi hành động sẽ lặp lại nếu điều kiện `count <= number` vẫn tiếp tục được thỏa mãn. Còn nếu `count > number`, vòng while đi đến điểm kết thúc, trao điều khiển cho lệnh tiếp theo trong chuỗi lệnh của chương trình.



Hình 3.6: Sơ đồ một vòng lặp while.

Chương trình hoàn chỉnh trong Hình 3.7 minh họa cách sử dụng vòng lặp `while` để in ra các số nguyên (biến `count`) từ 1 cho đến một ngưỡng giá trị do người dùng nhập vào từ bàn phím (lưu tại biến `number`). Kèm theo là kết quả của các lần chạy khác nhau với các giá trị khác nhau của `number`. Đặc biệt, khi người

dùng nhập giá trị 0 cho number, thân vòng while không chạy một lần nào, thể hiện ở việc không một số nào được in ra màn hình. Lí do là vì nếu number bằng 0 thì biểu thức count <= number ngay từ đầu vòng while đã có giá trị false.

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

int main()
{
   int count, number;

   cout << "Enter a number: ";
   cin >> number;

   count = 1;
   while (count <= number)
   {
      cout << count << ", ";
      count++;
   }
   cout << "BOOOOOM!" << endl;

   return 0;
}
```

Kết quả chạy chương trình

```
Enter a number: 2
1, 2, BOOOOOM!

Enter a number: 1
1, BOOOOOM!

Enter a number: 0
BOOOOOM!
```

Hình 3.7: Ví dụ về vòng lặp while.

Vòng lặp vô tận

Một trong những lỗi chương trình thường gặp là một vòng lặp không thể kết thúc, thân vòng lặp đó được lặp đi lặp lại mà không bao giờ ngừng. Một vòng lặp như vậy được gọi là **vòng lặp vô tận**. Thông thường, một vài lệnh trong thân một vòng while hay do-while sẽ làm thay đổi giá trị của một vài biến để biểu thức điều kiện cho giá trị false (hoặc 0), chẳng hạn lệnh tăng biến đếm

`count` trong Hình 3.7. Nếu (các) biến đó không được thay đổi giá trị một cách thích hợp hoặc khi điều kiện lặp được viết không chính xác (có thể do thuật toán sai hay nhầm lẫn khi viết mã chương trình), ta sẽ nhận được một vòng lặp vô tận do điều kiện lặp không bao giờ nhận giá trị `false`.

Hình 3.8 minh họa một vòng lặp vô tận khi biến `count` tăng dần đến vô hạn nhưng luôn nhận giá trị lẻ và không bao giờ có thể có giá trị bằng 12 để kết thúc vòng lặp `while`. Đây có thể gọi là lỗi lô-gic. Nếu đoạn chương trình có nhiệm vụ tính tổng các số 1, 3, 5, 7, 9, 11 thì điều kiện lặp nên được sửa thành `(count < 12)` để chương trình có thể chạy đúng.

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

int main(int argc, char *argv[])
{
    int sum;
    int count = 1;

    while (count != 12) {
        sum = sum + count;
        count = count + 2;
    }
    return 0;
}
```

Hình 3.8: Ví dụ về vòng lặp vô tận.

3.3.2. Vòng `do-while`

Vòng `do-while` rất giống với vòng `while`, khác biệt chính là ở chỗ thân vòng lặp sẽ được thực hiện trước, sau đó mới kiểm tra điều kiện lặp, nếu đúng thì quay lại chạy thân vòng lặp, nếu sai thì dừng vòng lặp. Khác biệt đó có nghĩa rằng thân của vòng `do-while` luôn được chạy ít nhất một lần, trong khi thân vòng `while` có thể không được chạy lần nào.

Công thức của vòng `do-while` tổng quát là:

```
do
   thân_vòng_lặp
while (điều_kiện_lặp);
```

Tương tự như ở vòng `while`, *thân_vòng_lặp* của vòng `do-while` có thể chỉ gồm một lệnh hoặc thường gặp hơn là một chuỗi lệnh được bọc trong một cặp ngoặc {}. Lưu ý dấu chấm phẩy đặt cuối toàn bộ khối `do-while`.

Công thức tổng quát của vòng `do-while` ở trên tương đương với công thức sau nếu dùng vòng `while`:

```
 thân_vòng_lặp
while (điều_kiện_lặp)
   thân_vòng_lặp
```

Để minh họa hoạt động của hai cấu trúc lặp `while` và `do-while`, ta so sánh hai đoạn mã dưới đây:

```
count = 1;
while (count <= number) {
    cout << count << ", ";
    count++;
}
```

```
count = 1;
do {
    cout << count << ", ";
    count++;
} while (count <= number);
```

Hai đoạn mã chỉ khác nhau ở chỗ một bên trái dùng vòng `while`, bên phải dùng vòng `do-while`, còn lại, các phần thân vòng lặp, điều kiện, khởi tạo đều giống hệt nhau. Đoạn bên trái được lấy từ ví dụ trong mục trước, nó in ra các số từ 1 đến `number`. Đoạn mã dùng vòng `do-while` bên phải cũng thực hiện công việc giống hệt đoạn bên trái, ngoại trừ một điểm: khi `number` nhỏ hơn 1 thì nó vẫn đếm 1 trước khi dừng vòng lặp – thân vòng lặp chạy 01 lần trước khi kiểm tra điều kiện.

Để minh họa rõ hơn, ta xem xét Hình 3.9 là bản sửa đổi của ví dụ trong Hình 3.7 bằng cách thay cấu trúc `while` bằng cấu trúc `do-while` và giữ nguyên các phần còn lại. Kết quả chạy chương trình cho thấy hoạt động của chương trình trong Hình 3.9 không hoàn toàn như mong muốn. Cụ thể, khi `number` được nhập giá trị 0, chương trình vẫn đếm một lần trong khi đáng ra không nên đếm lần nào như chương trình nguyên gốc trong Hình 3.7. Đó là một minh họa về sự khác biệt giữa hoạt động của hai cấu trúc `while` và `do-while`. Trong bài toán cụ thể này, cấu trúc `do-while` không phải là lựa chọn tốt so với cấu trúc `while`.

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

int main()
{
   int count, number;

   cout << "Enter a number: ";
   cin >> number;

   count = 1;
   do {
      cout << count << ", ";
      count++;
   } while (count <= number);

   cout << "BOOOOOM!" << endl;

   return 0;
}
```

Kết quả chạy chương trình

```
Enter a number: 2
1, 2, BOOOOOM!

Enter a number: 1
1, BOOOOOM!

Enter a number: 0
1, BOOOOOM!
```

***Hình 3.9: Ví dụ về vòng lặp* `do-while`.**

Tận dụng đặc điểm chạy một lần trước khi kiểm tra điều kiện, chương trình trong Hình 3.10 làm nhiệm vụ nhập từ bàn phím một chuỗi số kết thúc bằng số 0 và tính tổng của chúng. Công việc lặp đi lặp lại là nhập vào một số (`input`) rồi cộng dồn giá trị của số đó vào tổng (`sum`). Điều kiện lặp là số vừa nhập phải khác 0. Rõ ràng, không cần và không nên kiểm tra điều kiện lặp trước khi lần đầu thực hiện thân vòng lặp. Trong trường hợp này, cấu trúc `do-while` là lựa chọn tốt hơn so với cấu trúc `while`.

Chương trình trong Hình 3.10 còn là một ví dụ về một vòng lặp không dùng con đếm để điều khiển vòng lặp mà dùng một biến canh hay cờ trạng thái để đánh

dấu điểm kết thúc. Ở đây, 0 là giá trị canh đánh dấu phần tử dữ liệu cuối cùng mà người dùng nhập vào chương trình.

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

int main ()
{
    int total = 0;
    int input;

    do {
        cout << "Enter a number (0 to stop) ";
        cin >> input;
        total = total + input;
    } while (input != 0);

    cout << "The sum of the numbers you entered is " << total;
    return 0;
}
```

Kết quả chạy chương trình

```
Enter a number (0 to stop) 2
Enter a number (0 to stop) 4
Enter a number (0 to stop) 1
Enter a number (0 to stop) 0
The sum of the numbers you entered is 7
```

Hình 3.10: Vòng lặp `do-while` dùng biến canh.

3.3.3. Vòng `for`

Vòng `for` là cấu trúc hỗ trợ việc viết các vòng lặp mà số lần lặp được kiểm soát bằng biến đếm. Chẳng hạn, đoạn mã giả sau đây mô tả thuật toán in ra các số từ 1 đến `number`:

Làm nhiệm vụ sau đây đối với mỗi giá trị của `count` *từ* `1` *đến* `number`*:*
  In `count` *ra màn hình*

Đoạn mã giả đó có thể được viết bằng vòng `for` của C++ như sau:

```
for (count = 1; count <= number; count++)
    cout << count << ", ";
```

Với `number` có giá trị bằng 3, đoạn trình trên cho kết quả in ra màn hình là:

```
1, 2, 3,
```

Cấu trúc tổng quát của vòng lặp `for` là:

`for (` *khởi_tạo; điều_kiện_lặp; cập_nhật*`)`
  thân_vòng_lặp

Trong đó, biểu thức *khởi_tạo* thường khởi tạo con đếm điều khiển vòng lặp, *điều_kiện_lặp* xác định xem thân vòng lặp có nên chạy tiếp hay không (điều kiện này thường chứa ngưỡng cuối cùng của con đếm), và biểu thức *cập_nhật* làm tăng hay giảm con đếm. Cũng tương tự như ở các cấu trúc `if`, `while`..., nếu *thân_vòng_lặp* có nhiều hơn một lệnh thì cần phải bọc nó trong một cặp ngoặc {}. Lưu ý rằng cặp ngoặc đơn bao quanh bộ ba *khởi_tạo*, *điều_kiện_lặp*, *cập_nhật*, cũng như hai dấu chấm phẩy ngăn cách ba thành phần đó, là các thành bắt buộc của cú pháp cấu trúc `for`. Ba thành phần đó cũng có thể là biểu thức rỗng nếu cần thiết, nhưng kể cả khi đó vẫn phải có đủ hai dấu chấm phẩy.

Trong hầu hết các trường hợp, cấu trúc `for` tổng quát ở trên tương đương với cấu trúc lặp `while` sau (ngoại lệ được nói đến trong Mục 0):

```
khởi_tạo;
while (điều_kiện_lặp)
{
  thân_vòng_lặp
  cập_nhật;
}
```

Ta có thể khai báo biến ngay trong phần *khởi_tạo* của vòng `for`, chẳng hạn đối với biến con đếm. Nhưng các biến được khai báo tại đó chỉ có hiệu lực ở bên trong cấu trúc lặp (chi tiết về phạm vi của biến sẽ được nói đến trong Mục 4.4). Ví dụ:

```
for (int count = 1; count <= number; count++)
    cout << count << ", ";
```

Hình 3.11 minh họa cách sử dụng vòng lặp `for` để tính điểm trung bình từ điểm của 10 môn học (số môn học lưu trong biến `subjects`). Người dùng sẽ được yêu cầu nhập từ bàn phím điểm số của 10 môn học trong khi chương trình cộng dồn tổng của 10 điểm số này. Công việc mà chương trình cần lặp đi lặp lại 10 lần là: nhập điểm của một môn học, cộng dồn điểm đó vào tổng điểm. Đầu tiên vòng `for` sẽ tiến hành bước khởi tạo với mục đích chính là khởi tạo biến đếm. Việc khởi tạo chỉ được tiến hành duy nhất một lần. Trong ví dụ này, biến `count` được khai báo ngay tại vòng `for` và khởi tạo giá trị bằng 0. Tiếp theo vòng `for` sẽ tiến hành kiểm tra điều kiện lặp `count < subjects`. Nếu điều kiện sai, vòng lặp `for` sẽ kết thúc. Nếu điều kiện đúng, thân vòng lặp `for` sẽ được thực hiện (nhập một giá trị kiểu `float` rồi cộng dồn vào biến `sum`). Sau đó là bước cập nhật với nhiệm vụ tăng biến đếm thêm 1. Kết quả là vòng lặp sẽ chạy 10 lần với các giá trị `count` bằng 0, 1, .., 9 (khi `count` nhận giá trị 10 thì điều kiện lặp không còn đúng và vòng lặp kết thúc).

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

int main()
{
    float sum = 0;
    int subjects = 10;

    cout << "Enter the marks for " << subjects << " subjects: ";
    for (int count = 0; count < subjects; count++) {
        float mark;
        cin >> mark;
        sum += mark;
    }
    cout << "Average mark = " << sum/subjects;

    return 0;
}
```

Hình 3.11: Ví dụ về vòng lặp `for`.

Xét một bài toán khác: Tính lãi gửi tiền tiết kiệm. Một người gửi một sổ tiết kiệm kì hạn 12 tháng với số tiền ban đầu là 1.000.000 VNĐ, lãi suất 12% một năm. Giả sử tiền lãi hàng năm không được rút ra mà gộp chung vào vốn. Số tiền

nằm trong sổ tiết kiệm sau mỗi năm (kì hạn 12 tháng) được tính theo công thức sau:

$$amount = principal * (1 + interest\_rate)^{year}$$

trong đó *amount* là số tiền trong sổ tiết kiệm, *principal* là vốn ban đầu, *interest_rate* là lãi xuất hàng năm, và *year* là số năm đã gửi. Hãy tính và in ra số tiền trong sổ tiết kiệm vào thời điểm cuối các kì 12 tháng hàng năm trong vòng 5 năm theo dạng

```
Year  Amount in deposit
1     xxxx
2     xxxx
...
```

Thuật toán cho bài toán đó như sau:

Khởi tạo các biến amount, principal, interestRate
In đề mục hai cột Year và Amount in deposit ra màn hình
Với year chạy từ 1 đến 5, thực hiện các công việc sau:
  tính amount sau year năm theo công thức
  in amount theo cột

Chương trình hoàn chỉnh cho trong Hình 3.12. Trong đó có một số điểm cần chú ý.

Thứ nhất, vấn đề định dạng các giá trị số khi in ra màn hình. Định dạng mặc định khi in ra màn hình là định dạng khoa học, chẳng hạn số 1120000 ở hệ thập phân sẽ có dạng 1.12e+006. Ta sẽ nhìn thấy kết quả này nếu xóa bỏ dòng sau khỏi chương trình:

```
cout << fixed << setprecision( 2 ); // set number display
format
```

Dòng trên không hiển thị gì ra màn hình mà chỉ đặt cấu hình cho các lần in tiếp theo. Cụ thể, `fixed` giúp quy định rằng các giá trị thực in ra theo dạng có dấu chấm thập phân (hệ đo lường Anh-Mỹ, tương đương với dấu phảy thập phân của hệ mét), còn `setprecision( 2 )` quy định rằng các giá trị thực được in ra với độ chính xác 2 chữ số sau dấu chấm. Đó là hai trong nhiều phép định dạng luồng dữ liệu (*stream manipulator*). Các phép định dạng này nằm trong thư viện iomanip của thư viện chuẩn. Do đó, để sử dụng chúng, ta phải có định hướng tiền xử lý `#include <iomanip>` ở đầu tệp mã nguồn (xem phần đầu của chương trình trong Hình 3.12).

Thứ hai là sử dụng hàm thư viện toán học để tính lũy thừa. C++ không có phép tính lũy thừa. Thay vào đó, thư viện <cmath> trong thư viện chuẩn C++ cung cấp hàm pow. Hàm `pow( x, y )` cho kết quả là giá trị của phép tính x^y. Để dùng hàm này, ta cũng cần có định hướng tiền xử lý `#include <cmath>` ở đầu tệp mã nguồn.

```
#include <iostream>
#include <iomanip>
#include <cmath> // standard C++ math library

using namespace std;

int main()
{
    double amount; // amount at end of each year
    double principal = 1000000; // initial amount
    double interestRate = .12;

    cout << "Year\tAmount on deposit" << endl; // display
headers

    cout << fixed << setprecision( 2 ); // set number display
format

    // calculate amount on deposit for each of five years
    for ( int year = 1; year <= 5; year++)                {
        amount = principal * pow( 1.0 + interestRate, year );
        cout << year << "\t" << amount << endl;
    }

    return 0;
}
```

Kết quả chạy chương trình

```
Year    Amount on deposit
1       1120000.00
2       1254400.00
3       1404928.00
4       1573519.36
5       1762341.68
```

Hình 3.12: Tính tiền tiết kiệm hàng năm sử dụng vòng `for`.

3.4. Thuật toán và các cấu trúc điều khiển lồng nhau

3.5. Các lệnh `break` và `continue`

Như đã giới thiệu ở các mục trước, các vòng lặp `while`, `do-while`, và `for` đều kết thúc khi kiểm tra biểu thức điều kiện được giá trị `false` và chạy tiếp thân vòng lặp trong trường hợp còn lại. Các lệnh `break` và `continue` là các lệnh nhảy cho phép thay đổi luồng điều khiển đó.

Lệnh `break` khi được thực thi bên trong một cấu trúc lặp hay một cấu trúc `switch` có tác dụng lập tức chấm dứt cấu trúc đó, chương trình sẽ chạy tiếp ở lệnh nằm tiếp sau cấu trúc đó. Lệnh `break` thường được dùng để kết thúc sớm vòng lặp (thay vì đợi đến lượt kiểm tra điều kiện lặp) hoặc để bỏ qua phần còn lại của cấu trúc `switch` (như tại các ví dụ trong Mục 3.2.2).

Về ví dụ sử dụng lệnh `break` trong vòng lặp. Chẳng hạn, nếu ta sửa ví dụ trong Hình 3.11 để vòng `for` ngừng lại khi người dùng nhập điểm số có giá trị âm, ta có chương trình trong Hình 3.13. Với cài đặt này, khi người dùng nhập một điểm số có giá trị âm, điều kiện `(mark < 0)` sẽ cho kết quả `true`, chương trình thoát khỏi vòng `for` và chạy tiếp từ lệnh `if` nằm sau đó. Trong trường hợp đó, biến `count` chưa kịp tăng đến ngưỡng `subjects` (điều kiện lặp của vòng `for` chưa kịp bị phá vỡ). Do đó, biểu thức `(count >= subjects)` trong lệnh `if` sau đó có nghĩa "vòng `for` có chạy đủ `subjects` lần hay không?" hoặc "vòng `for` có bị ngắt giữa chừng bởi lệnh `break` hay không?", hay là "dữ liệu nhập vào có thành công hay không?".

```
#include <iostream>

using namespace std;

int main()
{
    float sum = 0;
    int count, subjects = 10;

    cout << "Enter the marks for " << subjects << " subjects: ";
    for (count = 0; count < subjects; count++) {
        float mark;
        cin >> mark;
        if (mark < 0) break;
        sum += mark;
    }
    if (count >= subjects)  // successful
        cout << "Average mark = " << sum/subjects;
    else
        cout << "Error: Invalid mark!";

    return 0;
}
```

Hình 3.13: Ví dụ về lệnh `break`.

Lệnh `continue` nằm trong một vòng lặp có tác dụng kết thúc lần lặp hiện hành của vòng lặp đó. Hình 3.14 là một bản sửa đổi khác của chương trình trong Hình 3.11. Trong phiên bản này, chương trình không ghi nhận điểm số có giá trị âm, cũng không kết thúc chương trình sau khi báo lỗi như bản trong Hình 3.13, mà yêu cầu nhập lại cho đến khi nào thành công. Khi gặp điểm số âm được nhập vào (biến `mark`), lệnh `continue` được thực thi có tác dụng bỏ qua đoạn lệnh ghi nhận điểm ở nửa sau của thân vòng `while` (đoạn cộng dồn vào tổng `sum` và tăng biến đếm `count`). Lần lặp được thực hiện sau đó sẽ yêu cầu nhập lại điểm cho môn học đang nhập dở (xem kết quả chạy chương trình trong Hình 3.14).

```
#include <iostream>

using namespace std;

int main()
{
    float sum = 0;
    int count = 0, subjects = 3;

    cout << "Enter the marks of " << subjects << " subjects.\n";
    while (count < subjects) {
        float mark;
        cout << "#" << count + 1 << ": ";
        cin >> mark;
        if (mark < 0)
        {
            cout << mark << " ignored\n";
            continue;
        }
        sum += mark;
        count++;
    }
    cout << "Average mark = " << sum/count;

    return 0;
}
```

Kết quả chạy chương trình

```
Enter the marks of 3 subjects.
#1: 8.0
#2: 7.2
#3: -5
-5 ignored
#3: 10.0
Average mark = 8.4
```

Hình 3.14: Ví dụ về lệnh `continue`.

Một điểm cần lưu ý là các lệnh `break` hay `continue` chỉ có tác dụng đối với vòng lặp trong cùng chứa nó. Chẳng hạn, nếu có hai vòng lặp lồng nhau và lệnh `break` nằm trong vòng lặp bên trong, thì khi được thực thi, lệnh `break` đó chỉ có tác dụng kết thúc vòng lặp bên trong.

Một vòng lặp không có `break` hay `continue` có cấu trúc đơn giản và dễ hiểu vì thời điểm kiểm tra điều kiện lặp là điểm duy nhất quyết định lặp hay dừng. Với

một lệnh break kết thúc sớm vòng lặp hoặc một lệnh continue kết thúc sớm lần lặp hiện hành, cấu trúc lặp sẽ khó hiểu. Vậy nên người ta khuyên nên tránh hai lệnh này nếu có thể.

3.6. Biểu thức điều kiện trong các cấu trúc điều khiển

Hầu hết các cấu trúc điều khiển mà ta nói đến trong chương này đều dùng đến một thành phần quan trọng: biểu thức điều kiện. Trong các ví dụ trước, ta mới chỉ dùng đến các điều kiện đơn giản, chẳng hạn `count <= number` hay `grade == 'A'`, với duy nhất một phép so sánh. Khi cần viết những điều kiện phức tạp hơn, cần đến nhiều điều kiện nhỏ, ta có thể kết hợp chúng bằng các phép toán lôgic `&&` (AND – và), `||` (OR – hoặc) và `!` (NOT – phủ định). Ví dụ:

Khi kiểm tra điều kiện `80 ≤ score < 90`, bất đẳng thức toán học này cần được tách thành hai điều kiện đơn. Bất đẳng đúng khi cả hai điều kiện đơn đều thỏa mãn. Đó là khi ta cần dùng phép toán lôgic `&&` (AND).

