

NGUYỄN VIỆT HƯƠNG

NGÔN NGỮ LẬP TRÌNH C++ VÀ CẤU TRÚC DỮ LIỆU



NHÀ XUẤT BẢN GIÁO DỤC

TS. NGUYỄN VIỆT HƯƠNG

NGÔN NGỮ LẬP TRÌNH C++ VÀ CẤU TRÚC DỮ LIỆU

(Tái bản lần thứ tư)

NHÀ XUẤT BẢN GIÁO DỤC

Bản quyền thuộc HEVOBCO – Nhà xuất bản Giáo dục.

04 – 2008/CXB/139 – 1999/GD

Mã số: 7B558y8 – DAI

LỜI NÓI ĐẦU

Tư duy lập trình trong những năm gần đây đã có sự biến chuyển lớn. Đó là sự ra đời của phương pháp lập trình hướng đối tượng. Với phương pháp lập trình này, những người lập trình không còn lo ngại trước những chương trình lớn và phức tạp. Lập trình hướng đối tượng cho ta một cách tổ chức các chương trình. Nó làm việc với toàn bộ tổ chức của chương trình chứ không quan tâm tới các chi tiết của hoạt động lập trình. Bởi vậy, những chương trình phức tạp trở nên đơn giản, dễ hiểu. Trong khi những phương pháp lập trình trước đây, vì chỉ quan tâm đến những chi tiết lập trình nên đã bế tắc và thậm chí thất bại trước những vấn đề phức tạp. Ngoài ra, lập trình hướng đối tượng rất gần với thế giới thực. Có thể nói là nó cho phép ta mô hình hóa thế giới thực vào trong chương trình vì nó đối xử với các đối tượng chương trình như các thực thể của thế giới thực. Chính sự gần gũi này làm chương trình rõ ràng hơn, dễ hiểu hơn và dễ quản lý hơn.

Năm 1980 tại Bell Laboratories ở Murry Hill, bang New Jersey của Mỹ, Bjarne Stroustrup đã đưa ra phiên bản đầu tiên của C++ dưới tên "C có lớp" vì C++ dựa trên cơ sở của C. Năm 1983 tên này được đổi thành C++. Ngôn ngữ lập trình này đã trở thành ngôn ngữ lập trình hướng đối tượng nổi bật trong những năm 1990 vì nó là thể hiện của phương pháp luận hướng đối tượng. Ngoài ra, nó còn là sự thể hiện của nhiều ý tưởng độc đáo khác giúp tạo ra những chương trình linh hoạt, bền vững và dễ mở rộng. Việc phát triển C++ vẫn còn là "công việc đang tiến hành" nghĩa là vẫn còn có những đổi mới trong tương lai. Tuy nhiên, cuối cùng, ngôn ngữ C++ sẽ thay thế vai trò thống lĩnh của ngôn ngữ C.

Vì những lý do nêu trên, giáo trình này sẽ trình bày những ý tưởng của ngôn ngữ lập trình C++ đồng thời với hướng dẫn cách ứng dụng những điều vừa học thông qua các ví dụ. Phương pháp này đảm bảo cho người đọc hiểu đầy đủ từ ý tưởng này sang ý tưởng khác. Ở phần hai, giáo trình này còn trình bày cách cài đặt một số cấu trúc dữ liệu cơ bản bằng ngôn ngữ lập trình C++. Vì thời gian có hạn, phần hai chưa trình bày được đầy đủ như tác giả mong muốn.

Đây là giáo trình giảng dạy của khoa Điện tử – Viễn thông, trường Đại học Bách khoa Hà Nội và là tài liệu tham khảo cho sinh viên các trường Đại học kỹ thuật và Đại học khoa học tự nhiên cũng như kỹ sư các ngành.

Tác giả chân thành cảm ơn những người đã đóng góp cho quá trình soạn thảo giáo trình này và mọi ý kiến góp ý để giáo trình sẽ ngày càng được hoàn thiện hơn, xin gửi về địa chỉ : khoa Điện tử – Viễn thông, trường Đại học Bách khoa Hà Nội.

Tác giả

PHẦN I

NGÔN NGỮ LẬP TRÌNH C++

CHƯƠNG 1

CÁI NHÌN ĐẦU TIÊN VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VÀ C++

1.1. TẠI SAO CHÚNG TA CẦN LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG ?

Lập trình hướng đối tượng (Object Oriented Programming - OOP) đã được phát triển bởi vì những cách lập trình trước đây bộc lộ nhiều hạn chế. Để hiểu được những gì mà OOP làm, chúng ta cần biết những hạn chế này là gì và chúng phát sinh từ các ngôn ngữ truyền thống như thế nào.

1.1.1. Các ngôn ngữ thủ tục

Pascal, C, BASIC, Fortran và các ngôn ngữ lập trình truyền thống tương tự là các ngôn ngữ thủ tục (**procedural languages**). Nghĩa là, mỗi câu lệnh trong ngôn ngữ bảo máy làm một việc gì đó chẳng hạn như nhận dữ liệu nhập vào, cộng các số, đưa kết quả ra màn hình v.v... Một chương trình trong ngôn ngữ thủ tục là một danh sách các lệnh.

Vì các chương trình rất nhỏ nên không cần nguyên tắc tổ chức (thường gọi là mô hình) nào khác một danh sách các lệnh. Người lập trình tạo danh sách các lệnh và máy tính thực hiện chúng.

1.1.2. Chia thành các hàm

Khi chương trình trở nên quá lớn thì một danh sách các lệnh trở thành công kênh. Người lập trình chỉ có thể hiểu được một chương trình dài hơn trăm lệnh khi nó được tách thành các đơn vị nhỏ hơn. Vì lý do này, hàm (function) được chấp nhận như một cách làm cho các chương trình dễ hiểu hơn đối với những người tạo ra chúng, (thuật ngữ hàm được sử dụng trong C và C++, trong các ngôn ngữ khác khái niệm này có thể gọi là chương trình con, thủ tục). Một chương trình được chia thành các hàm và mỗi hàm có một chức năng được định nghĩa rõ ràng và một giao diện với các hàm khác trong chương trình cũng được định nghĩa rõ ràng.

Ý tưởng chia một chương trình thành các hàm có thể được mở rộng thành một nhóm các hàm gọi là một module, nhưng về nguyên tắc không khác vì các lệnh thực hiện các nhiệm vụ xác định.

1.1.3. Những vấn đề tồn tại với lập trình có cấu trúc

Khi các chương trình trở nên lớn và phức tạp hơn thì ngay cả lập trình có cấu trúc cũng bắt đầu bộc lộ những dấu hiệu khó khăn. Có thể chúng ta đã nghe những chuyện này sinh về việc viết chương trình của các công ty phần mềm. Để án quá phức tạp, nhờ kế hoạch, thêm nhiều người lập trình, sự phức tạp tăng lên, giá tăng vọt và thảm họa từ đó mà ra.

Những thất bại này làm cho ngôn ngữ thủ tục tự bộc lộ những điểm yếu của nó. Dù có cài đặt cách lập trình có cấu trúc tốt như thế nào đi nữa thì các chương trình lớn cũng trở nên quá phức tạp.

Nguyên nhân nào dẫn đến thất bại này của ngôn ngữ thủ tục? Một trong những lý do quyết định là vai trò của dữ liệu.

1. Dữ liệu không được coi trọng

Ngôn ngữ thủ tục chỉ quan tâm đến làm một việc gì đó, chẳng hạn đọc vào từ bàn phím, đưa ra màn hình, kiểm tra để tìm lỗi v.v... Việc chia nhỏ thành các hàm vẫn không ngoài mục đích đó. Những gì chúng làm có thể phức tạp hơn, trừu tượng hơn nhưng vẫn chỉ nhằm vào hành động.

Chuyện gì xảy ra với dữ liệu trong mô hình này? Dữ liệu, xét cho cùng, là lý do cho sự tồn tại của một chương trình. Phần quan trọng của một chương trình kiểm kê không phải là một hàm hiển thị dữ liệu hay một hàm kiểm tra việc nhập vào sai mà chính là dữ liệu của nó. Nhưng dữ liệu trong các ngôn ngữ thủ tục lại bị đặt ở cấp thứ hai trong tổ chức của chương trình.

Ví dụ, trong một chương trình kiểm kê, dữ liệu tạo nên bản kiểm kê có thể được đọc từ đĩa vào bộ nhớ, ở đó nó được đối xử như các biến toàn cục. Toàn cục có nghĩa là các biến tạo thành dữ liệu được khai báo ở ngoài bất kỳ hàm nào để chúng có thể truy nhập tới tất cả các hàm. Các hàm này thực hiện nhiều thao tác khác nhau trên dữ liệu đó. Chúng đọc, phân tích, cập nhật, sắp xếp, hiển thị và ghi nó trở lại đĩa v.v...

Trong hầu hết các ngôn ngữ như Pascal và C, cũng có các biến cục bộ, nó ẩn bên trong một hàm. Nhưng các biến cục bộ không thể dùng cho các dữ liệu quan trọng yêu cầu được truy nhập bởi nhiều hàm khác nhau.

Giả sử một người lập trình mới được thuê viết một hàm phân tích dữ liệu kiểm kê này theo một cách nhất định. Do không quen với sự khôn khéo trong lập trình, người lập trình đó tạo ra một hàm vô tình làm hỏng dữ liệu. Điều này rất dễ xảy ra bởi vì tất cả các hàm đều có quyền truy nhập tới cùng một dữ liệu.

Một vấn đề khác là vì nhiều hàm truy nhập tới cùng một dữ liệu nên cách lưu trữ dữ liệu trở thành một vấn đề lớn. Việc sắp đặt dữ liệu không thể thay đổi nếu không thay đổi các hàm truy nhập nó. Ví dụ, nếu thêm vào các mục dữ liệu mới thì cần phải thay đổi tất cả các hàm truy nhập dữ liệu đó để chúng cũng có thể truy nhập các mục mới này. Thật khó tìm tất cả các hàm như vậy và thậm chí thay đổi tất cả chúng sao cho đúng còn khó hơn.

Những gì cần là một cách hạn chế truy nhập tới dữ liệu, ẩn nó đối với tất cả các hàm trừ các hàm tối cần thiết. Điều này sẽ bảo vệ dữ liệu, đơn giản hóa việc bảo quản và cho nhiều lợi ích khác.

2. Môi quan hệ với thế giới thực

Các ngôn ngữ thủ tục thường khó cho việc thiết kế. Vấn đề là các thành phần của chúng - các hàm và các cấu trúc dữ liệu - rất khó mô hình hóa được thế giới thực. Ví dụ, cho rằng chúng ta đang viết chương trình để tạo một giao diện người sử dụng đồ họa: các menu, các cửa sổ v.v... Ngay bây giờ, chúng ta cần các hàm nào, cần cấu trúc dữ liệu nào? Câu trả lời không rõ ràng. Sẽ tốt hơn nếu các cửa sổ và các menu tương ứng với các phần tử chương trình.

3. Các kiểu dữ liệu mới

Có các vấn đề khác với các ngôn ngữ truyền thống. Một vấn đề trong đó là tạo các kiểu dữ liệu mới. Các ngôn ngữ máy tính điển hình có sẵn một vài kiểu dữ liệu: kiểu số nguyên, kiểu dấu phẩy động, kiểu ký tự v.v... Làm gì nếu muốn tạo ra kiểu dữ liệu mới cho riêng mình? Có thể chúng ta muốn làm việc với các số phức, hoặc các tọa độ hai chiều, hoặc các giá trị này. Khả năng tạo các kiểu dữ liệu của người sử dụng được gọi là sự mở rộng (extensibility) bởi vì nó cho phép mở rộng khả năng của ngôn ngữ. Các ngôn ngữ truyền thống thường không thể mở rộng.

1.1.4. Cách tiếp cận hướng đối tượng

Ý cơ bản đằng sau các ngôn ngữ hướng đối tượng là kết nối vào một thực thể chương trình cả dữ liệu và các hàm thao tác trên dữ liệu đó. Một thực thể như vậy được gọi là một đối tượng (object).

Các hàm của một đối tượng được gọi là các hàm thành viên trong C++ bởi vì chúng thuộc một lớp đối tượng cụ thể, điển hình để cung cấp cách duy nhất nhằm truy nhập dữ liệu của nó. Nếu

chúng ta muốn đọc một mục dữ liệu trong một đối tượng thì phải gọi một hàm thành viên trong đối tượng đó. Nó sẽ đọc mục dữ liệu đó và trả về giá trị cho ta. Chúng ta không thể truy nhập trực tiếp dữ liệu. Dữ liệu được ẩn đi, để nó an toàn tránh những thay đổi ngẫu nhiên. Dữ liệu và các hàm của nó được bao bọc vào trong một thực thể. Sự bao bọc và cất giấu dữ liệu là các thuật ngữ chính trong việc mô tả các ngôn ngữ hướng đối tượng.

Nếu chúng ta muốn thay đổi dữ liệu trong một đối tượng thì chúng ta phải biết chính xác hàm nào tương tác với nó; tức là các hàm thành viên trong đối tượng đó. Không có hàm nào khác có thể truy nhập dữ liệu. Điều này giúp cho đơn giản hóa việc viết, gỡ rối và bảo quản chương trình.

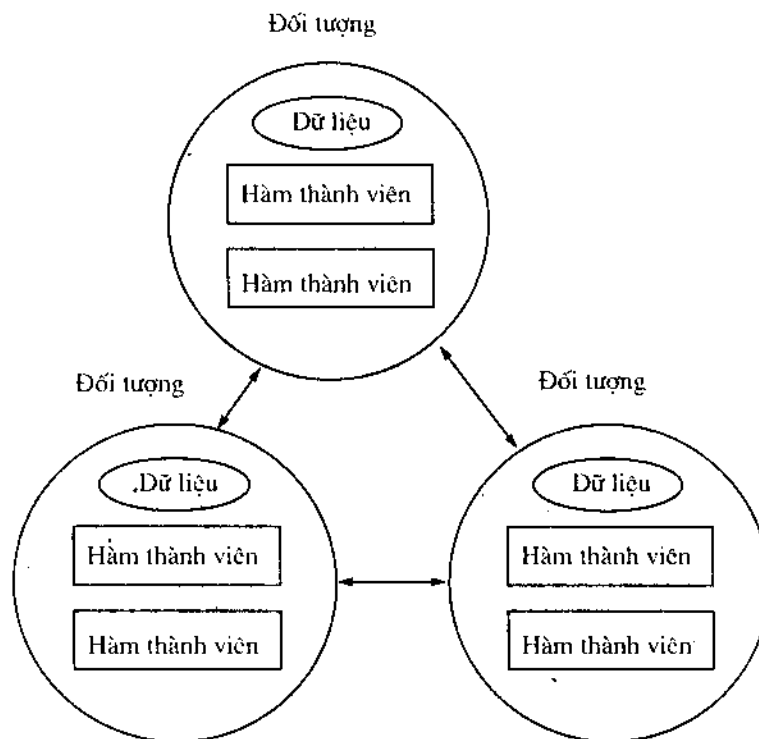
Một chương trình C++ điển hình bao gồm một số đối tượng liên lạc với các đối tượng khác bằng việc gọi một trong các hàm thành viên của nó. Hình 1-1 cho thấy tổ chức của một chương trình C++.

Lưu ý là những gì gọi là các hàm thành viên trong C++ thì trong các ngôn ngữ khác gọi là các phương pháp (methods), ví dụ như ngôn ngữ SMALLTALK, một ngôn ngữ lập trình hướng đối tượng đầu tiên. Cũng vậy, các mục dữ liệu có thể được gọi là các biến cụ thể. Gọi hàm thành viên của một đối tượng thường được coi là gửi thông báo tới đối tượng đó. Những thuật ngữ này được những người viết C++ sử dụng.

1.2. ĐẶC ĐIỂM CỦA NGÔN NGỮ HƯỚNG ĐỐI TƯỢNG

1.2.1. Các lớp

Trong OOP, các đối tượng là cụ thể của các lớp. Hầu hết các ngôn ngữ máy tính đều có sẵn các kiểu dữ liệu. Ví dụ, kiểu int (là một kiểu số nguyên) được định nghĩa trước trong C++. Bởi vậy, trong chương trình chúng ta có thể khai báo bao nhiêu biến kiểu int cũng được. Ví dụ:



Hình 1-1. Tổ chức hướng đối tượng.

```
int day;  
int count;  
int divisor;  
int answer;
```

Tương tự như vậy chúng ta có thể định nghĩa nhiều đối tượng của cùng một lớp. Một lớp đóng vai trò như một bản thiết kế hoặc một mẫu. Nó xác định dữ liệu gì và các hàm gì có trong các đối tượng của lớp đó. Định nghĩa lớp không tạo ra bất kỳ đối tượng nào, y như chỉ tồn tại kiểu int, sẽ không tạo ra bất kỳ biến int nào.

Một lớp chỉ là một mô tả của một số đối tượng tương tự nhau. Một đối tượng là một cụ thể của một lớp bởi vì một đối tượng là một ví dụ "thực" của một mô tả được cung cấp bởi lớp. Vì vậy, có một tên thông dụng được dùng cho dữ liệu của một đối tượng là dữ liệu cụ thể. Nó có tên này bởi vì dữ liệu là riêng cho từng đối tượng.

1.2.2. Sự kế thừa

Ý tưởng về các lớp dẫn đến ý tưởng về sự kế thừa. Trong cuộc sống hàng ngày, chúng ta sử dụng khái niệm lớp được chia thành các lớp con. Lớp động vật được chia thành cá, chim, động vật có vú... Lớp xe được chia thành xe con, xe buýt, xe tải, xe máy...

Nguyên tắc phân chia là mỗi lớp con có các đặc điểm chung với lớp mà nó rút ra: xe con, xe tải, xe buýt và xe máy, tất cả đều có tay lái, động cơ, được dùng để vận chuyển người và hàng hóa. Đây là các đặc điểm của xe. Ngoài các đặc điểm này, mỗi lớp con còn có các đặc điểm cụ thể của riêng nó. Ví dụ: các xe buýt có chỗ ngồi cho nhiều người, trái lại, xe tải có thùng xe để chở hàng hóa.

Cũng như vậy, một lớp OOP có thể được dùng như lớp cơ sở cho một hoặc nhiều lớp con khác. Trong C++, lớp đầu tiên được gọi là lớp cơ sở, các lớp khác được định nghĩa là sử dụng các đặc điểm của lớp cơ sở, nhưng thêm các đặc điểm riêng của nó được gọi là các lớp dẫn xuất.

Sự kế thừa hơi giống việc dùng các hàm để đơn giản hóa một chương trình thủ tục truyền thống. Nếu chúng ta thấy ba phần khác nhau của một chương trình thủ tục hầu như cùng làm một công việc thì chúng ta có thể rút ra các phần tử chung của ba phần này và đặt chúng vào một hàm. Ba phần của chương trình có thể gọi hàm này để thực hiện những hành động chung. Tương tự như vậy, một lớp cơ sở chứa các phần tử chung cho một nhóm các lớp dẫn xuất. Như các hàm trong một chương trình thủ tục, sự kế thừa làm ngắn chương trình hướng đối tượng và làm rõ mối quan hệ giữa các phần tử chương trình.

1.2.3. Sự sử dụng lại

Khi một lớp được viết, được gỡ rối, nó có thể được phân phối đến những người lập trình khác nhau để sử dụng trong các chương trình của họ. Điều này gọi là sự sử dụng lại (reusability). Nó tương tự như cách mà một thư viện hàm trong một ngôn ngữ thủ tục có thể được sáp nhập vào các chương trình khác nhau.

Tuy nhiên, trong OOP, khái niệm kế thừa (inheritance) cung cấp một sự mở rộng quan trọng đối với ý tưởng sử dụng lại. Một người lập trình có thể lấy một lớp đã có, không cần thay đổi nó, thêm vào một số đặc điểm và khả năng khác cho nó để tạo ra một lớp mới cho riêng mình. Việc này được thực hiện bằng việc rút một lớp mới. Lớp mới sẽ kế thừa tất cả những khả năng của lớp cũ, ngoài ra còn có những đặc điểm mới của riêng nó.

Các phần mềm có sẵn được sử dụng lại là một lợi ích cơ bản của OOP.

1.2.4. Tạo các kiểu dữ liệu mới

Một trong những lợi ích của các đối tượng là chúng cho người lập trình một cách thuận tiện để tạo ra các kiểu dữ liệu mới. Cho rằng chúng ta phải làm việc với các vị trí hai chiều (chẳng hạn các

tọa độ x và y) trong chương trình. Chúng ta muốn biểu diễn các phép toán trên các giá trị vị trí này với các phép toán số học thông thường như:

$$Vt1 = Vt2 + Vt0;$$

Ở đây các biến $Vt1$, $Vt2$ và $Vt0$ biểu diễn các cặp số độc lập. Bằng cách tạo một lớp hợp nhất hai giá trị này và khai báo $Vt1$, $Vt2$, $Vt0$ là các đối tượng của lớp này, chúng ta có thể tạo một kiểu dữ liệu mới.

1.2.5. Sự đa hình thái và chồng tên

Chú ý rằng các phép toán $+$ và $=$ được dùng trong phép toán vị trí ở trên không giống như cách chúng làm với các kiểu có sẵn, chẳng hạn như `int`. Các đối tượng $Vt1$, $Vt2$, $Vt0$ không được định nghĩa trước trong C^{++} mà là các đối tượng được người lập trình định nghĩa thuộc lớp **Vitri**. Các toán tử $+$ và $=$ làm thế nào biết được cách tính toán trên các đối tượng đó? Chúng ta phải định nghĩa các phép toán mới cho các đối tượng này.

Sử dụng các toán tử và các hàm theo các cách khác nhau, tùy thuộc vào những gì chúng tính toán, được gọi là sự đa hình thái (polymorphism and overloading - một thứ có nhiều dạng khác nhau). Khi một toán tử đã tồn tại, chẳng hạn $+$ hoặc $=$ được cho khả năng tính toán trên các kiểu dữ liệu thêm vào, ta nói là nó được chồng. Các hàm được chồng khi nhiều hàm có cùng tên nhưng các đối số khác nhau. Sự chồng hàm làm cho chương trình dễ viết và dễ hiểu.

1.2.6. C^{++} và C

C^{++} được rút ra từ ngôn ngữ C. Nói đúng ra, nó là một sự phát triển cao của C. Hầu hết các câu lệnh trong C dùng được trong C^{++} , mặc dù đảo ngược lại không đúng. Các phần tử quan trọng nhất thêm vào C để tạo thành C^{++} liên quan đến lớp, đối tượng và lập trình hướng đối tượng. Bởi vậy, ban đầu C^{++} được gọi là "C có các lớp". Tuy nhiên, C^{++} cũng có nhiều đặc điểm khác tốt hơn C rất nhiều, như cách vào ra cout, cin...

CHƯƠNG 2

SỰ KẾ THỪA

Sự kế thừa là khái niệm trung tâm thứ hai trong lập trình hướng đối tượng (OOP) sau khái niệm lớp. Trên thực tế, mặc dù khái niệm về sự kế thừa chỉ là phụ đối với ý tưởng về các lớp song nó rất mạnh trong OOP. Vì sao vậy? Bởi vì sự kế thừa cho phép sử dụng lại. Sự sử dụng lại có nghĩa là lấy một lớp đã có sẵn và sử dụng nó trong một chương trình mới. Bằng việc sử dụng lại các lớp, ta có thể giảm được thời gian và sức lực cần thiết để viết một chương trình và làm cho phần mềm trở nên mạnh và tin cậy hơn. Nếu viết phần mềm để kinh doanh thì không thể bỏ qua những tiết kiệm có được từ sử dụng lại.

Trong chương này, đầu tiên chúng ta sẽ nói về những khái niệm cơ bản của sự kế thừa, sau đó sẽ đi vào những chi tiết khó xử lý bao gồm cách quản lý hàm tạo (constructor) trong sự kế thừa và sự kế thừa bội.

2.1. GIỚI THIỆU VỀ SỰ KẾ THỪA

Trong phần này chúng ta sẽ bắt đầu bằng việc xem xét hai động cơ chủ yếu dẫn đến sự kế thừa: trợ giúp việc sử dụng lại và cung cấp một cơ sở mới cho việc tổ chức chương trình. Chúng ta sẽ kết thúc phần này bằng việc giới thiệu cú pháp tạo một lớp kế thừa từ lớp khác.

2.1.1. Sự sử dụng lại

Để hiểu tại sao sự kế thừa là quan trọng cần nhìn lại lịch sử của sự kế thừa. Những người lập trình đã luôn luôn cố gắng tránh viết lại cùng một mã lệnh hai lần. Sự kế thừa là giải pháp mới nhất cho vấn đề này.

1. Viết lại mã

Cách đầu tiên để sử dụng lại chỉ đơn giản là viết lại mã lệnh đã có. Giả sử chúng ta có vài mã lệnh làm việc trong một chương trình cũ nhưng không phải làm những gì mà chúng ta muốn trong một chương trình mới. Copy mã lệnh cũ vào file nguồn mới, sửa đổi một chút để chấp nhận chúng trong môi trường mới, rồi sau đó chạy chương trình. Trong tất cả các khả năng đều sinh ra lỗi mới. Chúng ta phải gỡ rối lại tất cả các mã lệnh. Và chúng ta thấy tiếc vì đã không viết ngay mã lệnh mới.