```
if (score >= 80 && score < 90)
    grade = 'B';
```

Khi một trong hai điều kiện xảy ra, hoặc tiền đã hết hoặc túi đã đầy, thì không thể mua thêm hàng. Trường hợp này, ta cần dùng phép toán lôgic `||` (OR).

```
if (moneyLeft <= 0 || bagIsFull)
    cout << "Can't buy anything more!";
```

Tiếp tục lặp trong khi dữ liệu vào chưa có giá trị bằng giá trị canh – đánh dấu điểm cuối của chuỗi dữ liệu:

```
while ( !(input == 0))
    ...
```

Ở đây, ta có thể dùng phép phủ định. Hoặc nhiều người chọn cách đơn giản hơn là dùng phép so sánh khác (`!=`):

```
while (input != 0)
    ...
```

Một điều cần đặc biệt lưu ý khi dùng toán tử so sánh bằng (`==`): tránh nhầm lẫn với phép gán (`=`). Nhầm lẫn dạng này sẽ dẫn đến những lỗi lôgic rất khó phát hiện. Chẳng hạn câu lệnh sẽ in ra lời khen `"Excellent!"` nếu xếp loại học lực đạt loại A:

```
if (grade == 'A')
    cout << "Excellent!";
```

Nếu ta viết nhầm thành:

```
if (grade = 'A')
    cout << "Excellent!";
```

thì khi kiểm tra điều kiện, câu lệnh này sẽ sửa giá trị của biến `grade` thành `'A'` thay vì so sánh. Biểu thức gán `grade = 'A'` có giá trị bằng giá trị được gán (`'A'`), giá trị này khác 0 nên được hiểu là `true`. Khi biểu thức điều kiện của lệnh `if` này luôn có giá trị `true`, đoạn lệnh sẽ hiển thị lời khen cho mọi loại học lực.

Một gợi ý cách tránh lỗi này là nên tập thói quen viết các biểu thức so sánh theo kiểu `('A' == grade)` thay vì `(grade == 'A')` theo thói quen thông thường. Với vị trí giá trị hằng ở bên trái, biến ở bên phải, thì khi ta chẳng may viết nhầm `==` thành `=` thì trình biên dịch sẽ phát hiện lỗi biên dịch "sai cú pháp của phép gán `'A' = grade`" thay vì im lặng do không thể phát hiện lỗi lôgic.

Bài tập

1. Nhập vào từ bàn phím một số nguyên nằm trong khoảng từ 0 đến 10 là điểm của một sinh viên. Hãy sử dụng cấu trúc `switch` để phân loại sinh viên đó:
   - Điểm < 5: kém
   - 5 ≤ Điểm ≤ 7: trung bình
   - 8 ≤ Điểm ≤ 9: giỏi
   - Điểm = 10: Xuất sắc
2. Nhập vào từ bàn phím hai số nguyên a và b. Nếu
   - a > b: hiện ra màn hình dòng chữ "`a is greater than b`".
   - a < b: hiện ra màn hình dòng chữ "`a is smaler than b`".
   - a = b: hiện ra màn hình dòng chữ "`a is equal to b`".
3. Nhập vào từ bàn phím ba số a, b, c. Hãy kiểm tra xem ba số đó có thỏa mãn là độ dài các cạnh của một tam giác hay không? Nếu có, hiện ra màn hình thông báo về diện thích của tam giác đó. Nếu không, thông báo ra màn hình ba số đó không phải là 3 cạnh của một tam giác.
4. Nhập từ bàn phím hai số nguyên. Tính thương của số lớn chia cho số bé.
5. Nhập từ bàn phím ba số nguyên. Tính tổng của số nhỏ nhất và số lớn nhất.
6. Nhập vào từ bán phím danh sách các số nguyên thể hiện cho điểm của một lớp học. Quá trình nhập dừng lại khi số nhập vào là một số âm. Hãy cho biết số lượng sinh viên trong lớp và tính điểm trung bình của lớp học vừa nhập.
7. Nhập vào từ bàn phím một số nguyên dương, hãy tính tổng tất cả các số nguyên dương lẻ nhỏ hơn số nhập từ bàn phím.
8. Nhập từ bàn phím một số nguyên dương. Hãy kiểm tra xem số nguyên dương đó có phải là số nguyên tố hay không.

Chương 4. Hàm

Các chương trình ví dụ mà ta gặp từ đầu đến giờ đều ngắn với duy nhất một hàm chính – hàm `main`. Khi cần giải quyết bài toán phức tạp hơn, hàm `main` sẽ dài hơn, và phức tạp hơn, dẫn đến chương trình khó đọc, khó hiểu, lập trình viên dễ mắc lỗi, và lỗi khó tìm.

Để giải quyết các bài toán lớn và phức tạp, người ta thường sử dụng chiến lược **chia để trị**. Nghĩa là, chia bài toán thành một chuỗi các bài toán nhỏ hơn, dễ hơn, và chúng có thể được giải một cách ít nhiều độc lập với nhau, sau đó kết hợp lại để có một lời giải hoàn chỉnh cho bài toán ban đầu. Trong lập trình, chia để trị được thể hiện bằng việc chia chương trình thành các **chương trình con**. Đoạn mã chứa trong thân một chương trình con sẽ được thực thi mỗi khi chương trình con đó được gọi (hay kích hoạt). Chương trình con chính là một trong các cơ chế cho phép mô-đun hóa chương trình.

Bên cạnh chiến lược chia để trị, việc sử dụng chương trình con còn mang lại cho lập trình viên khả năng **tái sử dụng** mã, sử dụng các chương trình con như là các khối lắp ghép để xây dựng các chương trình mới. Chương trình con cũng giúp tránh được các đoạn mã giống nhau lặp đi lặp lại trong một chương trình.

Người ta khuyên rằng độ dài mỗi chương trình con không nên vượt quá một trang màn hình để lập trình viên có thể kiểm soát tốt hoạt động của chương trình con đó.

Trong C++, tất cả các chương trình con đều được thể hiện bởi cấu trúc **hàm**. Ngay cả chương trình chính (`main`) cũng là một hàm. Một hàm được kích hoạt khi nó được gọi từ bên trong một hàm khác, chẳng hạn hàm `pow` được gọi từ trong hàm `main` của chương trình trong Hình 3.12. Khi hàm được gọi thực hiện xong công việc của mình, nó trả kết quả về cho nơi gọi nó hoặc đơn giản là trả quyền điều khiển về cho nơi gọi nó. Có thể so sánh hàm được gọi như là một người thợ, còn hàm gọi nó như là người chủ. Người chủ giao việc cho người thợ A, người thợ A thực hiện công việc và khi làm xong thì báo cáo kết quả lại cho người chủ. Người thợ A trong khi làm nhiệm vụ của mình cũng có thể yêu cầu một người thợ khác (B) thực hiện một nhiệm vụ con trong đó. Đến đây quan hệ giữa người thợ A và người thợ B cũng tương tự như quan hệ giữa người chủ và người thợ B.

Chương này sẽ nói về cách định nghĩa các chương trình con (hoặc các hàm) và các vấn đề liên quan như biến địa phương, cách truyền dữ liệu vào trong hàm.

4.1. Các hàm có sẵn

Các ngôn ngữ lập trình bậc cao thường cung cấp rất nhiều các hàm có sẵn cho người dùng sử dụng. Các hàm đó được chia thành các nhóm hay các thư viện khác nhau. Khi muốn sử dụng hàm có sẵn đó, ta chỉ cần chỉ ra cho chương trình biết là hàm đó nằm ở thư viện nào (sử dụng `#include`) và sau đó có thể sử dụng hàm đó một cách bình thường. Ví dụ như tại chương trình trong Hình 3.12, ta đã sử dụng hàm `pow` và khai báo include thư viện cmath chứa nó.

Một số thư viện của C++ hay được sử dụng:

- **iostream**: cung cấp các hàm liên quan đến nhập và xuất dữ liệu
- **cmath**: cung cấp các hàm liên quan đến toán học.
- **cstdlib**: cung cấp các hàm cho các mục đích chung như: quản lý bộ nhớ động, sinh số ngẫu nhiên, tìm kiếm-sắp xếp…
- **cstring**: cung cấp các hàm xử lý chuỗi kí tự như: tính độ dài, sao chép chuỗi…

Có thể tra cứu chi tiết về các thư viện này tại [4].

Ví dụ, Bảng 4.1 liệt kê danh sách một số hàm toán học thông dụng.

Bảng 4.1: Các hàm toán học trong thư viện cmath.

Hàm	**Miêu tả**
ceil(x)	số nguyên nhỏ nhất không nhỏ hơn x
cos(x)	hàm lượng giác cos x (x đo bằng radian)
exp(x)	e^x
fabs(x)	giá trị tuyệt đối của x
floor(x)	số nguyên lớn nhất không lớn hơn x
fmod(x, y)	phần dư của phép chia x/y (lấy giá trị kiểu thực)

log(x)	loga cơ số e của x (ln x)
log10(x)	loga cơ số 10 của x (log x)
pow(x, y)	x^y
sin(x)	hàm lượng giác sin x (x đo bằng radian)
sqrt(x)	căn bậc 2 của x (x là giá trị không âm)
tan(x)	hàm lượng giác tan x (x đo bằng radian)

4.2. Cấu trúc chung của hàm

Ngoài việc sử dụng các hàm có sẵn trong thư viện, lập trình viên còn tự xây dựng các hàm của riêng mình. Sau khi được định nghĩa, một hàm có thể được sử dụng nhiều lần. Hàm được định nghĩa theo công thức:

```
kiểu_trả_về tên_hàm (tham_số_1, tham_số_2,…)
{
        thân_hàm
}
```

trong đó:

- *kiểu_trả_về* là kiểu giá trị mà hàm sẽ trả về. Đây có thể là một trong các kiểu dữ liệu có sẵn của C++ hoặc một kiểu dữ liệu người dùng tự định nghĩa. Trong C++, nếu hàm không trả về giá trị nào, thì *kiểu_trả_về* được thay thế bằng từ khóa `void`.
- *tên_hàm* là định danh của hàm, cách đặt tên hàm tuân thủ theo cách đặt định danh và giống cách đặt tên biến.
- *tham_số_1*, *tham_số_2*,…là danh sách các tham số đầu vào của hàm. Tên đầy đủ của chúng là các **tham số hình thức** (*formal parameter*)[4]. Đây là phương tiện để truyền dữ liệu vào trong hàm.
- *thân_hàm* là nơi chứa chuỗi các lệnh.

[4] Còn được gọi tắt là "tham số".

Hình 4.1 mô tả cách khai báo, cài đặt và sử dụng một hàm tính diện tích hình chữ nhật (`calculateArea`) trong C++. Hàm `calculateArea` nhận hai tham số đầu vào là `rectangleLength` và `rectangleWidth`, nó tính diện tích hình chữ nhật (lưu tại biến `area`) và trả lại kết quả (lệnh `return area;`).

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

//Khai báo và định nghĩa một hàm mới
int calculateArea (int rectangleLength, int rectangleWidth)
{
    int area = rectangleLength * rectangleWidth;
    return area;  //hàm trả lại một giá trị
}

//hàm chính của chương trình
int main ()
{
    int length = 3;
    int width = 2;
    int area = calculateArea (length, width);
    cout << "The area is: " << area;
    return 0;
}
```

Hình 4.1: Ví dụ về khai báo và sử dụng hàm.

Lưu ý:

Việc khai báo và định nghĩa (thân hàm) hàm có thể tách ra thành hai phần khác nhau. Ta sẽ gặp ở những ví dụ ở các chương sau.

Biến được khai báo trong một hàm được gọi là biến địa phương của hàm đó, nghĩa là nó chỉ có phạm vi là bên trong thân hàm đó. Giả sử trong chương trình của ta có hai hàm, mỗi hàm khai báo một biến có tên giống nhau, ví dụ `area`. Ta sẽ có hai biến khác nhau tình cờ có tên giống nhau, và việc sửa đổi biến `area` trong hàm này sẽ không có ảnh hưởng gì đến biến `area` trong hàm kia. Tức là, đối với máy tính, hai biến này không có liên quan gì đến nhau.

4.3. Cách sử dụng hàm

Để sử dụng một hàm, ta gọi tên hàm, theo sau là cặp ngoặc đơn chứa các giá trị truyền vào cho hàm (xem cách gọi hàm `calculateArea` từ trong hàm `main` ở

Hình 4.1). Các giá trị truyền vào cho hàm được gọi là các **đối số** (*argument*)[5]. Như trong lời gọi hàm `calculateArea(length, width)` tại Hình 4.1, `length` và `width` là các đối số. Một hàm có thể được gọi nhiều lần với các đối số khác nhau.

Khi chương trình thực thi một lời gọi hàm, giá trị của các đối số được sao chép vào các tham số hình thức tương ứng, nghĩa là giá trị của `length` được chép vào `rectangleLength` và giá trị của `width` được chép vào `rectangleWidth`, rồi đoạn mã định nghĩa hàm được thực thi. Cuối cùng, lệnh `return` trả về một giá trị (là giá trị của biểu thức nằm ngay sau từ khóa `return`) cho đoạn mã chương trình đã gọi hàm này. Đồng thời, luồng điều khiển của chương trình cũng quay trở lại với nơi gọi hàm và các lệnh nối tiếp phía sau lời gọi hàm sẽ được thực thi.

4.4. Biến toàn cục và biến địa phương

Chương 2 đã giới thiệu các khái niệm cơ bản về biến. Trong mục này, ta sẽ bàn kĩ hơn về các loại biến, hiệu lực và phạm vi của biến.

Chúng ta đã biết rằng biến có các thuộc tính như: tên, kiểu, kích thước, giá trị. Ngoài ra, mỗi biến còn có các đặc điểm khác, cụ thể là:

- thời gian sống: biến tồn tại bao lâu trong bộ nhớ
- phạm vi: biến có thể được sử dụng tại những nơi nào trong chương trình

4.4.1. Phạm vi của biến

Phạm vi (*scope*) của một biến là những nơi chúng ta có thể sử dụng biến đó. Một biến có thể có phạm vi toàn cục hay phạm vi địa phương.

Biến toàn cục (*global variable*) là biến được khai báo bên ngoài tất cả các hàm (biến `totalApples` trong Hình 2.1). Biến toàn cục có thể được sử dụng trong toàn bộ chương trình, ngay cả bên trong các hàm, miễn là đoạn mã sử dụng biến nằm sau dòng lệnh khai báo biến đó.

Biến địa phương (*local variable*) là biến được khai báo bên trong một hàm hoặc một khối lệnh (trong C++, một khối lệnh được gói trong cặp ngoặc {}). Biến địa phương chỉ có thể sử dụng bên trong khối lệnh mà nó được khai báo tại

[5] Một số tài liệu dùng thuật ngữ "tham số thực sự" hay "tham đối" để chỉ đối số. Một số tài liệu khác dùng thuật ngữ "tham số" để gọi chung cho cả tham số hình thức lẫn đối số.

đó. Khối lệnh này có thể là thân của một hàm hoặc một khối cấu trúc điều khiển (xem Chương 3). Chẳng hạn, tại chương trình trong Hình 4.2 có ba biến `x` khác nhau với các phạm vi khác nhau: biến `x` được khai báo trong hàm `foo` có phạm vi là hàm `foo`; biến `x` được khai báo ngay tại dòng đầu tiên của thân hàm `main` có phạm vi là toàn bộ thân hàm `main` do cặp ngoặc {} gần nhất chính là cặp ngoặc bao quanh thân hàm `main`; và biến `x` được khai báo bên trong thân vòng `for` của hàm `main` chỉ có phạm vi bên trong vòng `for`. Đây là các biến hoàn toàn độc lập. Sửa đổi đối với biến này hoàn toàn không có ảnh hưởng gì đối với các biến kia (thể hiện ở các dòng output: con đếm `x` của vòng `for` thay đổi giá trị trong khi hai biến `x` còn lại không có gì thay đổi).

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

void foo( void );

int main()
{
    int x = 5;   // local variable to main

    foo();

    for (int i = 1; i <= 3; i++) {
        int x = i * i;    // local variable to for loop
        cout << "Local x in for loop is " << x << endl;
    }
    cout << "Local x in main is " << x << endl;
}

void foo()
{
    int x = 0;   // local variable to foo
    cout << "Local x in foo is " << x << endl;
}
```

Kết quả chạy chương trình:

```
Local x in foo is 0
Local x in for loop is 1
Local x in for loop is 4
Local x in for loop is 9
Local x in main is 5
```

Hình 4.2: Ví dụ về phạm vi của biến.

4.4.2. Thời gian sống của biến

Trong trường hợp mặc định với lệnh khai báo thông thường, một biến sẽ được tạo ra khi chương trình chạy đến lệnh khai báo biến đó và tồn tại cho đến khi chương trình ra khỏi phạm vi của biến đó. Cụ thể, các biến toàn cục có phạm vi là toàn bộ chương trình, nên chúng được sinh ra và khởi tạo đúng một lần và "sống" cho đến khi chương trình kết thúc.

Với các biến địa phương được khai báo theo kiểu thông thường, phạm vi của chúng kéo từ lệnh khai báo chúng cho đến hết khối chương trình chứa lệnh khai báo đó. Do đó, các biến này được tạo ra mỗi khi chương trình chạy đến lệnh khai báo biến và bị hủy đi khi chương trình chạy đến hết khối. Người ta còn gọi các biến địa phương này là **biến tự động** (*automatic variable*).

Trường hợp ngoại lệ trong C++ là các biến địa phương được khai báo với từ khóa `static`. Các biến này tuy là biến địa phương nhưng "sống" cho đến khi chương trình kết thúc mặc dù nó không thể được truy nhập tới trong khi chương trình đang chạy ở ngoài phạm vi của nó. Các biến static này giữ nguyên giá trị của mình giữa các lần gọi hàm. Người ta gọi các biến địa phương loại này là **biến tĩnh** (*static variable*).

Chương trình trong Hình 4.3 là một ví dụ minh họa sự khác nhau giữa biến tĩnh và biến thông thường. Trong hàm `staticVariableTest` có hai biến địa phương: `x` là biến tự động và `y` là biến tĩnh. Cả hai đều được khai báo với giá trị khởi tạo bằng 0, cùng được in ra màn hình trước khi tăng thêm 1. `staticVariableTest` được gọi bốn lần từ trong hàm `main`, mỗi lần lại in ra màn hình một dòng thông báo giá trị của `x` và `y`. Kết quả in ra màn hình cho thấy x có giá trị bằng 0 trong tất cả các lời gọi hàm, trong khi biến tĩnh y mỗi lần lại tăng thêm 1. Lần nào bước vào hàm `staticVariableTest`, `x` cũng nhận giá trị khởi tạo (0). Trong khi đó, `y` chỉ được khởi tạo bằng 0 đúng một lần, nó không bị hủy khi hàm kết thúc mà nó vẫn "sống" và giữ hiệu ứng của phép tăng ở cuối hàm `staticVariableTest` từ lần gọi hàm trước cho đến lần gọi hàm sau.

```
#include <iostream>

using namespace std;

void staticVariableTest()
{
    int x = 0;
    static int y = 0;
    cout << "x = " << x << ", y = " << y << endl;
    x++; y++;
}

int main()
{
    staticVariableTest();
    staticVariableTest();
    staticVariableTest();
    staticVariableTest();

    return 0;
}
```

Kết quả chạy chương trình:

```
x = 0, y = 0
x = 0, y = 1
x = 0, y = 2
x = 0, y = 3
```

Hình 4.3: Ví dụ về biến tĩnh.

4.5. Tham số, đối số, và cơ chế truyền tham số cho hàm

Trong phần này chúng ta sẽ tìm hiểu kĩ về các cơ chế truyền tham số cho hàm. Có hai cơ chế chính là **truyền giá trị** và **truyền tham chiếu** với điểm khác nhau căn bản: một bên chỉ cho phép hàm được gọi thao tác trên bản sao dữ liệu của nơi gọi hàm, bên kia cho phép hàm được gọi truy nhập trực tiếp.

4.5.1. Truyền giá trị

Trong cơ chế **truyền giá trị** hay **truyền bằng giá trị** (*pass-by-value*), tham số được truyền bằng giá trị, nghĩa là bản sao của đối số được chép vào tham số tương ứng trong hàm chứ bản thân đối số thì không được truyền vào hàm. Các thay đổi đối với bản sao sẽ không có ảnh hưởng gì đối với bản gốc, cho nên, khi hàm thực hiện các sửa đổi đối với một tham số thì các sửa đổi này không có ảnh

hưởng gì tới đối số tương ứng được dùng cho lời gọi hàm. Tất cả các ví dụ trong chương này cho đến đây đều dùng cơ chế truyền giá trị.

Xem thêm minh họa trong Hình 4.4. Việc tham số hình thức `limit` bị thay đổi giá trị từ bên trong hàm `sum` không có ảnh hưởng gì đối với đối số `upperLimit` (một biến của hàm `main`) được dùng khi gọi hàm này. Sau khi lời gọi hàm được thực hiện xong, `upperLimit` vẫn giữ nguyên giá trị là 9.

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

int sum(int limit)
{
    int total = 0;
    for ( ; limit > 0; limit--)
        total += limit;
    return total;
}

int main ()
{
    int upperLimit = 9;
    int total = sum (upperLimit);
    cout << "The sum of numbers from 1 to " << upperLimit
         << " is " << total << endl;
    return 0;
}
```

Kết quả chạy chương trình:

```
The sum of numbers from 1 to 9 is 45
```

Hình 4.4: Ví dụ về truyền giá trị cho hàm.