2. Thư viện hàm

Để giảm các lỗi sinh ra bởi sự thay đổi mã lệnh, những người lập trình đã cố tạo ra các phần tử chương trình tự túc dưới dạng các hàm. Hy vọng là các hàm đó đủ tổng quát để có thể sử dụng mà không cần thay đổi trong nhiều tình huống lập trình. Các công ty phần mềm đã gộp các nhóm hàm như vậy vào với nhau, đưa vào các thư viện hàm và bán cho những người lập trình khác.

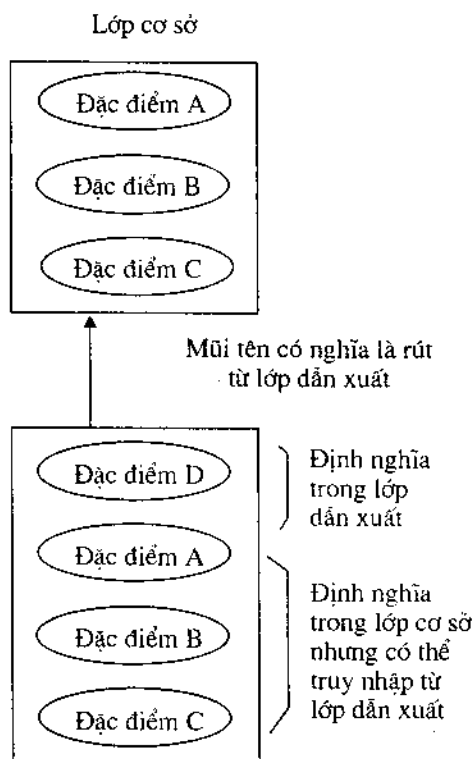
Các thư viện hàm là một bước đi đúng hướng nhưng các hàm lại không mô hình hóa được thế giới thực bởi vì chúng không chứa dữ liệu quan trọng. Thông thường tất cả các hàm đều yêu cầu thay đổi để làm việc được trong một môi trường mới, tất cả các công ty cung cấp các thư viện hàm đều cung cấp các mã nguồn làm cho việc thay đổi như vậy dễ dàng hơn, nhưng sự thay đổi đó lại sinh ra lỗi.

3. Thư viện lớp

Một cách mới rất mạnh cho việc sử dụng lại trong OOP là thư viện lớp, bởi vì một lớp mô hình hóa một thực thể thế giới thực gần gũi hơn, nó cần ít sự thay đổi hơn là hàm để thích ứng trong một tình huống mới. Quan trọng hơn, OOP cung cấp một cách để thay đổi một lớp mà không cần thay đổi mã lệnh của nó. Điều dường như không thể này có được bằng cách dùng sự kế thừa để rút ra một lớp mới từ một lớp cũ. Lớp cũ (gọi là lớp cơ sở) không bị thay đổi nhưng lớp mới (gọi là lớp dẫn xuất) có thể sử dụng cả những đặc điểm của lớp cũ và những đặc điểm của riêng nó. Hình 2-1 minh họa điều này.

2.1.2. Sự kế thừa và thiết kế chương trình

Ngoài việc làm cho việc thay đổi các chương trình cổ sẵn dễ dàng hơn, sự kế thừa còn có một lợi ích khác. Nó cung cấp một cách để liên kết một thành phần chương trình này với một thành phần chương trình khác. Mỗi quan hệ mới này làm cho việc thiết kế chương trình linh động hơn và cấu trúc chương trình phản ánh mối quan hệ thế giới thực chính xác hơn. Sự kế thừa đôi khi còn gọi là quan hệ "loại". Để thấy được ý nghĩa này, đầu tiên chúng ta cùng xét một loại quan hệ khác là quan hệ hợp thành.



Hình 2-1. Sự kế thừa.

1. Hợp thành: Một quan hệ "có"

Nếu có một lớp employee (nhân viên) và một trong những mục dữ liệu trong lớp này là tên của nhân viên thì chúng ta có thể nói rằng đối tượng employee "có" tên. Đối tượng employee cũng có thể "có" lương, mã nhân viên... Loại quan hệ này gọi là sự hợp thành bởi vì đối tượng employee là hợp của các biến trên.

Dữ liệu thành viên của lớp có thể chứa đối tượng của các lớp khác cũng như các biến của các kiểu dữ liệu cơ bản. Chúng ta có thể hình dung một lớp xe đạp bao gồm đối tượng khung, hai đối tượng bánh và một đối tượng tay lái.

Sự hợp thành trong OOP mô hình hóa các tình huống thế giới thực trong đó các đối tượng là hợp thành của các đối tượng khác.

2. Sự kế thừa: Một quan hệ "loại"

Sự kế thừa trong OOP phản ánh khái niệm mà chúng ta gọi là sự khái quát hóa trong thế giới thực. Nếu chúng ta có một chiếc xe đạp đua, một chiếc xe đạp địa hình và một chiếc xe đạp thiếu nhi thì chúng ta có thể nói những xe đạp này là cụ thể rõ ràng của một khái niệm tổng quát hơn gọi là xe đạp. Tất cả các loại xe đạp đều có đặc điểm: hai bánh, một khung... Nhưng một xe đạp đua ngoài các đặc điểm chung này còn có đặc điểm là lớp nhỏ và nhẹ. Một xe đạp địa hình cũng có tất cả những đặc điểm của một xe đạp, ngoài ra còn có lớp to, dày và phanh tốt.

Tương tự, có thể có công nhân, người quản lý, nhà khoa học và thư ký trên bảng lương của một công ty. Những người này là cụ thể rõ ràng của một loại tổng quát hơn là nhân viên.

Hàng ngày chúng ta dùng sự khái quát hóa để nhận thức thế giới thực. Chúng ta biết rằng mèo, cừu và người là các loài động vật có vú; xe tải, xe con và xe buýt là các loại xe; PC và Macintoshes là các loại máy tính. Loại quan hệ này rất thông dụng trong đời sống hàng ngày đến nỗi mà rất có ý nghĩa để mô hình hóa chúng trong chương trình máy tính. Đó là những gì mà sự kế thừa làm.

3. Không hoàn toàn là một cây gia tộc

Sự kế thừa được so sánh với mối quan hệ gia tộc, đó là lý do nó mang cái tên là sự kế thừa. Tuy nhiên sự kế thừa trong OOP không hoàn toàn giống sự kế thừa trong gia tộc loài người. Vì một điều, một lớp con có thể kế thừa một lớp cha. Lớp con (lớp dẫn xuất) có xu hướng có nhiều đặc điểm hơn lớp cha (lớp cơ sở) trong khi đó loài người thường thiếu những phẩm chất có được từ cả bố và mẹ.

2.1.3. Cú pháp của sự kế thừa

Ví dụ đơn giản nhất về sự kế thừa yêu cầu hai lớp: một lớp cơ sở và một lớp dẫn xuất. Lớp cơ sở không cần cú pháp đặc biệt nào nhưng lớp dẫn xuất phải chỉ rõ nó được rút ra từ lớp cơ sở nào. Điều này được thực hiện bằng cách đặt một dấu hai chấm vào sau tên của lớp dẫn xuất, theo sau là từ khóa **public** và tên của lớp cơ sở.

```
Class Base           //lớp cơ sở
{
    //hàm và dữ liệu thành viên
};
Class Derv:public Base //lớp dẫn xuất
{
    //hàm và dữ liệu thành viên
};
```

Dấu hai chấm (giống như mũi tên trong sơ đồ sự kế thừa) có nghĩa là rút ra từ. Bởi vậy, lớp **Derv** được rút ra từ lớp **Base** và kế thừa tất cả dữ liệu và các hàm mà lớp cơ sở **Base** có (mặc dù chúng ta không chỉ ra ở đây). Từ khóa **public** chúng ta sẽ nói tới sau.

Chúng ta cùng xem một ví dụ chi tiết hơn. Bản 2-1 có lớp **Parent** và một lớp **child** rút ra từ đó.

Listing 2-1 INHERIT

```
//inherit.cpp
//khung các lớp minh họa sự kế thừa
#include<iostream.h>
class Parent
{
private:
    float f1;           //dữ liệu của Parent
public:
    void pget()
    {
        cout<<"\nNhập vào giá trị float:";
        cin>>f1;
    }
    void pdisplay()
    {
        cout<<"f1="<<f1;
    }
};
class Child:public Parent
{
```

```

private:
    int intv;           //du lieu cua Parent
public:
    void cget()
    {
        pget();
        cout<<"\nNhap vao gia tri int:";
        cin>>intv;
    }
    void cdisplay()
    {
        pdisplay();
        cout<<"int="<<intv;
    }
};

void main()
{
    Child ch;           //tao mot doi tuong Child
    cout<<"Nhap du lieu cho doi tuong child";
    ch.cget();
    cout<<"Du lieu trong doi tuong Child la:";
    ch.cdisplay();
}

```

Sau đây là tương tác với chương trình:

Nhan du lieu cho doi tuong Child

Nhap vao gia tri float: 456.789

Nhap vao gia tri int: 123

Du lieu trong doi tuong Child la: flov=456.789, int=123

Kết quả dường như là bí ẩn. Mặc dù đối tượng **ch** của lớp **Child** chứa duy nhất một mục dữ liệu số nguyên nhưng kết quả đưa ra màn hình cho ta thấy nó cũng đã lưu trữ và lấy lại thành công mục dữ liệu float. Việc này xảy ra như thế nào?

2.1.4. Thuộc tính kế thừa

Một đối tượng của lớp dẫn xuất kế thừa tất cả các hàm và dữ liệu thành viên của lớp cơ sở. Bởi vậy, đối tượng **ch** không chỉ chứa mục dữ liệu **intv** mà còn chứa cả mục dữ liệu **flav**. Đối tượng **ch** cũng có thể truy nhập, ngoài các hàm thành viên của nó là **cget()** và **cdisplay()**, còn có các hàm thành viên từ **Parent** là **pget()** và **pdisplay()**.

1. Truy nhập dữ liệu lớp cơ sở

Mặc dù một đối tượng của lớp dẫn xuất có thể chứa dữ liệu được định nghĩa trong lớp cơ sở nhưng chưa hẳn dữ liệu này có thể truy nhập được từ lớp dẫn xuất. Cho rằng chúng ta muốn hàm **cdisplay()** trong **Child** hiển thị dữ liệu **flav** của **Parent** cũng như dữ liệu **intv** của nó, chúng ta xét xem có thể viết lại hàm **cdisplay()** như sau được không:

```

class Child
{
    void cdisplay()
    {
        cout<<"flav="<<flav;
        cout<<"intv="<<intv;
    }
};

```

Câu trả lời là "Không !". Không thể truy nhập **flov** từ lớp **Child** bởi vì **flov** là thành viên riêng (**private**) của **Parent**. Thông thường trong C++, các thành viên chung (**public members**) - dữ liệu và các hàm - có thể truy nhập được từ bên ngoài lớp nhưng các thành viên riêng thì không.

Các hàm của lớp dẫn xuất không ở trong lớp cơ sở nhưng chúng có liên quan gần gũi với lớp cơ sở hơn là các hàm ở trong các lớp không có quan hệ gì. Tuy nhiên, chúng không thể truy nhập các thành viên riêng của lớp cơ sở (sau này chúng ta sẽ thấy một loại đặc biệt, **protected**, cho phép kiểu truy nhập như vậy).

2. Gọi hàm của lớp cơ sở

Chúng ta không cần truy nhập trực tiếp dữ liệu **flov** từ các hàm trong đối tượng **Child**. Không như **flov** là **private**, các hàm thành viên của **Parent** là **public**. Bởi vậy, chúng ta có thể gọi chúng để quản lý **flov**. Từ hàm **cdisplay()** chúng ta gọi hàm **pdisplay()** để hiển thị giá trị của **flov**. Lớp dẫn xuất không cần truy nhập trực tiếp biến **flov**.

2.1.5. Sự chồng hàm trong lớp cơ sở và lớp dẫn xuất

Trong chương trình INHERIT, các hàm truy nhập và hiển thị dữ liệu trong lớp cơ sở và lớp dẫn xuất là khác nhau. Điều này rất dễ nhầm và không phải tinh thần của OOP : sử dụng các tên khác nhau cho các hàm làm cùng một công việc cơ bản giống nhau. Sẽ tốt hơn nếu cả hai hàm này có cùng tên, khi đó tên hàm sẽ rõ hơn và dễ nhớ hơn. Vì vậy, hoàn toàn đúng khi làm chồng (còn gọi là làm trùng tên) các hàm trong lớp cơ sở và lớp dẫn xuất. Nếu chúng ta đặt tên cho cả hai hàm này là **display()** thì chúng ta có dạng như sau:

```
class Parent
{
    void display()
    {}
};
class Child:public Parent
{
    void display()
    {
        //display(); //không được gọi display() trong Child
        Parent::display(); //được gọi display() trong Parent
    }
};
```

Nếu chúng ta dùng tên **display()** trong một hàm của lớp dẫn xuất, trình biên dịch cho rằng chúng ta gọi hàm trong lớp dẫn xuất. Điều đó sẽ là một thảm họa từ bên trong hàm **display()**, bởi vì một hàm gọi đi gọi lại chính nó sẽ phá hủy chương trình. Trong lớp dẫn xuất, để tham chiếu tới một hàm trong lớp cơ sở, ta phải dùng toán tử quy định phạm vi :: (**scope resolution operator**) và tên lớp cơ sở :

Parent::display()

Điều này để cho trình biên dịch biết rõ là chúng ta muốn gọi hàm nào. Bản chương trình 2-2 là một phiên bản đã sửa chữa cho thấy cách mà chúng ta làm trùng tên các hàm thành viên, gọi hai hàm hiển thị **display()** và hai hàm nhập dữ liệu **get()**.

Listing 2-2 INHERIT2

```
//inherit2.cpp
//su dung ham chong
class Parent
```



```

{
private:
    float flow;
public:
    void get()
    {
        cout<<"\n Nhap vao gia tri float:";
        cin>>flow;
    }
    void display()
    {
        cout<<"flow="<<flow;
    }
};
class Child:public Parent
{
private:
    int intv;
public:
    void get()
    {
        Parent::get();
        cout<<"\n Nhap vao gia tri int:";
        cin>>intv;
    }
    void display()
    {
        Parent::display();
        cout<<"intv="<<intv;
    }
};
////////////////////////////////////
void main()
{
    Child ch;
    cout<<"Nhan du lieu cho doi tuong Child";
    ch.get();
    cout<<"Du lieu trong doi tuong Child la ";
    ch.display();
}

```

Bảng 2-1 tóm tắt cách truy nhập các hàm chồng tên và các hàm thông thường từ lớp cơ sở và lớp dẫn xuất. Giả sử có một hàm thành viên **basefunc()** trong lớp cơ sở, một hàm thành viên **dervfunc()** trong lớp dẫn xuất và một hàm chồng tên **func()** trong cả hai lớp.

Bảng 2-1. Truy nhập hàm từ lớp cơ sở và lớp dẫn xuất

Hàm là thành viên của	Trạng thái chồng	Truy nhập hàm từ lớp cơ sở	Truy nhập hàm từ lớp dẫn xuất
Lớp cơ sở	Tên khác nhau	Basefunc()	Basefunc()
Lớp cơ sở	Chồng tên	Func()	Base::func()
Lớp dẫn xuất	Tên khác nhau	Không biết hàm	Dervfunc()
Lớp dẫn xuất	Chồng tên	Không biết hàm	Func()

2.2. THIẾT KẾ CHƯƠNG TRÌNH : LỚP EMPLOYEE

Chúng ta cùng xem một ví dụ tương đối thực tế về sự kế thừa. Chúng ta sẽ bắt đầu với lớp **employee** (một lớp nhân viên). Chương trình này tập trung vào thiết kế chương trình trên khía cạnh kế thừa: đó là dùng sự kế thừa như một cách cơ bản để liên kết các thành phần chương trình với nhau. Chương trình này ít liên quan tới khái niệm sử dụng lại. Vấn đề này chúng ta sẽ nói ở phần tiếp theo.

Cho rằng chúng ta muốn lập trình một cơ sở dữ liệu về nhân viên của một công ty. Các nhân viên thường được chia thành các loại khác nhau. Trong công ty tưởng tượng này, chúng ta giả sử có người quản lý, nhà khoa học và công nhân.

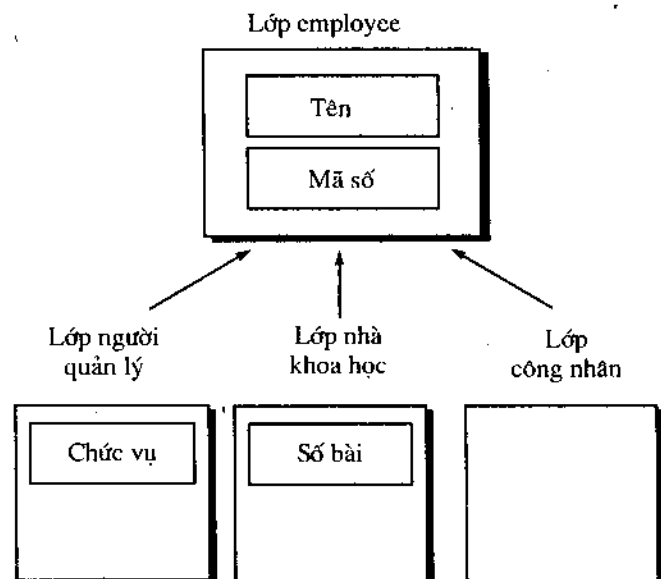
2.2.1. Phân cấp lớp

Khi gặp một tình huống trong đó nhiều thứ khác nhau có thể được mô tả như là nhiều loại của một thứ khác, chúng ta sẽ thấy tình huống đó có thể được mô hình hóa trong một chương trình như một sự kế thừa. Người quản lý, nhà khoa học và công nhân là các loại nhân viên. Đây là sự mô hình hóa trong C++ :

class employee	//lớp nhân viên
{ ...};	
class manager:public employee	//lớp người quản lý
{ };	
class scientist:public employee	//lớp nhà khoa học
{ };	
class laborer:public employee	//lớp công nhân
{ };	

Ba lớp **manager**, **scientist** và **laborer** được rút ra từ lớp **employee**. Tất cả các nhân viên không cần biết chúng thuộc lớp dẫn xuất nào đều có những phần tử chung. Vì mục đích của cơ sở dữ liệu, chúng ta giả thiết rằng tất cả chúng đều có tên và mã số. Trong kế thừa, những phần tử chung như vậy được đặt ở lớp cơ sở. Các lớp dẫn xuất có những đặc điểm riêng của chúng. Đặc điểm quan trọng về người quản lý là chức vụ, chẳng hạn là giám đốc hay phó giám đốc. Đối với nhà khoa học, những dữ liệu quan trọng là số bài mà nhà khoa học viết cho các tạp chí khoa học. Đối với công nhân, không có đặc điểm nào khác tên và mã công nhân. Hình 2-2 cho thấy điều này.

Chương trình EMPINH trong bản 2-3 là một ví dụ minh họa sự phân cấp lớp như trên.



Hình 2-2. Phân cấp lớp trong EMPINH.

Listing 2-3 EMPINH

```
//empinh.cpp
//mo hinh hoa su ke thua
#include<iostream.h>
class employee
{ private:
    enum {LEN=30}; //do dai cuc dai cua ten
    char name[LEN]; //ten nhan cong
    unsigned long number; //ma nhan cong
public:
    void getdata()
    {
        cout<<"\n Nhap vao ten:";cin>>name;
        cout<<"\n Nhap vao ma so:";cin>>number;
    }
    void putdata()
    {
        cout<<"\n Name="<<name;
        cout<<"\n Ma so="<<number;
    }
};
////////////////////////////////////
class manager:public employee
{
private:
    enum {LEN=40}; //do dai cuc dai cua chuc vu
    char title[LEN]; //chuc vu
public:
    void getdata()
    {
        employee::getdata();
        cout<<"\n Chuc vu:";cin>>title;
    }
    void putdata()
    {
        employee::putdata();
        cout<<"\n Chuc vu="<<title;
    }
};
////////////////////////////////////
class scientist:public employee
{
private:
    int pubs; //so bai viet
public:
    void getdata()
    {
        employee::getdata();
        cout<<"\n So bai viet:";cin>>pubs;
    }
    void putdata()
    {
        employee::putdata();
        cout<<"\n So bai viet="<<pubs;
```

```

    }
};
////////////////////////////////////
class laborer:public employee
{
};
////////////////////////////////////
void main()
{
    manager m1,m2;
    scientist s1;
    laborer l1;
    cout<<endl;
    cout<<"\nNhap du lieu cho manager 1:";
    m1.getdata();
    cout<<"\nNhap du lieu cho manager 2:";
    m2.getdata();
    cout<<"\nNhap du lieu cho scientist 1:";
    s1.getdata();
    cout<<"\nNhap du lieu cho laborer 1:";
    l1.getdata();
    cout<<"\n Du lieu tren manager 1"; //hien thi du lieu cua vai nhan cong
    m1.putdata();
    cout<<"\n Du lieu tren manager 2";
    m2.putdata();
    cout<<"\n Du lieu tren scientist 1";
    s1.putdata();
    cout<<"\n Du lieu tren laborer 1";
    l1.putdata();
}

```

Các hàm thành viên của lớp **manager** và **scientist** gọi các hàm lớp cơ sở trong **employee** để lấy tên nhân viên và mã số từ người sử dụng và hiển thị chúng. Chúng cũng quản lý những dữ liệu riêng như chức vụ đối với **manager**, số bài viết đối với **scientist**. Lớp **laborer** hoàn toàn dựa vào lớp **employee** cả về dữ liệu và hàm thành viên.

2.2.2. Lớp trừu tượng

Chúng ta có thể nghĩ là nên sử dụng các đối tượng của lớp **employee** thay cho các đối tượng của lớp **laborer** bởi vì chúng có cùng dữ liệu và các hàm. Tuy nhiên điều này đi ngược với tổ chức chương trình. Như đã sử dụng ở đây, lớp **employee** không phải là lớp để thực sự thể hiện các đối tượng. Mục đích duy nhất của lớp **employee** là phục vụ như một lớp tổng quát để các lớp khác có thể rút ra từ nó.

Để hiểu rõ hơn, hãy tưởng tượng có một động vật tên là Toby. Nếu có ai đó hỏi Toby là gì, có thể chúng ta sẽ nói: "Ồ, nó là một loại động vật có vú". Mọi người lại hỏi: "Vâng, nhưng nó là loại động vật có vú gì?" Tất cả các động vật đều có nguồn gốc từ một loài cụ thể hơn là động vật có vú. Động vật có vú là một loài không cụ thể, nó chỉ là một loài tổng quát.

Một lớp mà chúng ta dự định không thể hiện bằng bất kỳ đối tượng nào, chỉ là lớp cơ sở cho các lớp khác gọi là **lớp trừu tượng**. Một thuận lợi của việc dùng lớp trừu tượng **employee** là nếu chúng ta quyết định, đôi khi sau thiết kế phân cấp lớp, thêm một mục dữ liệu vào lớp **laborer**, lúc đó chúng ta có thể thay đổi lớp **laborer** mà không cần thay đổi lớp **employee**, bởi vì thay đổi **employee** có thể ảnh hưởng tới tất cả các lớp dẫn xuất của **employee**.

2.3. SỰ SỬ DỤNG LẠI: LỚP STACK CẢI TIẾN

Trong phần trước chúng ta đã xem một chương trình mà ở đó sự kế thừa làm việc như một phần tử thiết kế chính. Trong phần này, chúng ta sẽ minh họa một chương trình ở đó sự kế thừa cho phép sử dụng lại. Chúng ta sẽ bắt đầu với lớp **Stack**.

Giả sử có một lớp **Stack** đã được một người nào đó viết, chúng ta thấy phiên bản **Stack** này không cảnh báo cho người sử dụng biết khi có quá nhiều mục dữ liệu đặt vào **Stack** hoặc có quá nhiều mục dữ liệu lấy ra khỏi **Stack**. Chúng ta cũng sửa lại tình huống này bằng cách rút ra một lớp mới từ **Stack** để kiểm tra trên và trên dưới của **Stack**. Chúng ta sẽ gọi lớp thu được này là **Stack2**.

2.3.1. Sự sử dụng lại

Tưởng tượng rằng một công ty (hay một nhóm người lập trình) đã viết lớp **Stack** đầu tiên. Họ đã mất nhiều năm để viết và gỡ rối lớp và lớp **Stack** bây giờ là một phần chương trình đáng tin cậy. Sau này một nhóm lập trình khác trong một công ty khác có được một thư viện chứa lớp **Stack**. Lớp này nổi bật bởi tốc độ và sự đa năng của nó, nhưng công ty đã phát hiện là nó cần sự bảo vệ lỗi tốt hơn. Bởi vậy, nhóm người lập trình thứ hai này rút ra một lớp mới từ lớp cũ. Vì không thay đổi gì trong lớp **Stack** ban đầu nên những đặc điểm ban đầu sẽ tiếp tục làm việc tốt. Lớp mới chỉ đơn giản là có được thêm vào một vài đặc điểm mới và chỉ những phần này mới cần được gỡ rối.