Truyền giá trị được xem là lựa chọn an toàn do nó đảm bảo rằng hàm được gọi sẽ không gây hiệu ứng phụ không mong muốn đối với dữ liệu của nơi gọi hàm. Sử dụng cơ chế truyền giá trị là một trong các khía cạnh của phong cách lập trình tốt. Tuy nhiên, cơ chế này có một nhược điểm: nếu dữ liệu được truyền vào hàm có kích thước lớn thì sẽ tốn phí về thời gian chạy và bộ nhớ.

4.5.2. Truyền tham chiếu

Khác với truyền giá trị, cơ chế **truyền tham chiếu** hay **truyền bằng tham chiếu** (*pass-by-reference*) cho phép hàm được gọi truy nhập trực tiếp tới dữ liệu của nơi gọi hàm và sửa dữ liệu đó nếu muốn. So với truyền giá trị, cơ chế truyền

tham chiếu giúp chương trình có hiệu năng cao hơn do tránh được chi phí cho việc sao chép dữ liệu. Tuy nhiên, truyền tham chiếu làm cho chương trình kém an toàn do nguy cơ hàm được gọi làm rối loạn dữ liệu của hàm gọi. Mục này nói về một cơ chế mà C++ cung cấp để truyền tham số bằng tham chiếu. Ngoài ra, Chương 6 sẽ nói đến một dạng truyền tham chiếu khác đặc thù của C/C++: con trỏ.

Trước tiên chúng ta tìm hiểu về khái niệm tham chiếu trong ngôn ngữ lập trình C++. **Tham chiếu** (*reference*) của một biến về bản chất là một biệt danh hay một cái tên khác của chính biến đó. Các thao tác trên tham chiếu của một biến cũng có hiệu quả giống hệt như thao tác trên chính biến đó và ngược lại. Sau khi khai báo, tham chiếu được sử dụng trong chương trình như các biến khác. Ví dụ dòng:

```
int &numberStick = numberUSB;
```

khai báo `numberStick` là một tham chiếu tới một biến kiểu `int` có tên `numberUSB`. Lưu ý rằng trong khai báo có dấu &. Ngoài ra, tham chiếu phải được khởi tạo ngay khi khai báo, C++ không chấp nhận tình trạng một tham chiếu không chiếu đi đâu cả.

Hình 4.5 minh họa cách khai báo và sử dụng tham chiếu. Biến `numberStick` được khai báo (có dấu & vào đằng trước tên biến) là một tham chiếu tới biến `numberUSB` (stick là một tên gọi khác của USB stick). Mọi tính toán và tác động trên biến `numberStick` sẽ tương tự như trên biến `numberUSB` và ngược lại vì thực chất hai biến này là một.

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

int main()
{
    int numberUSB = 10;
    int &numberStick = numberUSB;

    numberStick ++;
    cout << numberUSB << " " << numberStick;
    return 0;
}
```

Hình 4.5: Ví dụ về khai báo và sử dụng tham chiếu.

Trong nhiều trường hợp, chúng ta muốn việc thay đổi giá trị của một tham số trong hàm sẽ làm thay đổi giá trị của đối số tương ứng. Ví dụ, khi ta viết một hàm nhập giá trị cho các biến từ bàn phím.

Để làm như vậy, ta có thể sử dụng cơ chế **truyền tham chiếu**. Tức là, tham số hình thức sẽ là một tham chiếu tới đối số tương ứng. Trong cơ chế này, các sửa đổi đối với tham số hình thức ở bên trong hàm được thực hiện trên chính đối số dùng cho lời gọi hàm.

Hình 4.6 minh họa cơ chế truyền tham chiếu với hàm `getStudent`. Hàm này nhập hai số nguyên từ bàn phím và lưu chúng vào hai tham số `id` và `mark` (đã được khai báo là tham chiếu). Việc lưu giá trị vào `id` và `mark` sẽ làm thay đổi giá trị của đối số là các biến tương ứng của hàm `main`.

Khác biệt duy nhất về cú pháp giữa truyền giá trị và truyền tham chiếu là ở danh sách khai báo tham số hình thức. Để quy định tham số nào sẽ được truyền bằng tham chiếu, cần có dấu `&` đặt ngay sau tên kiểu dữ liệu tại khai báo tham số đó trong định nghĩa hàm. Trong ví dụ hàm `getStudent`, có các dấu `&` đặt tại khai báo của các tham số `id` và `mark`.

```
void getStudent (int &id, int &mark);
```

Lưu ý: Hàm `getStudent` không cần trả về giá trị gì nên nó đã được khai báo với từ khóa `void` cho kiểu giá trị trả về và không cần có lệnh `return` kèm giá trị trong thân hàm.

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

void getStudent (int &id, int &mark)
{
     cout << "Enter student’s id and mark: ";
     cin >> id >> mark;
     return;
}

int main()
{
    int id1, mark1;
    getStudent (id1, mark1);
    cout << "Student " << id1 << " gets mark " << mark1 << endl;
    int id2, mark2;
    getStudent (id2, mark2);
    cout << "Student " << id2 << " gets mark " << mark2 << endl;
    cout << "Total mark: " << (mark1 + mark2);
    return 0;
}
```

Kết quả chạy chương trình

```
Enter student's id and mark: 1 5
Student 1 gets mark 5
Enter student's id and mark: 2 6
Student 2 gets mark 6
Total mark: 11
```

Hình 4.6: Ví dụ về truyền tham biến cho hàm.

Như đã nói ở trên, việc cho phép hàm được gọi truy nhập trực tiếp tới dữ liệu của nơi gọi hàm gây nguy cơ hàm được gọi phá rối dữ liệu của nơi gọi hàm. Vậy có cách nào để giảm bớt hoặc ngăn chặn nguy cơ này? Đó là hằng tham chiếu. Một khi một tham chiếu được khai báo là hằng, trình biên dịch sẽ đảm bảo rằng tham chiếu đó sẽ chỉ được dùng để đọc chứ không thể sửa đổi biến mà nó chiếu tới.

4.5.3. Tham số mặc định

Khi khai báo hàm, ta có thể đặt sẵn một giá trị mặc định cho các tham số. Các tham số này phải đứng cuối danh sách. Để đặt giá trị mặc định, ta chỉ cần viết dấu gán và giá trị mặt định cho tham số tại khai báo hàm. Nếu tham số đó không được truyền giá trị khi gọi hàm, giá trị mặc định sẽ được dùng. Ngược

lại, nếu một giá trị được truyền vào tham số thì giá trị mặc định sẽ bị bỏ qua để dùng giá trị được truyền vào.

Hình 4.7 minh họa về cách khai báo và sử dụng tham số mặc định. Trong đó, tham số hình thức `upper` của hàm `sum` có giá trị mặc định là 9. Lần gọi hàm `sum` thứ nhất cung cấp đối số cho cả hai tham số (`lower` = 1 và `upper` = 5) và ta thu được kết quả là 15. Lần gọi hàm `sum` thứ hai chỉ cung cấp đối số có giá bằng 1 cho tham số `lower`, còn tham số `upper` sẽ nhận giá trị mặc định (9), ta thu được kết quả là 45.

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

int sum(int lower, int upper=9)
{
    int total = 0;
    for ( ; lower <= upper; lower ++)
        total += lower;
    return total;
}

int main ()
{
    int lowerLimit = 1;
    int upperLimit = 5;

    int total = sum (lowerLimit, upperLimit);
    cout << "Total is " << total << endl;

    total = sum (lowerLimit);
    cout << "Total is " << total << endl;
    return 0;
}
```

Kết quả chạy chương trình

```
Total is 15
Total is 45
```

Hình 4.7: Ví dụ về tham số mặc định.

Về nguyên tắc, trình biên dịch sẽ duyệt danh sách các đối số truyền vào từ trái qua phải, và lần lượt gắn các đối số cho các tham số hình thức tương ứng. Khi kết thúc danh sách đối số, những tham số hình thức chưa được truyền giá trị sẽ

nhận giá trị mặc định. Lưu ý: số đối số truyền vào phải nhiều hơn hoặc bằng số tham số hình thức không được gắn giá trị mặc định.

4.6. Hàm trùng tên

Trong C++, hai hàm khác nhau có thể có tên trùng nhau (*function overloading*) nếu danh sách tham số (chỉ quan tâm đến kiểu dữ liệu mà không quan tâm đến tên của tham số) của hai hàm là khác nhau.

Ví dụ,

```
void print (int x);
void print (float x);
void print (int x, int y);
```

là ba hàm khác nhau trong C++. Mỗi khi cần dịch một lời gọi hàm có tên `print`, trình biên dịch kiểm tra kiểu dữ liệu của đối số và số lượng các đối số trong lời gọi hàm đó để xác định xem hàm được gọi là hàm nào trong ba hàm `print` đã được định nghĩa.

Ví dụ trong Hình 4.8 minh họa việc khai báo và sử dụng các hàm trùng tên `print` liệt kê ở trên. Cả ba hàm được gọi từ trong hàm `main` với danh sách tham số khác nhau. Kết quả chạy chương trình chứng tỏ trình biên dịch đã gọi đúng hàm cho các bộ đối số khác nhau.

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

void print (int x)
{
    cout << "int: " << x << endl;
}

void print (double x)
{
    cout << "double: " << x << endl;
}

void print (int x, int y)
{
    cout << "pair: " << x << " " << y << endl;
}

int main ()
{
    print(1);
    print(10.5);
    print(1,2);
    return 0;
}
```

Kết quả chạy chương trình:

```
int: 1
double: 10.5
pair: 1 2
```

Hình 4.8: Ví dụ về hàm trùng tên.

Các hàm trùng tên được phân biệt bởi chữ kí của hàm. Chữ kí của một hàm bao gồm tên của hàm đó và danh sách kiểu tham số theo thứ tự khai báo.

Theo quy tắc đó, ba lệnh khai báo hàm dưới đây có chữ kí trùng nhau nên không được C++ chấp nhận làm các hàm trùng tên:

```
int   print (int x);
float print (int x);
int   print (int y);
```

Khi kết hợp sử dụng hàm trùng tên và tham số mặc định, cần lưu ý tránh trường hợp một hàm khi bỏ qua các tham số mặc định lại có lời gọi giống hệt với một

hàm trùng tên với nó. Ví dụ khi ta có hai hàm trùng tên, một hàm có danh sách tham số rỗng, hàm kia cho phép tất cả các tham số có thể lấy giá trị mặc định. Tình trạng này sẽ gây lỗi biên dịch do tình trạng nhập nhằng không thể xác định một lời gọi hàm ứng với hàm nào.

C++ còn cho phép sử dụng cơ chế hàm trùng tên để định nghĩa các phiên bản khác của các phép toán đã có sẵn để dùng cho các kiểu dữ liệu không cơ bản. Chẳng hạn lập trình viên có thể định nghĩa kiểu dữ liệu số phức và viết phép cộng (+) dành cho kiểu dữ liệu này. Nội dung chi tiết về chủ đề này nằm ngoài phạm vi của cuốn sách này, người đọc có thể tìm hiểu tại [2].

4.7. Hàm đệ quy

Trong các ví dụ từ đầu cuốn sách, ta đã làm quen với việc một hàm gọi một hàm khác. Đệ quy là dạng gọi hàm đặc biệt: hàm đệ quy là hàm chứa lời gọi tới chính nó, trực tiếp hoặc gián tiếp (qua một hàm khác).

Dạng hàm này được dùng để biểu diễn các thuật toán đệ quy – đưa một bài toán về một bài toán tương tự nhưng có kích thước nhỏ hơn. Đây là chủ đề phức tạp, sẽ được nói đến một cách đầy đủ hơn trong các môn học cao hơn của ngành Khoa học Máy tính. Phần này chỉ giới thiệu về đệ quy ở mức đơn giản nhất.

Ví dụ điển hình thường được dùng để minh họa cho hàm đệ quy là bài toán tính giai thừa của một số. Giai thừa của *n* được tính theo công thức đệ quy:

$$n! = n * (n-1) * (n-2) * (n-3) ... * 1$$

tương đương với:

$$n! = n * (n-1)!$$

Nghĩa là, giả sử f(n) là giai thừa bậc n, f(n) được định nghĩa như sau:

f(0) = 1
*f(n) = n * f(n-1)*

Trong đó, dòng thứ nhất là trường hợp cơ bản, dòng thứ hai là công thức đệ quy.

Hàm đệ quy viết bằng C++ về cơ bản là giống hệt với công thức đệ quy ở trên. Hình 4.9 mô tả hàm đệ quy để tính giá trị của *n*! Hàm đệ quy `factorial` bao gồm một khối lệnh `if-else` chia hai trường hợp cơ bản và tổng quát. Trường hợp cơ bản xảy ra nếu `n ≤ 1`, khi đó hàm kết thúc tính toán và trả về giá trị 1 là

giai thừa của n. Trường hợp còn lại (n > 1), hàm gọi đệ quy để tính giá trị của (n - 1)! rồi trả về kết quả là tích của n và (n - 1)!.

Trong hai phần chính của hàm đệ quy, trường hợp cơ bản nhìn qua có vẻ đơn giản nhưng thực ra lại có vai trò rất quan trọng trong hàm đệ quy. Nếu thiếu hoặc có nhầm lẫn khi viết trường hợp này, hoặc sơ suất trong bước đệ quy làm nó không thể hội tụ về trường hợp cơ bản thì chương trình có nguy cơ đệ quy vô tận đến khi cạn kiệt bộ nhớ dành cho chương trình. Hiện tượng này tương tự như vòng lặp vô tận.

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

unsigned long factorial (int n)
{
    if (n <= 1) // test for base case
        return 1;    // base case: n! = 1
    else   //recursion step
        return (n * factorial (n-1)); // calculate (n-1)! first
}

int main ()
{
    int n = 5;
    cout << n << "! = " << factorial (n);
    return 0;
}
```

Hình 4.9: Ví dụ về hàm đệ quy tính n!.

Một điểm cần nhỏ chú ý trong chương trình tính giai thừa là kiểu trả về cho hàm tính giai thừa. Giá trị của giai thừa tăng rất nhanh so với n: 10! đã đạt đến giá trị 3628800; nếu dùng kiểu int (có kích thước 4 byte ở hầu hết các bản cài đặt C++) thì giai thừa của 16 bắt đầu vượt ra ngoài khoảng giá trị của int (hãy thử bằng cách sửa chương trình ví dụ trong Hình 4.9). Do đó, để có thể đáp ứng giá trị lớn hơn của n, kiểu dữ liệu cho giá trị giai thừa phải có kích thước lớn. Tài liệu C++ chuẩn quy định kiểu long int có kích thước lớn nhất trong các kiểu nguyên và gồm ít nhất 4 byte. Kiểu unsigned long int (viết ngắn gọn là unsigned long) chỉ gồm các giá trị không âm nên có thể lưu giá trị từ 0 tới ít nhất 4294967295, ta nên chọn kiểu dữ liệu này cho kiểu trả về của hàm tính giai thừa.

Bài tập

1. Viết một hàm nhận tham số đầu vào là 5 số nguyên, và trả về kết quả là trung bình cộng của 5 số nguyên đó. Viết một hàm nhận tham số đầu vào là 5 số nguyên, và trả về kết quả là trung bình cộng của 5 số nguyên đó.
2. Hãy tìm những hàm trùng nhau trong danh sách các hàm sau
   - int foo (int x)
   - int foo (float x)
   - float foo (int y)
   - float foo (int x, int y)
   - float foo (int x, float y)
   - int foo (int student, float teacher)
3. Khi nào thì nên truyền đối số cho hàm theo kiểu truyền giá trị, khi nào thì nên truyền theo kiểm tham chiếu?
4. Viết một chương trình có một hàm với 5 tham số dạng tham chiếu để nhận vào 5 số nguyên. Trong 5 tham số đầu vào thì có 3 tham số với giá trị mặc định là 13, 25, 35. Tìm và liệt kê ra tất cả các cặp số có tổng bằng 50.
5. Viết chương trình với một hàm getMin tính giá trị nhỏ nhất của 3 số. Hàm trên dựa vào hàm getMin tính giá trị nhỏ nhất của hai số. Lưu ý, hai hàm này trùng tên.
6. Nhập từ bán phím một số nguyên x, hãy tính và hiện ra màn hình giá trị căn bậc hai của x và e^x
7. Tìm hiểu hàm *rand* để sinh ra các số ngẫu nhiên. Hãy viết chương trình sinh ra 9 số nguyên ngẫu nhiên nằm trong phạm vi từ 2 đến 99.
8. Dãy số Fibonacci được tính như sau:

   `f (0) = 0; f (1) = 1; f(n) = f (n-1) + f(n-2).`

   Hãy viết chương trình với hàm đệ quy để tính `f(k)`, với `k` là giá trị nhập từ bàn phím.
9. Tìm hiểu về ý nghĩa và cách sử dụng hàm inline trong C++.

Chương 5. Mảng và xâu kí tự

Các kiểu dữ liệu cơ bản như đã giới thiệu trong Chương 2 không đủ để biểu diễn các loại dữ liệu mà các bài toán đòi hỏi. Một ví dụ là khi chương trình cần lưu và xử lý một chuỗi các phần tử dữ liệu cùng kiểu, chẳng hạn như danh sách sinh viên trong trường hoặc danh sách điểm thi của một sinh viên, để có thể sắp xếp, tìm kiếm, và tính toán các con số thống kê trên chuỗi dữ liệu đó. Đa số các ngôn ngữ lập trình cung cấp các kiểu dữ liệu có cấu trúc để phục vụ các nhiệm vụ này, trong đó, mảng là cấu trúc dữ liệu thông dụng nhất.

5.1. Mảng một chiều

Mảng một chiều là một chuỗi hữu hạn các phần tử dữ liệu thuộc cùng một kiểu dữ liệu, đặt tại các ô nhớ liên tiếp trong bộ nhớ. Mỗi phần tử trong mảng có một chỉ số khác nhau. Mảng cho phép định vị và truy nhập đến từng phần tử bằng cách sử dụng chỉ số của phần tử đó.

Ví dụ, điểm số cho 7 môn thi của một sinh viên có thể được lưu trữ trong một mảng có kích thước bằng 7 (nghĩa là có 7 ô nhớ) thay vì khai báo 7 biến khác nhau cho điểm thi từng môn. Mảng `score` có thể được hình dung như sau:

`score`

30	85	76	90	72	80	88
0	1	2	3	4	5	6

trong đó, mỗi ô biểu diễn một phần tử của mảng – trong trường hợp này là một giá trị thuộc kiểu `int` – và được đánh số lần lượt từ 0 đến 6.

Cũng như một biến bình thường, mảng phải được khai báo trước khi sử dụng. Cú pháp khai báo một mảng trong C++ có dạng:

kiểu_dữ_liệu tên_mảng `[`*số_phần_tử*`];`

trong đó, *kiểu_dữ_liệu* là một kiểu dữ liệu hợp lệ (chẳng hạn `int`, `float`, `char`, `bool`…), *tên_mảng* là một định danh hợp lệ, và *số_phần_tử* (luôn đặt trong cặp ngoặc vuông) quy định số lượng phần tử mà mảng cần chứa.

Ví dụ, mảng `score` trong ví dụ ở trên được khai báo như sau:

```
int score [7];
```

Lưu ý: giá trị số phần tử đặt trong cặp ngoặc vuông phải là một hằng số, do các mảng khai báo kiểu này thuộc bộ nhớ tĩnh và phải có kích thước được xác định trước khi chương trình thực thi. Mảng với kích thước động sẽ được nói đến trong chương sau.

Kết quả của lệnh khai báo như trên là ta có 7 biến kiểu `int` với "tên" của chúng là `score[0]`, `score[1]`, `score[2]`, `score[3]`, `score[4]`, `score[5]`, và `score[6]`.

Hình 5.1 minh họa việc khai báo và sử dụng mảng một chiều. Lưu ý sự khác nhau về ngữ nghĩa của giá trị bên trong cặp ngoặc vuông: tại lệnh khai báo mảng thì nó là kích thước mảng, còn khi truy nhập phần tử của mảng thì nó là chỉ số của phần tử mảng.

5.1.1. Khởi tạo mảng

Khi khai báo một mảng, ta có thể khởi tạo giá trị cho các phần tử của mảng theo cách sau:

```
int score [7] = {60, 70, 89, 75, 88, 34, 90};
```

Nếu không được khởi tạo, các phần tử của mảng sẽ có giá trị không xác định cho đến khi ta gán cho chúng một giá trị nào đó.

5.1.2. Trách nhiệm kiểm soát tính hợp lệ của chỉ số mảng

Đối với mảng được khai báo với kích thước n, chỉ số của các phần tử trong mảng đó là các số nguyên từ 0 đến n–1. Ngoài ra, các giá trị khác đều không hợp lệ. Việc truy nhập mảng bằng các chỉ số không hợp lệ, chẳng hạn khi truy nhập đến `score[-1]` hay `score[n]`, có thể dẫn đến các thay đổi không mong muốn đối với dữ liệu ở vùng bộ nhớ bên ngoài mảng (có thể thuộc về các biến khác). Trong nhiều ngôn ngữ lập trình, việc này được kiểm soát tự động để tránh trường hợp truy nhập với chỉ số không hợp lệ. Tuy nhiên, trong C++ việc truy nhập đến các phần tử của mảng với chỉ số nhỏ hơn 0 hoặc lớn hơn n–1 không hề phạm lỗi cú pháp, việc truy nhập ra ngoài mảng không gây lỗi khi dịch nhưng có thể gây lỗi khi chạy, lập trình viên có trách nhiệm kiểm soát các giá trị chỉ số mảng để tránh trường hợp này.

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

const int NUMBER_COURSES = 7;

int main()
{
   int score[NUMBER_COURSES];
   float sum = 0;

   for (int course = 0; course < NUMBER_COURSES; count++)
   {
      cout << "Enter the score for course #" << course << ": ";
      cin >> score[course];
      sum = sum + score[course];
   }
   cout << "The average score is " << sum/NUMBER_COURSES;

   return 0;
}
```

Kết quả chạy chương trình

```
Enter the score for course #0: 30
Enter the score for course #1: 85
Enter the score for course #2: 76
Enter the score for course #3: 90
Enter the score for course #4: 72
Enter the score for course #5: 80
Enter the score for course #6: 88
The average score is 74.42857
```

Hình 5.1: Ví dụ về khai báo và sử dụng mảng.