Trong tình huống thực chúng ta sẽ làm việc với nhiều file, nhưng để đơn giản chúng ta sẽ đưa ra cả lớp cơ sở và lớp dẫn xuất như một phần trên cùng một file nguồn. Dưới đây là bản chương trình **STACKINH**.

Listing 2-4 STACKINH

```
//stackinh.cpp
//tao lop stack cai tien dung su ke thua
#include<iostream.h>
#include<conio.h>
#include<process.h> //dung cho exit()

////////////////////////////////////
class Stack //mot stack luu tru cuc dai SIZE so nguyen
{
protected:
    enum {SIZE=20}; //kha nang cua Stack
    int st[SIZE]; //so nguyen duoc luu trong mang
    int top;
public:
    Stack() //ham tao khong doi so
    {top=-1;}
    void push(int var) //dat mot muc du lieu vao Stack
    {st[++top]=var;}
    int pop() //lay mot muc du lieu ra khoi Stack
    {return st[top--];}
};
////////////////////////////////////
class Stack2:public Stack
{
public:
    void push(int var) //dat mot muc du lieu vao Stack
    {
        if(top>=SIZE-1)
```

```

        {cout<<"Loi:Stack tran tren.";exit(-1);}
        Stack::push(var);
    }
    int pop()          //lay mot muc du lieu ra khoi Stack
    {return st[top--];}
};
////////////////////
void main()
{
    Stack2 s;          //tao mot doi tuong Stack2

    s.push(11);         //dat 3 muc du lieu vao Stack
    s.push(12);
    s.push(13);
    cout<<s.pop()<<endl; //lay cac muc du lieu va hien thi chung
    cout<<s.pop()<<endl;
    cout<<s.pop()<<endl;
    getch();
}

```

Ở đây chúng ta nhận thấy xuất hiện một đối tượng rõ ràng hơn.

Lớp **Stack2** không có dữ liệu riêng của nó. Nhiệm vụ của nó là "gói" các hàm **pop()** và **push()** của lớp **Stack** trong các hàm cải tiến **pop()** và **push()** của riêng nó. Các hàm cải tiến này bắt đầu với lệnh **if** để kiểm tra bảo đảm cho người sử dụng lớp không lấy ra hoặc đặt vào quá nhiều mục dữ liệu. Sau đó chúng gọi hàm **push()** và **pop()** của lớp cơ sở để thực hiện lưu trữ và lấy lại dữ liệu.

Các thành viên có thể truy nhập được của một lớp được gọi là giao diện của lớp, bởi vì chúng là những gì mà người sử dụng lớp tương tác với lớp. Ở đây, giao diện của lớp **Stack2** giống lớp **Stack**. Trên thực tế, lớp **Stack2** đã cải trang thành lớp **Stack**. Tuy nhiên, một đối tượng **Stack2** có nội dung bên trong rõ ràng hơn.

2.3.2. Hàm tạo lớp cơ sở

Lưu ý là có một hàm tạo trong lớp cơ sở **Stack** nhưng không có trong lớp dẫn xuất **Stack2**. Khi chúng ta tạo một đối tượng của lớp dẫn xuất, trình biên dịch sẽ kiểm tra xem có hàm tạo trong lớp dẫn xuất không. Nếu có, trình biên dịch sẽ sắp xếp để nó được gọi; nếu không có hàm tạo lớp dẫn xuất, trình biên dịch sẽ gọi hàm tạo lớp cơ sở thay thế (điều này chỉ đúng cho các hàm tạo không có đối số như chúng ta sẽ thấy trong các phần tiếp theo). Trong lớp **Stack**, nhiệm vụ của hàm tạo là khởi tạo chỉ số **top** về đỉnh của ngăn xếp rỗng.

2.3.3. Định danh truy nhập **protected**

Trên thực tế, không phải là chúng ta không thay đổi gì đối với lớp cơ sở **Stack** trước khi rút ra lớp **Stack2** từ nó. Chúng ta đã thay đổi định danh truy nhập cho dữ liệu từ **private** sang **protected**.

Một định danh **private** chỉ cho phép các hàm thành viên của chính lớp đó truy nhập. Ngược lại, định danh **public** có thể cho bất kỳ hàm nào trong chương trình truy nhập. Còn định danh **protected** cho phép truy nhập đối với các hàm hoặc là thành viên của lớp đó hoặc là thành viên của lớp dẫn xuất nó. Vì vậy, định danh **protected** được gọi là định danh hướng gia tộc.

Những người tạo **Stack** cần làm cho dữ liệu của họ là **protected** (được bảo vệ) khi tạo lớp lần đầu tiên để cho, nếu ai đó muốn rút ra một lớp khác từ nó, họ có thể truy nhập dữ liệu của **Stack**.

Trên thực tế có một bất lợi khi đặt dữ liệu là **protected**. Chúng ta sẽ quay lại vấn đề này khi tập trung vào các định danh truy nhập ở mục 2.5.

2.3.4. Hàm không được kế thừa

Có một vài hàm đặc biệt không được tự động kế thừa. Như chúng ta đã thấy trong mục 2.1, nếu có một hàm **func()** trong một lớp cơ sở **alpha** và nó không được chồng (**overloaded**) trong lớp dẫn xuất beta, lúc đó một đối tượng của lớp dẫn xuất có thể gọi hàm lớp cơ sở này:

```
beta bb;  
bb.func();
```

Hàm **func()** được tự động kế thừa bởi **beta**. Với giả thiết là hàm **func()** chỉ thao tác trên dữ liệu lớp cơ sở; nó không cần tương tác với lớp dẫn xuất.

Tuy nhiên có một vài hàm mà chúng ta biết trước là chúng sẽ làm những việc khác nhau trong lớp cơ sở và lớp dẫn xuất. Đó là toán tử chồng =, các hàm tạo (**constructor**) và các hàm hủy (**destructor**).

Trước tiên chúng ta xét hàm tạo. Hàm tạo lớp cơ sở phải tạo dữ liệu lớp cơ sở và hàm tạo lớp dẫn xuất phải tạo dữ liệu lớp dẫn xuất. Bởi vì các hàm tạo lớp cơ sở và lớp dẫn xuất tạo dữ liệu khác nhau nên một hàm tạo không thể thay cho một hàm tạo khác. Do đó các hàm tạo không được tự động kế thừa.

Tương tự, toán tử = trong lớp dẫn xuất phải gán các giá trị cho dữ liệu của lớp dẫn xuất và toán tử = trong lớp cơ sở phải gán các giá trị cho dữ liệu của lớp cơ sở. Đây là các công việc khác nhau, bởi vậy toán tử này cũng không được tự động kế thừa.

Bây giờ chúng ta sẽ xét hàm hủy. Một hàm hủy lớp dẫn xuất hủy dữ liệu của lớp dẫn xuất. Nó không hủy các đối tượng lớp cơ sở; nó phải gọi hàm hủy lớp cơ sở để làm việc này. Hơn nữa, các hàm hủy này làm các công việc khác nhau nên chúng không được tự động kế thừa.

Nếu định nghĩa rõ một trong các hàm không kế thừa này trong lớp cơ sở, chẳng hạn toán tử = và không định nghĩa hàm này trong lớp dẫn xuất, trình biên dịch sẽ tạo một phiên bản mặc định. Các hàm tạo copy mặc định và toán tử = mặc định thực hiện việc sao chép và gán có suy nghĩ.

2.4. HÀM TẠO VÀ SỰ KẾ THỪA

Như chúng ta đã thấy, các hàm tạo là các hàm hơi đặc biệt. Bởi vậy chúng đóng vai trò khác thường trong sự kế thừa là điều không đáng ngạc nhiên. Trong phần này, chúng ta sẽ xem xét vai trò của nó, chỉ ra khi nào cần các hàm tạo, khi nào không và giải thích cách mà một hàm tạo có thể gọi một hàm tạo khác dùng cú pháp đặc biệt.

2.4.1. Chuỗi rất lớn các hàm tạo

Khi chúng ta định nghĩa một đối tượng của một lớp dẫn xuất, không chỉ hàm tạo của nó được thực hiện mà hàm tạo trong lớp cơ sở cũng được thực hiện. Trên thực tế, hàm tạo lớp cơ sở được thực hiện trước. Bởi vì đối tượng lớp cơ sở là đối tượng con - một phần - của đối tượng lớp dẫn xuất và chúng ta cần tạo các bộ phận trước khi tạo toàn thể. Hình 2-3 chỉ ra mối quan hệ giữa các đối tượng và đối tượng con.

Các hàm tạo của tất cả các đối tượng con của một đối tượng được gọi trước hàm tạo của đối tượng đó. Ví dụ sau minh họa điều này:

Listing 2-5 INCONDES

```
//incondes.cpp  
//kiểm tra hàm tạo và hàm hủy trong sự kế thừa
```

```

#include<iostream.h>
#include<conio.h>
class Parent
{
public:
    Parent()
    {cout<<"\n Parent constructor";}
    ~Parent()
    {cout<<"\n Parent destructor";}
};
////////////////////////////////////
class Child:public Parent
{
public:
    Child()
    {cout<<"\n Child constructor";}
    ~Child()
    {cout<<"\n Child destructor";}
};
////////////////////////////////////
void main()
{
    cout<<"\nStarting ";
    Child ch;           //tao mot doi tuong Child
    cout<<"\nTerminating ";
    getch();
}

```

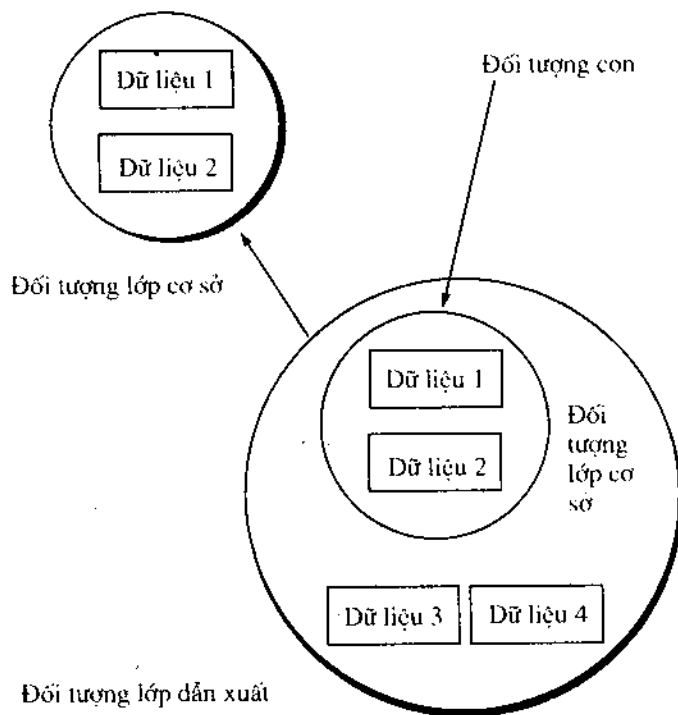
Chương trình này mô tả một lớp **Parent** và một lớp **Child** rút ra từ **Parent**. Trong hàm **main()** chúng ta tạo một đối tượng **Child**. Chúng ta đã cài đặt rõ ràng các hàm tạo và hàm hủy ở cả hai lớp, có thể nhận ra điều này khi chúng thực hiện. Đây là kết quả đưa ra từ chương trình **incondes.cpp**:

```

Starting
Parent constructor
Child constructor
Terminating
Parent destructor
Child destructor

```

Như chúng ta thấy, hàm tạo lớp cơ sở được gọi trước sau đó hàm tạo lớp dẫn xuất mới được gọi. Khi đối tượng **Child** kết thúc, hàm hủy được gọi theo thứ tự ngược lại: đối tượng lớp dẫn xuất được hủy trước sau đó đến đối tượng lớp cơ sở.



Hình 2-3. Sự kế thừa và các đối tượng con.

Trình biên dịch sắp xếp để tất cả các hàm tạo và hàm hủy được gọi một cách tự động. Thậm chí, nếu chúng ta không định nghĩa rõ ràng hàm tạo và hàm hủy như đã làm ở đây thì trình biên dịch sẽ gọi các hàm tạo và hàm hủy mặc định để tạo và hủy các đối tượng và đối tượng con. Hoạt động này sẽ xảy ra dù có hay không có các hàm tạo và hàm hủy của riêng chúng ta.

Hàm tạo mặc định của một đối tượng luôn luôn được gọi trước hàm tạo mà chúng ta định nghĩa. Khi hàm tạo của chúng ta bắt đầu thực hiện, đối tượng và tất cả các đối tượng con của nó đã tồn tại và được khởi tạo.

2.4.2. Khi nào cần các hàm tạo lớp dẫn xuất

Chúng ta đã thấy trong chương trình STACKINH ở phần trước là chúng ta không cần viết một hàm tạo không đối số cho lớp dẫn xuất nếu lớp cơ sở đã có một hàm tạo như vậy.

1. Hàm tạo không có đối số

Chúng ta không thể dùng một hàm tạo có đối số khi nó được định nghĩa rõ ràng trong một lớp mà chúng ta sẽ thể hiện một đối tượng. Có nghĩa là, thậm chí nếu chúng ta có một hàm tạo n đối số trong lớp cơ sở thì chúng ta vẫn phải định nghĩa một hàm tạo n đối số trong lớp dẫn xuất. Bản chương trình 2-6 CONINH cho thấy điều này làm việc như thế nào.

Listing 2-6 CONINH

```
//coninh.cpp
//minh hoa ham tao va su ke thua
class Mu
{
    private:
        int mudata;           //du lieu
    public:
        Mu():mudata(0)         //ham tao khong doi so
        {}
        Mu(int m):mudata(m)    //ham tao co doi so
        {}
};
////////////////////////////////////
class Nu:public Mu             //lop dan xuat
{
};                             //khong co ham tao
////////////////////////////////////
void main()
{
    Nu n1;                     //co the dung ham tao khong doi so trong Mu
    Nu n2(20);                 //loi: khong the dung ham tao mot doi so
}
```

2. Hàm tạo có đối số

Nếu chúng ta thêm một hàm tạo một đối số vào lớp dẫn xuất Nu thì chúng ta cũng phải thêm vào một hàm tạo không đối số, cũng như với bất kỳ lớp nào. Bản chương trình 2-7 CONSINH có những hàm tạo thêm vào này.

Listing 2-7 CONSINH

```
//consinh.cpp
//minh hoa ham tao va su ke thua
class Mu
{
    private:
```

```

    int mudata;           //du lieu
public:
    Mu():mudata(0)        //ham tao khong doi so
    {
    }
    Mu(int m):mudata(m)    //ham tao co doi so
    {
    }
};
class Nu:public Mu        //lop dan xuất
{
public:
    Nu()                  //ham tao khong doi so
    {
    }
    Nu(int n):Mu(n)        //ham tao co doi so
    {
    }
};                          //khong co ham tao
////////////////////
void main()
{
    Nu n1;                //goi ham tao khong doi so
    Nu n2(20);             //loi: khong the dung ham tao mot doi so
}

```

Trong hàm main() chương trình đã định nghĩa thành công các đối tượng **Nu** sử dụng cả hàm tạo không đối số và hàm tạo có đối số. Hàm tạo một đối số truyền đối số của nó cho hàm tạo trong **Mu** sử dụng cú pháp mới:

```

Nu(int n):Mu(n)
{ }

```

Tất cả những gì sau dấu hai chấm được gọi là danh sách bộ khởi tạo, ở đây sự khởi tạo được thực hiện trước khi hàm tạo bắt đầu thực hiện. Cú pháp này giống cú pháp dùng trong các hàm tạo để khởi tạo các biến thuộc kiểu dữ liệu cơ bản, nhưng ở đây nó được dùng để gọi một hàm tạo. Trên thực tế, những hoạt động này giống nhau về khái niệm, bởi vậy không có gì là ngạc nhiên khi chúng dùng cùng một cú pháp. Trong trường hợp này thì thực hiện khởi tạo một đối tượng lớp còn trong trường hợp khác lại khởi tạo một biến thuộc kiểu dữ liệu cơ bản, nhưng C++ đối xử với các đối tượng lớp và các biến kiểu dữ liệu cơ bản như nhau.

Trong chương trình này, hàm tạo lớp dẫn xuất chỉ đơn giản truyền đối số n cho hàm tạo lớp cơ sở. Nó không có hành động gì nên trong hai dấu ngoặc nhọn là rỗng. Đây là tình huống thường gặp trong lớp dẫn xuất: các hàm tạo có thân hàm rỗng bởi vì công việc của nó được làm trong danh sách bộ khởi tạo.

2.4.3. Thêm chức năng cho hàm tạo lớp dẫn xuất

Không phải tất cả các hàm tạo lớp dẫn xuất đều có thân hàm rỗng. Cho rằng chúng ta muốn thêm một đặc điểm cho lớp **airtime** (một lớp thời gian sử dụng trong hàng không), chúng ta muốn ngăn cản người sử dụng lớp khởi tạo các đối tượng **airtime** với các giá trị giờ và phút quá lớn. Giờ cần nhỏ hơn 24 và phút cần nhỏ hơn 60. Nếu người sử dụng viết :

```

airtime Departure(11,65);

```

thì chúng ta muốn hàm tạo cảnh báo bởi vì giá trị phút không thể lớn hơn 59.

Để thêm đặc điểm mới này cho **airtime**, chúng ta rút ra một lớp mới **airtime2** có lệnh kiểm tra lỗi cần thiết trong hàm tạo hai đối số của nó. Điều này được minh họa trong bản chương trình 2-8 là AIRINH.

Listing 2-8 AIRINH

```
//airinh.cpp
//ham tao va su ke thua trong lop airtime
#include<iostream.h>
#include<process.h>          //cho exit()
#include<conio.h>            //cho getch()
class airtime
{
protected:
    int hours;
    int minutes;
public:
    airtime():hours(0),minutes(0)          //ham tao khong doi so
    {}
    airtime(int h,int m):hours(h),minutes(m) //ham tao hai doi so
    {}
    void display()const                    //dua ra man hinh
    {
        cout<<hours<<":"<<minutes;
    }
};
////////////////////////////////////
class airtime2:public airtime
{
public:
    airtime2():airtime()                  //ham tao khong doi so
    {}
    airtime2(int h,int m):airtime(h,m)    //ham tao hai doi so
    {
        if(minutes>59 || hours>23)
        {
            cout<<"\nLoi:gia tri airtime khong hop le"
            <<hours<<":"<<minutes;
            exit(-1);
        }
    }
};
////////////////////////////////////
void main()
{
    airtime2 at0;                          //tot, ham tao khong doi so duoc goi
    cout<<"\nat0=";
    at0.display();
    airtime2 at1(10,45);                    //tot, ham tao hai doi so duoc goi
    cout<<"\nat1=";
    at1.display();
    airtime2 at2(10,65);                    //loi, ham tao hai doi so duoc goi
    cout<<"\nat2=";
    at2.display();
    getch();
}
```

Kết quả đưa ra từ chương trình AIRINH:

```

at0=0:0
at1=10:45
Loi:gia tri airtime khong hop le

```

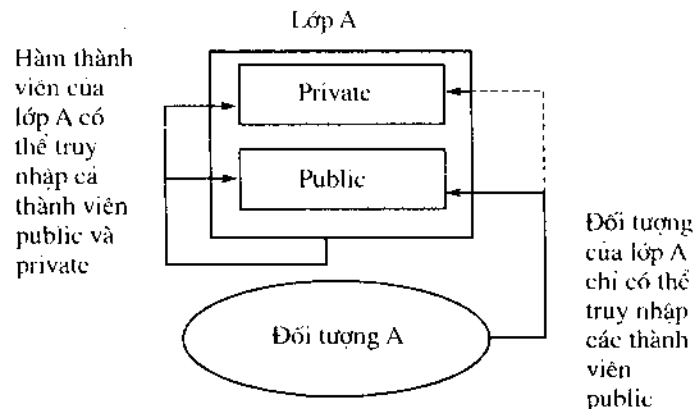
Như chúng ta thấy, hàm tạo **airtime2** đã tìm kiếm thành công giá trị **minutes** vượt ra khỏi khoảng cho phép và cảnh báo người sử dụng.

2.5. ĐIỀU KHIỂN TRUY NHẬP

Chúng ta đã được giới thiệu về các định danh truy nhập và cách mà chúng điều khiển truy nhập đối với các hàm thành viên lớp cơ sở. Trong phần này chúng ta sẽ nói kỹ về các định danh truy nhập và chỉ ra khi nào thì cần dùng định danh nào. Chúng ta cũng sẽ nói tới một tình huống khác: sự kế thừa chung (kế thừa **public**), sự kế thừa được bảo vệ (kế thừa **protected**) và sự kế thừa riêng (kế thừa **private**).

2.5.1. Ôn lại sự truy nhập

Khi chưa có sự kế thừa, các hàm thành viên lớp có thể truy nhập tới tất cả những gì có trong lớp dù nó là **public** hay **private**, nhưng các đối tượng của lớp đó chỉ có thể truy nhập tới các thành viên **public**, điều này được chỉ ra ở hình 2-4.



Hình 2-4. Định danh truy nhập không có sự kế thừa.

Khi sự kế thừa xuất hiện, các khả năng truy nhập khác tăng lên cho lớp dẫn xuất. Các hàm thành viên của lớp dẫn xuất có thể truy nhập các thành viên **public** và **protected** của lớp cơ sở nhưng vẫn không thể truy nhập các thành viên **private**. Các đối tượng của lớp dẫn xuất chỉ có thể truy nhập các thành viên **public** của lớp cơ sở. Tình huống này được chỉ ra ở bảng 2-2.

Bảng 2-2. Sự kế thừa và khả năng truy nhập

Định danh truy nhập	Có thể truy nhập từ trong lớp	Có thể truy nhập từ lớp dẫn xuất	Có thể truy nhập từ các đối tượng ngoài lớp
Public	Có	Có	Có
Protected	Có	Có	Không
Private	Có	Không	Không

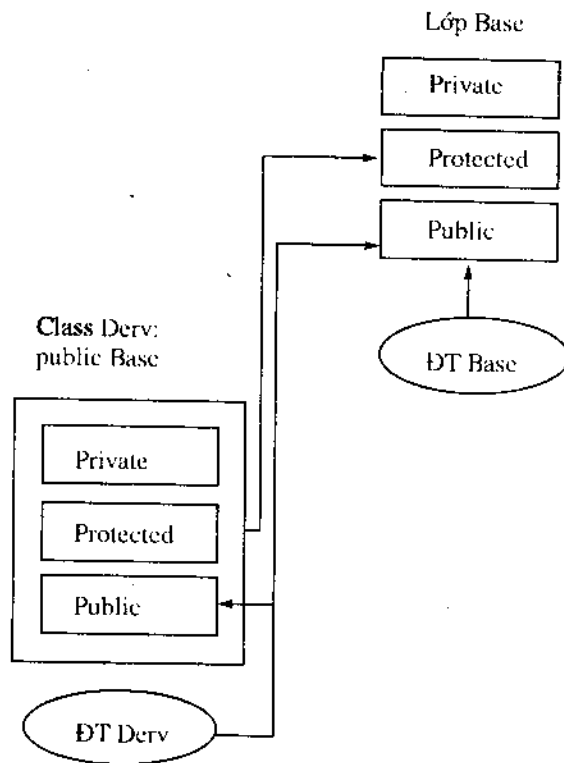
Mặc dù một lớp được rút ra từ lớp cơ sở nhưng các tình huống đối với lớp cơ sở không thay đổi, bởi vì chúng không biết về lớp dẫn xuất. Hình 2-5 chỉ ra mối quan hệ này.

2.5.2. Để dữ liệu ở private

Dữ liệu trong lớp cơ sở nên để ở **private** hay **protected**? Chúng ta đã có ví dụ cho cả hai cách này. Đối với các chương trình ví dụ để dữ liệu ở **protected** có thuận lợi là: không cần viết thêm các hàm lớp cơ sở để truy nhập dữ liệu từ lớp dẫn xuất. Tuy nhiên, đây không phải là cách tốt nhất.

Nói chung, dữ liệu lớp nên để **private** (tất nhiên có những ngoại lệ). Dữ liệu **public** có thể bị thay đổi bởi bất kỳ hàm nào ở bất kỳ đâu trong chương trình, điều này nên tránh. Dữ liệu **protected** có thể bị thay đổi bởi các hàm trong bất kỳ lớp dẫn xuất nào.

Bất kỳ người nào rút ra một lớp từ một lớp khác đều có quyền truy nhập tới dữ liệu **protected** của lớp cơ sở. Sẽ an toàn và tin cậy hơn nếu lớp dẫn xuất không thể truy nhập trực tiếp dữ liệu lớp cơ sở.



Hình 2-5. Định danh truy nhập có kế thừa.

Chúng ta nên nhớ lại rằng một giao diện lớp bao gồm các hàm dùng để truy nhập cái gì đó. Thiết kế các lớp cho chúng ta hai giao diện: một giao diện **public** để các đối tượng của lớp sử dụng và một giao diện **protected** để các lớp dẫn xuất sử dụng. Không nên để giao diện nào truy nhập trực tiếp dữ liệu. Một thuận lợi của việc để dữ liệu lớp cơ sở ở **private** là chúng ta có thể thay đổi nó mà không làm hỏng các lớp dẫn xuất. Chỉ cần giao diện giống nhau là chúng ta có thể thay đổi dữ liệu bên trong lớp.

Đây có thể là một ý tưởng tốt để ngăn cản các hàm lớp dẫn xuất thay đổi dữ liệu lớp cơ sở, các hàm như vậy chỉ được đọc dữ liệu lớp cơ sở. Để có truy nhập chỉ đọc này, chúng ta viết các hàm như bình thường. Các hàm này nên để ở **protected**, bởi vì chúng là phần giao diện **protected** được dùng bởi lớp dẫn xuất và các lớp không phải là lớp dẫn xuất không thể truy nhập được.

Ví dụ sau đây cho thấy sự sắp xếp này của dữ liệu **private** và các hàm truy nhập **protected**. Nó là một phiên bản sửa đổi của chương trình STACKINH trong mục 2.3.

Listing 2-9 STAPROFU (STack PROtected FUnction)

```
//staprofu.cpp
//tao stack tot hon dung ke thua
//du lieu lop co so la private, cac ham thanh vien la protected
#include<conio.h> //cho getch()
#include<iostream.h> //cho exit()
////////////////////////////////////
```

```

class Stack          //mot stack lưu trữ tối SIZE số nguyên
{
protected:
    enum {SIZE=20}; //khả năng của stack
private:
    int st[SIZE];    //số nguyên lưu trong mảng
    int top;         //chỉ số của mục dữ liệu cuối cùng đặt vào
protected:
    int get_top() const //tra về giá trị top hiện tại
    {return top;}
public:
    Stack()          //hàm tạo không đối số
    {top=-1;}
    void push(int var)
    {st[++top]=var;}
    int pop()
    {return st[top--];}
};
/////////////////////////////////////////////////////////////////
class Stack2:public Stack
{
public:
    void push(int var)
    {
        if(get_top()>=SIZE-1)
            {cout<<"Lỗi: Stack tràn trên";exit(-1);}
        Stack::push(var);    //gọi hàm push() trong Stack
    }
    int pop()
    {
        if(get_top()<0)
            {cout<<"Lỗi: Stack tràn dưới";exit(-1);}
        return Stack::pop();    //gọi hàm pop() trong Stack
    }
};
/////////////////////////////////////////////////////////////////
void main()
{
    Stack2 s;          //tạo một đối tượng Stack
    s.push(11);
    s.push(12);
    s.push(13);
    cout<<s.pop()<<endl;    //lấy ra 3 mục dữ liệu
    cout<<s.pop()<<endl;
    cout<<s.pop()<<endl;
    cout<<s.pop()<<endl;    //lỗi:lấy ra quá nhiều
    getch();
}

```

Như chúng ta có thể thấy, mảng **st** và chỉ số **top** tạo thành các mục dữ liệu chính trong **Stack** bây giờ là **private**. Chỉ có các hàm thành viên của **Stack** mới có thể truy nhập chúng. Khi chúng ta rút ra **Stack2** từ **Stack**, hàm **push()** và **pop()** của **Stack2** có thể gọi hàm **push()** và **pop()** của **Stack** để lưu trữ và lấy lại dữ liệu. Tuy nhiên, trong **Stack2** chúng ta cũng cần đọc giá trị của **top** để có thể kiểm tra xem **Stack** đã đầy chưa. Để làm được điều này, chúng ta đặt hàm **get_top()** trong

Stack, hàm này ở **protected** nên lớp dẫn xuất có thể thực hiện nó; để đảm bảo lớp dẫn xuất không thể dùng nó để thay đổi bất kỳ cái gì trong **Stack** nó là hàm hằng **const**.

Đối với người sử dụng lớp thông thường, **Stack2** làm việc y như trong chương trình **STACKINH** (hàm **main()** y như khi chỉ có **Stack**). Tuy nhiên, trong chương trình **STAPROFU** dữ liệu của **Stack** an toàn tránh những tổn hại do người sử dụng lớp viết lệnh sai. Nói chung, đây là cách tốt hơn là đặt dữ liệu ở **protected**.

Chú ý rằng chúng ta đặt hằng **enum SIZE** ở **protected** để **Stack2** có thể sử dụng. Bởi vì **SIZE** phải được định nghĩa trước **st**, nó là **private**, nên chúng ta dùng hai **protected** trong dữ liệu của **Stack**.

2.5.3. Sự kế thừa **public** và **private**

Chúng ta đã xét cách các định danh truy nhập có thể được áp dụng cho dữ liệu lớp cơ sở để điều khiển việc truy nhập từ lớp dẫn xuất. Bây giờ chúng ta cùng xét một cách sử dụng khác của các định danh truy nhập: điều khiển cách mà các lớp được kế thừa.

1. Sự kế thừa **public**

Trong sự kế thừa chúng ta thường muốn dùng định danh truy nhập **public**

```
class alpha
{};
class beta:public alpha
{};
```

Đây được gọi là sự kế thừa chung (sự kế thừa **public**), đôi khi còn gọi là sự rút ra **public** (**public derivation**). Với loại kế thừa này, các đối tượng của lớp dẫn xuất có thể truy nhập các thành viên **public** của lớp cơ sở, như ở bên trái hình 2-5.

Trong sự kế thừa **public**, hàm ý là một đối tượng của lớp **B** là một "loại" đối tượng của lớp **A**. Một đối tượng của lớp dẫn xuất có đầy đủ các đặc điểm của lớp cơ sở, cộng thêm một vài đặc điểm của riêng nó.

2. Trình biên dịch và sự kế thừa **public**

Trình biên dịch rất coi trọng mối quan hệ "loại" trong sự kế thừa **public**. Nó để cho chúng ta sử dụng một đối tượng lớp dẫn xuất trong nhiều tình huống mà ở đó cần một đối tượng lớp cơ sở bởi vì đối tượng lớp dẫn xuất là một loại đối tượng lớp cơ sở. Chương trình ví dụ sau đây, **KINDOF**, minh họa sự mềm dẻo này (ở đây chương trình không định nghĩa hàm **anyfunc()** và không muốn được liên kết hay thực hiện).

Listing 2-10 **KINDOF**

```
//kindof.cpp
//một đối tượng lớp dẫn xuất là một loại đối tượng lớp cơ sở
class alpha                                //lớp cơ sở
{
    public:
        void memfunc()                    //hàm thành viên public
        {}
};
class beta:public alpha                    //lớp dẫn xuất
{};
void main()
{
```

```

void anyfunc(alpha);      //khai bao, ham co mot doi so
alpha aa;                //doi tuong kieu alpha
beta bb;                 //doi tuong kieu beta
aa=bb;                   //doi tuong beta duoc gan cho bien alpha
anyfunc(bb);              //doi tuong beta duoc truyen nhu doi so alpha
}

```

Như chúng ta thấy, một đối tượng của lớp dẫn xuất có thể gọi một hàm thành viên trong lớp cơ sở chỉ cần hàm đó có kiểu truy nhập **public**:

```
bb.anyfunc();
```

Tuy nhiên, chúng ta có thể gán một đối tượng lớp dẫn xuất cho một biến của lớp cơ sở.

```
aa=bb;
```

Trình biên dịch rất vui với điều này bởi vì **bb** là một loại **alpha**. Tương tự, chúng ta có thể truyền một đối tượng lớp dẫn xuất cho một hàm có một đối số lớp cơ sở.

Lợi ích thực sự của sự mềm dẻo này sẽ rõ hơn khi chúng ta nói về con trỏ và hàm ảo.

Lưu ý là với cách khác điều này không làm việc. Ví dụ chúng ta không thể nói:

```
bb=aa;
```

bởi vì đối tượng **aa** không phải là một loại đối tượng **bb**.

3. Sự kế thừa *private*

Bây giờ chúng ta đã thấy trình biên dịch đối xử với sự kế thừa **public** như thế nào, tiếp theo chúng ta cùng xem sự kế thừa **private**, nó có dạng như sau:

```

class alpha
{ };
class beta:private alpha
{ };

```

Bằng cách thay thế **private** cho **public** chúng ta sẽ thay đổi toàn bộ mối quan hệ kế thừa một cách đáng ngạc nhiên (chúng ta tạm thời dùng khả năng **protected** vì nó rất giống **private**).

Về cơ bản, khi một lớp được kế thừa **private**, các đối tượng của nó không thể truy nhập bất kỳ cái gì có trong lớp cơ sở, bất kể định danh truy nhập nào được dùng trong lớp cơ sở. Điều này được chỉ ra ở bên phải hình 2-5.

Kết quả là toàn bộ lớp cơ sở ẩn đối với các đối tượng của lớp dẫn xuất. Đối với các lớp dẫn xuất có thể coi như không có lớp cơ sở. Bởi vậy, các đối tượng của lớp dẫn xuất không biết rằng chúng là một loại đối tượng lớp cơ sở; chúng không biết gì về lớp cơ sở mặc dù lớp cơ sở là một phần của lớp dẫn xuất. Điều này giống sự hợp thành hơn - một quan hệ "có" (chúng ta đã giới thiệu về sự hợp thành trong mục 2.1). Lớp dẫn xuất có một đối tượng của lớp cơ sở nhưng chúng không biết gì về nó.

Bởi vì sự kế thừa **private** rất giống sự hợp thành nên sử dụng sự hợp thành thay thế thường tốt hơn. Trong đoạn chương trình dưới đây cho thấy điều này, chúng ta chỉ đơn giản cài một đối tượng của lớp **alpha** vào lớp **beta** và quên sự kế thừa đi:

```

class alpha
{ };
class beta
{private:

```



```
alpha obj;
};
```

Đối tượng **alpha** được đặt ở **private** nên nó vẫn được che đậy đối với các đối tượng **beta**. Sự hợp thành làm cho mối quan hệ giữa hai lớp rõ hơn, ít phức tạp hơn và đối với các đối tượng lớp dẫn xuất vẫn làm việc như vậy. Chúng ta sẽ thấy một ví dụ về sự hợp thành trong mục 2.7.

4. Trình biên dịch và sự kế thừa **private**

Trình biên dịch không hiểu rằng sự kế thừa **private** giống sự hợp thành hơn là sự kế thừa. Vì vậy, trình biên dịch sẽ không dễ dãi với các đối tượng lớp dẫn xuất được dùng thay cho các đối tượng lớp cơ sở. Một phiên bản mở rộng của chương trình KINDOF cho thấy sự kế thừa **public** và **private**. Bản chương trình 2-11 là KINDOF2.

Listing 2-11 KINDOF2

```
class alpha                                //lop co so
{
    public:
        void memfunc()                    //ham thanh vien public
        {}
};
class beta:public alpha                    //ke thua public
{};
class gama:private alpha                  //ke thua private
{};

void main()
{
    void anyfunc(alpha);                  //khai bao, ham co mot doi so
    alpha aa;                             //doi tuong kieu alpha
    beta bb;                              //doi tuong kieu beta
    gama gg;                              //doi tuong kieu gama
    bb.memfunc();                         //tot
    gg.memfunc();                         //loi: memfunc() khong truy nhap
    anyfunc(bb);                          //tot, doi tuong beta duoc truyen nhu doi so alpha
    anyfunc(gg);                          //loi: khong the chuyen doi gama thanh alpha
    aa=bb;                                //tot, doi tuong beta duoc gan cho bien alpha
    aa=gg;                                //loi: khong the chuyen doi gama thanh alpha
}
```

Một lớp mới **gama** được kế thừa **private** từ **alpha**. Trình biên dịch sẽ không cho phép chúng ta gọi một hàm thành viên của **alpha** từ một đối tượng **gama**, truyền một đối tượng **gama** tới một hàm có đối số **alpha**, hay gán một đối tượng **gama** cho một biến **alpha**.

2.5.4. Sự kế thừa **protected**

Sự kế thừa **protected** tương tự sự kế thừa **private**: cả hai đều giống sự hợp thành (một quan hệ "có"). Tuy nhiên, các lệnh trong các hàm thành viên của lớp dẫn xuất **protected** có thể truy nhập các thành viên **public** và **protected** trong lớp cơ sở, trái lại - như chúng ta đã thấy - các lệnh trong hàm thành viên của lớp dẫn xuất **private** thì không thể. Sự kế thừa **protected** thường không được sử dụng và nó có trong ngôn ngữ chỉ để cho đầy đủ.

2.5.5. Tóm tắt sự truy nhập

Chúng ta cùng tóm tắt những gì đã bàn về sự truy nhập và các loại kế thừa khác nhau. Bảng 2-3 chỉ ra sự truy nhập tới các thành viên lớp cơ sở (thường là dữ liệu), ở đó các thành viên của lớp dẫn xuất có thể được phép truy nhập. Sự kế thừa **private** làm cho tất cả các thành viên không bao giờ có thể truy nhập.

Bảng 2-3. Sự truy nhập của các hàm thành viên lớp dẫn xuất

	Kế thừa private	Kế thừa protected	Kế thừa public
Dữ liệu lớp cơ sở là private	Không thể truy nhập	Không thể truy nhập	Không thể truy nhập
Dữ liệu lớp cơ sở là protected	Không thể truy nhập	Có thể truy nhập	Có thể truy nhập
Dữ liệu lớp cơ sở là public	Không thể truy nhập	Có thể truy nhập	Có thể truy nhập

Bảng 2-4 chỉ ra sự truy nhập tới các thành viên lớp cơ sở (thường là các hàm) mà ở đó có thể cho phép sự truy nhập đối với các đối tượng lớp dẫn xuất được định nghĩa ngoài mô tả lớp, chẳng hạn như trong hàm main(). Chỉ có các thành viên **public** được kế thừa **public** là có thể truy nhập.

Bảng 2-4. Sự truy nhập của các đối tượng lớp dẫn xuất

	Kế thừa private	Kế thừa protected	Kế thừa public
Hàm lớp cơ sở là private	Không thể truy nhập	Không thể truy nhập	Không thể truy nhập
Hàm lớp cơ sở là protected	Không thể truy nhập	Không thể truy nhập	Không thể truy nhập
Hàm lớp cơ sở là public	Không thể truy nhập	Không thể truy nhập	Có thể truy nhập

2.6. SỰ KẾ THỪA NHIỀU MỨC

Sự kế thừa mà chúng ta nói đến từ trước đến giờ là sự kế thừa hai mức. Còn có sự kế thừa nhiều mức. Không chỉ có lớp **beta** có thể rút ra từ lớp **alpha**, lớp **gama** cũng có thể rút ra từ **beta**, lớp **delta** có thể rút ra từ **gama** v.v...

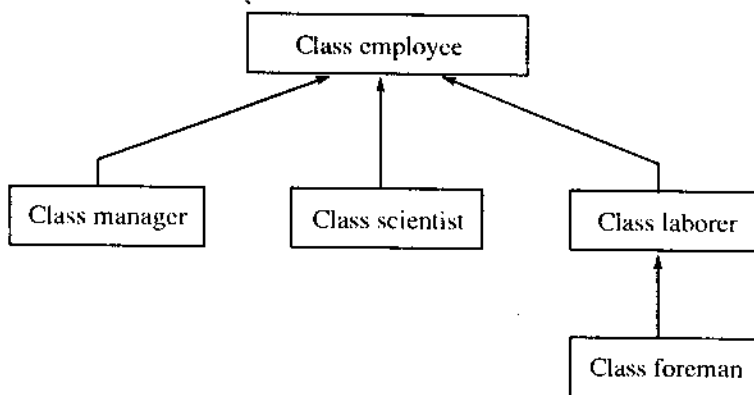
```
class alpha
{ };
class beta:public alpha
{ };
class gama:public beta
{ };
class delta:public gama
{ };
```

Mọi thứ làm việc như mong đợi. Mối quan hệ giữa **delta** và **gama** y như giữa **beta** và **alpha**.

Một lớp có thể truy nhập tới tất cả các lớp tổ tiên của nó. Trong sự kế thừa **public**, các hàm thành viên **delta** có thể truy nhập tới dữ liệu **public** hoặc **protected** trong **gamma**, **beta** và **alpha**. Tất nhiên là chúng không thể truy nhập tới các thành viên của bất kỳ lớp nào trừ của chính nó.

2.6.1. Ví dụ rút ra lớp foreman từ lớp laborer

Trong chương trình EMPINH ở mục 2.2, chúng ta đã kiểm tra một chương trình mà trong đó có một vài nhân viên được rút ra từ lớp nhân viên **employee**. Bây giờ chúng ta thêm một đốc công (**foreman**) vào chương trình này. Một đốc công là một loại công nhân đặc biệt, bởi vậy lớp đốc công **foreman** được rút ra từ lớp công nhân **laborer**, như chỉ ra ở hình 2-6.



Hình 2-6. Phân cấp lớp trong EMPGRAND.

Các đốc công giám sát những hoạt động của công nhân. Họ chịu trách nhiệm về chỉ tiêu sản xuất cho nhóm của họ. Khả năng của một đốc công được đánh giá bằng phần trăm chỉ tiêu sản xuất mà họ đạt được. Mục dữ liệu **quotas** trong **foreman** biểu diễn số phần trăm này (ví dụ, một đốc công thường đạt 72,5% chỉ tiêu). Bản 2-12 là chương trình EMPGRAND.

Listing 2-12 EMPGRAND

```

//empgrand.cpp
//hơn hai mục kế thừa
//foreman rút ra từ laborer
#include<iostream.h>
const int LEN=80;           //độ dài cực đại của tên

class employee               //lớp nhân viên
{
private:
    char name[LEN];          //tên nhân viên
    unsigned long number;    //mã nhân viên
public:
    void getData()
    {
        cout<<"\nNhập vào tên: "; cin>>name;
        cout<<"\nNhập mã nhân viên: "; cin>>number;
    }
    void putData()
    {
        cout<<"\nTên: "<<name;
    }
}
  
```

```

        cout<<"\nMa: "<<number;
    }
};
//-----
class manager:public employee    //lop nguoi quan ly
{
    private:
        char title[LEN];          //chuc vu, vi du giam doc
    public:
        void getData()
        {
            employee::getData();
            cout<<"\nChuc vu: ";cin>>title;
        }
        void putData()
        {
            employee::putData();
            cout<<"\nChuc vu: "<<title;
        }
};
//-----
class scientist:public employee    //lop nha khoa hoc
{
    private:
        int pubs;                //so sach xuat ban
    public:
        void getData()
        {
            employee::getData();
            cout<<"\nSo sach xuat ban: ";cin>>pubs;
        }
        void putData()
        {
            employee::putData();
            cout<<"\nSo sach xuat ban: "<<title;
        }
};
//-----
class laborer:public employee    //lop cong nhan
{
};
//-----
class foreman:public laborer    //lop doc cong
{
    private:
        float quotas;            //phan tram chi tieu dat duoc
    public:
        void getData()
        {
            employee::getData();
            cout<<"\nChi tieu dat duoc (%): "; cin>>quotas;
        }
        void putData()
        {

```

```

        employee::putData();
        cout<<"\nChi tiêu đạt được là: "<<quotas<<' %';
    }
};
//-----
void main()
{
    laborer labo;
    foreman fore;
    cout<<"\n\nNhập dữ liệu cho công nhân:";
    labo.getData();
    cout<<"\n\nNhập dữ liệu cho đốc công:";
    fore.getData();

    cout<<"\n\nDữ liệu của công nhân:";
    labo.putData();
    cout<<"\n\nDữ liệu của đốc công:";
    fore.putData();
}

```

Chú ý rằng một phân cấp lớp không giống như một sơ đồ tổ chức. Một phân cấp lớp do những đặc điểm chung tổng quát tạo thành. Lớp càng tổng quát càng ở vị trí cao trong phân cấp lớp. Lớp công nhân là một kiểu nhân viên tổng quát hơn lớp đốc công, bởi vậy lớp công nhân ở trên lớp đốc công trong sơ đồ phân cấp lớp. Trái lại, trên sơ đồ tổ chức của một công ty, đốc công lại ở trên công nhân bởi vì trên sơ đồ tổ chức, người có quyền lực càng cao thì càng ở vị trí cao trên sơ đồ. Đây là một vài mẫu tương tác với chương trình:

```

Nhập dữ liệu cho công nhân:
Nhập vào tên: Hung
Nhập mã nhân viên: 294504
Nhập dữ liệu cho đốc công:
Nhập vào tên: Bình
Nhập mã nhân viên: 294508
Chỉ tiêu đạt được (%): 90
Dữ liệu của công nhân:
Tên: Hung
Mã: 294504
Dữ liệu của đốc công:
Tên: Bình
Mã: 294508
Chỉ tiêu đạt được là: 909504

```

2.6.2. Ví dụ về hàm tạo

Khi có hơn hai mức kế thừa trong phân cấp lớp thì việc quản lý các hàm tạo có danh sách khởi tạo có thể trở nên hơi vụng về. Bản 2-13 là một ví dụ về các hàm tạo này.

Listing 2-13 INHCON

```

//inhcon.cpp
//kiểm tra các hàm tạo cơ sở trong su kế thừa
#include<iostream.h>
class Gparent
{
private:
    int intv;

```

```

    float flow;
public:
    Gparent(int i,float f):intv(i),flov(f)
    {
    }
    void display()
    {cout<<intv<<" , "<<flov<<" ;"}
};
class Parent:public Gparent
{
private:
    int intv;
    float flow;
public:
    Parent(int i1,float f1,int i2,float f2):
        Gparent(i1,f1),          //khởi tạo Gparent
        intv(i2),flov(f2)        //khởi tạo Parent
    {
    }
    void display()
    {
        Gparent::display();
        cout<<intv<<" , "<<flov<<" ;"
    }
};
class Child:public Parent
{
private:
    int intv;
    float flow;
public:
    Child(int i1,float f1,int i2,float f2,int i3,float f3):
        Parent(i1,f1,i2,f2),      //khởi tạo Parent
        intv(i3),flov(f3)         //khởi tạo Child
    {
    }
    void display()
    {
        Parent::display();
        cout<<intv<<" , "<<flow;
    }
}; //ket thuc mo ta lop
void main()
{
    Child ch(1,1,1,2,2,3,3.3);
    cout<<"\nDu lieu trong ch la = ";
    ch.display();
}

```

Lớp **Child** được kế thừa từ lớp **Parent**, lớp **Parent** lại được kế thừa từ lớp **Gparent**. Mỗi lớp có một mục dữ liệu **int** và một mục dữ liệu **float**. Hàm tạo trong mỗi lớp có đủ đối số để khởi tạo dữ liệu cho lớp đó và lớp tổ tiên. Điều này có nghĩa là có hai đối số cho hàm tạo lớp **Gparent**, bốn đối số cho hàm tạo lớp **Parent** (nó phải khởi tạo **Gparent** và chính nó) và sáu đối số cho **Child** (nó phải khởi tạo **Gparent**, **Parent** và chính nó). Mỗi hàm tạo gọi hàm tạo lớp cơ sở của nó.

Trong hàm **main()**, chúng ta tạo một đối tượng kiểu **Child**, khởi tạo nó với sáu giá trị và sau đó hiển thị nó. Đây là kết quả đưa ra từ chương trình:

Du lieu trong ch = 1 , 1.1 ; 2 , 2.2 ; 3 , 3.3

Đối tượng **Child** ch, đối tượng con **Parent** trong đối tượng **Child** và đối tượng con **Gparent** trong đối tượng con **Parent**, tất cả đều được khởi tạo trước khi hàm tạo **Child** bắt đầu thực hiện. Đó là lý do tại sao tất cả các lời gọi hàm tạo con xuất hiện trên danh sách khởi tạo. Chúng phải được khởi tạo trước dấu ngoặc nhọn mở của hàm tạo đó. Vì vậy, khi hàm tạo bắt đầu thực hiện phải đảm bảo rằng tất cả các đối tượng con có liên quan phải được tạo và khởi tạo.

Ngoài ra, không thể bỏ qua một thế hệ nào khi gọi các hàm tạo tổ tiên trong danh sách khởi tạo. Sự thay đổi sau đây của hàm tạo **Child**:

```
Child(int i1,float f1,int i2,float f2,int i3,float f3):
    Gparent(i1,f1),          //lỗi: không thể khởi tạo Gparent
    intv(i3),flov(f3)
{ }
```

Lời gọi hàm tạo **Gparent** là không hợp lệ bởi vì lớp **Gparent** không phải là lớp cơ sở trực tiếp của **Child**.