5.1.3. Mảng làm tham số cho hàm

Có thể dùng mảng làm tham số cho hàm. Hình 5.2 là kết quả của việc sửa chương trình trong Hình 5.1, đưa hai nhiệm vụ nhập dữ liệu cho mảng và tính điểm trung bình vào hai hàm và truyền mảng `score` vào trong hai hàm đó.

Để ý rằng tuy các tham số mảng không được khai báo với kí tự "`&`" trong phần khai báo và định nghĩa hàm, nhưng thực chất chúng là các tham chiếu (thay vì giá trị). Nói cách khác, khi chương trình thực thi, các hàm không tạo các bản sao riêng của các mảng được truyền làm tham số (việc tạo bản sao này có thể có chi phí rất cao về bộ nhớ.)

Vậy làm thế nào khi ta cần quy định rằng một hàm không được sửa đổi một tham số mảng nào đó? (ví dụ không nên cho hàm `average` quyền sửa mảng `score`.) Giải pháp mà C++ cung cấp là từ khóa `const`. Ta sửa phần khai báo tham số của hàm `average` từ

```
float average(int score[], int size);
```

thành

```
float average(const int score[], int size);
```

Kết quả là trình biên dịch sẽ không chấp nhận các dòng lệnh sửa giá trị của tham số mảng `score` ở bên trong hàm `average`. Nên sử dụng từ khóa `const` cho tất cả các tham số mà hàm không có nhu cầu thay đổi.

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

const int NUMBER_COURSES = 7;

void loadScore(int score[], int size)
{
   for (int course = 0 ; course < size ; course++)
   {
      cout << "Enter the score for course #" << course << ": ";
      cin >> score[course];
   }
}

float average(int score[], int size)
{
   float sum = 0;
   for (int course = 0 ; course < size ; course++)
      sum = sum + score[course];
   return sum/size;
}

int main()
{
   int score[NUMBER_COURSES];

   loadScore(score, NUMBER_COURSES);

   float averageScore = average(score, NUMBER_COURSES);
   cout << "The average score is " << averageScore;

   return 0;
}
```

Hình 5.2: Ví dụ về mảng làm tham số của hàm.

5.2. Mảng nhiều chiều

Cấu trúc mảng có thể có nhiều hơn một chiều. Ví dụ, để lưu một bàn cờ ca-rô, ta có thể dùng một mảng hai chiều chứa các kí tự, kí tự '.' biểu diễn một ô trống trên bàn cờ, các kí tự 'x' và 'o' lần lượt biểu diễn các kí hiệu trong trò chơi cờ ca rô. Khai báo mảng hai chiều như sau:

```
const int BOARD_HEIGHT = 22;
const int BOARD_WIDTH = 26;
char board[BOARD_HEIGHT][BOARD_WIDTH];
```

Ta dùng hai chỉ số hàng và cột để truy nhập từng phần tử trong mảng, ví dụ:

```
board[10][2] = 'x';
```

Cũng như mảng một chiều, mảng nhiều chiều cũng có thể dùng làm tham số cho hàm.

Lưu ý, khi khai báo mảng nhiều chiều như là tham số của hàm, chúng ta phải khai báo kích thước tất cả các chiều của mảng, ngoại trừ chiều đầu tiên có thể bỏ trống.

```
void clearBoard(char board[][SCREEN_WIDTH]);
```

là một khai báo hợp lệ, còn

```
void clearBoard(char board[][]);
```

là một khai báo không hợp lệ.

5.3. Xâu kí tự

Chúng ta đã dùng đến các xâu kí tự, chẳng hạn "`The average score is`" trong. Các ngôn ngữ lập trình đều có cấu trúc **xâu kí tự** (*string*), trong đó quy định cách tham chiếu tới từng phần tử trong xâu. Kèm theo đó là một bộ các hàm và thủ tục để thực hiện các phép toán trên dữ liệu xâu, chẳng hạn như xác định độ dài xâu, so sánh nội dung xâu, ghép xâu.

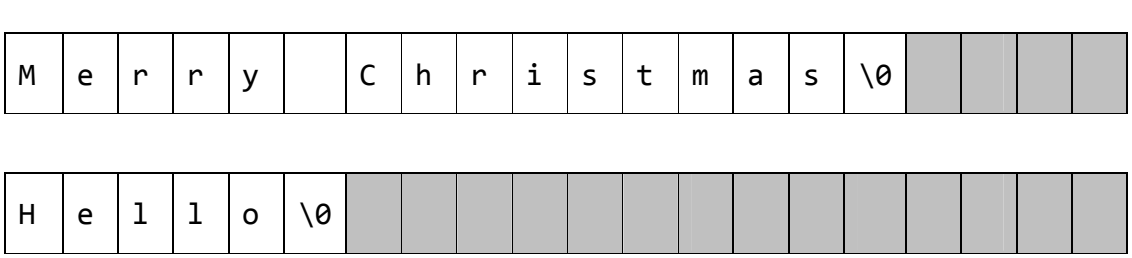
Với hầu hết các ngôn ngữ lập trình, trong đó có C++, xâu kí tự được đại diện bằng một mảng một chiều gồm các phần tử kiểu char. Ví dụ, mảng char sau đây gồm 20 phần tử kiểu char.

```
char greeting [20];
```

Về lý thuyết, mảng trên có thể chứa một chuỗi các ký tự với độ dài không quá 20. Nhưng ta còn có thể lưu tại đó chuỗi kí tự có độ dài ngắn hơn, chẳng hạn "`Merry Christmas`" hay "`Hello`".

Do mảng char có thể lưu xâu kí tự có độ dài nhỏ hơn kích thước của mảng, ta cần có cơ chế xác định điểm cuối hay độ dài của xâu. Ở một số ngôn ngữ lập trình như Pascal, cơ chế đó là độ dài xâu được ghi vào phần tử đầu tiên trong mảng (có chỉ số bằng 0). C++ lại sử dụng kí tự `null` ('`\0`') để đánh dấu kết thúc

của xâu kí tự. Ví dụ, mảng `greeting` có thể dùng để lưu xâu "`Merry Christmas`" hay "`Hello`" như trong Hình 5.3.



Hình 5.3: Xâu ký tự lưu trong mảng.

Lưu ý: các kí tự '`\0`' nằm ở cuối xâu để đánh dấu kết thúc xâu; còn các ô màu xám kí hiệu các phần tử mảng nằm ngoài xâu và có giá trị không xác định. Khi khai báo mảng để lưu trữ một xâu kí tự, ta cần nhớ rằng phải khai báo kích thước mảng đủ lớn để chứa cả kí tự null nằm cuối xâu.

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

const int MAX_NAME_LENGTH = 100;

int main()
{
    char name[MAX_NAME_LENGTH];
    cout << "What is your first name? ";
    cin >> name;
    cout << "Hi " << name << "!";
    return 0;
}
```

Kết quả chạy chương trình

```
What is your first name? Ellen
Hi Ellen!
```

Hình 5.4: Ví dụ về xâu kí tự.

Hình 5.4 minh họa một chương trình ví dụ nhập một xâu kí tự và ghi ra màn hình. Ta có thể thấy rằng mảng kí tự (hay xâu) `name` được sử dụng như là một biến bình thường đối với các lệnh vào ra dữ liệu.

5.3.1. Khởi tạo giá trị cho xâu kí tự

Có hai cách khởi tạo giá trị cho xâu kí tự ngay tại lệnh khai báo:

1. Khởi tạo xâu theo cách khởi tạo mảng:

```
char greeting[20] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

2. Dùng hằng giá trị để khởi tạo xâu:

```
char greeting[20] = "Hello";
```

Đối với cách thứ nhất, ta phải tự điền kí tự null ở cuối xâu. Còn cách thứ hai, các biểu diễn giá trị xâu có dạng dùng cặp dấu nháy kép được tự động kèm thêm kí tự null.

Lưu ý rằng chúng ta đang nói về công đoạn khởi tạo giá trị cho xâu chứ không nói về lệnh gán giá trị cho cả xâu. Cũng như các loại mảng nói chung, đối với mảng char (hay xâu ký tự), không được phép dùng một lệnh để gán trị hàng loạt cho các phần tử trong mảng. Các lệnh sau sẽ gây lỗi khi dịch:

```
greeting = {'H', 'e', 'l', 'l', 'o', '\0'};
greeting = "Merry Christmas";
```

5.3.2. Thư viện xử lý xâu kí tự

Thư viện `cstring` của thư viện C++ cung cấp một loạt các hàm tiện ích cho việc xử lý xâu kí tự, chẳng hạn như `strlen` trả về độ dài xâu, `strcpy` sao chép xâu, và `strcmp` so sánh xâu.

5.4. Tìm kiếm và sắp xếp dữ liệu trong mảng

Tìm kiếm dữ liệu trong mảng là xác định xem một giá trị nào đó (được gọi là **khóa**) có mặt trong mảng hay không, nếu có thì nó nằm ở vị trí nào. Sắp xếp mảng là sắp đặt lại vị trí của các phần tử mảng theo một trong hai thứ tự: giá trị tăng dần hoặc giảm dần. Ví dụ, một danh sách từ có thể được sắp xếp theo thứ tự từ điển, danh sách sinh viên được sắp xếp theo số sinh viên. Mục này giới thiệu một số thuật toán cơ bản cho việc tìm kiếm và sắp xếp dữ liệu trong mảng.

5.4.1. Tìm kiếm tuyến tính

Thuật toán tìm kiếm tuyến tính (*linear search*) duyệt tuần tự từng phần tử trong một mảng, so sánh khóa với mỗi phần tử. Khi gặp một phần tử khớp với khóa, thuật toán kết thúc và trả về chỉ số của phần tử đó. Nếu duyệt đến hết mảng mà vẫn không tìm thấy phần tử nào khớp với khóa, thuật toán kết luận là khóa tìm kiếm không có trong mảng. Ví dụ, với mảng {3, 18, 2, 3, 10, 1}, nếu ta thực hiện tìm kiếm với khóa bằng 10, thuật toán tìm kiếm tuyến tính sẽ duyệt lần lượt từ phần tử đầu tiên (3) cho tới khi gặp phần tử thứ năm (10) và trả về

kết quả là chỉ số mảng của phần tử đó (với mảng C++ đánh số từ 0, kết quả của thuật toán trong trường hợp này sẽ là 4).

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

int linearSearch(int a[], int size, int key)
{
    for (int i = 0 ; i <= size ; i++)
        if (a[i] == key) return i;
    return -1;
}

int main()
{
    int array[100] = {10, 3, 4, 7, 2, 15};
    int arraySize = 6;
    int searchKey;

    cout << "Enter the search key: ";
    cin >> searchKey;

    int pos = linearSearch(array, arraySize, searchKey);
    if (pos >= 0)
        cout << searchKey << " is found at entry " << pos;
    else cout << searchKey << " is not found";

    return 0;
}
```

Kết quả chạy chương trình:

```
Enter the search key: 3
3 is found at entry 1
Enter the search key: 45
45 is not found
```

Hình 5.5: Ví dụ về tìm kiếm tuyến tính.

Chương trình trong Hình 5.5 minh họa thuật toán tìm kiếm tuyến tính trong mảng. Trong đó, thuật toán tìm kiếm được cài trong hàm `linearSearch` trả về chỉ số của phần tử mảng khớp với khóa tìm kiếm hoặc trả về `-1` nếu không tìm thấy.

5.4.2. Tìm kiếm nhị phân

Thuật toán tìm kiếm tuyến tính được giới thiệu ở mục trước tuy đơn giản nhưng có độ phức tạp O(n) nên cho hiệu quả không cao đối với dữ liệu lớn. **Tìm kiếm nhị phân** (*binary search*) là một thuật toán tìm kiếm phức tạp hơn nhưng cho hiệu quả cao hơn. Ngoài ra, nó còn đòi hỏi mảng đầu vào đã được sắp xếp.

Giả sử mảng đã được sắp xếp tăng dần, thuật toán tìm kiếm nhị phân hoạt động như sau: Ở lần lặp đầu tiên, lấy phần tử nằm giữa mảng. Nếu phần tử đó khớp với khóa thì thuật toán kết thúc. Nếu nó có giá trị lớn hơn khóa thì chắc chắn nửa sau của mảng chứa toàn các phần tử lớn hơn khóa, và do đó không phải cái cần tìm. Nghĩa là giới hạn tìm kiếm giờ chỉ còn là nửa đầu của mảng. Còn nếu phần tử ở giữa có giá trị nhỏ hơn khóa, với lập luận tương tự như trên, ta có giới hạn cần tìm kiếm thu hẹp lại chỉ còn là nửa sau của mảng. Như vậy, tại mỗi lần lặp, thuật toán so sánh khóa với phần tử nằm giữa phần mảng cần tìm kiếm để hoặc là thấy phần tử đó khớp với khóa hoặc là loại bỏ một nửa mảng ra khỏi phạm vi cần tìm kiếm. Thuật toán kết thúc khi tìm thấy một phần tử có giá trị bằng khóa hoặc khi phạm vi cần tìm kiếm bị giảm xuống thành một mảng con có kích thước bằng 0.

Ví dụ, cho mảng {1, 3, 4, 7, 10, 12, 15}. Để tìm phần tử có giá trị 10, đầu tiên thuật toán lấy phần tử nằm giữa mảng là 7 và so sánh với 10. Vì 7 nhỏ hơn 10, thuật toán bỏ qua nửa đầu của mảng, phạm vi tìm kiếm thu hẹp thành đoạn mảng {10, 12, 15}. Tại lần lặp tiếp theo, thuật toán lại lấy phần tử nằm giữa là 12 và so sánh với khóa 10. Vì 12 lớn hơn 10 nên nửa sau của đoạn {10, 12, 15} bị bỏ qua, phạm vi tìm kiếm được thu hẹp lại chỉ còn đoạn mảng gồm một phần tử {10}. Lần lặp thứ ba, phần tử nằm giữa mảng (phần tử duy nhất còn lại trong đoạn cần tìm) có giá trị trùng với khóa nên thuật toán kết luận đã tìm thấy phần tử có giá trị 10.

Thuật toán được minh họa trong chương trình Hình 5.6.

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

int binarySearch(int a[], int size, int key)
{
    int start = 0, end = size;
    while (end > start)
    {
        cout << end << " " << start << endl;
        int middle = (end + start) / 2;
        if (a[middle] == key) return middle;
        else if (a[middle] > key) end = middle;
        else start = middle + 1;
    }
    return -1;
}

int main()
{
    int array[100] = {1, 3, 4, 7, 10, 12, 15};
    int arraySize = 7;
    int searchKey;

    cout << "Enter the search key: ";
    cin >> searchKey;

    int pos = binarySearch(array, arraySize, searchKey);
    if (pos >= 0)
        cout << searchKey << " is found at entry " <<  pos;
    else cout << searchKey << " is not found";

    return 0;
}
```

Hình 5.6: Ví dụ về tìm kiếm nhị phân.

5.4.3. Sắp xếp chọn

Sắp xếp chọn (*selection sort*) là một trong những thuật toán sắp xếp đơn giản nhất, nhưng lại có hiệu quả không cao đối với dữ liệu có kích thước lớn (mảng có số phần tử lớn). Hoạt động của thuật toán này rất đơn giản. Ở lần lặp thứ nhất, ta tìm phần tử nhỏ nhất của toàn mảng rồi tráo vị trí của nó với phần tử đầu tiên (giả thiết rằng ta đang cần sắp xếp mảng theo thứ tự tăng dần). Lần lặp thứ 2, ta tìm phần tử nhỏ thứ nhì trong toàn mảng, nghĩa là phần tử nhỏ nhất của đoạn mảng tính từ phần tử thứ hai trở đi, rồi tráo vị trí của nó với phần tử thứ

hai trong mảng. Thuật toán tiếp tục cho đến lần lặp cuối cùng – khi phần tử lớn nhì mảng được chọn ra và tráo với phần tử có chỉ số sát cuối, dẫn đến việc phần tử lớn nhất mảng nằm tại vị trí cuối mảng. Tổng quát, kết quả của lần lặp thứ *i* là phần tử nhỏ thứ *i* của mảng được xếp vào vị trí thứ *i* trong mảng.

Hình 5.7 minh họa thuật toán sắp xếp chọn.

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

void selectionSort(int a[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        // find the smallest entry in a[i],...a[size-1]
        int smallest = i;
        for (int j = i + 1; j < size; j++)
            if (a[j] < a[smallest]) smallest = j;

        // swap a[i] with that smallest entry
        int temp = a[i];
        a[i] = a[smallest];
        a[smallest] = temp;
    }
}

int main()
{
    int array[100], arraySize;

    cout << "Enter array size: ";
    cin >> arraySize;
    cout << "Enter the array: ";
    for (int i = 0; i < arraySize; i++) cin >> array[i];

    selectionSort(array, arraySize);

    cout << "Sorted array: \n";
    for (int i = 0; i < arraySize; i++)
        cout << array[i] << " ";

     return 0;
}
```

Hình 5.7: Sắp xếp chọn.

Bài tập

1. Nhập vào một danh sách các số nguyên từ bàn phím. Hãy thực hiện các nhiệm vụ sau và ghi kết quả ra màn hình
   - Tìm số lớn nhất trong danh sách
   - Tìm số nhỏ nhất trong danh sách
   - Tính trung bình cộng các số trong danh sách
   - Sắp xếp danh sách theo thứ tự tăng dần
2. Nhập vào từ bàn phím danh sách tên các bạn sinh viên trong lớp. Hãy tìm hiểu và cài đặt thuật toán sắp xếp chọn (selection sort) để sắp xếp danh sách tên theo thứ tự tăng dần. Hiện danh sách sau khi đã sắp xếp ra màn hình.
3. Nhập vào từ bàn phím một danh sách các số nguyên. Hãy liệt kê ra tất cả các bộ số (a, b, c) mà tổng của chúng bằng 25.
4. Một quả đồi được chia thành một lưới ô vuông có kích thước m*n ô vuông. Mỗi ô vuông chứa một số nguyên đại diện cho độ cao tại ô vuông đó. Hãy tìm:
   - Ô vuông có độ cao lớn nhất
   - Ô vuông có độ cao nhỏ nhất
   - Tất cả các hình vuông có kích thước 3*3, mà ô vuông ở giữa cao hơn các ô vuông xung quanh
   - Tất cả các hình vuông có kích thước 3*3, mà ô vuông ở giữa thấp hơn các ô vuông xung quanh
5. Nhập vào từ bàn phím một xâu kí tự. Sử dụng kiểu dữ liệu `char[]` trong C++ để ghi ra màn hình các kết quả sau:
   - Độ dài của xâu kí tự vừa nhập
   - Xâu kí tự sau khi đã loại bỏ tất cả các dấu '?', '!'
   - Xâu kí tự sau khi chèn vào giữa xâu từ "C++".
   - Xâu kí tự sau khi đã xóa bỏ 5 kí tự nằm ở giữa

6. Nhập một danh sách tên các bạn sinh viên trong lớp, hãy đếm xem có bao nhiêu bạn có tên với chữ cái bắt đầu là ‘T’, ‘C’, ‘V’, ‘A’, ‘Q’. Tìm tên có số bạn trùng nhau là nhiều nhất.

7. Nhập một xâu ký tự từ bàn phím là danh sách tên các sinh viên. Hai tên đứng liền nhau được cách nhau bởi ít nhất một dấu cách. Hãy tính xem có bao nhiêu sinh viên trong danh sách. Ví dụ:

Dữ liệu vào	Kết quả tương ứng
Vinh tuan vinh blah blah	5

8. Nhập một xâu ký tự từ bàn phím. Hãy chuẩn hoá xâu bằng cách loại bỏ các ký tự không phải chữ cái, dấu cách, dấu phảy, dấu chấm. Các từ cách nhau bởi đúng một dấu cách và chỉ viết hoa chữ cái đầu tiên. Ví dụ:

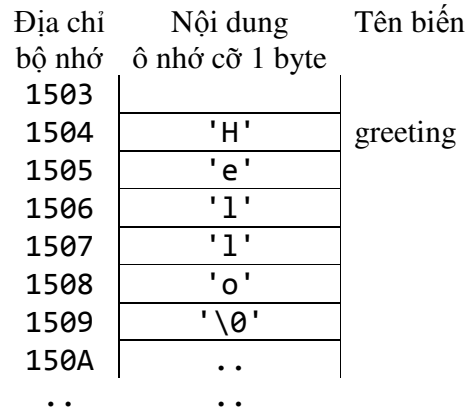
Dữ liệu vào	Kết quả tương ứng
tOi te1N !LA ng2uyen v3an aN;.	Toi Ten La Nguyen Van An.

Chương 6. Con trỏ và bộ nhớ

Con trỏ là một công cụ mà một số ngôn ngữ lập trình bậc cao, đặc biệt là C và C++, cung cấp để cho phép chương trình quản lý và tương tác trực tiếp với bộ nhớ. Nó giúp chương trình linh động và hiệu quả. Tuy nhiên, sử dụng con trỏ cũng dễ dẫn đến sai sót và khó phát hiện ra lỗi.

6.1. Bộ nhớ máy tính

Để hiểu về con trỏ, trước tiên ta sẽ tìm hiểu về bộ nhớ máy tính. Ta có thể hình dung bộ nhớ máy tính là một loạt các ô nhớ nối tiếp nhau, mỗi ô nhớ có kích thước nhỏ nhất là một byte. Các ô nhớ đơn này được đánh số liên tục, ô nhớ sau có số thứ tự hơn số thứ tự của ô nhớ liền trước là 1. Tức là ô nhớ 1505 sẽ đứng sau ô nhớ 1504 và đứng trước ô nhớ 1506 (xem Hình 6.1).



Hình 6.1: Biến* `greeting`*– xâu kí tự "Hello"– trong bộ nhớ.

6.2. Biến và địa chỉ của biến

Biến trong một chương trình là tên của một vùng bộ nhớ được dùng để lưu dữ liệu. Việc truy nhập dữ liệu tại các ô nhớ đó được thực hiện thông qua tên biến. Tức là, ta không phải quan tâm đến vị trí vật lý của ô bộ nhớ.

Khi một biến được khai báo, phần bộ nhớ cần thiết sẽ được cấp phát cho nó tại một vị trí cụ thể trong bộ nhớ, vị trí đó được gọi là địa chỉ bộ nhớ của biến đó. Thông thường, hệ điều hành sẽ quyết định vị trí của biến trong bộ nhớ khi chương trình chạy.

Để lấy địa chỉ của một biến, trong C++ ta sử dụng toán tử tham chiếu địa chỉ (&). Khi đặt toán tử tham chiếu trước tên của một biến, ta có một biểu thức có giá trị là địa chỉ ca biến đó trong bộ nhớ. Ví dụ:

```
short apples = 9; //khai báo biến apples có giá trị 9
cout << apples;   //hiện ra số 9 là giá trị của apples
cout << &apples;  //hiện ra địa chỉ của biến apples
```

Trong thực tế, trước khi chương trình chạy, chúng ta không thể biết được địa chỉ của biến `apples`.

6.3. Biến con trỏ

Biến **con trỏ** là một loại biến đặc biệt, được dùng để lưu giữ địa chỉ ô nhớ. Tức là, giá trị của chúng là địa chỉ của ô nhớ trong bộ nhớ. Hay nói một cách hình tượng, biến con trỏ chỉ đến một ô nhớ trong bộ nhớ. Biến con trỏ được khai báo như sau:

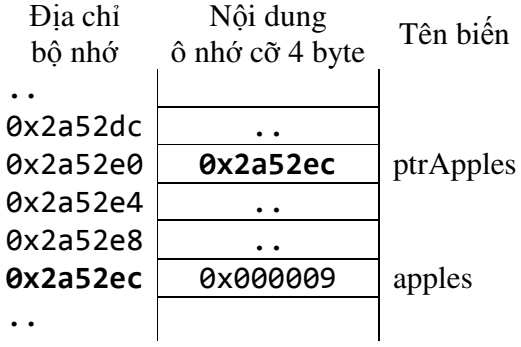
*data_type *ptr;*

trong đó *data_type* là kiểu của dữ liệu được lưu ở ô nhớ mà con trỏ *ptr* sẽ chỉ đến. Để truy nhập dữ liệu tại ô nhớ mà biến con trỏ chỉ đến ta sử dụng toán tử *.

Hình 6.3 minh họa về việc khai báo và sử dụng biến con trỏ. Trong ví dụ, ta khai báo một biến con trỏ `ptrApples` dùng để lưu giữ địa chỉ ô nhớ của biến `apples`. Lệnh

```
ptrApples = &apples;
```

sẽ gán địa chỉ của biến `apples` cho biến con trỏ `ptrApples` và tạo hiệu ứng tương tự như trong Hình 6.2



***Hình 6.2: Biến con trỏ* `ptrApples` *có giá trị là địa chỉ của biến* `apples`.**

Lưu ý, các tác động trên ô nhớ được chỉ đến bởi biến con trỏ `ptrApples` cũng chính là các tác động được thực hiện trên biến apples và ngược lại. Cụ thể là lệnh

```
apples++;
```

sẽ tăng giá trị của biến `apples` lên một, đồng nghĩa với việc tăng giá trị ở ô nhớ do biến con trỏ `ptrApples` chỉ đến lên 1.

Tương tự, lệnh

```
*ptrApples += 1;
```

sẽ tăng giá trị tại ô nhớ mà biến con trỏ `ptrApples` thêm 1, đồng nghĩa với việc tăng giá trị của biến `apples` thêm 1.

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int apples = 9;
    cout << "apples: " << apples << endl;
    cout << "Address of apples: " << &apples << endl;

    int *ptrApples;
    ptrApples = &apples;

    cout << "ptrApples: " << ptrApples << endl;
    cout << "Value at ptrApples: " << *ptrApples << endl;

    apples ++;
    cout << "apples: " << apples << endl;
    cout << "Value at ptrApples: " << *ptrApples << endl;

    *ptrApples += 1;
    cout << "apples: " << apples << endl;
    cout << "Value at ptrApples: " << *ptrApples << endl;

    return 0;
}
```

Kết quả chạy chương trình

```
apples: 9
Address of apples: 0x2a52ec
ptrApples: 0x2a52ec
Value at ptrApples: 9
apples: 10
Value at ptrApples: 10
apples: 11
Value at ptrApples: 11
```

Hình 6.3: Khai báo và sử dụng biến con trỏ.

6.4. Mảng và con trỏ

Mảng và con trỏ có liên hệ chặt chẽ với nhau. Giả sử, khi khai báo một mảng

```
int score[7];
```

ta đã khai báo một vùng nhớ liên tiếp gồm 7 ô, mỗi ô chứa một số nguyên.

Trong một số ngôn ngữ lập trình bậc cao, cụ thể là C++, mảng được cài đặt bằng con trỏ. Cụ thể là, biến mảng (`score`) có thể được coi là một biến con trỏ trỏ tới ô nhớ đầu tiên trong mảng.

Để truy cập đến phần tử có chỉ số `i` trong mảng `score`, ta có thể sử dụng `score[i]` hoặc `*(score + i)`. Ví dụ, `*score` và `score[0]` đều có ý nghĩa là phần tử đầu tiên trong mảng. Cách dùng con trỏ để thao tác với mảng được minh họa trong Hình 6.4.

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int score[7] = {1,2,3,4,5,6,7};
    int sum = 0;

    for (int count = 0; count < 7; count ++)
        sum += *(score+count);

    return 0;
}
```

Hình 6.4: Sử dụng mảng như con trỏ.

6.5. Bộ nhớ động

Trong tất cả các chương trình ví dụ trước, ta mới chỉ dùng bộ nhớ trong phạm vi các biến được khai báo trong mã nguồn với kích thước được xác định trước khi chương trình chạy. Với C++, tất cả các biến được khai báo trong chương trình đều nằm trong bộ nhớ tĩnh và có kích thước xác định trước. Nếu chúng ta cần đến lượng bộ nhớ mà kích thước chỉ có thể biết được khi chương trình chạy thì sao? Chẳng hạn khi chương trình của ta cần làm việc với một mảng mà kích thước của nó không xác định được khi chương trình chưa chạy, và ta cũng không thể ước lượng độ dài tối đa mà cần đọc dữ liệu vào để xác định độ dài cần thiết.

Giải pháp là sử dụng bộ nhớ động. Không như các biến thông thường chỉ cần khai báo, các biến nằm trong vùng bộ nhớ động cần được cấp phát khi muốn sử dụng và giải phóng một cách tường minh khi không còn được dùng đến.

Trong C++, con trỏ cùng với các toán tử `new` và `delete` là công cụ để thao tác với dữ liệu nằm trong bộ nhớ động.

6.5.1. Cấp phát bộ nhớ động

Khi một biến con trỏ được khai báo, ví dụ

```
int *ptrApples;
```

giá trị của biến con trỏ `ptrApples` chưa được xác định, nghĩa là nó chưa trỏ vào một vùng nhớ xác định trước nào. Chúng ta cũng có thể xin cấp phát một vùng nhớ động, và vùng nhớ này được quản lý (trỏ tới) bởi biến con trỏ `ptrApples` bằng cách sử dụng toán tử cấp phát bộ nhớ động `new` như sau:

```
ptrApples = new int;
```

Lênh này sẽ cấp phát một vùng nhớ để chứa một số nguyên, địa chỉ vùng nhớ này được lưu giữ bởi biến `ptrApples`. Quá trình cấp phát bộ nhớ này được thực hiện trong quá trình chạy chương trình và được gọi là **cấp phát bộ nhớ động**.

Sau khi xin cấp phát, ta có thể tiến hành lưu giữ, cập nhật giá trị ở vùng nhớ này thông qua địa chỉ của nó (hiện lưu tại biến `ptrApples`).

6.5.2. Giải phóng bộ nhớ động

Do lượng bộ nhớ động là có hạn, dữ liệu đặt trong bộ nhớ động nên được giải phóng khi dùng xong để dành bộ nhớ cho các yêu cầu cấp phát tiếp theo.

Khi chúng ta không sử dụng đến vùng nhớ này nữa, chúng ta nên giải phóng vùng nhớ này bằng toán tử `delete`, ví dụ như sau.

```
delete ptrApples;
```

Vùng nhớ mà biến con trỏ ptrApples trỏ tới sẽ được giải phóng và có thể được tái sử dụng cho một lần cấp phát bộ nhớ động khác.

Hình 6.5 minh họa về việc cấp phát, sử dụng, và giải phóng bộ nhớ động.

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int *ptrApples;
    ptrApples = new int;

    *ptrApples = 5;
    cout << "value at ptrApples: " << *ptrApples << endl;
    *ptrApples += 2;
    cout << "value at ptrApples: " << *ptrApples << endl;

    delete ptrApples;

    return 0;
}
```

Hình 6.5: Cấp phát, sử dụng và giải phóng bộ nhớ động.

Lưu ý: nếu vì sơ xuất của người lập trình hay vì lý do nào đó mà tại một thời điểm nào đó chương trình không còn lưu hoặc có cách nào tính ra địa chỉ của một vùng nhớ được cấp phát động, thì vùng bộ nhớ này được xem là bị "mất". Nghĩa là chương trình không có cách gì truy nhập hay giải phóng vùng bộ nhớ này nữa, nó sẽ được hệ điều hành thu hồi sau khi chương trình kết thúc. Ví dụ, đoạn chương trình sau làm thất thoát phần bộ nhớ được cấp phát do lệnh `new` thứ nhất:

```
ptr1 = new int;
ptr2 = new int;
ptr1 = ptr2;
```

6.6. Mảng động và con trỏ

Trong các chương trình ví dụ từ trước đến giờ, kích thước của mảng phải được xác định trước khi khai báo. Tức là, kích thước của mảng là một hằng số cho trước. Trong nhiều trường hợp, chúng ta khó có thể xác định trước được kích thước của mảng ngay từ khi lập trình, mà kích thước này phụ thuộc vào dữ liệu đầu vào cũng như quá trình chạy chương trình.

Để khắc phục tình trạng trên, C++ cho phép chúng ta sử dụng con trỏ để tạo mảng có kích thước động nằm trong bộ nhớ động. Những mảng được tạo theo

cách đó được gọi là mảng động, bởi kích thước mảng và việc cấp phát bộ nhớ cho mảng được xác định và tiến hành trong quá trình chạy chương trình. Việc khai báo mảng động được tiến hành thành hai bước:

1. **Khai báo con trỏ mảng:** Trước tiên, khai báo một biến con trỏ:

data_type `*`*array_name*`;`

trong đó *data_type* là kiểu dữ liệu của mảng, *array_name* là tên mảng.

2. **Xin cấp phát bộ nhớ:** Sau khi khai báo, chúng ta xin cấp phát bộ nhớ cho biến con trỏ bằng toán tử `new []` theo cấu trúc sau:

array_name `= new` *data_type* `[`*size*`];`

hệ thống sẽ cấp cho chương trình một vùng nhớ liên tiếp có thể chứa được *size* phần tử có kiểu *data_type*. Địa chỉ của phần tử đầu tiên được chỉ đến bởi biến *array_name*. Lưu ý, *size* có thể là một hằng số, hoặc là một biến.

Sau khi đã cấp phát bộ nhớ thành công, ta có thể đối xử với biến con trỏ *array_name* như một mảng thông thường với *size* phần tử. Tức là, để truy cập đến phần tử thứ *i* trong mảng, ta có thể sử dụng *array_name*`[`*i*`]` hoặc `*(`*array_name* `+` *i*`)`.

Giải phóng bộ nhớ: Khi chúng ta không sử dụng đến mảng động nữa, chúng ta phải tiến hành giải phóng bộ nhớ đã xin cấp phát bằng cách sử dụng toán tử `delete []` như sau:

`delete []` *array_name*`;`

Lưu ý khi giải phóng bộ nhớ động cho một mảng, nếu ta sử dụng lệnh

`delete` *array_name*`;`

thì chỉ có ô nhớ *array_name*`[0]` được giải phóng, còn các ô nhớ tiếp theo của mảng không được giải phóng.

Hình 6.6 minh họa việc sử dụng mảng động để tính tổng điểm của một danh sách sinh viên nhập vào từ bàn phím. Số lượng sinh viên không được xác định trước mà được người dùng nhập vào từ bàn phím.

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int numberCourses;
    cout << "Enter the number of courses: ";
    cin >> numberCourses;

    int *courses;
    courses = new int[numberCourses];

    for (int count = 0; count < numberCourses; count++) {
        cout << "Enter the mark of course #" << count << ": ";
        cin >> courses[count];
    }

    int totalMark = 0;
    for (int count = 0; count < numberCourses; count++)
        totalMark += courses[count];

    cout << "Total mark: " << totalMark << endl;

    delete [] courses;

    return 0;
}
```

Hình 6.6: Mảng động và con trỏ.

6.7. Truyền tham số là con trỏ

Ngoài hai phương pháp truyền dữ liệu vào trong hàm đã được giới thiệu trong Mục 4.5, truyền tham số là giá trị và truyền tham số là tham chiếu tới đối số, C++ còn cho phép ta sử dụng phương pháp thứ ba: truyền tham số là con trỏ.

Trong Hình 6.7, biến con trỏ `courses` được truyền vào hàm `getMark` và `calculateMark`. Dẫn đến tham số hình thức của hàm `getMark` và `calculateMark` chính là con trỏ tới mảng `courses` trong hàm `main`.

```
#include <iostream>

using namespace std;

void getMark (int *courses, int numberCourses)
{
    for (int count = 0; count < numberCourses; count++) {
        cout << "Enter the mark of course # " << count << ": ";
        cin >> courses[count];
    }
}

int calculateMark (const int *courses, int numberCourses)
{
    int totalMark = 0;
    for (int count = 0; count < numberCourses; count++)
        totalMark += courses[count];
    return totalMark;
}

int main(int argc, char *argv[])
{
    int numberCourses;
    cout << "Enter the number of courses: ";
    cin >> numberCourses;
    int *courses;
    courses = new int[numberCourses];

    getMark (courses, numberCourses);
    int totalMark = calculateMark (courses, numberCourses);
    cout << "Total mark: " << totalMark << endl;

    delete courses;

    return 0;
}
```

Hình 6.7: Tham số của hàm là con trỏ.

Trong những trường hợp ta muốn dùng con trỏ để truyền lượng lớn dữ liệu vào hàm nhưng lại không muốn cho hàm sửa đổi dữ liệu, ta có thể dùng từ khóa `const` để thiết lập quyền của hàm đối với tham số. Xem ví dụ hàm `calculateMark` ở Hình 6.7, trong đó, từ khóa `const` ở phía trước tham số hình thức `courses`

```
int calculateMark (const int *courses, int numberCourses)
```

quy định rằng hàm `calculate` phải coi dữ liệu `int` mà `courses` trỏ tới là hằng và không được phép sửa đổi.

Về cách sử dụng từ khóa `const` khi khai báo một biến con trỏ, ta có 4 lựa chọn:

1. không quy định con trỏ hay dữ liệu được trỏ tới là hằng
   `float * ptr;`
2. quy định dữ liệu được trỏ tới là hằng, con trỏ thì được sửa đổi.
   `const float * const ptr;`
3. quy định dữ liệu trỏ tới không phải là hằng, nhưng biến con trỏ thì là hằng
   `float * const ptr;`
4. quy định cả con trỏ lẫn dữ liệu nó trỏ tới đều là hằng.
   `const float * const ptr;`

Sử dụng `const` cho những hoàn cảnh thích hợp là một phần của phong cách lập trình tốt. Nó giúp giảm quyền hạn của hàm xuống tới mức vừa đủ, chẳng hạn một hàm chỉ có chức năng đọc và tính toán dữ liệu thì không nên có quyền sửa dữ liệu, từ đó giảm nguy cơ của hiệu ứng phụ không mong muốn.

Một điểm quan trọng cần lưu ý là tình trạng con trỏ có giá trị null hoặc không xác định do chưa gán bằng địa chỉ của biến nào. Nếu con trỏ `ptr` có giá trị null hoặc không xác định thì việc truy nhập `*ptr` sẽ dẫn đến lỗi run-time hoặc lỗi lô-gic cho chương trình. Ta cần chú ý khởi tạo giá trị của các biến con trỏ và kiểm tra giá trị trước khi truy nhập. Ngoài ra, còn có một lời khuyên là nên sử dụng tham chiếu thay cho con trỏ bất cứ khi nào có thể.

Bài tập

1. Trình bày sự khác biệt, ưu điểm, nhược điểm giữa biến tĩnh và biến động.
2. Tính giá trị của `apples`, `*ptrApp`, `*ptrApp2` của đoạn mã sau:

```
int apples;
int *ptrApp = &apples;
int *ptrApp2 = ptrApp;
apples += 2;
*ptrApp --;
*ptrApp2 += 3;
```

3. Xác định kết quả của đoạn mã sau:

```
int *p1 = new int;
int *p2;
*p1 = 5;
p2 = p1;
cout << "*p1 = " << *p1 << endl;
cout << "*p2 = " << *p2 << endl;
*p2 = 10;
cout << "*p1 = " << *p1 << endl;
cout << "*p2 = " << *p2 << endl;
p1 = new int;
*p1 = 20;
cout << "*p1 = " << *p1 << endl;
cout << "*p2 = " << *p2 << endl;
```

4. Nhập từ bàn phím vào một danh sách các số nguyên. Hãy sử dụng cấu trúc mảng động để lưu giữ dãy số nguyên này và tìm số lượng số nguyên chia hết cho số nguyên đầu tiên trong dãy.
5. Trình bày mối quan hệ giữa mảng và con trỏ.
6. Phân tích sự khác biệt, nhược điểm, ưu điểm của việc sử dụng mảng động và mảng tĩnh.
7. Sử dụng cấu trúc mảng động để lưu giữ một danh sách tên các sinh viên nhập vào từ bàn phím. Hãy sắp xếp danh sách tên sinh viên tăng dần theo độ dài của tên. Hiện ra màn hình danh sách sau khi đã sắp xếp.

8. Một bàn cờ có kích thước m*n ô vuông. Trạng thái trên mỗi ô vuông được biểu diễn bởi một kí tự in thường từ ‘A’ đến ‘Z’. Sử dụng mạng động hai chiều để lưu giữ trạng thái của bàn cờ nhập từ bàn phím. Tìm và hiện ra màn hình:

   - Các hàng thỏa mãn điều kiện tất cả các ô cùng một trạng thái
   - Các cột thỏa mãn điều kiện tất cả các ô cùng một trạng thái
   - Các đường chéo thỏa mãn điều kiện tất cả các ô cùng một trạng thái

Chương 7. Các kiểu dữ liệu trừu tượng

Các kiểu dữ liệu có sẵn của một ngôn ngữ lập trình, đặc biệt là các kiểu dữ liệu cơ bản, đôi khi không đủ để biểu diễn dữ liệu của bài toán cần giải quyết. Thông thường, ta cần gộp một vài thành phần dữ liệu có liên quan với nhau lại để biểu diễn một phần tử dữ liệu phức hợp mới. Chẳng hạn:

- Dữ liệu cần thiết để mô tả một ngày (`Date`) gồm 3 thành phần: ngày, tháng, năm. Ví dụ, dữ liệu về ngày Quốc khánh của Việt Nam gồm 3 thành phần: 2 (ngày), 9 (tháng), 1945 (năm).
- Dữ liệu cần thiết để mô tả `Student` bao gồm ba thành phần: `studentNumber` thuộc kiểu `int`, `birthday` thuộc kiểu `Date`, `name` thuộc kiểu `char [50]`.

Kiểu dữ liệu trừu tượng là kiểu dữ liệu phức hợp được cấu tạo từ các thành phần dữ liệu thuộc các kiểu dữ liệu khác nhau (có thể là kiểu có sẵn hoặc kiểu dữ liệu mà chúng ta tự định nghĩa). Đây là cơ chế cho phép chúng ta tự định nghĩa các kiểu dữ liệu mới, chẳng hạn như `Date` và `Student` như miêu tả ở trên.

Đối với mỗi một kiểu dữ liệu trừu tượng, ta cần định nghĩa một loạt các thao tác xử lý dữ liệu cho nó. Ví dụ, với kiểu dữ liệu `Date` có thể cần đến các thao tác xử lý dữ liệu như in ra màn hình, và các phép tính đối với dữ liệu ngày tháng như: tính ngày hôm qua, tính ngày mai, tính khoảng cách giữa hai ngày, …

Các ngôn ngữ lập trình bậc cao có thể chia thành hai loại: ngôn ngữ lập trình hướng thủ tục, và ngôn ngữ lập trình hướng đối tượng. Kiểu dữ liệu trừu tượng ở hai loại ngôn ngữ này có sự khác biệt như sau:

- Đối với các **ngôn ngữ lập trình hướng thủ tục**, các kiểu dữ liệu có cấu trúc chỉ dừng lại ở việc đóng gói các thành phần dữ liệu có liên quan lại với nhau. Phần xử lý dữ liệu được đặt tại các hàm và thủ tục độc lập.
- Các **ngôn ngữ lập trình hướng đối tượng** đi xa hơn một bước: cho phép đóng gói cả các hàm xử lý dữ liệu vào trong kiểu dữ liệu, coi chúng như là một phần không thể tách rời của kiểu dữ liệu. Trong các ngôn ngữ này, kiểu dữ liệu trừu tượng được gọi là các **lớp đối tượng** (*class*).