2.7. SỰ HỢP THÀNH

Sự hợp thành (composition) là đặt một đối tượng bên trong một đối tượng khác hay đứng về phía lập trình là định nghĩa một đối tượng của một lớp ở bên trong mô tả của một lớp khác. Chúng ta đã nói đến sự hợp thành một vài lần trước đây, ở đó chúng ta so sánh nó với sự kế thừa **private**. Đây là ví dụ về sự hợp thành mà chúng ta đã dùng:

```
class alpha
{ };
class beta
{
    private:
        alpha obj;
};
```

2.7.1. Gán các đối tượng **airtime** vào lớp **flight**

Đây là một ví dụ điển hình về sự hợp thành. Chúng ta đã có nhiều ví dụ về lớp **airtime**, nó biểu diễn các giá trị thời gian giờ và phút. Bây giờ cho rằng chúng ta cần viết một chương trình sắp xếp các chuyến bay và chúng ta muốn làm việc với các đối tượng chuyến bay (ví dụ, chuyến bay 962 tới Atlanta bây giờ đã sẵn sàng mở cửa ở đường bay số 32).

Để đơn giản, chúng ta chỉ để ba mục dữ liệu trong lớp chuyến bay: số chuyến bay, thời gian khởi hành và thời gian đến (được lưu trữ dưới dạng các giá trị **airtime**). Các hàm thành viên của lớp chuyến bay **flight** sẽ nhận các giá trị cho ba mục dữ liệu này từ người sử dụng và hiển thị chúng. Bản 2-14 trình bày chương trình **AIRSCHED**.

Listing 2-14 AIRSCHED

```
//airsched.cpp
//minh hoa su hop thanh: doi tuong airtime trong lop flight
#include<iostream.h>
class airtime
{
    private:
        int hours;          //0 - 23
```

```

    int minutes;        //0 - 59
public:
    airtime():hours(0),minutes(0)
    {
    }
    airtime(int h,int m):hours(h),minutes(m)
    {
    }
    void display() const
    {
        cout<<hours<<": "<<minutes;
    }
    void get()
    {
        char dummy;
        cout<<"\nNhap vao thoi gian (dang 12:59): ";
        cin>>hours>>dummy>>minutes;
    }
};
//-----
class flight
{
private:
    long fnumber;        //so chuyen bay
    airtime departure;    //thoi gian khoi hanh
    airtime arrival;      //thoi gian den
public:
    flight():fnumber(0)//ham tao khong doi so
    {
    }
    void get()            //lay du lieu tu nguoi su dung
    {
        char dummy;
        cout<<"\nNhap vao so chuyen bay: ";
        cin>>fnumber;
        cout<<"\nThoi gian khoi hanh: ";
        departure.get();
        cout<<"\nThoi gian khoi den: ";
        arrival.get();
    }
    void display() const    //dua ra man hinh
    {
        cout<<"\nSo chuyen bay = "<<fnumber;
        cout<<"\nThoi gian khoi hanh = ";
        departure.display();
        cout<<"\nThoi gian khoi den = ";
        arrival.display();
    }
};
//-----
void main()
{
    flight flarr[100];        //mang luu tru 100 doi tuong flight
    int total=0;              //so chuyen bay trong mang
    char ch;                  //cho 'c' hoac 'k'
    do
    {
        //lay du lieu cho chuyen bay
        flarr[total++].get();

```



```

        cout<<"\nCo nhap nua khong(c/k)? ";
        cin>>ch;
    }
    while(ch!='k');
    //hien thi du lieu cho cac chuyen bay
    for(int j=0;j<total;j++)
    {
        cout<<endl;
        flarr[j].display();
    }
}

```

Trong hàm `main()`, chương trình để cho người sử dụng nhập vào dữ liệu cho bao nhiêu chuyến bay tùy thích. Nó lưu các chuyến bay này vào trong một mảng. Sau đó nó hiển thị dữ liệu cho tất cả các chuyến bay trong mảng. Đây là mẫu tương tác với chương trình khi có hai chuyến bay được nhập vào.

Trong chương trình, sự hợp thành dường như là một sự lựa chọn tự nhiên; chúng ta sử dụng `airtime` như các kiểu dữ liệu cơ bản. Chúng ta không sử dụng sự kế thừa để kết nối lớp `airtime` với lớp `flight` bởi vì các đối tượng `airtime` rất giống với các biến thuộc các kiểu dữ liệu cơ bản. Tuy nhiên không phải tất cả các tình huống đều rõ ràng như vậy.

2.7.2. Khi nào sử dụng sự kế thừa tốt hơn sự hợp thành ?

Khi mối quan hệ "loại" giữa các lớp là quan trọng. Cho rằng chúng ta có một mảng kiểu `employee`. Thật là tốt nếu có thể lưu trữ bất kỳ loại nhân viên nào trong mảng này - người quản lý, nhà khoa học, công nhân hay bất kỳ nhân viên nào - như thế này:

```

employee emparray[SIZE];
emparray[0]=laborer1;
emparray[1]=scientist;
emparray[0]=laborer2;
...

```

Cách duy nhất mà chúng ta có thể làm được điều này là kế thừa nhiều loại nhân viên từ lớp `employee`. Sự hợp thành không thể tạo các mối quan hệ cần thiết giữa các lớp (có thể lưu trữ các đối tượng của các lớp khác nhau trong một mảng lớp cơ sở là một đặc điểm quan trọng của C++ nhưng lý do quan trọng sẽ không rõ, phải chờ đến khi chúng ta nói về các hàm ảo).

2.8. SỰ KẾ THỪA BỘI

Sự kế thừa bội xảy ra khi một lớp kế thừa từ hai hay nhiều lớp cơ sở như thế này:

```

class Base1
{ };
class Base2
{ };

class Derv:public Base1,public Base2
{ };

```

`Derv` được rút ra từ cả `Base1` và `Base2`. Trong mô tả của lớp dẫn xuất, các lớp cơ sở được phân cách nhau bằng một dấu phẩy (hoặc nhiều dấu phẩy nếu có hơn hai lớp cơ sở) và mỗi lớp cơ sở có định danh truy nhập riêng của nó. Lớp dẫn xuất `Derv` kế thừa tất cả dữ liệu và các hàm thành viên từ cả hai lớp `Base1` và `Base2`.

CHƯƠNG 3

CON TRỎ

Con trỏ có tiếng là khó hiểu. Trong chương này, chúng ta cùng tìm hiểu về con trỏ, ít nhất là về khái niệm, qua đó chúng ta có thể thấy chúng thực sự không phải là phức tạp.

Phần đầu chúng ta thảo luận về con trỏ trên phương diện lý thuyết thuần túy, hầu hết dùng các kiểu dữ liệu cơ bản làm ví dụ. Sau khi hiểu kỹ về con trỏ, chúng ta cùng xem cách mà con trỏ được dùng điển hình trong môi trường C++.

Một ứng dụng quan trọng của con trỏ là cấp phát động bộ nhớ, được thực hiện trong C++ với từ khóa **new** và **delete**. Vấn đề này sẽ được xem xét trong nhiều ví dụ. Một con trỏ đặc biệt gọi là **this** trỏ tới các đối tượng của riêng nó cũng cần được xem xét ở đây.

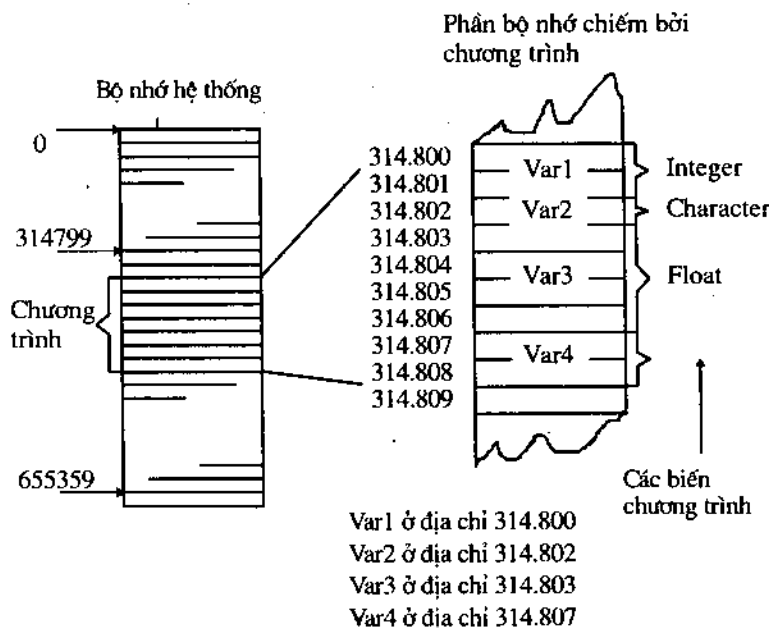
Con trỏ được dùng để tạo những cấu trúc phức tạp lưu trữ dữ liệu.

3.1. ĐỊA CHỈ VÀ CON TRỎ

Ý tưởng cơ bản đằng sau con trỏ không phải phức tạp. Đầu tiên chúng ta cần biết về địa chỉ bộ nhớ và cần biết rằng địa chỉ có thể lưu trữ các biến.

3.1.1. Địa chỉ (hàng con trỏ)

Mỗi byte trong bộ nhớ máy tính có một địa chỉ. Địa chỉ là các số, y như các số nhà trên một phố. Các số bắt đầu từ 0 và từ đó tăng lên 1, 2, 3... Nếu ta có 1MB bộ nhớ thì địa chỉ cao nhất sẽ là 16.777.215. Các số đó là lớn đối với địa chỉ phố nhưng với máy tính lại không phải như vậy.



Hình 3-1. Địa chỉ bộ nhớ.

Bất kỳ chương trình nào khi được nạp vào trong bộ nhớ đều chiếm một khoảng xác định các địa chỉ này. Điều đó có nghĩa là mỗi biến và mỗi hàm trong chương trình bắt đầu từ một địa chỉ cụ thể. Hình 3-1 cho thấy điều này.

3.1.2. Toán tử địa chỉ &

Chúng ta có thể tìm ra địa chỉ chiếm bởi một biến bằng cách dùng toán tử địa chỉ & (address of operator). Bản chương trình 3-1 trình bày một chương trình ngắn VARADDR để minh họa cách làm này:

Listing 3-1 VARADDR

```
//varaddr.cpp
// địa chỉ của các biến
#include<iostream.h>
#include<conio.h>
void main()
{
    int var1=11;           //định nghĩa và khởi tạo các biến
    int var2=12;
    int var3=13;
    cout<<endl<<&var1     //đưa ra địa chỉ các biến
        <<endl<<&var2
        <<endl<<&var3;
    getch();
}
```

Chương trình đơn giản này định nghĩa ba biến nguyên và khởi tạo chúng bằng các giá trị 11, 12, 13. Sau đó đưa địa chỉ của các biến này ra màn hình.

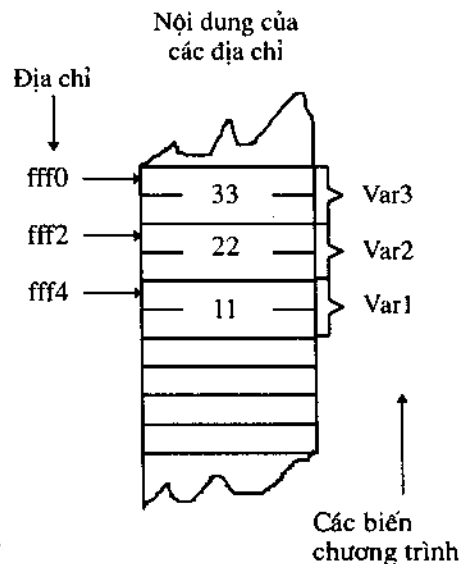
Địa chỉ thực chiếm bởi các biến trong một chương trình tùy thuộc vào nhiều yếu tố, chẳng hạn như loại máy tính mà chương trình đang chạy, kích thước hệ điều hành và liệu có chương trình nào khác đang tồn tại trong bộ nhớ không. Vì những lý do này mà kết quả đưa ra ở các máy có thể khác nhau. Đây là kết quả đưa ra trên một máy tính:

```
0x8f4ffff4 <-----địa chỉ của var1
0x8f4ffff2 <-----địa chỉ của var2
0x8f4ffff0 <-----địa chỉ của var3
```

Nên nhớ rằng địa chỉ của một biến không giống như nội dung của nó. Nội dung của ba biến là 11, 22 và 33. Hình 3-2 cho thấy ba biến này trong bộ nhớ.

Toán tử chèn << đưa ra địa chỉ ở dạng cơ số 16, địa chỉ được chỉ ra bởi tiếp đầu ngữ ox đứng trước mỗi số. Địa chỉ của var1 là 8f4ffff4. Đây là cách thông thường để trình bày địa chỉ bộ nhớ. Chúng ta cần biết là mỗi biến bắt đầu tại một địa chỉ duy nhất. Tuy nhiên, cần chú ý là trong kết quả đưa ra ở trên, mỗi địa chỉ khác địa chỉ kế nó 2 byte. Đó là bởi vì trong chương trình chạy trên máy tính, mỗi số nguyên chiếm 2 byte bộ nhớ.

Nếu dùng biến kiểu **char** chúng ta sẽ có các địa chỉ liên kế nhau, bởi vì kiểu **char** chiếm một byte bộ nhớ; nếu dùng kiểu **double** thì các địa chỉ sẽ cách nhau 8 byte.



Hình 3-2. Địa chỉ và nội dung của nó.

Địa chỉ xuất hiện theo thứ tự giảm dần bởi vì các biến tự động (**automatic variable** - là các biến được định nghĩa trong một hàm) được lưu trữ trên **stack**, nó phát triển theo hướng đi xuống (từ địa chỉ cao đến địa chỉ thấp) trong bộ nhớ. Nếu ta dùng các biến ngoài (**external variable** - được định nghĩa bên ngoài các hàm) thì các biến đó sẽ có địa chỉ tăng dần, bởi vì các biến ngoài được lưu trữ trên **heap**, nó phát triển theo hướng đi lên. Chúng ta không cần phải lo lắng quá nhiều về vấn đề này bởi vì trình biên dịch quản lý chi tiết giúp chúng ta.

3.1.3. Biến con trỏ

Các địa chỉ tự nó hơi hạn chế. Ta có thể tìm ra nơi mà dữ liệu được lưu trong bộ nhớ, nhưng việc đưa ra các địa chỉ thật sự chẳng có ích gì. Khi khả năng lập trình đã cao, yêu cầu cần thêm vào một ý tưởng: các biến lưu trữ các địa chỉ. Chúng ta đã thấy kiểu biến lưu trữ các ký tự, các số nguyên và các số với dấu phẩy động. Một địa chỉ cũng là một số và nó có thể lưu trữ trong một biến. Một biến giữ một giá trị địa chỉ gọi là một biến con trỏ hay đơn giản là con trỏ. Nếu một con trỏ chứa địa chỉ của một biến thì ta có thể nói con trỏ trỏ tới biến đó. Đây là lý do mà nó có tên là con trỏ.

Kiểu dữ liệu của biến con trỏ là gì? Nó không giống với kiểu của biến mà địa chỉ của nó được con trỏ lưu trữ; một con trỏ trỏ tới kiểu **int** không nghĩa là có kiểu **int**. Chúng ta có thể nghĩ tới một kiểu dữ liệu con trỏ được gọi bằng một cái tên gì đó chẳng hạn như **pointer** hay **ptr**. Tuy nhiên những vấn đề này hơi phức tạp.

3.1.4. Con trỏ trỏ tới các kiểu cơ bản

Bản chương trình 3-2, **PTRVAR**, trình bày cú pháp cho các biến con trỏ giữ địa chỉ của các biến nguyên **int**.

Listing 3-2 PTRVAR

```
//ptrvar.cpp
//con trỏ (biến địa chỉ)
#include<iostream.h>
void main()
{
    int var1=11;          //hai biến nguyên

    int var2=22;
    cout<<endl<<&var1 //đưa ra địa chỉ của các biến
        <<endl<<&var2;
    int* ptr;             //con trỏ trỏ tới các số nguyên
    ptr=&var1;             //trỏ tới var1
    cout<<endl<<ptr;       //đưa ra giá trị của con trỏ
    ptr=&var2;             //trỏ tới var2
    cout<<endl<<ptr;       //đưa ra giá trị của con trỏ
}
```

Chương trình này định nghĩa hai biến nguyên, **var1** và **var2** và khởi tạo chúng bằng các giá trị 11 và 22. Sau đó đưa ra địa chỉ của chúng.

Tiếp theo chương trình định nghĩa một biến con trỏ trong dòng lệnh:

```
int* ptr;
```

Đây có thể là một cú pháp hơi đặc biệt. Dấu sao (*) có nghĩa là con trỏ trỏ tới. Do đó, đọc từ phải sang trái, câu lệnh định nghĩa biến con trỏ **ptr** là một con trỏ trỏ tới **int**. Đây là một cách khác của việc nói rằng biến này có thể giữ địa chỉ của biến nguyên.

Vậy có gì không đúng với ý tưởng kiểu con trỏ đa năng có thể giữ địa chỉ của bất kỳ kiểu dữ liệu nào? Ví dụ, giả sử gọi nó là pointer thì ta có thể viết:

```
pointer ptr3;
```

Vấn đề là trình biên dịch cần biết loại biến mà con trỏ trỏ tới (lý do sẽ rõ khi chúng ta bàn về con trỏ và mảng ở phần sau). Những người thiết kế trình biên dịch đã tìm ra một cụm tên mới như `pointer_to_int` và `pointer_to_char`, nhưng họ đã sử dụng một ký hiệu ngắn gọn hơn cho phép chỉ cần học một ký hiệu mới. Đây là cú pháp dùng trong C++. Nó cho phép khai báo một con trỏ trỏ tới bất kỳ kiểu dữ liệu nào, chỉ cần dùng tên kiểu và dấu sao:

```
char* cptr;  
int* iptr;  
float* fptr;  
...
```

Dấu sao (trong ngữ cảnh này) là viết tắt của con trỏ hay đầy đủ hơn, con trỏ trỏ tới kiểu bên trái của nó.

3.1.5. Những ý kiến khác nhau về cú pháp

Nhiều người lập trình viết các định nghĩa con trỏ với các dấu sao gần biến hơn tên kiểu:

```
char *cptr;
```

Điều này không có vấn đề gì với trình biên dịch, nhưng đặt dấu sao gần tên kiểu nhấn mạnh rằng dấu sao là phần của tên kiểu (con trỏ trỏ tới char), không phải là phần của tên biến.

Nếu định nghĩa hơn một con trỏ của cùng một kiểu trên một dòng thì chỉ cần chèn tên kiểu chỉ một lần nhưng phải đặt các dấu sao trước mỗi tên biến:

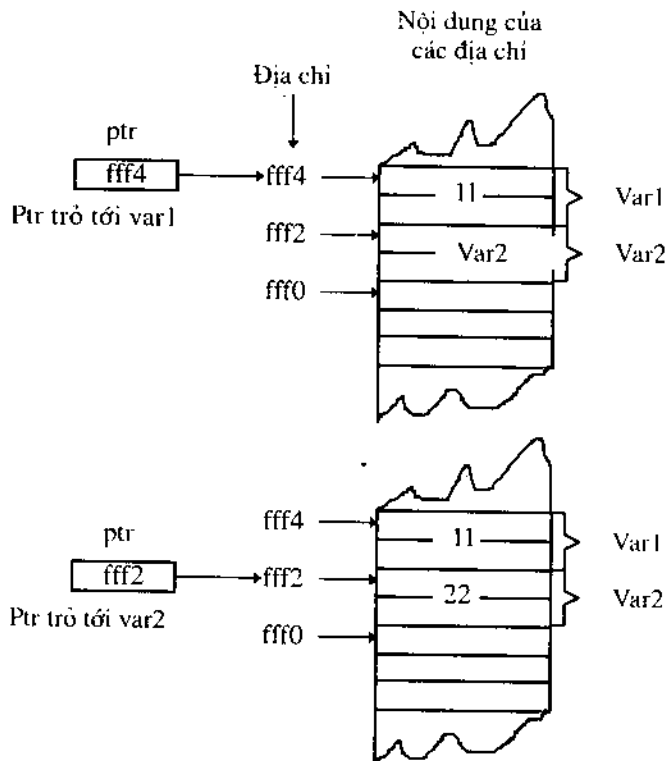
```
char* ptr1,* ptr2,* ptr3; //ba  
//biến kiểu char hoặc có thể dùng  
//dấu sao cạnh tên biến:
```

```
char *ptr1,*ptr2,*ptr3; //ba  
//biến kiểu char:
```

3.1.6. Con trỏ phải có giá trị

Một địa chỉ như `0x8f4ffff4` có thể xem như một hằng con trỏ. Khi một biến được đặt vào một địa chỉ cụ thể trong bộ nhớ nó sẽ không di chuyển được (nếu có thì do sự thay đổi của hệ điều hành, kết quả người lập trình không thấy). Bởi vậy các giá trị địa chỉ sẽ là hằng chỉ cần chương trình tiếp tục chạy.

Mặt khác, một con trỏ như `ptr` có thể xem như một biến con trỏ. Y như biến nguyên `var1` có thể được gán giá trị hằng 11, biến con trỏ `ptr` cũng có thể được gán giá trị hằng `0x8f4ffff4`.



Hình 3-3. Thay đổi giá trị trong ptr.

Khi định nghĩa một biến lần đầu tiên, nó không giữ giá trị (trừ khi khởi tạo nó). Nó có thể chứa một giá trị vô nghĩa. Trong trường hợp con trỏ, giá trị vô nghĩa này là một địa chỉ của cái gì đó trong bộ nhớ nhưng không phải là những gì chúng ta muốn. Bởi vậy, trước khi một con trỏ được dùng, phải gán một địa chỉ xác định cho nó. Trong chương trình PTRVAR, ptr lúc đầu được gán địa chỉ của var1 trong dòng lệnh:

```
ptr=&var1; //đặt địa chỉ của var1 vào ptr
```

Sau đó chương trình đưa ra giá trị chứa trong ptr, nó giống địa chỉ đưa ra với &var1. Tiếp đến, cùng một biến con trỏ ptr lại được gán địa chỉ của var2 và giá trị này được đưa ra màn hình. Hình 3-3 chỉ ra hoạt động của chương trình PTRVAR.

3.1.7. Con trỏ trỏ tới đối tượng

Các đối tượng được lưu trong bộ nhớ máy tính, bởi vậy con trỏ có thể trỏ tới các đối tượng y như có thể trỏ tới các biến của các kiểu dữ liệu cơ bản. Ví dụ, ta có thể định nghĩa một đối tượng và sau đó đặt địa chỉ của nó vào trong một con trỏ:

```
employee emp1;           <-----định nghĩa đối tượng của lớp employee
employee* ptrobj=&emp1; <-- đặt địa chỉ của đối tượng trong ptrobj
```

Biến **ptrobj** thuộc kiểu **employee***, hoặc là con trỏ trỏ tới **employee**. Bây giờ chúng ta cùng viết lại chương trình PTRVAR để làm việc với các đối tượng của lớp **employee**. Bản 3-3 trình bày chương trình PTROBJ.

Listing 3-3 PTROBJ

```
//ptrobj.cpp
//con trỏ trỏ tới các đối tượng
#include<iostream.h>
#include<string.h>           //cho strcpy()
class employee               //lớp nhân viên
{
private:
    enum {LEN=30};           //độ dài của tên
    char name[LEN];          //tên nhân viên
    unsigned long number;    //mã nhân viên
public:                       //hàm tạo hai đối số
    employee(char* na,unsigned long nu):number(nu)
    {strcpy(name, na);}
};
////////////////////////////////////
void main()
{
    employee emp1("Thang",123123L);
    employee emp2("Binh",234234L);

    cout<<"\nCác giá trị địa chỉ";
    cout<<endl<<&emp1           //đưa ra các giá trị địa chỉ
    <<endl<<&emp2;
    employee* ptr;             //con trỏ trỏ tới employee
    cout<<"\nCác giá trị con trỏ";
    ptr=&emp1;                 //trỏ tới emp1
    cout<<endl<<ptr;            //đưa ra giá trị con trỏ
    ptr=&emp2;                 //trỏ tới emp2
    cout<<endl<<ptr;            //đưa ra giá trị con trỏ
}
```

Hoạt động của chương trình này tương tự như chương trình PTRVAR. Tuy nhiên, từ kết quả đưa ra ta thấy các đối tượng **employee** chiếm nhiều bộ nhớ hơn các biến **int**.

Kết quả như sau:

```
cac gia tri dia chi
    ox29072222
    ox29072200
cac gia tri con tro
    ox29072222
    ox29072200
```

Nếu trừ 222 cho 200 ta được 22 ở cơ số 16, nó bằng 34 ở cơ số 10. Kết quả này là có nghĩa bởi vì một đối tượng **employee** chứa một xâu 30 ký tự và một số **unsigned long** 4 byte.

3.1.8. Truy nhập các biến được trỏ tới

Giả sử ta không biết tên của một biến nhưng lại biết địa chỉ của nó. Vậy liệu có truy nhập được tới nội dung của biến đó không?

Có một cú pháp đặc biệt để truy nhập tới giá trị của một biến bằng cách dùng địa chỉ của nó thay cho tên. Bản 3-4, PTRACC, cho thấy nó được làm như thế nào.

Listing 3-4 PTRACC

```
//ptracc.cpp
//truy nhap bien duoc tro toi
#include<iostream.h>
void main()
{
    int var1=11;           //hai bien nguyen
    int var2=22;
    int* ptr;              //con tro tro toi cac so nguyen

    ptr=&var1;              //tro toi var1
    cout<<"\nVar1="<<*ptr;
    ptr=&var2;              //tro toi var2
    cout<<"\nVar2="<<*ptr;
}
```

Chương trình này rất giống chương trình PTRVAR nhưng thay vì đưa ra giá trị địa chỉ trong ptr, nó đưa ra giá trị nguyên được lưu giữ tại địa chỉ được lưu trong ptr. Kết quả như sau:

```
Var1=11
Var2=22
```

Biểu thức truy nhập các biến **var1** và **var2** là ***ptr**, nó xuất hiện trong các lệnh **cout** ở trên.

Khi dấu sao được dùng ở bên trái của một tên biến con trỏ, như trong biểu thức ***ptr**, nó được gọi là toán tử gián tiếp. Nó có nghĩa là giá trị của biến được trỏ tới bởi biến con trỏ ở bên phải nó. Khi **ptr** được gán địa chỉ của **var1**, biểu thức ***ptr** có giá trị 11 bởi vì đó là giá trị của **var1**. Khi **ptr** được thay đổi bằng địa chỉ của **var2** thì biểu thức ***ptr** có giá trị là 22 bởi vì **var2** là 22. Toán tử gián tiếp đôi khi được gọi là toán tử nội dung (**contents of operator**). Hình 3-4 cho thấy hoạt động này.

Chúng ta không chỉ dùng con trỏ để hiển thị giá trị của một biến mà còn có thể dùng nó để thực hiện bất kỳ hoạt động gì mà ta muốn thực hiện trực tiếp trên biến. Bản 3-5 trình bày chương trình PTRTO, ở đó sử dụng một con trỏ để gán một giá trị cho một biến và sau đó gán giá trị đó cho biến khác.

Listing 3-5 PTRTO

```
//ptrto.cpp
//truy nhập su dung con tro
#include<iostream.h>
void main()
{
    int var1,var2;      //hai bien nguyen
    int* ptr;           //con tro tro toi cac so nguyen
    ptr=&var1;          //tro toi var1
    *ptr=37;            //giong var1=37
    var2=*ptr;          //giong nhu var2=var1
    cout<<endl<<var1 //dua ra gia tri cua con tro
        <<endl<<var2;
}
```

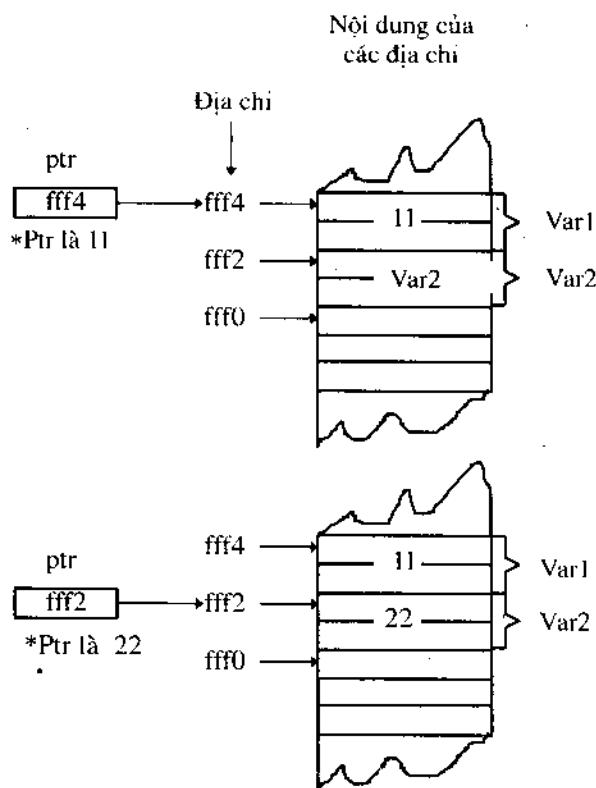
Nhớ rằng dấu sao được dùng như toán tử gián tiếp có ý nghĩa khác dấu sao được dùng để khai báo biến con trỏ. Toán tử gián tiếp đứng trước tên biến và có nghĩa là giá trị của biến được trỏ tới bởi con trỏ. Dấu sao được dùng trong khai báo có nghĩa là con trỏ trỏ tới.

```
int* ptr;    //khai báo
*ptr=37;     //gián tiếp
```

Việc sử dụng toán tử gián tiếp (**indirection operator**) để truy nhập giá trị được lưu trong một địa chỉ được gọi là sự định địa chỉ gián tiếp (**indirection addressing**), hay đôi khi gọi là sự tham chiếu ngược.

Sau đây là tóm tắt những gì chúng ta đã biết từ trước tới nay:

```
int v;        //định nghĩa biến v
              //thuộc kiểu int
int* p;       //định nghĩa p là con
              //trỏ trỏ tới int
p=&v;         //gán địa chỉ của biến v
              //cho con trỏ v
v=3;         //gán 3 cho v
*p=3;        //cũng gán 3 cho v
```



Hình 3-4. Truy nhập qua con trỏ.

Hai câu lệnh cuối cho thấy sự khác nhau giữa sự định địa chỉ thông thường và sự định địa chỉ gián tiếp, ở đó tham chiếu tới cùng một biến dùng con trỏ giữ địa chỉ của nó.

Hai cách này gần giống việc giao một lá thư tới một người nào đó. Nếu lái xe tới nhà người đó và nhét thư vào trong nhà thì đó là sự định địa chỉ trực tiếp. Cũng có thể viết địa chỉ trên phong bì và đặt lá thư đó vào hộp thư công cộng. Người đưa thư sẽ đọc địa chỉ và xem lá thư đó tới nơi nào. Đó là sự định địa chỉ gián tiếp bởi vì nó trực tiếp bởi người thứ ba.

3.1.9. Con trỏ trỏ tới kiểu void

Trước khi tiếp tục bàn về các con trỏ, chúng ta cần biết một kiểu đặc biệt của các kiểu dữ liệu con trỏ. Thông thường, địa chỉ đặt vào một con trỏ phải giống kiểu con trỏ. Chúng ta không thể gán địa chỉ của một biến **float** cho một con trỏ trỏ kiểu **int**. Tuy nhiên, có một ngoại lệ đối với quy tắc này. Đó là một loại con trỏ đa năng có thể trỏ tới bất kỳ kiểu dữ liệu nào. Nó được gọi là con trỏ trỏ tới kiểu **void** và được định nghĩa như sau:

```
void* ptr;           //ptr có thể trỏ tới kiểu dữ liệu bất kỳ.
```

Những con trỏ như vậy có những kiểu sử dụng đặc biệt, chẳng hạn truyền các con trỏ tới các hàm tương tác trên một vài kiểu dữ liệu khác nhau.

Ví dụ tiếp theo sử dụng một con trỏ trỏ tới **void** cũng cho thấy, nếu không dùng **void** thì phải cẩn thận khi gán cho con trỏ một địa chỉ cùng kiểu với con trỏ. Bản 3-6 trình bày chương trình **PTRVOID**.

Listing 3-6 PTRVOID

```
//ptrvoid.cpp
//ptr trỏ tới void
#include<iostream.h>
void main()
{
    int intvar;           //biến nguyên
    float flovar;         //biến số dấu phẩy động
    int* ptring;          //định nghĩa con trỏ trỏ tới int
    float* ptrflo;        //định nghĩa con trỏ trỏ tới float
    void* ptrvoid;        //định nghĩa con trỏ trỏ tới void
    ptring=&intvar;        //được, gán int* cho int*
    //ptring=&flovar;      //lỗi, gán float* cho int*
    //ptrflo=&intvar;      //lỗi, gán int* cho float*
    ptrflo=&flovar;        //được, gán float* cho float*
    ptrvoid=&flovar;       //được, gán float* cho void*
    ptrflo=&intvar;        //được, gán float* cho void*
}
```

Chúng ta có thể gán địa chỉ của **intvar** cho **ptring** bởi vì chúng đều là kiểu **int***, nhưng không thể gán địa chỉ của **flovar** cho **ptring** bởi vì kiểu đầu tiên là kiểu **float*** còn kiểu thứ hai là **int***. Tuy nhiên kiểu **ptrvoid** có thể được gán với bất kỳ kiểu con trỏ nào, chẳng hạn **int*** hoặc **float***, bởi vì nó là một con trỏ trỏ tới **void**.

Bất kỳ lúc nào có thể cũng nên tránh dùng kiểu **void**. Luôn nhớ rằng trong C++ con trỏ chỉ chứa địa chỉ của một kiểu xác định là một biện pháp quan trọng để tránh các lỗi lập trình. Sử dụng kiểu **void** mất đi đặc điểm an toàn này. Tuy nhiên, đôi khi nó cũng quan trọng như chúng ta sẽ thấy.

3.2. CON TRỎ, MẢNG VÀ HÀM

Con trỏ có thể được sử dụng trong nhiều tình huống. Hai tình huống thú vị là khi con trỏ được sử dụng để truy nhập các phần tử mảng và khi chúng được sử dụng làm các đối số hàm. Kết hợp hai tình huống này với nhau ta sẽ thấy con trỏ rất hữu ích khi mảng được truyền như đối số hàm.

3.2.1. Con trỏ và mảng

Giữa con trỏ và mảng có một sự liên kết chặt chẽ. Đoạn chương trình sau sẽ ôn lại về mảng:

```
int intarray[5]={31,54,77,52,93};    //mảng
```

```
for(int j=0;j<5;j++)
    cout<<endl<<intarray[j]; //dua ra cac phan tu cua mang
```

Lệnh trên đưa ra lần lượt các phần tử. Ví dụ khi $j = 3$, biểu thức `intarray[j]` có giá trị là `intarray[3]`, nó là phần tử thứ 4, số 52. Đây là kết quả của chương trình:

```
31
54
77
52
93
```

1. Phần tử mảng và ký hiệu con trỏ

Chúng ta sẽ thật ngạc nhiên khi các phần tử của mảng có thể được truy nhập bằng cách dùng ký hiệu con trỏ. Ví dụ tiếp theo, như ví dụ trước nhưng dùng ký hiệu con trỏ:

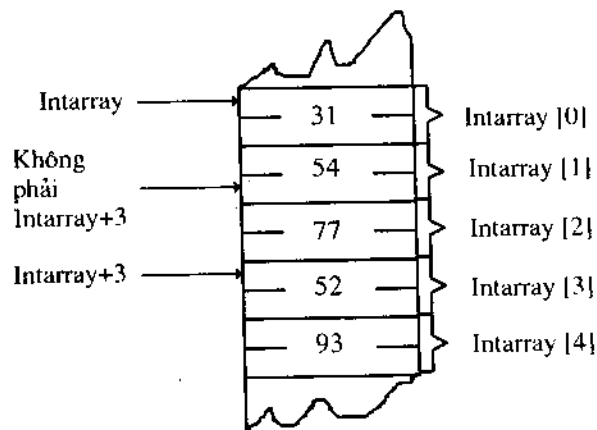
```
int intarray[5]={31,54,77,52,93}; //mang
for(int j=0;j<5;j++)
    cout<<endl<<*(intarray + j); //dua ra cac phan tu cua mang
```

Biểu thức `*(intarray + j)` ở đây có tác dụng y như `intarray[j]` trong ví dụ trước và kết quả đưa ra hoàn toàn giống nhau. Nhưng chúng ta giải thích biểu thức `*(intarray + j)` như thế nào? Cho rằng $j = 3$, lúc đó biểu thức sẽ là `*(intarray + 3)`. Chúng ta muốn biểu thức này biểu diễn phần tử thứ tư của mảng (tức 52).

Nhớ rằng tên của một mảng biểu diễn địa chỉ của nó. Bởi vậy biểu thức `intarray + j` là một địa chỉ mà đã cộng gì đó với nó. Chúng ta có thể muốn 3 byte được cộng vào `intarray` nhưng cách đó không tạo ra kết quả chúng ta muốn: `intarray` là một mảng số nguyên và 3 byte cộng vào mảng này sẽ ở giữa phần tử thứ hai, nó chẳng có nghĩa gì. Chúng ta muốn số nguyên thứ tư trong mảng chứ không phải byte thứ tư, như trong hình 3-5 chỉ ra.

Trình biên dịch C++ đủ thông minh để lấy kích thước của dữ liệu làm số lượng khi nó thực hiện phép tính số học trên các địa chỉ của dữ liệu. Nó biết rằng `intarray` là một mảng kiểu `int` bởi vì nó được khai báo như vậy. Khi nó thấy biểu thức `intarray+3`, nó hiểu đây là địa chỉ của số nguyên thứ tư trong mảng `intarray` chứ không phải byte thứ tư.

Nhưng chúng ta lại muốn giá trị của phần tử thứ tư này chứ không phải địa chỉ. Để có được giá trị chúng ta sử dụng toán tử gián tiếp `*` (**indirection operator**). Biểu thức thu được khi $j = 3$ là `*(intarray+3)`, đây là nội dung của phần tử mảng thứ tư, tức 52.



Hình 3-5. Địa chỉ và nội dung của nó.

Bây giờ chúng ta đã biết tại sao một khai báo con trỏ phải bao gồm cả kiểu của biến được trỏ tới. Trình biên dịch cần biết một con trỏ là con trỏ trỏ tới `int` hay con trỏ trỏ tới `float` để nó thực hiện đúng phép toán khi truy nhập tới các phần tử của mảng. Nó nhân giá trị của chỉ số với kích thước của kiểu dữ liệu được trỏ tới.

2. Hằng con trỏ và biến con trỏ

Cho rằng thay vì cộng `j` vào `intarray` để đi qua các địa chỉ của mảng, chúng ta sử dụng toán tử tăng một đơn vị `++` (**increment operator**). Chúng ta có thể viết `*(intarray++)` được không?

Câu trả lời là không, lý do là chúng ta không thể tăng một hằng (hay thay đổi nó bằng bất kỳ cách nào). Biểu thức `intarray` là địa chỉ mà ở đó hệ thống đã chọn để đặt mảng. Mảng sẽ vẫn còn ở địa chỉ này cho đến khi chương trình kết thúc; do đó, `intarray` là một hằng. Chúng ta nói `intarray++` có khác gì nói `7++`.

Tuy nhiên, mặc dù chúng ta không thể tăng một địa chỉ nhưng chúng ta có thể tăng một con trỏ giữ một địa chỉ. Đoạn chương trình sau cho thấy cách này:

```
int intarray[5]={31,54,77,52,93}; //mang
int* pprint;                      //con trỏ trỏ tới int
pprint=intarray
for(int j=0;j<5;j++)
    cout<<endl<<*(pprint++);      //đưa ra các phần tử của mảng
```

Biến `pprint` bắt đầu với địa chỉ giống `intarray`, do đó cho phép phần tử mảng đầu tiên, `intarray[0]`, có giá trị 31, được truy nhập trước. Nhưng bởi vì `pprint` là một biến, không phải là một hằng, nên nó có thể tăng được. Sau khi nó được tăng, nó trỏ tới phần tử mảng thứ hai là `intarray[1]`. Biểu thức `*(pprint++)` lúc đó biểu diễn nội dung của phần tử mảng thứ hai là 54. Vòng lặp làm cho biểu thức truy nhập lần lượt từng phần tử của mảng.

3.2.2. Con trỏ và hàm

Có ba cách truyền các đối số tới một hàm: truyền theo giá trị, truyền theo tham chiếu và truyền theo con trỏ. Nếu hàm có ý định thay đổi các biến trong chương trình gọi nó thì các biến này không truyền theo giá trị bởi vì hàm sẽ có duy nhất một bản sao của biến. Tuy nhiên, một đối số tham chiếu hoặc con trỏ được dùng để thay đổi biến trong chương trình gọi nó.

1. Truyền các biến đơn giản

Đầu tiên chúng ta ôn lại cách truyền các đối số theo tham chiếu và sau đó so sánh cách này với cách truyền các đối số theo con trỏ. Đoạn chương trình sau trình bày cách truyền theo tham chiếu:

```
void main()
{
    void centimize(double&);           //khai báo hàm
    double var=10.0;                  //biến có giá trị 10
    cout<<endl<<"var="<<var<<"inches";
    centimize(var);                   //chuyển sang centimeter
    cout<<endl<<"var="<<var<<"centimeters";
}
void centimize(double& v)              //định nghĩa hàm
{ v *=v * 2.54;                       //v y như var
}
```

Ở đây ta muốn chuyển đổi một biến `var` trong hàm `main()` từ `inch` sang `centimet`, nên ta đã truyền biến đó theo tham chiếu tới hàm `centimize()` (nhớ rằng dấu `&` theo sau kiểu dữ liệu `double` trong khai báo hàm này chỉ ra rằng đối số được truyền theo tham chiếu). Hàm `centimize()` nhân biến ban đầu với giá trị 2.54. Lưu ý cách mà hàm tham chiếu tới biến đó, nó chỉ đơn giản đưa vào tên đối số `v`; `v` và `var` là hai tên khác nhau của cùng một biến.

Một khi nó đã chuyển đổi `var` sang `centimeter`, `main()` hiển thị kết quả. Đây là kết quả đưa ra:

var=25.4 centimeters

Ví dụ tiếp theo chỉ ra một tình huống tương tự có sử dụng con trỏ:

```
void main()
{
    void centimize(double*);           //khai báo
    double var=10.0;                  //biến có giá trị 10
    cout<<endl<<"var="<<var<<"inches";
    centimize(&var);                  //chuyển sang centimeter
    cout<<endl<<"var="<<var<<"centimeters";
}

void centimize(double* ptrd)          //định nghĩa hàm
{
    *ptrd *=2.54;                     //ptrd giống như var
}
```

Kết quả đưa ra giống như trước. Hàm **centimize()** được khai báo có một đối số là một con trỏ tới **double**:

```
void centimize(double* );
```

Khi hàm **main()** gọi hàm này, nó cung cấp địa chỉ của một biến làm đối số:

```
centimize(&var);
```

Nhớ rằng đây không phải là biến như trong truyền theo tham chiếu mà là địa chỉ của biến. Bởi vì hàm **centimize()** được truyền một địa chỉ, nó phải dùng toán tử gián tiếp, ***ptrd**, để truy nhập tới giá trị lưu trong địa chỉ này:

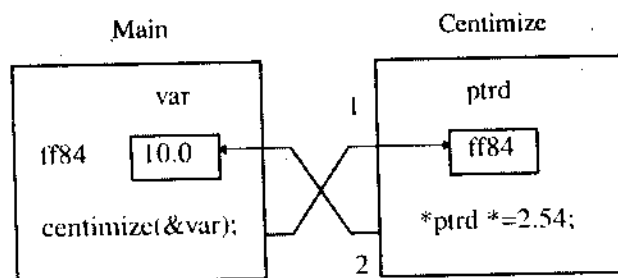
```
*ptrd *=2.54;           //nhân nội dung của ptrd với 2.54
```

Tất nhiên câu lệnh này giống như

```
*ptrd = *ptrd * 2.54;
```

Dấu sao đứng một mình có nghĩa là phép nhân.

Bởi vì **ptr** chứa địa chỉ của **var** nên bất kỳ hành động gì được thực hiện với ***ptrd** cũng được thực hiện với **var**. Hình 3-6 cho thấy việc thay đổi **ptrd** trong chương trình gọi như thế nào.



- 1- main() truyền địa chỉ của var cho ptrd trong centimize().
- 2- centimize() dùng địa chỉ để truy nhập var.

Hình 3-6. Con trỏ được truyền tới hàm.

Truyền con trỏ như một đối số tới hàm trong một số trường hợp giống truyền một tham chiếu. Cả hai đều cho phép biến trong chương trình gọi bị thay đổi bởi hàm. Tuy nhiên cơ chế thì khác nhau. Một tham chiếu là một bí danh của biến ban đầu, trái lại một con trỏ lại là địa chỉ của biến ban đầu.

2. Truyền mảng như các đối số

Chúng ta đã biết về mảng được truyền như các đối số tới hàm và các phần tử của chúng được hàm truy nhập. Điều này được thực hiện bằng cách dùng ký hiệu mảng. Tuy nhiên dùng ký hiệu con trỏ thay cho ký hiệu mảng khi truyền mảng tới hàm thông dụng hơn nhiều. Đoạn chương trình sau cho thấy cách này:

```
#include<iostream.h>
#include<conio.h>
const int MAX=5;
void main()
{
    void centimize(double*);
    double varray[MAX]={ 10.0,43.1,95.9,59.7,87.3 };
    centimize(varray);           //chuyen cac phan tu mang sang cm
    for(int j=0;j<MAX;j++)
        cout<<endl<<"varray["<<j<<"]="<<varray[j]<<"cm";
    getch();
}
void centimize(double* ptrd)
{
    for(int j=0;j<MAX;j++)
        *ptrd++ *=2.54;
}
```

Khai báo cho hàm `centimize()` giống như trong ví dụ ở phần trước, đối số của hàm là một con trỏ tới `double`. Trong ký hiệu mảng nó được viết như sau:

```
void centimize(double[]);
```

Đó là vì ở đây `double*` tương đương với `double[]` mặc dù cú pháp con trỏ được dùng thông dụng hơn.

Vì tên của mảng là địa chỉ của mảng nên không cần toán tử địa chỉ & khi gọi hàm:

```
centimize(varray);    //truyen dia chi mang
```

Trong `centimize()`, địa chỉ mảng này được đặt vào biến `ptrd`. Để trỏ lần lượt tới từng phần tử của mảng chúng ta chỉ cần tăng `ptrd`.

Kết quả đưa ra như sau:

```
varr[0]=25.4    cm
varr[1]=109.474 cm
varr[2]=243.586 cm
varr[3]=151.638 cm
varr[4]=221.742 cm
```

Đây là một câu hỏi về cú pháp: Làm sao biết được biểu thức `*ptrd++` tăng con trỏ mà không tăng nội dung của con trỏ? Nói cách khác trình biên dịch giải thích như `*(ptr++)`, điều chúng ta mong muốn, hay như `(*ptr)++`? Biết rằng dấu sao * và toán tử ++ có cùng mức ưu tiên. Tuy nhiên, các toán tử có cùng mức ưu tiên được xét theo cách thứ hai: sự kết hợp. Sự kết hợp liên quan tới việc

trình biên dịch thực hiện các phép toán bắt đầu từ bên phải hay bên trái. Nếu một nhóm toán tử có sự kết hợp từ phải thì trình biên dịch thực hiện phép toán bên phải của biểu thức trước, sau đó đến các phép toán bên trái. Các toán tử một ngôi như * và ++ có sự kết hợp từ bên phải, do đó biểu thức được giải thích là *(ptrd++), nó tăng con trỏ chứ không phải tăng những gì con trỏ trỏ tới. Nghĩa là con trỏ được tăng lên trước, sau đó toán tử gián tiếp * được áp dụng cho kết quả này.

3.2.3. Con trỏ và chuỗi

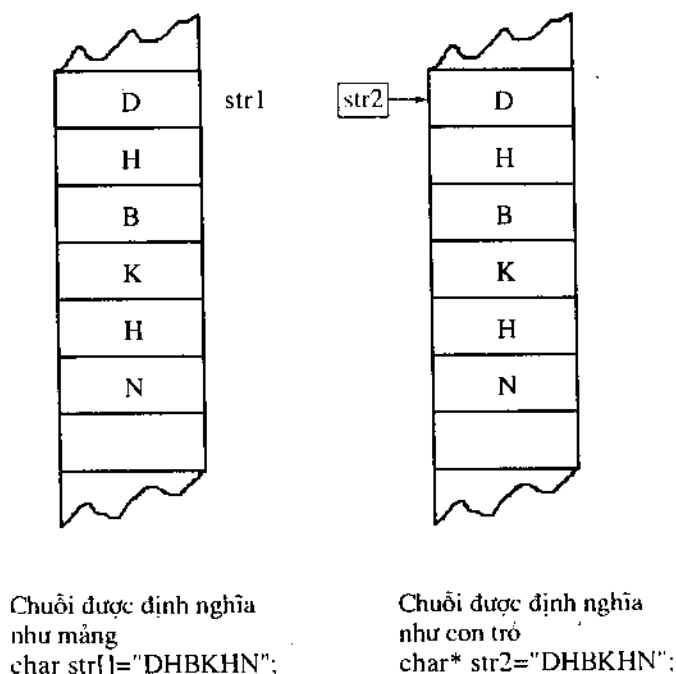
Chúng ta biết chuỗi chỉ đơn giản là các mảng ký tự. Bởi vậy, ký hiệu con trỏ có thể áp dụng cho các ký tự y như có thể áp dụng cho bất kỳ mảng nào. Đây là một thuật ngữ thông dụng trong C++. Trong phần này chúng ta cùng xem xét một vài khía cạnh của mối quan hệ giữa chuỗi và con trỏ.