7.1. Định nghĩa kiểu dữ liệu trừu tượng bằng cấu trúc struct

Các ngôn ngữ lập trình bậc cao sử dụng các cấu trúc khá tương tự nhau để mô tả dữ liệu trừu tượng. Một trong các cấu trúc mà C++ cung cấp cho công việc này

là cấu trúc `struct`. Ví dụ, kiểu dữ liệu trừu tượng `Time` có thể được định nghĩa như sau trong C++:

```
struct Time {
   int hour;
   int minute;
   int second;
};
```

Trong đó, `hour`, `minute`, `second` là các trường hay các thành viên dữ liệu của cấu trúc `Time`.

Chú ý rằng các trường thuộc cùng một cấu trúc phải có tên khác nhau tuy rằng có thể trùng tên với các trường của một cấu trúc khác. Ngoài ra, phần định nghĩa một cấu trúc phải được kết thúc bằng một dấu chấm phẩy.

Sau khi đã định nghĩa kiểu dữ liệu `Time`, ta có thể sử dụng nó y như các kiểu dữ liệu khác. Ta có thể khai báo biến, mảng, con trỏ, tham chiếu kiểu `Time`, ví dụ:

```
Time dinnerTime;
Time appointment[10];
Time *timePtr = &dinnerTime;
Time &timeRef = dinnerTime;
```

Lưu ý: mặc dù cùng dùng từ khóa `struct`, nhưng `struct` của C++ không tương đương với `struct` của ngôn ngữ lập trình C.

Trong C++, biến thuộc kiểu dữ liệu cấu trúc cũng được đối xử như các kiểu dữ liệu thông thường. Ta có thể thực hiện phép gán giá trị của biến `Time` này cho biến `Time` khác, ví dụ:

```
appointment[1] = dinnerTime;
```

Kết quả là giá trị của các thành viên dữ liệu của `dinnerTime` được sao chép vào các thành viên dữ liệu tương ứng của `appointment[1]`. Lưu ý rằng phép gán mặc định của C++ là phép gán nông, nghĩa là chỉ có giá trị của các thành viên được sao chép. Do đó nếu một trong các thành viên dữ liệu là con trỏ tới một vùng nhớ thì chỉ có giá trị của con trỏ được sao chép chứ nội dung của vùng nhớ thì không.

Để truy cập các thành viên dữ liệu của cấu trúc, ta có hai toán tử dấu chấm (`.`) và mũi tên (`->`). Toán tử mũi tên (`->`) dùng để truy nhập các thành viên qua một

con trỏ đến đối tượng. Toán tử dấu chấm (`.`) được dùng trong các trường hợp còn lại. Ví dụ, để in thành viên `hour` của biến `dinnerTime` ra màn hình:

```
cout << dinnerTime.hour;
```

hoặc

```
cout << timeRef.hour;
```

hoặc thông qua biến con trỏ `timePtr` như sau

```
cout << timePtr->hour;
```

Trong đó `timePtr` đang chứa địa chỉ của biến `dinnerTime`, còn `timeRef` là một tham chiếu của biến `dinnerTime`.

Lưu ý rằng biểu thức `(*timePtr)` cho ta chính biến `dinnerTime`, do đó ba biểu thức `(*timePtr).hour`, `dinnerTime.hour`, và `timePtr->hour` tương đương với nhau. Ngoài ra, cặp ngoặc trong biểu thức `(*timePtr).hour` là cần thiết do toán tử * không được ưu tiên bằng toán tử `->`.

Hình 7.1 minh họa cách khai báo và sử dụng `struct` để khai báo cấu trúc dữ liệu `Time`. Cũng như các kiểu dữ liệu khác, kiểu dữ liệu có cấu trúc cũng có thể được truyền vào trong các hàm dưới dưới dạng tham số (xem Hình 7.2). Trong ví dụ đó, phần mã chương trình có nhiệm vụ in một giá trị kiểu `Time` ra màn hình được chuyển thành hàm `print` với tham số là một tham chiếu tới biến `Time` cần in. Trong ví dụ này, tham số của hàm `print` được quy định là hằng tham chiếu (từ khóa `const`) để hàm này không có quyền sửa dữ liệu nằm trong biến kiểu `Time` được truyền vào.

Khi sử dụng cơ chế truyền bằng giá trị, chẳng hạn `print(Time)`, một bản sao của đối số kiểu Time sẽ được truyền vào hàm tương tự như đối với tham số thuộc các kiểu dữ liệu khác. Tuy nhiên, để tránh việc phải tốn chi phí tính toán và bộ nhớ cho việc sao chép các cấu trúc mà không phải cấp quyền sửa một cách không cần thiết, người ta thường dùng tham số là hằng tham chiếu như tại hàm `print(const Time & t)` trong Hình 7.2 thay vì dùng kiểu truyền giá trị.

Một lựa chọn khác là dùng tham số là con trỏ, chẳng hạn

```
void print(Time * ptrTime)
```

Và nếu không muốn cho print quyền sửa dữ liệu của biến Time được truyền cho nó, ta quy định tham số là con trỏ tới hằng kiểu Time, nghĩa là:

```
void print (const Time * ptrTime)
```

Để ý rằng cấu trúc `Time` trong Hình 7.2 chỉ bao gồm dữ liệu. Phần chương trình xử lý dữ liệu của `Time` được đặt ở bên ngoài cấu trúc dữ liệu `Time`, tại các hàm có chức năng thao tác dữ liệu `Time`. Đây chính là đặc điểm của phong cách lập trình hướng thủ tục: dữ liệu được khai báo một nơi và xử lý dữ liệu một nơi. Cụ thể, trong ví dụ của ta, các hàm `print`, `setTime` và `main` tuy nằm ngoài và độc lập với `Time` nhưng lại có toàn quyền thao tác dữ liệu nằm trong các biến kiểu `Time`, chẳng hạn như gán cho một biến Time giá trị không hợp lệ 3:100:-1. Đây là giới hạn của các ngôn ngữ lập trình thủ tục. Các ngôn ngữ lập trình hướng đối tượng đi xa hơn và cung cấp cơ chế tự bảo vệ cho các kiểu dữ liệu. Ta có thể sửa cấu trúc `Time` trong Hình 7.2 để sử dụng cơ chế đó, mục tiếp theo sẽ nói về vấn đề này.

```
// define struct Time and test it.
#include <iostream>

using namespace std;

// Time structure definition
struct Time {
   int hour;     // 0-23 (24-hour clock format)
   int minute;   // 0-59
   int second;   // 0-59
}; // end struct Time

int main()
{
    Time dinnerTime, midnight; // variables of new type Time

    dinnerTime.hour = 18;    // set hour member of dinnerTime
    dinnerTime.minute = 30;  // set minute member of dinnerTime
    dinnerTime.second = 0;   // set second member of dinnerTime

    midnight.hour = 0;    // set hour member of midnight
    midnight.minute = 0;  // set minute member of midnight
    midnight.second = 0;   // set second member of midnight

    cout << "Dinner will be held at "
         << dinnerTime.hour << ":"
         << dinnerTime.minute << ":"
         << dinnerTime.second << endl;

    cout << "Lights will be switched off at "
         << midnight.hour << ":"
         << midnight.minute << ":"
         << midnight.second << endl;

   return 0;
}
```

Kết quả chạy chương trình

```
Dinner will be held at 18:30:0
Lights will be switched off at 0:0:0
```

Hình 7.1: Ví dụ về khai báo và sử dụng cấu trúc dữ liệu struct.

```
// define struct Time and test it.
#include <iostream>
#include <iomanip>

using namespace std;

// Time structure definition
struct Time {
    int hour;     // 0-23 (24-hour clock format)
    int minute;   // 0-59
    int second;   // 0-59
}; // end struct Time

// print time to the screen
void print (const Time &t)
{
    cout << t.hour << ":" << t.minute << ":" << t.second;
}

// set hour, minute, and second of a Time structure
void setTime (Time &t, int hour, int minute, int second)
{
    t.hour = hour; t.minute = minute; t.second = second;
}

int main()
{
    Time dinnerTime, midnight; // variables of the type Time

    setTime(dinnerTime, 18, 30, 0);
    setTime(midnight, 0, 0, 0);

    cout << "Dinner will be held at ";
    print(dinnerTime);
    cout << endl;
    cout << "Lights will be switched off at ";
    print(midnight);

    return 0;
}
```

Hình 7.2: Ví dụ về dùng dữ liệu cấu trúc làm tham số cho hàm.

7.2. Định nghĩa kiểu dữ liệu trừu tượng bằng cấu trúc `class`

Như đã nói ở trên, ta có thể đóng gói các hàm xử lý dữ liệu vào bên trong kiểu dữ liệu trừu tượng. Ví dụ, các hàm `print` và `setTime` trong Hình 7.2 chính là một tiện ích cho kiểu dữ liệu `Time` và nên được đóng gói vào bên trong kiểu dữ liệu này.

Cấu trúc class cho phép ta thực hiện việc đóng gói đó. Khi thực hiện công việc này, ta bắt đầu bước từ lập trình hướng thủ tục sang lập trình hướng đối tượng.

Hình 7.3 minh họa cách khai báo và cài đặt lớp đối tượng `Time` sử dụng cấu trúc `class`. Một biến khi khai báo thuộc lớp `Time` thì biến đó được gọi là một đối tượng `Time`.

So với `struct Time` trong Hình 7.2, `class Time` khác ở hai điểm: (1) bao gồm cả các hàm `setTime` và `print`; (2) có thêm nhãn quyền truy nhập public. Các hàm `setTime` và `print` của lớp đối tượng `Time` trong Hình 7.3 hoạt động giống như các hàm tương ứng trong Hình 7.2, chỉ khác ở chỗ ta không cần truyền tham số là một đối tượng `Time` cho các hàm này (chi tiết sẽ được giải thích trong mục sau).

Về mặt hoạt động, cài đặt của `class Time` trong Hình 7.3 hoàn toàn tương đương với `struct Time` trong Hình 7.2. Cụ thể, hàm `main` hay một hàm nào khác vẫn có thể tạo các đối tượng `Time` với dữ liệu không hợp lệ. Nói cách khác, so với `struct Time` trong Hình 7.2, `class Time` trong Hình 7.3 mới chỉ tiến một bước là đóng gói dữ liệu với phần xử lý.

Cải tiến trong Hình 7.4 là bước tiếp theo, thực hiện được nhiệm vụ kiểm soát dữ liệu. Trong cài đặt này, các thành viên dữ liệu được giới hạn là chỉ được truy nhập từ bên trong `class Time`, còn hàm `setTime` đảm bảo dữ liệu được gán cho các thành viên dữ liệu phải có giá trị hợp lệ. Chi tiết sẽ được giải thích trong các mục sau.

Lưu ý rằng ta hoàn toàn có thể dùng từ khóa `struct` thay vì `class` trong tất cả các trường hợp, nhưng theo thông lệ, `struct` thường được dùng cho các lớp đối tượng chỉ có dữ liệu còn `class` dùng cho các lớp gồm cả dữ liệu và hàm, nên ta chuyển sang dùng cấu trúc `class` kể từ ví dụ này.

```
// define class Time and test it.
#include <iostream>
using namespace std;

// class Time definition
class Time {
public:
    void print () {
        cout << hour << ":" << minute << ":" << second;
    }
    void setTime (int h, int m, int s) {
        hour = h; minute = m; second = s;
    }

    int hour;     // 0-23 (24-hour clock format)
    int minute;   // 0-59
    int second;   // 0-59
}; // end class Time

int main()
{
    Time dinnerTime, midnight; // variables of the type Time

    setTime(dinnerTime, 18, 30, 0);
    setTime(midnight, 0, 0, 0);

    cout << "Dinner will be held at ";
    print(dinnerTime);
    cout << endl;
    cout << "Lights will be switched off at ";
    print(midnight);

    return 0;
}
```

Hình 7.3: Đóng gói các hàm xử lý dữ liệu vào trong class Time.

```
// define class Time and test it.
#include <iostream>
using namespace std;

// class Time definition
class Time {
public:
    void print () {
       cout << hour << ":" << minute << ":" << second;
    }
    bool setTime (int h, int m, int s);

private:
    int hour;     // 0-23 (24-hour clock format)
    int minute;   // 0-59
    int second;   // 0-59
    bool isValid (int h, int m, int s) {
        return (h >= 0 && h < 24) &&
               (m >= 0 && m < 60) && (s >= 0 && s < 60);
    }
}; // end class Time

bool Time::setTime (int h, int m, int s)
{
    if (! isValid(h,m,s)) return false;
    hour = h; minute = m; second = s;
    return true;
}

int main()
{
    Time dinnerTime, midnight; // variables of the type Time

    setTime(dinnerTime, 18, 30, 0);
    setTime(midnight, 0, 0, 0);

    cout << "Dinner will be held at ";
    print(dinnerTime);
    cout << endl;
    cout << "Lights will be switched off at ";
    print(midnight);

    return 0;
}
```

Hình 7.4: Bổ sung cho class Time khả năng tự kiểm soát tính hợp lệ của dữ liệu.

7.2.1. Quyền truy nhập

Khi khai báo một lớp đối tượng, chúng ta mong muốn phân quyền truy nhập và sử dụng dữ liệu cũng như các hàm của lớp đó. Tức là, một số dữ liệu hay hàm được truy nhập và sử dụng rộng rãi bởi tất cả các hàm thuộc hay không thuộc lớp đối tượng đó. Bên cạnh đó, chúng ta cũng mong muốn một số dữ liệu hay hàm được bảo vệ khỏi việc bị truy nhập và sử dụng từ bên ngoài. Tức là, chỉ các hàm thuộc lớp đó được phép truy nhập và sử dụng, các hàm bên ngoài không thuộc lớp đối tượng đó thì không được phép truy nhập và sử dụng.

Các ngôn ngữ lập trình hướng đối tượng cung cấp cho chúng ta cơ chế để phân quyền như vậy.

Nhãn quyền truy nhập (*member access specifier*) quy định quyền truy nhập đến các thành viên của lớp. Trong số đó có hai loại thường được sử dụng là:

- **public**: Dữ liệu hay hàm thuộc loại **công cộng** (được khai báo dưới nhãn `public`) có thể được truy nhập và sử dụng rộng rãi bởi tất cả các hàm thuộc hay không thuộc lớp đối tượng đó. Trong Hình 7.4, hàm `print` của lớp đối tượng Time là hàm public, và ta có thể gọi nó thông qua các biến `midnight` (`midnight.print`) hay `dinnerTime` (`dinnerTime.print`).

- **private**: Dữ liệu hay hàm thuộc loại **riêng tư** (được khai báo dưới nhãn `private`) chỉ được truy nhập và sử dụng bởi các hàm thành viên thuộc lớp đó. Các hàm bên ngoài không thuộc lớp đối tượng đó thì không được phép truy nhập và sử dụng. Trong Hình 7.4, các biến `hour`, `minute`, `second`, và hàm `isValid` thuộc diện `private` và chỉ có các hàm thành viên `print` và `setTime` là có thể truy nhập trực tiếp, chẳng hạn để đọc/ghi giá trị của `hour`. Trong khi, đó tại hàm `main`, nghĩa là ở ngoài lớp `Time`, ta không thể truy nhập vào các biến `hour`, `minute`, `second` hay gọi hàm `isValid`. Các lệnh sau đây không hợp lệ và sẽ gây lỗi khi biên dịch:

  ```
  dinnerTime.hour = 18;
  dinnerTime.isValid(10,3,4);
  ```

  Loại nhãn `private` này thường được dùng cho các hàm tiện ích chỉ cần dùng đến ở bên trong phạm vi lớp bởi các hàm thành viên khác.

Bên cạnh hai nhãn phổ biến trên, ta còn các loại nhãn phân quyền khác như `protected`, `friend` (ta sẽ không nói đến trong phạm vi khóa học này).

Nếu không dùng các từ khóa trên để quy định quyền truy nhập một các tường minh, thì quyền truy nhập mặc định của các thành viên của `class` là `private`, còn của `struct` là `public`. Các loại quyền truy nhập ngoài mặc định phải được quy định một cách tường minh.

Tuy các thành viên private của một lớp không thể được truy nhập trực tiếp từ ngoài nhưng chúng có thể được truy nhập gián tiếp thông qua các hàm mà lớp dành riêng cho công việc này. Chẳng hạn với các đối tượng thuộc lớp `Time`, hàm `print` cho phép đọc và hàm `setTime` cho phép ghi dữ liệu private. Ngoài ra, `setTime` còn có một chức năng quan trọng là kiểm tra tính hợp lệ của dữ liệu mới trước khi gán trị cho các thành viên dữ liệu (nó gọi hàm `isValid`). Nếu cần thiết, ta có thể bổ sung các hàm `getHour`, `setHour`, `getMinute`, `setMinute`, `getSecond`, `setSecond` để truy cập đến các thành viên dữ liệu private. Cùng với `setTime`, chúng được gọi là các **hàm truy cập** (*accessor*) của lớp `Time`, chúng kiểm soát dữ liệu được ghi vào đối tượng và quản lý định dạng của kết quả đọc dữ liệu của đối tượng.

7.2.2. Toán tử phạm vi và định nghĩa các hàm thành viên

Việc khai báo và định nghĩa một hàm có thể thực hiện đồng thời bên trong một lớp đối tượng (chẳng hạn hàm `print` trong Hình 7.4). Nhưng ta cũng có thể tách phần khai báo và định nghĩa một hàm ra hai nơi khác nhau: phần khai báo nằm bên trong khai báo lớp, phần định nghĩa nằm bên ngoài khai báo lớp (chẳng hạn hàm `setTime` trong Hình 7.4). Nếu đặt định nghĩa hàm ở bên ngoài này, ta phải dùng **toán tử phạm vi** (`::`) để chỉ rõ rằng ta đang định nghĩa một hàm thành viên của lớp `Time` chứ không phải một hàm thông thường.

Toán tử phạm vi chỉ rõ lớp đối tượng mà một thành viên thuộc về, cho phép hàm thành viên đó có được tính chất phạm vi như thể định nghĩa của nó nằm bên trong khối định nghĩa của lớp đối tượng chủ của nó. Ví dụ, phần thân hàm `setTime` được xem là nằm bên trong phạm vi của lớp Time, do đó, từ đây vẫn có thể truy nhập trực tiếp các biến thành viên loại private thuộc lớp `Time`.

Lưu ý, cũng như các thành viên dữ liệu, do đã nằm bên trong phạm vi của lớp nên các hàm thành viên của các lớp khác nhau có thể trùng tên.

Để ý các lời gọi hàm `setTime` từ hai đối tượng khác nhau `dinnerTime` và `midnight` trong Hình 7.3 hay Hình 7.4, lời gọi hàm từ đối tượng nào thì hàm hoạt động trên các thành viên dữ liệu của đối tượng đó. Tuy nhiên, ta không phải truyền đối tượng vào hàm như trong Hình 7.2 – khi hàm `print` là hàm

thông thường chứ không phải hàm thành viên. Lí do là vì khi gọi hàm thành viên từ một đối tượng, đối tượng đó được tự động truyền vào hàm như là một tham số ẩn.

7.2.3. Hàm khởi tạo và hàm hủy

Hình 7.5 là phiên bản mở rộng của cài đặt lớp `Time` trong Hình 7.4 (các phần `<...>` trong Hình 7.5 có nội dung giống như đoạn tương ứng trong Hình 7.4). Ngoài thành viên dữ liệu `note`, trong Hình 7.5 còn có thêm các hàm đặc biệt là `Time(),Time(int h, int m, int s)` và `~Time()`. Các hàm có tên `Time` được gọi là hàm khởi tạo (*constructor*), còn `~Time` được gọi là hàm hủy (*destructor*).

Hàm khởi tạo là hàm thành viên đặc biệt có chức năng khởi tạo các thành viên dữ liệu của đối tượng. Nó được gọi một cách tự động khi đối tượng được tạo, ví dụ khi một biến thuộc lớp đối tượng, chẳng hạn hàm khởi tạo `Time()` được gọi khi lệnh khai báo `midnightTime` được thực thi. Hàm khởi tạo phải trùng tên với tên lớp, không có giá trị trả về, và không có kiểu giá trị trả về. Một lớp có thể có vài hàm khởi tạo hoạt động theo nguyên tắc hàm trùng tên. Khi một đối tượng được tạo, trình biên dịch sẽ chọn gọi hàm nào có danh sách tham số khớp với các đối số được cho tại lệnh khai báo đối tượng. Trong Hình 7.5, đối tượng `dinnerTime` được khai báo với 3 đối số, cho nên hàm khởi tạo `Time (int h, int m, int s)` sẽ được tự động gọi khi chạy lệnh khao báo biến `dinnerTime`. Trong khi đó, biến `midnight` được khai báo không có đối số, nên hàm khởi tạo `Time()` không yêu cầu tham số sẽ được tự động gọi khi chạy lệnh khao báo biến `midnight`.

Hàm hủy, ví dụ `~Time()`, thực hiện chức năng ngược lại với hàm khởi tạo. Nó được gọi tự động khi một đối tượng bị hủy khi thời gian sống của nó đã kết thúc hoặc khi nó là một đối tượng được cấp phát bộ nhớ động và đang được giải phóng khỏi bộ nhớ bằng lệnh `delete`. Hàm hủy phải trùng tên với tên lớp nhưng có thêm dấu ngã (~) đặt trước. Hàm hủy không được có kết quả trả về.

Đối với một đối tượng có thành viên dữ liệu là bộ nhớ động được cấp phát trong thời gian sống của đối tượng, hàm hủy là nơi thích hợp để viết các lệnh giải phóng phần bộ nhớ được cấp phát đó, mà đoạn lệnh này sẽ được thực hiện tại thời điểm mà đối tượng bị hủy. Trong Hình 7.5, hàm hủy `~Time` chứa một lệnh giải phóng bộ nhớ động mà thành viên dữ liệu `note` trỏ tới. Công việc này sẽ được thực hiện đối với từng đối tượng `midnight` và `dinnerTime` khi chương trình chạy hết thời gian sống của chúng.

Cả hai loại hàm hủy và hàm tạo đều không bắt buộc. Nếu không được định nghĩa, trình biên dịch sẽ tự tạo các hàm tạo và hủy mặc định có nội dung rỗng.

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

class Time {
public:
    Time () {  //default constructor
        hour = minute = second = 0;
        note = 0;
    }
    Time (int h, int m, int s, const char* n) {
        setTime(h, m, s);
        note = new char [strlen(n) + 1];
        strcpy (note, n);
    }
    ~Time () { delete [] note; }  //destructor

    bool setTime (int h, int m, int s) { <...> }
    void print () {
        cout << hour << ":" << minute << ":" << second;
        if (note != 0) cout << " (" << note << ")";
    }

private:
    int hour;     // 0-23 (24-hour clock format)
    int minute;   // 0-59
    int second;   // 0-59
    char* note;
    bool isValid (int h, int m, int s) { <...> }
};

int main()
{
    Time dinnerTime (19, 30, 0, "first date");
    Time midnight;

    cout << "Dinner will be held at ";
    dinnerTime.print();
    cout << endl << "Light will be switched off at ";
    midnight.print();

    return 0;
}
```

Hình 7.5: Hàm tạo và hàm hủy.

7.3. Lợi ích của lập trình hướng đối tượng

Tại Hình 7.4, các nhãn quyền truy nhập `public` và `private` quy định rằng các thành viên dữ liệu (`hour`, `minute`, `second`) chỉ được phép truy nhập từ bên trong lớp `Time` (hàm `main` nằm ngoài phạm vi này), còn các hàm `setTime` và `print` có thể được gọi từ bất cứ đâu trong chương trình. Trong khi đó, tại Hình 7.2 các thành viên dữ liệu đó có thể được truy nhập từ bất cứ đâu (chẳng hạn hàm `main`) do có quyền truy nhập mặc định là public.

Điều đó có nghĩa là dữ liệu của `struct Time` có thể bị đọc và sửa đổi tùy ý tại bất cứ đoạn chương trình nào, trong khi đó, từ bên ngoài `class Time`, dữ liệu chỉ có thể được đọc bằng cách gọi hàm `print` và được ghi bằng cách gọi hàm thành viên `setTime` – nơi giá trị mới được kiểm tra tính hợp lệ (gọi hàm `isValid`) trước khi cập nhật.

Kết quả của các khác biệt trên là:

Để làm việc với dữ liệu kiểu dữ liệu `Time` trong Hình 7.2, các đoạn chương trình bên ngoài phải biết chi tiết cấu tạo của `Time`. Còn đối với kiểu `Time` trong Hình 7.4, các đoạn mã khác khỉ cần biết cách sử dụng các hàm `print` và `setTime` là đủ - đây chính là **giao diện** (*interface*) của `Time`.

Các đoạn chương trình bên ngoài cấu trúc `Time` ở Hình 7.2 có toàn quyền thao túng dữ liệu của các đối tượng `Time`, còn cấu trúc `Time` ở Hình 7.4 có thể tự đảm bảo được tính hợp lệ dữ liệu của mình bằng việc cho phép truy nhập có kiểm soát qua các hàm thành viên của chính nó.

Sự khác biệt này không có nhiều ý nghĩa đối với một chương trình nhỏ chỉ do một người viết, tuy nhiên, nó mang lại ích lợi quan trọng trong quá trình phát triển các phần mềm lớn hơn với nhiều mô đun và với sự tham gia của nhiều lập trình viên, trong đó có việc tăng tính mô đun và giảm lỗi lập trình.

Tóm lại, mỗi lớp đối tượng là khuôn mẫu hay mô hình cho việc tạo các đối tượng thuộc một kiểu nhất định. Lớp đối tượng định nghĩa hai loại thành viên và mỗi đối tượng thuộc lớp đó đều có:

- Các thành viên dữ liệu mô tả các thuộc tính của đối tượng. Ví dụ là `hour`, `minute`, và `second` của cấu trúc `Time`.
- Các hàm thành viên, hay hàm, mô tả hành vi của đối tượng (các hành động mà đối tượng có thể thực hiện). Ví dụ là các hàm `print`, `setTime` của cấu trúc `Time`.

Lập trình hướng đối tượng cho phép đơn giản hóa việc lập trình. Các mô đun chương trình có thể kết nối với nhau mà chỉ cần biết giao diện của nhau. Các chi tiết cài đặt được che dấu bên trong các mô-đun và bên ngoài không cần biết đến.

Lập trình hướng đối tượng còn giúp tăng tính tái sử dụng và khả năng tích hợp các mô đun phần mềm, thể hiện ở hai điểm: các thành viên của một lớp có thể là đối tượng thuộc lớp khác, và các lớp mới được tạo từ lớp cũ bằng quan hệ thừa kế (chủ đề này nằm ngoài phạm vi khóa học này).

7.4. Biên dịch riêng rẽ

C++ cho phép chia chương trình thành nhiều phần lưu tại các tệp khác nhau, Các tệp này có thể được biên dịch riêng rẽ, rồi liên kết (*link*) với nhau trước khi chạy chương trình.

Cách phân chia thông dụng nhất là đặt định nghĩa của một lớp cùng với định nghĩa các hàm thành viên của nó vào các tệp riêng chứ không đặt cùng chương trình sử dụng class đó. Với cách này, ta có thể xây dựng một thư viện các lớp đối tượng sao cho nhiều chương trình có thể dùng chung một lớp. Ta có thể biên dịch lớp đó đúng một lần và dùng nó trong nhiều chương trình khác nhau, tương tự như ta vẫn dùng các thư viện quen thuộc `iostream` và `cstring`.

Với mỗi lớp, ta còn có thể viết định nghĩa của nó tại hai tệp để tách giao diện của lớp (khối định nghĩa lớp gói trong cặp ngoặc {}) khỏi chi tiết cài đặt của lớp (định nghĩa các hàm thành viên). Khi đó, nếu ta thay đổi cài đặt của một lớp mà vẫn giữ nguyên giao diện của lớp đó thì ta chỉ phải dịch lại tệp chứa cài đặt của lớp đó. Các tệp khác, trong đó có các tệp chứa các chương trình sử dụng lớp đó, không cần phải sửa đổi gì và không cần biên dịch lại. Hình 7.6, Hình 7.7 và Hình 7.8 là kết quả của việc tách chương trình trong Hình 7.5 thành ba tệp riêng rẽ.

Thông lệ cho việc tách tệp như sau:

- Tệp header chứa định nghĩa lớp và các nguyên mẫu hàm thành viên, nghĩa là phần nằm trong cặp ngoặc {}, được đặt tên có phần mở rộng là .h (tệp time.h trong Hình 7.7). Tệp này phải được include trong tất cả các tệp sử dụng lớp Time (các tệp time.cpp và main.cpp).
- Tệp mã nguồn chứa định nghĩa của các hàm thành viên của lớp, thường trùng tên với tên tệp header tương ứng (ví dụ time.cpp) và có phần mở rộng là .cpp hay .cc.

Các tệp này được biên dịch riêng rẽ và liên kết với tệp chương trình chính (main.cpp) trước khi chạy chương trình.

Để ý mã nguồn của tệp time.h, ta thấy phần nội dung tệp được gói trong bộ định hướng tiền xử lý dưới đây để tránh việc nội dung tệp được include nhiều lần trong một chương trình.

```
#ifndef __TIME_H__
#define __TIME_H__
…
#endif
```

Đoạn tiền xử lý đó có nghĩa: nếu nhãn `__TIME_H__` chưa được định nghĩa thì định nghĩa `__TIME_H__`, và đoạn mã sau đó cho đến trước `#endif` mới được trình biên dịch xét đến. Còn nếu như `__TIME_H__` đã được định nghĩa khi trình biên dịch duyệt đến dòng `#ifndef __TIME_H__`, thì trình biên dịch sẽ bỏ qua phần mã tiếp theo cho đến hết `#endif`. Dòng `#define __TIME_H__` nằm trong tệp time.h có tác dụng đánh dấu trạng thái rằng time.h đã được include vào chương trình. Cái tên `__TIME_H__` là được đặt theo thông lệ để tương ứng với tên tệp time.h.

```
#include <iostream>
#include "time.h"

using namespace std;

int main()
{
    Time dinnerTime (19, 30, 0, "first date");
    Time midnight;

    cout << "Dinner will be held at ";
    dinnerTime.print();
    cout << endl << "Light will be switched off at ";
    midnight.print();

    return 0;
}
```

Hình 7.6: Tệp main.cpp chứa chương trình chính.

```
#ifndef TIME_H
#define TIME_H

class Time {
public:
    Time () {  //default constructor
        hour = minute = second = 0;
        note = 0;
    }
    Time (int h, int m, int s, const char* n);
    ~Time () { delete [] note; }  //destructor

    bool setTime (int h, int m, int s) ;
    void print ();

private:
    int hour;     // 0-23 (24-hour clock format)
    int minute;   // 0-59
    int second;   // 0-59
    char* note;

    bool isValid (int h, int m, int s);
};

#endif
```

Hình 7.7: Tệp time.h chứa định nghĩa của lớp Time.

```
#include <cstring>
#include <iostream>
#include "time.h"

using namespace std;

Time::Time (int h, int m, int s, const char* n)
{
    setTime(h, m, s);
    note = new char [strlen(n) + 1];
    strcpy (note, n);
}

bool Time::setTime (int h, int m, int s)
{
    if (! isValid(h,m,s)) return false;
    hour = h; minute = m; second = s;
    return true;
}

void Time::print()
{
    cout << hour << ":" << minute << ":" << second;
    if (note != 0) cout << " (" << note << ")";
}

bool Time::isValid (int h, int m, int s)
{
    return (h >= 0 && h < 24) &&
           (m >= 0 && m < 60) && (s >= 0 && s < 60);
}
```

Hình 7.8: Tệp time.cpp chứa cài đặt lớp Time.

Bài tập

1. Trình bày các sự giống nhau và khác biệt giữa cấu trúc dữ liệu struct, và cấu trúc dữ liệu class.

    - Tìm hiểu và trình bày về lập trình hướng đối tượng, các ngôn ngữ lập trình hướng đối tượng mà bạn biết. Sự khác biệt, ưu điểm, nhược điểm của lập trình hướng đối tượng so với lập trình hướng thủ tục.

2. Trình bày sự khác biệt giữa dữ liệu và phương thức thuộc loại private và thuộc loại public. Khi nào thì nền khai báo biến và phương thức thuộc loại private.

Sử dụng cấu trúc `struct` và cấu trúc `class` để lưu giữ dữ liệu và giải quyết các bài toán sau đây:

3. Viết một class chứa thông tin về ba cạnh của một tam giác và các phương thức:

    - Kiểm tra xem ba cạnh đó có thỏa mãn là ba cạnh của tam giác hay không.
    - Tính diện tích tam giác.
    - Kiểm tra xem nó là tam giác thường, tam giác cân, hay tam giác đều.

4. Nhập từ bàn phím thông tin về một ngày (một ngày gồm ba thành phần là ngày, tháng, năm), hãy tính toán và hiện ra màn hình:

    - Ngày hôm trước.
    - Ngày hôm sau.
    - Còn bao nhiêu ngày nữa thì đến ngày 1/1/2020.

5. Viết một class lưu trữ thông tin về một sinh viên (tên tuổi, ngày tháng năm sinh, quê quán, lớp học, lực học từ 0,0 đến 10,0) và các phương thức tính toán:

    - Sinh viên đó bao nhiêu tuổi tính đến ngày hôm nay.
    - Phân loại học lực của sinh viên đó (kém, trung bình, giỏi, xuất sắc).
    - Quê quán thuộc có thuộc ba thành thành phố lớn (Hà Nội, Hồ Chí Minh, Đà Nẵng) hay không.

6. Nhập vào từ bàn phím một danh sách sinh viên. Hãy tính và hiện ra màn hình:
    - Thông tin về tất cả các bạn tên là Vinh
    - Thông tin về tất cả các bạn quê ở Hà Nội
    - Cho biết tổng số bạn có học lực kém (<4), học lực trung bình (≥4 và <8), học lực giỏi (≥8).
7. Sử dụng cấu trúc dữ liệu class và mảng động để lưu giữ danh sách sinh viên một lớp học. Hãy kiểm tra xem có hai bạn sinh viên nào trùng nhau tất cả các thông tin.

Chương 8. Vào ra dữ liệu

Tất cả các chương trình mà ta đã gặp trong cuốn sách này đều lấy dữ liệu vào từ bàn phím và in ra màn hình. Nếu chỉ dùng bàn phím và màn hình là các thiết bị vào ra dữ liệu thì chương trình của ta khó có thể xử lý được khối lượng lớn dữ liệu, và kết quả chạy chương trình sẽ bị mất ngay khi ta đóng cửa sổ màn hình output hoặc tắt máy. Để cải thiện tình trạng này, ta có thể lưu dữ liệu tại các thiết bị lưu trữ thứ cấp mà thông dụng nhất thường là ổ đĩa cứng. Khi đó dữ liệu tạo bởi một chương trình có thể được lưu lại để sau này được sử dụng bởi chính nó hoặc các chương trình khác. Dữ liệu lưu trữ như vậy được đóng gói tại các thiết bị lưu trữ thành các cấu trúc dữ liệu gọi là **tệp** (*file*).

Chương này sẽ giới thiệu về cách viết các chương trình lấy dữ liệu vào (từ bàn phím hoặc từ một tệp) và ghi dữ liệu ra (ra màn hình hoặc một tệp).

8.1. Khái niệm dòng dữ liệu

Trong một số ngôn ngữ lập trình như C++ và Java, dữ liệu vào ra từ tệp, cũng như từ bàn phím và màn hình, đều được vận hành thông qua các **dòng dữ liệu** (*stream*). Ta có thể coi dòng dữ liệu là một kênh hoặc mạch dẫn mà dữ liệu được truyền qua đó để chuyển từ nơi gửi đến nơi nhận.

Dữ liệu được truyền từ chương trình ra ngoài theo một **dòng ra** (*output stream*). Đó có thể là dòng ra chuẩn nối và đưa dữ liệu ra màn hình, hoặc dòng ra nối với một tệp và đẩy dữ liệu ra tệp đó.

Chương trình nhận dữ liệu vào qua một **dòng vào** (*input stream*). Dòng vào thì có thể là dòng vào chuẩn nối và đưa dữ liệu vào từ màn hình, hoặc dòng vào nối với một tệp và nhận dữ liệu vào từ tệp đó.

Dữ liệu vào và ra có thể là các kí tự, số, hoặc các byte chứa các chữ số nhị phân.

Trong C++, các dòng vào ra được cài đặt bằng các đối tượng của các lớp dòng vào ra đặc biệt. Ví dụ, `cout` mà ta vẫn dùng để ghi ra màn hình chính là dòng ra chuẩn, còn `cin` là dòng vào chuẩn nối với bàn phím. Cả hai đều là các đối tượng dòng dữ liệu (khái niệm "đối tượng" này có liên quan đến tính năng hướng đối tượng của C++, khi nói về các dòng vào/ra của C++, ta sẽ phải đề cập nhiều đến tính năng này).

8.2. Tệp văn bản và tệp nhị phân

Về bản chất, tất cả dữ liệu trong các tệp đều được lưu trữ dưới dạng một chuỗi các bit nhị phân 0 và 1. Tuy nhiên, trong một số hoàn cảnh, ta không coi nội dung của một tệp là một chuỗi 0 và 1 mà coi tệp đó là một chuỗi các kí tự. Một số tệp được xem như là các chuỗi kí tự và được xử lý bằng các dòng và hàm cho phép chương trình và hệ soạn thảo văn bản của bạn nhìn các chuỗi nhị phân như là các chuỗi kí tự. Chúng được gọi là các **tệp văn bản** (*text file*). Những tệp không phải tệp văn bản là **tệp nhị phân** (*binary file*). Mỗi loại tệp được xử lý bởi các dòng và hàm riêng.

Chương trình C++ của bạn được lưu trữ trong tệp văn bản. Các tệp ảnh và nhạc là các tệp nhị phân. Do tệp văn bản là chuỗi kí tự, chúng thường trông giống nhau tại các máy khác nhau, nên ta có thể chép chúng từ máy này sang máy khác mà không gặp hoặc gặp phải rất ít rắc rối. Nội dung của các tệp nhị phân thường lấy cơ sở là các giá trị số, nên việc sao chép chúng giữa các máy có thể gặp rắc rối do các máy khác nhau có thể dùng các quy cách lưu trữ số không giống nhau. Cấu trúc của một số dạng tệp nhị phân đã được chuẩn hóa để chúng có thể được sử dụng thống nhất tại các platform khác nhau. Nhiều dạng tệp ảnh và âm thanh thuộc diện này.

Mỗi kí tự trong một tệp văn bản được biểu diễn bằng 1 hoặc 2 byte, tùy theo đó là kí tự ASCII hay Unicode. Khi một chương trình viết một giá trị vào một tệp văn bản, các kí tự được ghi ra tệp giống hệt như khi chúng được ghi ra màn hình bằng cách sử dụng `cout`. Ví dụ, hành động viết số 1 vào một tệp sẽ dẫn đến kết quả là 1 kí tự được ghi vào tệp, còn với số 1039582 là 7 kí tự được ghi vào tệp.

Các tệp nhị phân lưu tất cả các giá trị thuộc một kiểu dữ liệu cơ bản theo cùng một cách, giống như cách dữ liệu được lưu trong bộ nhớ máy tính. Ví dụ, mỗi giá trị `int` bất kì, 1 hay 1039582 đều chiếm một chuỗi 4 byte.

8.3. Vào ra tệp

C++ cung cấp các lớp sau để thực hiện nhập và xuất dữ liệu đối với tệp:

- `ofstream`: lớp dành cho các dòng ghi dữ liệu ra tệp
- `ifstream`: lớp dành cho các dòng đọc dữ liệu từ tệp
- `fstream`: lớp dành cho các dòng vừa đọc vừa ghi dữ liệu ra tệp.

Đối tượng thuộc các lớp này do quan hệ thừa kế nên cách sử dụng chúng khá giống với `cin` và `cout` – các đối tượng thuộc lớp `istream` và `ostream` – mà chúng ta đã dùng. Khác biệt chỉ là ở chỗ ta phải nối các dòng đó với các tệp.

```
#include <iostream>
#include <fstream>

using namespace std;

int main ()
{
    ofstream myfile;  // declare an output file stream object

    myfile.open ("hello.txt"); // open a file to write
    myfile << "Hello!\n";      // write a string to the file
    myfile.close();            // close the file

    return 0;
}
```

Hình 8.1: Các thao tác cơ bản với tệp văn bản.

Chương trình trong Hình 8.1 tạo một tệp có tên hello.txt và ghi vào đó một câu "`Hello!`" theo cách mà ta thường làm đối với `cout`, chỉ khác ở chỗ thay `cout` bằng đối tượng dòng `myfile` đã được nối với một tệp. Sau đây là các bước thao tác với tệp.

8.3.1. Mở tệp

Việc đầu tiên là nối đối tượng dòng với một tệp, hay nói cách khác là mở một tệp. Kết quả là đối tượng dòng sẽ đại diện cho tệp, bất kì hoạt động đọc và ghi đối với đối tượng đó sẽ được thực hiện đối với tệp mà nó đại diện. Để mở một tệp từ một đối tượng dòng, ta dùng hàm `open` của nó:

```
open (fileName, mode);
```

Trong đó, `fileName` là một xâu kí tự thuộc loại `const char *` với kết thúc là kí tự `null` (hằng xâu kí tự cũng thuộc dạng này), là tên của tệp cần mở, và `mode` là tham số không bắt buộc và là một tổ hợp của các cờ sau:

`ios::in`	mở để đọc
`ios::out`	mở để ghi
`ios::binary`	mở ở dạng tệp nhị phân
`ios::ate`	đặt ví trí bắt đầu đọc/ghi tại cuối tệp. Nếu cờ này không được đặt giá trị gì, vị trí khởi đầu sẽ là đầu tệp.
`ios::app`	mở để ghi tiếp vào cuối tệp. Cờ này chỉ được dùng cho dòng mở tệp chỉ để ghi.
`ios::trunc`	nếu tệp được mở để ghi đã có từ trước, nội dung cũ sẽ bị xóa để ghi nội dung mới.

Các cờ trên có thể được kết hợp với nhau bằng toán tử bit OR (|). Ví dụ, nếu ta muốn mở tệp people.dat theo dạng nhị phân để ghi bổ sung dữ liệu vào cuối tệp, ta dùng lời gọi hàm sau:

```
ofstream myfile;
myfile.open ("people.dat",
             ios::out | ios::app | ios::binary);
```

Trong trường hợp lời gọi hàm `open` không cung cấp tham số `mode`, chẳng hạn Hình 8.1, chế độ mặc định cho dòng loại `ostream` là `ios::out`, cho dòng loại `istream` là `ios::in`, và cho dòng loại `fstream` là `ios::in | ios::out`.

Cách thứ hai để nối một dòng với một tệp là khai báo tên tệp và kiểu mở tệp ngay khi khai báo dòng, hàm `open` sẽ được gọi với các đối số tương ứng. Ví dụ:

```
ofstream myfile ("hello.txt",
                 ios::out | ios::app | ios::binary);
```

Để kiểm tra xem một tệp có được mở thành công hay không, ta dùng hàm thành viên `is_open()`, hàm này không yêu cầu đối số và trả về một giá trị kiểu `bool` bằng `true` nếu thành công và bằng `false` nếu xảy ra trường hợp ngược lại

```
if (myfile.is_open()) { /* file now open and ready */ }
```

8.3.2. Đóng tệp

Khi ta hoàn thành các công việc đọc dữ liệu và ghi kết quả, ta cần đóng tệp để tài nguyên của nó trở về trạng thái sẵn sàng được sử dụng. Hàm thành viên này

không có tham số, công việc của nó là xả các vùng bộ nhớ có liên quan và đóng tệp:

```
myfile.close();
```

Sau khi tệp được đóng, ta lại có thể dùng dòng `myfile` để mở tệp khác, còn tệp vừa đóng lại có thể được mở bởi các tiến trình khác.

Hàm `close` cũng được gọi tự động khi một đối tượng dòng bị hủy trong khi nó đang nối với một tệp.

8.3.3. Xử lý tệp văn bản

Chế độ dòng tệp văn bản được thiết lập nếu ta không dùng cờ `ios::binary` khi mở tệp. Các thao tác xuất và nhập dữ liệu đối với tệp văn bản được thực hiện tương tự như cách ta làm với `cout` và `cin`.