1. Con trỏ trỏ tới hằng chuỗi

Đây là một ví dụ trong đó định nghĩa hai chuỗi, một dùng ký hiệu mảng như chúng ta đã biết, và một dùng ký hiệu con trỏ.

```
char st1[]="Định nghĩa là một mảng";
char* st2="Định nghĩa là một con trỏ";
cout<<endl<<st1;           //hiển thị hai chuỗi
cout<<endl<<st2;
//st1++;                    //không được, st1 là một hằng
st2++;                      //được, st2 là một con trỏ
cout<<endl<<st2;           //bây giờ st2 bắt đầu từ "inh nghĩa..."
```

Trong nhiều trường hợp hai kiểu hằng chuỗi này tương đương nhau. Chúng ta có thể đưa ra màn hình cả hai chuỗi như trong ví dụ, sử dụng chúng làm đối số hàm v.v... Nhưng có một sự khác nhau tinh tế là: **st1** là một địa chỉ - nghĩa là một hằng con trỏ - trái lại **st2** là một biến con trỏ. Bởi vậy, **st2** có thể thay đổi được còn **st1** thì không. Hình 3-7 cho thấy hai loại chuỗi này trong bộ nhớ.



Hình 3-7. Chuỗi như mảng và con trỏ.

Chúng ta có thể tăng `st2` bởi vì nó là một con trỏ, nhưng khi tăng xong nó không trỏ tới ký tự đầu tiên của chuỗi nữa. Đây là kết quả đưa ra:

Dinh nghĩa la mot mang
Dinh nghĩa la mot con tro
inh nghĩa la mot con tro

Một chuỗi định nghĩa là một con trỏ linh động hơn một chuỗi được định nghĩa là một mảng. Các ví dụ sau đây sử dụng sự mềm dẻo này.

2. Chuỗi là đối số của hàm

Đây là một ví dụ chỉ ra một chuỗi được sử dụng như một đối số hàm. Hàm `dispstr()` chỉ đơn giản đưa ra chuỗi được định nghĩa trong hàm `main()` bằng cách truy nhập lần lượt từng ký tự:

```
void main()
{
    void dispstr(char*);           //khai bao
    char str[]="DHBKHN";
    dispstr(str);                 //hien thi chuoi
}

void dispstr(char* ps)
{
    cout<<endl;                  //bat dau dong moi
    while(*ps)                    //cho den ky tu null
        cout<<*ps++;             //dua ra ky tu
}
```

Địa chỉ `str` được dùng làm đối số trong lời gọi hàm `dispstr()`. Địa chỉ này là một hằng nhưng bởi vì nó được truyền theo giá trị, một bản copy của nó được tạo ra trong hàm `dispstr()`. Bản copy này là một con trỏ, `ps`. Một con trỏ có thể thay đổi được nên hàm này đã tăng `ps` để hiển thị từng ký tự trong chuỗi. Biểu thức `*ps++` trả về lần lượt các ký tự. Vòng lặp quay vòng cho đến khi nó tìm thấy ký tự `null` (`'\0'`) ở cuối chuỗi. Bởi vì ký tự này có giá trị 0, nó biểu diễn `false`, nên vòng lặp `while` kết thúc ở đó. Tiết kiệm khoảng trống như vậy rất thông dụng trong C++ (trong C còn thông dụng hơn).

3. Copy một chuỗi dùng con trỏ

Chúng ta cùng xét một ví dụ về con trỏ lấy được các giá trị ký tự từ một chuỗi. Con trỏ cũng có thể được sử dụng để chèn các ký tự vào một chuỗi. Đoạn chương trình sau minh họa một hàm copy một chuỗi tới một chuỗi khác:

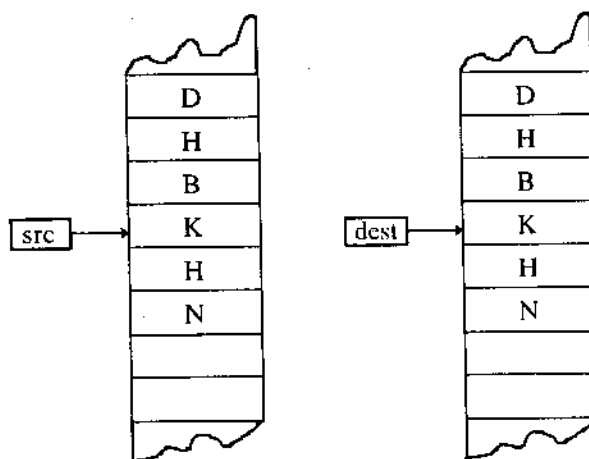
```
void main()
{
    void copystr(char*,char*);    //khai bao
    char* str1="DHBKHN";
    char str2[80];                //chuoi rong
    copystr(str2,str1);           //copy str1 toi str2
}

void copystr(char* dest,char* src)
{
    while(*src)                   //cho den ky tu null
        *dest++ = *src++;
    *dest='\0';
}
```

Ở đây, hàm **main()** gọi hàm **copystr()** để copy **str1** tới **str2**. Trong hàm này biểu thức

```
*dest++ = *src++;
```

lấy giá trị ký tự ở địa chỉ được trỏ tới bởi **src** và đặt nó vào địa chỉ được trỏ tới bởi **dest**. Sau đó cả hai con trỏ cùng tăng để lần tiếp theo của vòng lặp ký tự tiếp theo được copy. Vòng lặp kết thúc khi gặp ký tự **null** trong **src**; tại thời điểm này **null** được chèn vào **dest** và hàm kết thúc. Hình 3-8 cho thấy các con trỏ di chuyển qua các chuỗi như thế nào.



```
*Dest++ = *src++;
```

Hình 3-8. Copy một chuỗi dùng con trỏ.

4. Hàm thư viện chuỗi

Nhiều hàm thư viện mà chúng ta đã sử dụng cho các chuỗi có đối số chuỗi được xác định bằng ký hiệu con trỏ.

Lấy ví dụ, xem mô tả hàm **strcpy()** trong file tiêu đề **STRING.H**. Hàm này copy một chuỗi tới một chuỗi khác, chúng ta có thể so sánh nó với hàm **copystr()** mà chúng ta đã tạo ra ở trên. Đây là một cú pháp của hàm **strcpy()** điển hình:

```
char* strcpy(char* dest, const char* src);
```

Hàm này có hai đối số kiểu **char***. Tác dụng của định danh **const** trong đối số thứ hai là gì? Nó chỉ ra rằng **strcpy()** không thể thay đổi các ký tự được trỏ tới bởi **src**. Điều đó không có nghĩa là con trỏ **src** không thể thay đổi. Để con trỏ không thể thay đổi thì khai báo đối số là :

```
char* const src;
```

Chúng ta sẽ nói kỹ về **const** và **pointer** trong mục 3.5. Hàm **strcpy()** cũng trả về con trỏ trỏ tới **char**, đây là địa chỉ của chuỗi **dest**.

5. Mảng con trỏ trỏ tới các chuỗi

Y như việc có các mảng của các biến kiểu **int** hoặc **float**, cũng có mảng của các con trỏ. Một ứng dụng thường gặp của cấu trúc này là mảng của các con trỏ trỏ tới các chuỗi.

Cũng có mảng của các chuỗi. Nhưng như chúng ta đã thấy, bất lợi của việc dùng một mảng chuỗi là các mảng con giữ các chuỗi phải cùng độ dài, bởi vậy sẽ lãng phí bộ nhớ khi các chuỗi cần lưu trữ ngắn hơn độ dài của các mảng con (hình 3-10).

Chúng ta cùng xem cách dùng con trỏ để giải quyết vấn đề này. Đây là một mảng con trỏ trỏ tới các chuỗi hơn là một mảng của các chuỗi:

```
char* arrptrs[DAYS] = {"Sunday", "Monday", "Tuesday",  
                       "Wednesday", "Thursday", "Friday",  
                       "Saturday"};
```

```
for(int j=0; j<DAYS; j++)
```

```
    cout<<arrptrs[j]<<endl;
```

Đây là kết quả của đoạn chương trình này:

```
Sunday
```

```
Monday
```


Tuesday
Wednesday
Thursday
Friday
Saturday

Khi các chuỗi không là thành phần của mảng, C++ đặt chúng cạnh nhau trong bộ nhớ để không lãng phí bộ nhớ. Tuy nhiên, phải có một mảng lưu trữ các con trỏ trỏ tới các chuỗi để tìm ra chúng. Một chuỗi là một mảng kiểu **char**, bởi vậy một mảng các con trỏ trỏ tới các chuỗi là một mảng các con trỏ trỏ tới **char**. Đó là ý nghĩa của định nghĩa **arrptrs** trong chương trình trên. Bây giờ nhớ lại rằng một chuỗi luôn luôn được biểu diễn bởi một địa chỉ: địa chỉ của ký tự đầu tiên trong chuỗi. Các địa chỉ này được lưu trữ trong mảng. Hình 3-9 cho thấy điều này.

Mảng con trỏ là một cấu trúc mạnh trong C++.

6. Toán tử truy nhập thành viên (membership operator)

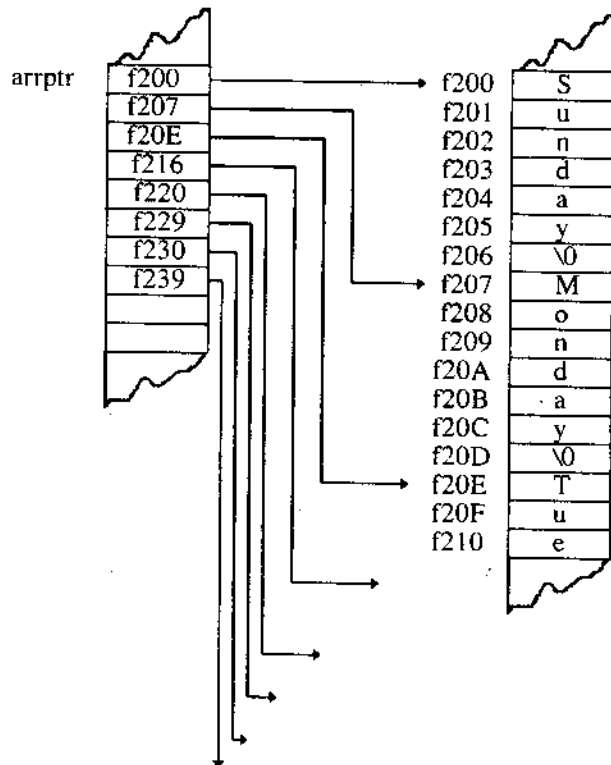
Chủ đề tiếp theo mặc dù không hoàn toàn liên quan tới chuỗi nhưng nên đề cập trước khi nói đến phần tiếp theo. Đây là một câu hỏi: giả sử có một con trỏ trỏ tới một đối tượng, liệu có thể truy nhập các hàm thành viên của đối tượng đó chỉ sử dụng con trỏ? Ví dụ sau minh họa cách làm này. Đầu tiên dùng cách thông thường, sử dụng chính tên đối tượng và toán tử chấm. Sau đó là cách dùng con trỏ.

Listing 3-7 DASHGRAT

```
//dashgrat.cpp
//minh họa việc truy nhập tới các thành viên dùng con trỏ
//toán tử gạch ngang lớn hơn (->)
#include<iostream.h>
class English //lớp đơn vị anh
{
private:
    int feet;
    float inches;
public:
    void getdist() //lay chieu dai tu nguoi su dung
    {
        cout<<"\nNhap vao feet: "; cin>>feet;
        cout<<"\nNhap vao inches: "; cin>>inches;
    }
    void showdist() //hien thi chieu dai
    {
        cout<<feet<<"'-"<<inches<<"'";
```

Mảng con trỏ

Các chuỗi



Hình 3-9. Mảng con trỏ trỏ tới các chuỗi.

```

    }
};
void main()
{
    English edist;           //tao doi tuong English
    English* ptreng=&edist;  //tao con tro tro toi doi tuong

    edist.getdist();         //truy nhap cac thanh vien doi tuong
    edist.showdist();        //voi toan tu cham
    ptreng->getdist();        //truy nhap cac thanh vien doi tuong
    ptreng->showdist();       //voi toan tu cham
}

```

Đây là kết quả của chương trình:

```

Nhap vao feet:4
Nhap vao inches:5.8
4'-5.8"

```

Nếu sử dụng toán tử truy nhập thành viên (.) với các con trỏ như các đối tượng thì sẽ không làm việc.

```
Ptreng.edist();    //không làm việc, ptreng không phải là đối tượng
```

Toán tử chấm yêu cầu định danh bên trái nó phải là một đối tượng. Bởi vì **ptreng** là một con trỏ trỏ tới đối tượng nên chúng ta cần một cú pháp khác. Có một cách khác là lấy đối tượng được trỏ tới bởi con trỏ:

```
(*ptreng).getdist(); //được nhưng không đẹp
```

Biểu thức ***ptreng** là một đối tượng, không phải một con trỏ, bởi vậy có thể dùng nó với toán tử chấm. Tuy nhiên cú pháp này hơi phức tạp. Một cách ngắn gọn hơn được thực hiện bởi toán tử truy nhập thành viên(->), một dấu gạch ngang và một dấu lớn hơn.

```
Ptreng->getdist(); //cách này tốt hơn
```

Toán tử -> làm việc với con trỏ giống như toán tử chấm làm việc với đối tượng.

3.3. QUẢN LÝ BỘ NHỚ VỚI NEW VÀ DELETE

Cơ chế được sử dụng rộng rãi nhất cho việc lưu trữ một số lượng lớn các biến và các đối tượng là mảng. Chúng ta đã gặp nhiều ví dụ trong đó các mảng được sử dụng để cấp phát bộ nhớ. Câu lệnh sau :

```
int arr[100];
```

cấp phát bộ nhớ cho 100 số nguyên. Mảng là một cấu trúc lưu trữ dữ liệu hữu ích nhưng có một vài trở ngại: chúng ta phải biết tại thời điểm viết chương trình mảng sẽ lớn bao nhiêu, không thể đợi cho đến khi chương trình chạy để xác định kích thước mảng. Cách sau đây không làm việc:

```
cin>>size;    //lấy kích thước từ người sử dụng
int arr[size]; //lỗi: kích thước mảng phải là hằng
```

Trình biên dịch yêu cầu kích thước mảng phải là hằng.

Trong nhiều tình huống chúng ta không biết phải cần bao nhiêu bộ nhớ cho đến khi chương trình chạy. Có thể chúng ta muốn để cho người sử dụng nhập vào số đối tượng nhân viên nhưng chúng ta không thể đoán được người sử dụng muốn nhập bao nhiêu.

3.3.1. Toán tử new

C++ cung cấp một cách khác để có được các khối bộ nhớ: toán tử **new** cấp phát bộ nhớ có kích thước xác định từ hệ điều hành và trả về con trỏ trỏ tới điểm bắt đầu của khối nhớ đó. Đoạn chương trình sau dùng **new** để lấy bộ nhớ cho một chuỗi:

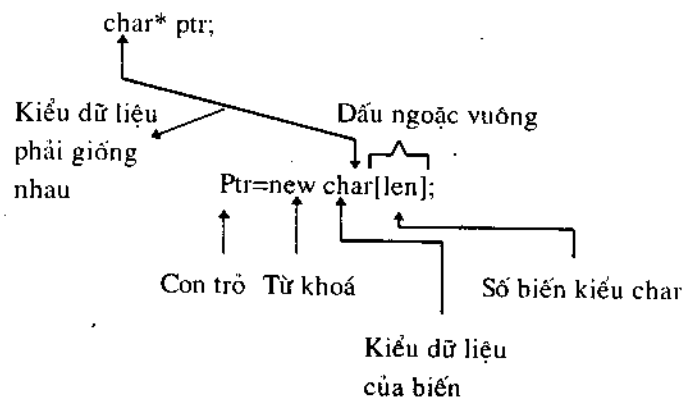
```
char* str="Ha Noi la thu do";  
int len=strlen(str);           //lay chieu dai cua str  
char* ptr;                     //tao mot con tro tro toi char  
ptr=new char[len+1];           //cap phat bo nho: size=str+'\0'  
strcpy(ptr,str);               //copy str toi vung nho moi  
cout<<"ptr="<<ptr;           //cho thay str bay gio o trong ptr  
delete[] ptr;                  //giai phong vung nho cua ptr
```

Trong biểu thức :

```
ptr=new char[len+1];
```

theo sau từ khóa **new** là kiểu của biến được cấp phát bộ nhớ và hai dấu ngoặc vuông, bên trong hai dấu ngoặc vuông là số lượng biến. Ở đây, chương trình cấp phát bộ nhớ cho **len+1** biến **char**, trong đó **len** là độ dài của **str** (tìm ra nhờ hàm thư viện **strlen()**); cộng thêm 1 để tạo một byte phụ cho ký tự **null** kết thúc chuỗi.

Toán tử **new** trả về một con trỏ trỏ tới điểm bắt đầu của vùng nhớ. Hình 3-10 chỉ ra cú pháp của lệnh **new** và con trỏ trỏ tới vùng nhớ nó lấy được.



Hình 3-10. Cú pháp của toán tử new.

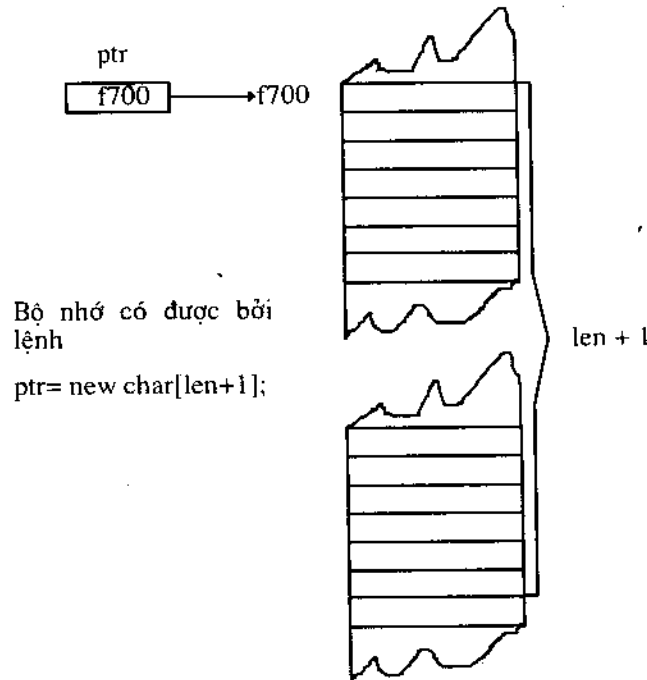
Khi sử dụng **new** nhớ dùng hai dấu ngoặc vuông bao quanh kích thước số; trình biên dịch sẽ không phản đối nếu dùng dấu ngoặc tròn () nhưng kết quả sẽ không đúng. Hình 3-11 chỉ ra bộ nhớ thu được bởi **new** và con trỏ trỏ tới nó.

Trong chương trình trên, hàm **strcpy()** được dùng để copy chuỗi **str** tới vùng nhớ mới tạo được **ptr** trỏ tới. Bởi vì vùng nhớ này có kích thước bằng độ dài của **str** nên chuỗi hoàn toàn vừa khít. Tiếp theo chương trình đưa ra nội dung của **ptr**, kết quả là:

```
ptr=Ha Noi la thu do
```

Toán tử **new** lấy bộ nhớ một cách tự động, nghĩa là trong khi chương trình đang chạy. Bộ nhớ được cấp phát từ vùng nhớ có tên là **heap** (đôi khi gọi là **free store**). **Heap** là vùng nhớ thứ ba được

sử dụng nhiều trong các chương trình C++. Chúng ta đã biết các biến động (automatic variable) được lưu trữ trong **Stack** và các biến ngoài (external variable) được lưu trữ trong **static storage area**.



Hình 3-11. Bộ nhớ thu được bởi toán tử new.

Những người lập trình C sẽ nhận ra rằng **new** tương đương họ hàm thư viện **malloc()**. Tuy nhiên **new** tốt hơn nhiều. Khi sử dụng **new** với các đối tượng, nó không chỉ cấp phát bộ nhớ cho các đối tượng mà còn tạo ra đối tượng với mục đích gọi hàm của đối tượng đó. Điều này đảm bảo đối tượng đó được khởi tạo đúng, rất cần để tránh các lỗi lập trình. Ngoài ra, **new** trả về một con trỏ trỏ tới kiểu dữ liệu thích hợp trong khi đó **malloc()** trả về con trỏ phải ép trỏ tới kiểu thích hợp. Nên dùng **new** cho các đối tượng, không bao giờ sử dụng **malloc()**.

3.3.2. Toán tử delete

Nếu chương trình chiếm nhiều vùng nhớ do **new** tạo ra thì cuối cùng tất cả các biến bộ nhớ đều bị chiếm và hệ thống sẽ bị phá hủy. Để đảm bảo an toàn và sử dụng có hiệu quả bộ nhớ, người ta tìm ra một toán tử tương ứng với **new** là **delete** để giải phóng bộ nhớ trả lại hệ điều hành. Trong chương trình câu lệnh :

```
delete[] ptr;
```

trả lại hệ điều hành tất cả các vùng nhớ do **new** trỏ tới.

Thực ra trong ví dụ này không cần đến **delete**, bởi vì bộ nhớ được tự động giải phóng khi chương trình kết thúc. Tuy nhiên, nếu cho rằng sử dụng **new** trong một hàm và nếu hàm đó sử dụng một biến cục bộ là con trỏ trỏ tới vùng nhớ mới được yêu cầu thì khi hàm kết thúc, con trỏ sẽ bị hủy nhưng vùng nhớ vẫn bị chiếm bởi chương trình. Vùng nhớ đó trở nên cô lập và chiếm một không gian nhớ mà không bao giờ có thể truy nhập. Bởi vậy, tốt nhất là xóa vùng nhớ khi không sử dụng đến nữa.

Xóa vùng nhớ nhưng không xóa con trỏ trỏ tới nó (trong ví dụ là **ptr**) và không thay đổi giá trị địa chỉ trong con trỏ. Tuy nhiên địa chỉ này không còn hợp lệ nữa; bộ nhớ nó trỏ tới có thể thay đổi

thành một cái gì đó hoàn toàn khác. Bởi vậy, không được sử dụng con trỏ trỏ tới vùng nhớ khi không còn dùng nữa.

Dấu ngoặc vuông theo sau **delete** trong ví dụ chỉ ra rằng đang xóa một mảng. Nếu tạo một biến đơn độc với **new** thì không cần dấu ngoặc vuông để xóa nó.

```
ptr= new int; //cấp phát bộ nhớ cho biến đơn int
```

```
.....
```

```
delete ptr; //không có dấu ngoặc vuông sau delete
```

Tuy nhiên, không được quên dấu ngoặc vuông khi xóa mảng đối tượng. Dùng chúng để đảm bảo rằng tất cả các thành phần của mảng được xóa và hàm hủy được gọi cho mỗi thành phần. Nếu quên dấu ngoặc vuông thì chỉ có phần tử đầu được xóa.

3.3.3. Lớp chuỗi dùng **new**

Toán tử **new** thường xuất hiện trong hàm tạo (constructor). Để có ví dụ chúng ta cùng viết một lớp **xString**. Ví dụ sau sử dụng **new** để có được lượng bộ nhớ chính xác.

Listing 3-8 NEWSTR

```
//newptr.cpp
//sử dụng new để lấy bộ nhớ cho chuỗi
#include<iostream.h>
#include<string.h> //cho strcpy()...
class xString
{
private:
    char* str; //con trỏ trỏ tới chuỗi
public:
    xString(char* s) //hàm tạo một đối số
    {
        int length=strlen(s); //chiều dài của đối số chuỗi
        str=new char[length+1]; //cấp phát bộ nhớ
        strcpy(str,s);
    }
    ~xString() //hàm hủy
    {
        delete[] str;
    }
    void display() //hiển thị chuỗi
    {
        cout<<str;
    }
};
void main()
{
    //sử dụng hàm tạo một đối số
    xString s1("Lớp xString sử dụng toán tử new");
    cout<<endl<<"s1=";
    s1.display();
}
```

Kết quả của chương trình là:

```
s1=Lớp xString sử dụng toán tử new
```

Lớp **xString** chỉ có một mục dữ liệu: một con trỏ trỏ tới **char**, **str**. Con trỏ này sẽ trỏ tới chuỗi thông thường do đối tượng **xString** lưu trữ. Tuy nhiên, không có mảng trong đối tượng để lưu trữ chuỗi. Chuỗi được lưu trữ ở một nơi khác; chỉ có con trỏ trỏ tới nó là thành viên.

3.3.4. Hàm hủy trong chương trình NEWSTR

Từ trước tới nay trong các ví dụ không có nhiều hàm hủy, nhưng bây giờ cấp phát bộ nhớ với **new**, các hàm hủy (destructor) trở nên quan trọng hơn. Nếu chúng ta cấp phát bộ nhớ khi tạo một đối tượng thì việc giải phóng bộ nhớ khi không cần đến đối tượng nữa là rất cần thiết. Cần nhớ lại rằng một hàm hủy là một hàm được gọi tự động khi đối tượng bị hủy. Hàm hủy trong NEWSTR như sau:

```
~xString()  
{  
    delete[] str;  
}
```

Hàm hủy này trả lại hệ thống vùng nhớ đã lấy khi đối tượng được tạo. Các đối tượng (giống như các biến khác) được hủy khi hàm mà trong đó chúng được định nghĩa kết thúc. Hàm này đảm bảo cho vùng nhớ chiếm bởi đối tượng **xString** sẽ trả lại hệ thống.