```
#include <iostream>
#include <fstream>

using namespace std;

int main ()
{
    ofstream courseFile ("courses.txt");
    if (courseFile.is_open())
    {
        courseFile << "1 Introduction to Programming\n";
        courseFile << "2 Mathematics for Computer Science\n";
        courseFile.close();
    }
    else cout << "Error: Cannot open file";

    return 0;
}
```

Kết quả chạy chương trình [tệp courses.txt]

```
1 Introduction to Programming
2 Mathematics for Computer Science
```

Hình 8.2: Ghi dữ liệu ra tệp văn bản.

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main ()
{
    ifstream file ("courses.txt");
    if (file.is_open())
    {
        while (! file.eof())
        {
            string line;
            getline (file,line);
            cout << line << endl;
        }
        file.close();
    }
    else cout << "Error! Cannot open file";

    return 0;
}
```

Kết quả chạy chương trình

```
1 Introduction to Programming
2 Mathematics for Computer Science
```

Hình 8.3: Đọc dữ liệu từ tệp văn bản.

Chương trình ví dụ trong Hình 8.2 ghi hai dòng văn bản vào một tệp. Chương trình trong Hình 8.3 đọc nội dung tệp đó và ghi ra màn hình. Để ý rằng trong chương trình thứ hai, ta dùng một vòng lặp để đọc cho đến cuối tệp. Trong đó, `myfile.eof()` là hàm trả về giá trị `true` khi chạm đến cuối tệp, giá trị `true` mà `myfile.eof()` trả về đã được dùng làm điều kiện kết thúc vòng lặp đọc tệp.

Kiểm tra trạng thái của dòng

Bên cạnh hàm `eof()` có nhiệm vụ kiểm tra cuối tệp, còn có các hàm thành viên khác dùng để kiểm tra trạng thái của dòng:

`bad()`	trả về `true` nếu một thao tác đọc hoặc ghi bị thất bại. Ví dụ khi ta cố viết vào một tệp không được mở để ghi hoặc khi thiết bị lưu trữ không còn chỗ trống để ghi.
`fail()`	trả về `true` trong những trường hợp `bad()` trả về `true` và khi có lỗi định dạng, chẳng hạn như khi ta đang định đọc một số nguyên nhưng lại gặp phải dữ liệu là các chữ cái.
`eof()`	trả về `true` nếu chạm đến cuối tệp
`good()`	trả về `false` nếu xảy tình huống mà một trong các hàm trên nếu được gọi thì sẽ trả về `true`.

Để đặt lại cờ trạng thái mà một hàm thành viên nào đó đã đánh dấu trước đó, ta dùng hàm thành viên `clear`.

Con trỏ get và put của dòng

Mỗi đối tượng dòng vào ra có ít nhất một con trỏ nội bộ. Con trỏ nội bộ của `ifstream` hay `istream` được gọi là **con trỏ get** hay **con trỏ đọc**. Nó chỉ tới vị trí mà thao tác đọc tiếp theo sẽ được thực hiện tại đó. Con trỏ nội bộ của `ofstream` hay `ostream` được gọi là **con trỏ put** hay **con trỏ ghi**. Nó chỉ tới vị trí mà thao tác ghi tiếp theo sẽ được thực hiện tại đó. Cuối cùng, `fstream` có cả con trỏ get và con trỏ put.

Để định vị vị trí hiện tại của các con trỏ get và put, ta có các hàm thành viên `tellg` và `tellp`. Các hàm này trả về một giá trị thuộc kiểu `pos_type`, là kiểu dữ liệu số nguyên biểu diễn vị trí hiện tại (tính từ đầu tệp) của con trỏ get của dòng (nếu gọi hàm `tellg`) hoặc con trỏ put của dòng (nếu gọi hàm `tellp`).

Để đặt lại vị trí của các con trỏ get và put, ta có các hàm `seekg(offset, direction)` và `seekp(offset, direction)` có công dụng di chuyển các con trỏ get và put tới vị trí `offset`. Trong đó, `offset` được tính từ đầu tệp nếu `direction` là `ios::beg` (giá trị mặc định của `direction`), từ cuối tệp nếu `direction` là `ios::end`, và từ vị trí hiện tại nếu direction là `ios::cur`.

Hình 8.4 minh họa cách sử dụng các con trỏ get và put để tính kích thước của một tệp văn bản.

```
#include <iostream>
#include <fstream>

using namespace std;

int main ()
{
    ifstream file ("courses.txt");

    long begin = file.tellg();
    file.seekg (0, ios::end);
    long end = file.tellg();
    file.close();

    cout << "The size is " << (end - begin) << " bytes.\n";

    return 0;
}
```

Kết quả chạy chương trình

```
The size is 65 bytes.
```

Hình 8.4: Dùng con trỏ dòng để xác định kích thước tệp.

8.3.4. Xử lý tệp nhị phân

Để đọc và ghi dữ liệu với tệp nhị phân, ta không thể dùng các toán tử <<, >> và các hàm như `getline` do dữ liệu có định dạng khác và các kí tự trắng không được dùng để tách giữa các phần tử dữ liệu. Thay vào đó, ta dùng hai hàm thành viên được thiết kế riêng cho việc đọc và ghi dữ liệu nhị phân một cách tuần tự là `write` và `read`. Cách dùng như sau:

```
output_stream.write(memory_block, size);
input_stream.read(memory_block, size);
```

Trong đó *memory_block* thuộc loại "con trỏ tới char" (char*), nó đại diện cho địa chỉ của một mảng byte lưu trữ các phần tử dữ liệu đọc được hoặc các phần tử cần được ghi ra dòng. Tham số *size* là một giá trị nguyên xác định số kí tự cần đọc hoặc ghi vào mảng đó.

Các chương trình trong Hình 8.5, Hình 8.6 và Hình 8.7 minh họa việc đọc và ghi dữ liệu kiểu người dùng tự định nghĩa từ tệp nhị phân.

```
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
using namespace std;

#define MAX_NAME_LENGTH 20

struct Student {
    char name [MAX_NAME_LENGTH + 1];
    float score;

    Student () { name[0] = '\0'; score = 0; }
    Student (const char*, float);
    void println() {
        cout << name << "\t" << score << endl;
    }
};

Student::Student (const char* n,float s)
{
    int length = strlen(n);
    if (length > MAX_NAME_LENGTH) length = MAX_NAME_LENGTH;
    strncpy(name, n, length);
    name[length] = '\0'; // mark the end of the string

    score = s;
}

#endif
```

Hình 8.5: Kiểu bản ghi đơn giản Student.

```
#include <iostream>
#include <fstream>
#include <cstring>

#include "student.h"

using namespace std;

int main ()
{
    ofstream myfile ("scores.dat", ios::binary | ios::out);

    if (myfile.is_open())
    {
        Student anne("anne", 10);
        Student julia("julia", 9);
        Student bob("bob", 3);

        myfile.write((char *)(&anne), sizeof(Student));
        myfile.write((char *)(&julia), sizeof(Student));
        myfile.write((char *)(&bob), sizeof(Student));

        myfile.close();
    }
    else cout << "Error: Cannot open file";

    return 0;
}
```

Hình 8.6: Ghi dữ liệu ra tệp nhị phân.

```
#include <iostream>
#include <fstream>
#include <cstring>

#include "student.h"

using namespace std;

int main ()
{
    ifstream myfile ("scores.dat", ios::binary);
    if (myfile.is_open())
    {
        while (! myfile.eof() )
        {
            Student student;
            myfile.read((char *)(&student), sizeof(student));
            if (myfile.good()) student.println();
        }
        myfile.close();
    }
    else cout << "Error! Cannot open file";

    return 0;
}
```

Hình 8.7: Đọc dữ liệu từ tệp nhị phân.

Bài tập

1. Nhập vào từ bàn phím một danh sách sinh viên. Mỗi sinh viên gồm có các thông tin sau đây: tên tuổi, ngày tháng năm sinh, nơi sinh, quê quán, lớp, học lực (từ 0 đến 9). Hãy ghi thông tin về danh sách sinh viên đó ra tệp văn bản student.txt

Sau khi thực hiện bài 1, hãy viết chương trình nhập danh sách sinh viên từ tệp văn bản student.txt rồi hiển thị ra màn hình:

- thông tin về tất cả các bạn tên là Vinh ra tệp văn bản vinh.txt
- thông tin tất cả các bạn quê ở Hà Nội ra tệp văn bản hanoi.txt
- Tổng số bạn có học lực kém (<4), học lực trung bình (≥4 và <8), học lực giỏi (≥8) ra tệp văn bản hocluc.txt.

- Sau khi thực hiện bài 2, hãy viết chương trình cho biết kích thước của tệp văn bản student.txt, vinh.txt, hanoi.txt, hocluc.txt. Kết quả ghi ra tệp văn bản all.txt.
- Viết chương trình kiểm tra xem tệp văn bản student.txt có tồn tại hay không? Nếu tồn tại thì hiện ra màn hình các thông tin sau:
  - Số lượng sinh viên trong tệp
  - Số lượng dòng trong tệp
  - Ghi vào cuối tệp văn bản dòng chữ "CHECKED"
  - Nếu không tồn tại, thì hiện ra màn hình dòng chữ "NOT EXISTED".
- Tệp văn bản number.txt gồm nhiều dòng, mỗi dòng chứa một danh sách các số nguyên hoặc thực. Hai số đứng liền nhau cách nhau ít nhất một dấu cách. Hãy viết chương trình tổng hợp các thông tin sau và ghi vào tệp văn bản info.txt những thông tin sau:
  - Số lượng số trong tệp
  - Số lượng các số nguyên
  - Số lượng các số thực

Lưu ý: Test chương trình với cả trường hợp tệp văn bản number.txt chứa một hay nhiều dòng trắng ở cuối tệp.

- Trình bày sự khác nhau, ưu điểm, nhược điểm giữa tệp văn bản và tệp văn bản nhị phân.

Phụ lục A. Phong cách lập trình

Phần này giới thiệu những điểm cơ bản mà lập trình viên nên làm theo để chương trình dễ đọc, dễ hiểu, và đạt được hiệu quả cao. Về cơ bản, chương trình cần viết tường minh, đơn giản, dễ hiểu. Không nên sử dụng các cấu trúc lệnh phức tạp dễ dẫn đến nhầm lẫn và khó tìm lỗi.

A.1. Chú thích

Trước và trong khi lập trình cần phải ghi chú thích cho các đoạn mã trong chương trình. Việc chú thích giúp chúng ta hiểu một cách rõ ràng và tường minh hơn về công việc chúng ta cần làm. Quan trọng hơn, chúng sẽ giúp chúng ta dễ dàng hiểu khi chúng ta quay lại kiểm tra hoặc tiếp tục làm việc với chương trình. Đặc biệt quan trọng là giúp chúng ta có thể chia sẻ và cùng phát triển chương trình theo nhóm trong một thời gian dài.

Cụ thể là, đối với mỗi hàm, đặc biệt là các hàm quan trọng, chúng ta cần xác định và ghi chú thích về những vấn đề cơ bản sau:

- Mục đích của hàm là gì?
- Biến đầu vào của hàm (tham biến) là gì?
- Các điều kiện rằng buộc của các biến đầu vào nếu có?
- Kết quả trả về của hàm là gì?
- Các rằng buộc của kết quả trả ra nếu có.
- Việc chú thích sẽ giúp chúng hiểu rõ ràng về yêu cầu của hàm.

Ví dụ:

```
// the function calculate the sum of two digits
// input: two integer numbers smaller than 10
// output: an integer number
getSum (int x, int y)
{
    int sum = x + y;
    return sum;
}
```

A.2. Chia nhỏ chương trình

Trong lập trình, người ta thường sử dụng chiến lược chia để trị, tức là chương trình được chia nhỏ ra thành các chương trình con. Việc chia nhỏ ra thành các chương trình con làm tăng tính môđun của chương trình và mang lại cho lập trình viên khả năng tái sử dụng mã.

Người ta khuyên rằng độ dài mỗi chương trình con không nên vượt quá một trang màn hình để lập trình viên có thể kiểm soát tốt hoạt động của chương trình con đó.

A.3. Biến toàn cục

Xu hướng chung là nên hạn chế sử dụng biến toàn cục. Khi nhiều hàm cũng sử dụng một biến toàn cục, việc thay đổi giá trị biến toàn cục của một hàm nào đó có thể dẫn đến những thay đổi không mong muốn ở các hàm khác. Biến toàn cục sẽ làm cho các hàm trong chương trình không độc lập với nhau.

A.4. Cách đặt tên biến

Tên biến nên dễ đọc, và gợi nhớ đến công dụng của biến hay kiểu dữ liệu mà biến sẽ lưu trữ. Đối với những biến gồm nhiều từ, thì các từ nên viết liền nhau và chữ cái đầu tiên của các từ phía sau nên được viết hoa. Ví dụ

```
numberStudents, savingAccount
```

Các biến hằng nên viết hoa. Nếu biến hằng gồm nhiều từ, thì các từ nên chia cách bằng dấu gạch chân. Ví dụ

```
PI, NUMBER_ITERATION
```

A.5. Cách đặt tên hàm

Tên hàm nên dễ đọc, và gợi nhớ đến mục đích của hàm. Tên hàm nên bắt đầu bằng một động từ. Đối với những hàm gồm nhiều từ, thì các từ nên viết liền nhau và chữ cái đầu tiên của các từ phía sau nên được viết hoa. Ví dụ

```
getSum,  getMin, calculateScores
```

A.6. Biểu thức

Các biểu thức cần viết đơn giản, gắn gọn và dễ hiểu. Các biểu thức dài có thể tách nhỏ ra sử dụng các biến trung gian. Nên sử dụng các cặp dấu ngoặc để trách sự nhập nhằng và nhầm lẫn về thứ tự thực hiện các phép toán trong biểu thức. Ví dụ:

```
sum = a * b + a * b * c – d/e
```

Có thể viết thành

```
ab = a * b;
sum = ab * (c + 1) – (d/e);
```

Lưu ý: Biểu thức điều kiện cũng nên tuân thủ theo nguyên tắc trên.

A.7. Vòng lặp

Không nên sử dụng nhiều các lệnh nhảy như `break`, hay `continue` để thoát ra khỏi vòng lặp. Với mỗi vòng lặp, nên xác định rõ ràng điều kiện để vòng lặp kết thúc.

A.8. Khối chương trình

Các đoạn chương trình cần được trình bày theo thành từng khối (sử dụng các dấu cách). Việc trình bày theo khối sẽ giúp chúng ta dễ dàng hiểu được cấu trúc và thứ tự thực hiện các lệnh. Ví dụ:

```
while (!done)
{
    doSomething ();
    done = check ();
}
```

A.9. Lớp

Mỗi lớp (class) nên tách ra thành hai tệp riêng biệt. Tệp header (.h) chứa khai báo về lớp, còn tệp nguồn (.cpp) chứa định nghĩa về các phương thức của lớp. Việc tách này sẽ dễ dàng cho chúng ta trong việc theo dõi, tìm hiểu và phát triển chương trình. Ví dụ:

```
time.h,  time.cpp
```

Phụ lục B. Dịch chương trình C++ bằng GNU C++

B.1. Dịch và liên kết

Quá trình tạo tệp thực thi được từ các tệp mã nguồn có hai bước.

Bước 1: biên dịch các tệp mã nguồn thành các tệp mã object. Các tệp có mở rộng .cpp hoặc .cc được dịch thành các tệp có mở rộng .o.

Bước 2: liên kết các tệp mã object thành một tệp thực thi được. Các tệp có mở rộng .o được liên kết với nhau thành một tập thực thi được (đuôi .exe trong môi trường DOS/Windows).

Nếu chương trình chỉ gồm một tệp mã nguồn, ví dụ count.cpp, ta chỉ cần chạy lệnh

g++ -o count.exe count.cpp

Kết quả là một tệp thực thi được có tên count.exe. Hoặc ta chỉ dùng lệnh

g++ count.cpp

Kết quả là một tệp thực thi được có tên mặc định (a.out trong môi trường Unix/Linux).

Nếu chương trình bao gồm nhiều tệp mã nguồn, chẳng hạn time_test.cpp (chứa hàm main), time.cpp, time.h, ta dịch từng tệp có đuôi cpp hoặc cc bằng lệnh sau

g++ -c file.cpp

Sau đó liên kết lại bằng lệnh

g++ time_test.o time.o -o time_test.exe

B.2. Tiện ích make của GNU

`make` là một hệ thống được thiết kế để xây dựng các chương trình từ các cây mã nguồn lớn. Từ đặc tả của lập trình viên về cây mã nguồn, tiện ích make sẽ gọi trình biên dịch và liên kết một cách hiệu quả nhất để xây dựng chương trình thực thi được. Để dùng tiện ích `make` cho một chương trình, lập trình viên phải tạo một tệp có tên `Makefile` nằm trong cùng thư mục với các tệp mã nguồn. Tệp này chứa các lệnh hướng dẫn make làm gì và làm thế nào để xây dựng được chương trình.

`Makefile` gồm các bộ đặc tả, mỗi bộ quản lý việc cập nhật một tệp.

Mỗi bộ đặc tả của một tệp `Makefile` gồm 3 phần: đích (*target*), các quan hệ phụ thuộc (*dependency*), và các chỉ thị (*instruction*). Công thức như sau:

`TargetFile: DependencyFile`$_1$ `DependencyFile`$_2$ `... DependencyFile`$_n$

trong đó, `TargetFile` là tệp cần cập nhật và mỗi `DependencyFile`$_i$ là một tệp mà `TargetFile` phụ thuộc vào nó. Dòng thứ hai của bộ đặc tả là một lệnh dịch `TargetFile`, lệnh này phải có một kí tự TAB đứng trước và kết thúc bằng kí tự xuống dòng. Ví dụ, bộ đặc tả đầu tiên trong `Makefile` của chúng ta như sau:

```
time_test.exe: time.o time_test.o
     g++ time.o time_test.o -o time_test.exe
```

trong đó dòng đầu chỉ ra rằng time_test.exe phụ thuộc hai tệp time.o và time_test.o, dòng thứ hai là lệnh dùng g++ dịch ra tệp time_test.exe.

Tiếp theo, time.o không có sẵn ngay khi biên dịch lần đầu, cho nên ta phải viết bộ đặc tả cho tệp này:

```
time.o: time.cc time.h
     g++ -c time.cc
```

Tương tự là bộ đặc tả cho time_test.o

```
time_test.o: time.cc time.h
     g++ -c time_test.cc
```

Do đó, Makefile sẽ có dạng:

```
time_test: time.o time_test.o
     g++ time.o time_test.o -o time_test

time.o: time.cc time.h
     g++ -c time.cc

time_test.o: time.cc time.h
     g++ -c time_test.cc
```

Và khi ta gõ từ dấu nhắc dòng lệnh

```
make
```

chương trình `make` sẽ đọc `Makefile` và thực hiện các công việc sau:

1. thấy rằng time_test.exe phụ thuộc vào time.o, và:
    a) Kiểm tra time.o, thấy nó phụ thuộc time.cc và time.h;

b) Xác định xem time.o đã lỗi thời chưa (cũ hơn bản mới nhất của các tệp time.cc và time.h)

c) nếu đã lỗi thời thì chạy lệnh `g++ -c time.cc` để tạo `time.o`

2. thấy rằng time_test.exe cũng phụ thuộc vào time_test.o, và

   a) Kiểm tra time_test.o, thấy nó phụ thuộc time_test.cc và time.h;

   b) Xác định xem time_test.o đã lỗi thời chưa (chưa được dịch từ bản mới nhất của các tệp time_test.cc và time.h)

   c) nếu đã lỗi thời thì chạy lệnh `g++ -c time_test.cc` để tạo time_test.o

3. thấy rằng tất cả các tệp mà time_test.exe phụ thuộc đều đã được cập nhật, nên nó chạy lệnh `g++ time.o time_test.o -o time_test` để tạo chương trình time_test.exe.

Tài liệu tham khảo

[1]. Bjarne Stroustrup, *The C++ Programming Language*, 3rd edition, Addison-Wesley, 1997.

[2]. Deitel & Deitel, *C++ How to Program*, 5th edition, Prentice Hall, 2005.

[3]. ISO/IEC JTC1/SC22/WG21. Stable release, *ISO/IEC 14882:2003* (2003)

[4]. Juan Soulie, *C++ Language Tutorial*, http://www.cplusplus.com.