Nếu nó không trả lại cho hệ thống thì vùng nhớ như thế sẽ không thể truy nhập được nữa (ít nhất là đến khi hệ thống được khởi động lại) và sẽ tạo thành một "lỗ thủng bộ nhớ", nếu lặp lại nhiều sẽ phá hủy hệ thống.

Có một đáp ứng sai trong việc dùng hàm hủy như trong chương trình NEWSTR. Nếu copy một đối tượng **xString** tới một đối tượng **xString** khác, ví dụ như câu lệnh `s2 = s1;` thì thực sự chỉ có một con trỏ trỏ tới chuỗi được copy, bởi vì con trỏ là dữ liệu duy nhất có trong đối tượng. Cả hai đối tượng bây giờ cùng trỏ tới một đối tượng trong bộ nhớ. Nhưng nếu bây giờ ta xóa đi một đối tượng chuỗi **xString**, hàm hủy sẽ xóa chuỗi `char*` làm cho con trỏ kia không còn hợp lệ nữa. Vấn đề này có thể khó nhận ra vì các đối tượng có thể xóa bằng nhiều cách không rõ ràng.

3.3.5. Tạo đối tượng với new

Các đối tượng, cũng như các biến kiểu dữ liệu cơ bản, có thể được tạo với toán tử **new**. Chúng ta đã có nhiều các ví dụ về các đối tượng được định nghĩa và cấp phát bộ nhớ bởi trình biên dịch:

```
English dist;           //định nghĩa
```

Đây là một đối tượng có tên là **dist** và được định nghĩa là của lớp **English**. Tuy nhiên, đôi khi chúng ta không biết tại thời điểm viết chương trình chúng ta cần tạo bao nhiêu đối tượng. Gặp trường hợp này nên dùng **new** để tạo các đối tượng trong khi chương trình đang chạy. Chúng ta đã biết **new** trả về một con trỏ trỏ tới đối tượng không tên. Bản 3-9 là một chương trình ngắn minh họa cách sử dụng này của **new**.

Listing 3-9 ENGLPTR

```
//englptr.cpp  
//truy nhập hàm thành viên bang con trỏ  
#include<iostream.h>  
class English  
{  
private:  
    int feet;  
    float inches;  
public:  
    void getdist()           //lay khoang cach tu nguoi su dung  
    {  
        cout<<"\nNhap feet:"; cin>>feet;  
        cout<<"\nNhap inches:"; cin>>inches;  
    }  
    void showdist()
```

```

        {
            cout<<feet<<"\'-"<<inches<<"\''";
        }
    };

void main()
{
    English* distptr;           //con tro tro toi English

    distptr= new English;       //con tro tro toi doi tuong English moi tao
    distptr->getdist();          //truy nhap thanh vien cua doi tuong voi
    distptr->showdist();          //toan tu new

    delete distptr;             //xoa doi tuong khoi bo nho
}

```

Chương trình tạo một đối tượng kiểu **English** sử dụng toán tử **new** và trả về một con trỏ trỏ tới nó tên là **distptr**. Sử dụng con trỏ này với toán tử **->** để truy nhập **getdist()** và **showdist()**. Đây là kết quả của chương trình:

```

    Nhap feet:5
    Nhap inches:5.6
    5'-5.6"

```

Chú ý rằng ngay trước khi chương trình kết thúc, nó xóa đối tượng mà **new** lấy được trước đó bằng **delete**. Đây luôn luôn là cách tốt nhất để đảm bảo rằng tất cả những gì được cấp phát với **new** cuối cùng đều được giải phóng bởi **delete**.

3.3.6. Mảng con trỏ trỏ tới các đối tượng

Một cấu trúc OOP thông dụng là mảng con trỏ trỏ tới các đối tượng. Cấu trúc này cho phép dễ dàng truy nhập tới một nhóm các đối tượng và linh động hơn là đặt chính các đối tượng vào trong một mảng. Bản 3-10 tạo một mảng con trỏ trỏ tới lớp **person**.

Listing 3-10 PTROBJS

```

//ptroajs.cpp
//mang con tro tro toi cac doi tuong
#include<iostream.h>
class person           //lop nguoi
{
    protected:
        char name[40];    //ten nguoi
    public:
        void getName()    //lay ten
        {
            cout<<"\nNhap vao ten:";
            cin>>name;
        }
        void printName()
        {
            cout<<"\nTen la "<<name;
        }
};

void main()

```

```

{
    person* persPtr[100];           //mang con tro tro toi doi tuong
    int n=0;                         //so nguoi trong mang
    char choice;

    do
    {
        persPtr[n]=new person;      //dat mot nguoi vao mang
        persPtr[n++]->getName();    //lay ten
        cout<<"Co nhap nua khong(c/k)?";
        cin>>choice;
    }
    while(choice=='c' || choice=='C');

    for(int j=0;j<n;j++)             //dua tat ca ra man hinh
    {
        cout<<"\nNguoi thu "<<j+1;
        persPtr[j]->printName();
    }

    //xoa tat ca khoi bo nho
    while(n) delete persPtr[--n];
}

```

Lớp **person** có một mục dữ liệu, **name**, lưu giữ một chuỗi biểu diễn tên của một người. Hai hàm thành viên **getName()** và **printName()** cho phép thiết lập tên và hiển thị chúng.

Cuối chương trình, vòng lặp **while** xóa các đối tượng được trở tới bởi tất cả các con trỏ trong mảng **persPtr**.

3.3.7. Hoạt động của chương trình

Hàm **main()** định nghĩa một mảng, **persPtr**, của một trăm con trỏ trỏ tới kiểu **person**. Trong vòng lặp **do**, nó sử dụng **new** để tạo một đối tượng **person**. Tiếp theo, nó yêu cầu người sử dụng nhập vào tên và cho đối tượng mới tạo tên này. Các con trỏ trỏ tới đối tượng **person** được lưu trữ trong mảng **persPtr**. Để chứng minh việc truy nhập tới các đối tượng dùng con trỏ thật là dễ, hàm **main()** đưa ra dữ liệu **name** cho mỗi đối tượng **person** chỉ đơn giản bằng cách sử dụng vòng lặp **for**.

Đây là vài mẫu tương tác với chương trình:

Nhap vao ten: Binh
Co nhap nua khong(c/k)?c

Nhap vao ten: Thang
Co nhap nua khong(c/k)?c

Nhap vao ten: Hung
Co nhap nua khong(c/k)?k

Nguoi thu 1
Ten la Binh
Nguoi thu 2
Ten la Thang
Nguoi thu 3
Ten la Hung

3.3.8. Truy nhập hàm thành viên

Mỗi phần tử của mảng **persPtr** được xác định bằng ký hiệu mảng là **persPtr[j]** (tương ứng với ký hiệu con trỏ ***(persPtr+j)**). Các phần tử là các con trỏ trỏ tới các đối tượng kiểu **person**. Để truy nhập tới các thành viên của đối tượng sử dụng con trỏ ta dùng toán tử **->**. Kết hợp tất cả lại chúng ta có cú pháp truy nhập hàm **getName()**:

```
persPtr[j] ->getName();
```

Câu lệnh này thực hiện hàm **getName()** của đối tượng **person** được trỏ tới bởi phần tử **j** của mảng **persPtr**.

3.4. THIS VÀ CONST

Trong phần này chúng ta cũng bàn về hai chủ đề ngắn nhưng ít liên quan: con trỏ **this** mà là một hàm thành viên có thể truy nhập tới đối tượng gọi nó và **const** được sử dụng với các con trỏ.

3.4.1. Con trỏ this

Các hàm thành viên của mọi đối tượng có thể truy nhập tới một loại con trỏ kỳ diệu có tên là **this**, nó trỏ tới chính đối tượng đó. Bởi vậy, bất kỳ hàm thành viên nào cũng có thể khám phá ra địa chỉ của đối tượng gọi nó. Bản 3-11 là một ví dụ ngắn, **WHERE**, cho thấy cơ chế này.

Listing 3-11 WHERE

```
//where.cpp
//con trỏ this
#include<iostream.h>

class where
{
private:
    char chararray[10];           //chiếm 10 byte
public:
    void reveal()
    { cout<<"\nĐịa chỉ của doi tuong la:"<<this; }
};

void main()
{
    where w1,w2,w3;              //tao 3 doi tuong
    w1.reveal();                  //xem chung o dau
    w2.reveal();
    w3.reveal();
}
```

Hàm **main()** trong ví dụ này tạo ra 3 đối tượng kiểu **where**. Sau đó chương trình yêu cầu một đối tượng đưa ra địa chỉ của nó dùng hàm thành viên **reveal()**. Hàm này đưa ra giá trị của con trỏ **this**. Đây là kết quả của chương trình:

```
Địa chỉ của doi tuong la:0x168728ce
Địa chỉ của doi tuong la:0x168728c4
Địa chỉ của doi tuong la:0x168728ba
```

Bởi vì dữ liệu trong mỗi đối tượng gồm có một mảng 10 byte nên mỗi đối tượng chiếm một phần bộ nhớ 10 byte (ce trừ đi c4 bằng 10 thập phân, ba trừ đi c4 cũng vậy).

Chú ý rằng con trỏ **this** không có trong các hàm thành viên tĩnh (static), bởi vì hàm thành viên tĩnh không liên quan đến một đối tượng cụ thể nào mà liên quan đến toàn bộ lớp.

1. Truy nhập dữ liệu thành viên với lớp

Khi gọi một hàm thành viên, nó sẽ tồn tại với con trỏ **this** của nó được thiết lập bằng địa chỉ của đối tượng. Con trỏ **this** có thể được đối xử như bất kỳ con trỏ nào trỏ tới một đối tượng và do đó có thể được dùng để truy nhập dữ liệu trong đối tượng mà nó trỏ tới, như chỉ ra trong ví dụ DOTHIS.

Listing 3-12 DOTHIS

```
//dothis.cpp
//con trỏ this tham chiếu tới du lieu
#include<iostream.h>
class what
{
private:
    int alpha;
public:
    void test()
    {
        this->alpha=11;
        cout<<"alpha="<<this->alpha;
    }
};
void main()
{
    what w;           //tao mot doi tuong
    w.test();
}
```

Kết quả là:

alpha=11

Chương trình này chỉ đơn giản đưa ra giá trị 11. Hàm thành viên **test()** truy nhập tới biến **alpha** sử dụng biểu thức:

this->alpha;

Nó có tác dụng hoàn toàn giống như truy nhập trực tiếp **alpha**. Cú pháp này làm việc nhưng nó không có mục đích gì ngoài việc chỉ ra rằng con trỏ **this** thực sự trỏ tới **w**, đối tượng gọi hàm thành viên **test()**.

2. Sử dụng **this** để trả về các giá trị

Một cách sử dụng thiết thực đối với con trỏ **this** là trả về các giá trị từ các hàm thành viên và cho các toán tử chồng. Con trỏ **this** làm gì trong những tình huống như vậy?

Trả về tham chiếu của đối tượng luôn luôn là một ý tưởng tốt, bởi vì điều này tránh việc tạo ra các đối tượng không cần thiết. Tuy nhiên, chúng ta không thể trả về một biến tự động (**automatic variable**) - định nghĩa trong hàm - theo tham chiếu, bởi vì các biến như vậy bị phá hủy khi chương trình kết thúc và tham chiếu trả về sẽ tham chiếu tới những gì không còn tồn tại nữa.

Có gì đó tồn tại lâu hơn một đối tượng cục bộ nếu trả nó về theo tham chiếu thì thật tốt. Việc trả về một đối tượng mà gọi một hàm có gì đáng chú ý? Một đối tượng sẽ tồn tại lâu hơn các hàm thành viên của nó. Các hàm thành viên được tạo ra và phá hủy mỗi khi chúng được gọi, nhưng đối tượng vẫn tồn tại cho đến khi bị phá hủy bởi tác động bên ngoài (như bị xóa). Do đó, trả về đối tượng gọi

hàm là một ý kiến tốt hơn trả về một đối tượng trung gian được tạo ra trong hàm đó. Con trỏ **this** làm việc này dễ dàng.

Bản 3-13 là chương trình ASSIGN, trong đó hàm operator=() trả về đối tượng gọi tới nó theo tham chiếu.

Listing 3-13 ASSIGN

```
//assign3.cpp
//tra ve noi dung cua con tro this
#include<iostream.h>
class alpha
{
private:
    int data;
public:
    alpha()           //ham tao khong doi so
    {
    }
    alpha(int d)      //ham tao mot doi so
    { data=d;}
    void display()     //hien thi du lieu
    {
        cout<<"data: ";
    }
    alpha& operator=(alpha a)    //toan tu = duoc chong
    {
        data=a.data;
        cout<<"\nToan tu gan duoc goi!";
        return *this;           //tra ve tham chieu toi alpha nay
    }
};

void main()
{
    alpha a1(37);
    alpha a2,a3;
    a3=a2=a1;                //goi toan tu chong =
    cout<<"\na1="; a1.display();
    cout<<"\na2="; a2.display();    //hien thi a2
    cout<<"\na3="; a3.display();    //hien thi a3
}
```

Trong chương trình này toán tử chong = được khai báo như sau:

```
alpha& operator=(alpha a)
```

Nó trả về tham chiếu. Câu lệnh cuối cùng trong hàm này sẽ là:

```
return *this;
```

Bởi vì **this** là một con trỏ trỏ tới đối tượng gọi hàm, ***this** chính là đối tượng đó và câu lệnh đã trả về đối tượng này. Đây là kết quả của chương trình ASSIGN:

Toan tu gan duoc goi!

Toan tu gan duoc goi!

a1=37

a2=37

a3=37

Mỗi lần gặp dấu = trong lệnh

a3=a2=a1;

hàm chồng operator=() được gọi, nó hiển thị Toán tử gan được gọi ! Cả 3 đối tượng có cùng giá trị, như chỉ ra ở trên.

Trá về tham chiếu từ các hàm thành viên và các toán tử chồng là cách dùng thông dụng và hiệu quả, tránh việc tạo ra các đối tượng trung gian không cần thiết. Điều này làm chương trình nhỏ hơn, nhanh hơn và ít sinh lỗi hơn.

3.4.2. Con trỏ và const

Đầu tiên chúng ta nói cách dùng **const** với các biến không phải con trỏ, sau đó nói cách dùng **const** cho địa chỉ trong con trỏ và nội dung của biến tại địa chỉ đó.

1. Biến const

Chúng ta đã biết, nếu muốn một biến không thay đổi trong thời gian chương trình tồn tại, ta làm cho nó là **const**. Ví dụ, định nghĩa sau :

```
const int var1=123;
```

làm cho bất kỳ thay đổi gì với **var1** sẽ sinh ra thông báo lỗi từ trình biên dịch. Nó có giá trị là 123 trong suốt thời gian chương trình chạy.

2. Hai vị trí cho const

Sẽ phức tạp hơn khi có sự tham gia của con trỏ. Khi định nghĩa một con trỏ ta có thể chỉ rõ rằng con trỏ đó là **const** hay giá trị mà nó trỏ tới là **const**, hoặc cả hai. Chúng ta cùng xét các khả năng này. Chúng ta sẽ bắt đầu với biến thông thường và định nghĩa ba con trỏ trỏ tới nó.

```
int a;                //biến int
const int* p=&a;       //con trỏ trỏ tới hàng nguyên
++p;                  //ok
++(*p);               //error:can't modify a const a
int* const q=&a;       //con trỏ trỏ tới int
++q;                  //error: can't modify a const q
++(*C++);             //ok
const int* const r=&a; //con trỏ hàng trỏ tới hàng int
++r;                  //error: can't modify a const r
++(*r);               //error: can't modify a const a
```

Nếu **const** được đặt trước kiểu dữ liệu (như trong định nghĩa p), kết quả là một con trỏ trỏ tới một biến hằng. Nếu **const** được đặt sau kiểu dữ liệu (như trong định nghĩa q), kết quả là một con trỏ hằng trỏ tới một biến. Trong trường hợp đầu không thể thay đổi giá trị trong biến mà con trỏ trỏ tới, trường hợp thứ hai không thể thay đổi địa chỉ trong con trỏ. Sử dụng **const** ở hai vị trí (như trong định nghĩa r) có nghĩa là cả con trỏ và biến nó trỏ tới không thể thay đổi.

const có thể được sử dụng ở bất kỳ đâu nếu có thể. Nếu một giá trị con trỏ không bao giờ thay đổi, để con trỏ là **const**. Nếu con trỏ không bao giờ được dùng để thay đổi những gì nó trỏ tới thì bắt nó là một con trỏ trỏ tới **const**.

Tất cả những điều này dường như không quan trọng trong ví dụ này, bởi vì biến a tự nó không phải là hằng. Tại sao phải lo lắng về việc thay đổi nó khi nó có thể thay đổi trực tiếp được? Những vấn đề này trở nên thực tế hơn khi có một biến đã là hằng mà ta lại muốn định nghĩa một con trỏ trỏ tới nó.

```

const int b=99;      // biến hằng
int* r=&b;            //error: can't convert const to non-const
const int* s=&b;      //ok
int* const t=&b;      //error: can't convert const to non-const

```

Trình biên dịch bảo vệ khỏi việc viết các lệnh có thể dẫn đến thay đổi giá trị của một biến hằng, dù là thay đổi qua con trỏ. Không thể gán địa chỉ của một biến hằng (như b trong ví dụ) trừ khi nó được định nghĩa là con trỏ trỏ tới biến hằng (như con trỏ s trong ví dụ). Đây vẫn là hằng bởi vì không thể sử dụng một con trỏ như vậy để thay đổi biến hằng ban đầu:

```

++(*s);              //error: can't modify a const

```

3. Đối số hàm và const

Một nơi dùng **const** quan trọng là khi truyền con trỏ tới hàm. Lệnh gọi hàm thường muốn chắc chắn rằng hàm sẽ không thay đổi những gì con trỏ trỏ tới. Hàm có thể đảm bảo điều này bằng cách xác định một con trỏ trỏ tới một biến hằng:

```

void func(const int* p)
{
    ++p;      //ok
    ++(*p);   //error: can't modify a const int
}

```

Không những thế, hàm có thể làm cho cả con trỏ và những gì nó trỏ tới sẽ không bị thay đổi:

```

void func(const int* const p)
{
    ++p;      //error: can't modify a const pointer
    ++(*p)    //error: can't modify a const int
}

```

Những kỹ thuật này giúp giảm những tác động phụ không mong đợi khi gọi một hàm.

4. Trả về các giá trị const

Một hàm trả về một con trỏ sẽ nguy hiểm khi có một lệnh gọi hàm bị lỗi đã sử dụng con trỏ để thay đổi hằng mà con trỏ trỏ tới (đây là vấn đề duy nhất đối với các biến **static** và **external** bởi vì các biến cục bộ bị phá hủy khi hàm kết thúc).

Để tránh điều này, trình biên dịch yêu cầu sử dụng một giá trị trả về **const** đối với một hàm trả về một con trỏ trỏ tới một hằng. Ví dụ, nếu định nghĩa một hằng **external** j

```

const int j=77;
thì không nói:
int* func()
{
    return &j; //error: can't convert const to non-const
}

```

Bởi vì điều này sẽ cho phép sử dụng con trỏ do hàm trả về để thay đổi j.

```
int* p=func();  
*p=88;          //no good, modify j
```

Bởi vậy phải khai báo hàm là:

```
const int* func()  
{  
    return &j;//ok  
}
```

Lúc đó gọi hàm phải gán giá trị trả về tới một hằng:

```
const int* p=func();    //ok
```

Bởi vì không thay đổi những gì do p trỏ tới nên j vẫn an toàn khỏi những thay đổi.

CHƯƠNG 4

HÀM ẢO VÀ HÀM BẠN

Phần đầu tiên và lớn nhất của chương này liên quan tới các hàm ảo (**virtual function**), một trong các đặc điểm chính của C++. Hàm ảo có thể thay đổi cấu trúc chương trình một cách kinh ngạc và mạnh mẽ. Đầu tiên chúng ta cùng xem xét những kiến thức cơ bản, sau đó là các ví dụ về cách sử dụng chúng, tiếp đến là một vài đặc điểm phụ.

Cuối chương này chúng ta sẽ nói về các hàm và các lớp bạn. Bạn (**friend**) là một nhượng bộ của C++ đối với thực tế. Sự bao bọc (**encapsulation**) nghiêm ngặt và cất giấu dữ liệu thường theo mong muốn, nhưng đôi khi có những tình huống phát sinh những ngoại lệ. **Friend** cung cấp một cách có tổ chức để mở rộng tự do sự nghiêm ngặt của C++ và không làm hại đến sự toàn vẹn dữ liệu. Chúng ta sẽ có nhiều tình huống trong **friend** được dùng.

4.1. GIỚI THIỆU VỀ HÀM ẢO

Có ba khái niệm chính trong lập trình hướng đối tượng: thứ nhất là lớp, thứ hai là sự kế thừa và thứ ba là sự đa hình thái (**polymorphism**). Hai khái niệm đầu chúng ta đã biết còn khái niệm thứ ba được cài đặt trong C++ bằng các hàm ảo.

4.1.1. Sự đa hình thái

Trong cuộc sống thực, thường có một bộ sưu tập của nhiều thứ khác nhau mà nếu đưa ra các lệnh giống nhau sẽ có các hành động khác nhau. Lấy sinh viên làm ví dụ, có thể có nhiều loại sinh viên trong một trường đại học: sinh viên Điện tử - Viễn thông, sinh viên Cơ khí, sinh viên Năng lượng... Giả sử hiệu trưởng của trường đại học này muốn gửi một hướng dẫn tới tất cả các sinh viên: "Điền đầy đủ vào các bản đăng ký nghề khi ra trường". Những loại sinh viên khác nhau sẽ có những bản đăng ký khác nhau bởi vì họ ở những khoa khác nhau. Người hiệu trưởng không cần gửi các thông báo khác nhau tới từng nhóm sinh viên ("Điền đầy đủ vào bản đăng ký nghề Điện tử - Viễn thông" đối với sinh viên khoa Điện tử - Viễn thông, "Điền đầy đủ vào bản đăng ký nghề Cơ khí" đối với sinh viên khoa Cơ khí v.v...). Đây là một thông báo áp dụng cho mọi người vì mọi người biết cách điền vào các bản đăng ký của riêng họ.

Đa hình thái có nghĩa là "có nhiều hình thái". Một chỉ thị của hiệu trưởng là đa hình thái bởi vì nó sẽ khác nhau với các loại sinh viên khác nhau. Đối với sinh viên khoa Điện tử - Viễn thông, chỉ thị sẽ là "Điền đầy đủ vào bản đăng ký nghề Điện tử - Viễn thông", đối với sinh viên khoa Cơ khí, chỉ thị sẽ là "Điền đầy đủ vào bản đăng ký nghề Cơ khí"...

Một cách điển hình, sự đa hình thái xuất hiện trong các lớp liên quan tới sự kế thừa. Trong C++ , sự đa hình thái có nghĩa là lời gọi tới một hàm thành viên sẽ làm cho các hàm khác nhau được thực hiện tùy thuộc vào kiểu đối tượng gọi hàm đó.

Điều này dường như hơi giống sự chồng hàm (**function overloading**) nhưng sự đa hình thái là một cơ chế khác và mạnh hơn nhiều. Một sự khác nhau giữa sự chồng hàm và sự đa hình thái là: đối với sự đa hình thái phải biết tự lựa chọn hàm để thực hiện, sự lựa chọn diễn ra trong khi chương trình đang chạy; đối với sự chồng hàm, sự lựa chọn là do trình biên dịch. Sự chồng hàm chỉ tiện lợi cho người sử dụng lớp, trái lại sự đa hình thái ảnh hưởng tới toàn bộ cấu trúc của một chương trình.

Tất cả những gì chúng ta vừa nói hơi trừu tượng, bởi vậy chúng ta cùng bắt đầu với một vài chương trình ngắn minh họa từng phần của tình huống này và rồi sau đặt tất cả chúng vào với nhau. Trong thực tế, các hàm ảo kết hợp với các con trỏ trỏ tới các đối tượng. Chúng ta sẽ xem chúng làm việc với nhau như thế nào.

4.1.2. Hàm thành viên thông thường được truy nhập với con trỏ

Ví dụ đầu tiên cho thấy những gì sẽ xảy ra khi một lớp cơ sở và các lớp dẫn xuất có tất cả các hàm trùng tên nhau và các hàm này được truy nhập bằng cách sử dụng con trỏ, không sử dụng các hàm ảo. Bản 4-1 là chương trình NOVIRT.

Listing 4-1 NOVIRT

```
//novirt.cpp
//cac ham thanh vien thong thuong duoc truy nhap bang cach su dung
//con tro tro toi doi tuong
#include<iostream.h>
class Base                                //lop co so
{
    public:
    void show()
    {
        cout<<"\nLop co so Base";
    }
};
class Derv1:public Base
{
    public:
    void show()
    {
        cout<<"\nLop dan xuat Derv1";
    }
};
class Derv2:public Base
{
    public:
    void show()
    {
        cout<<"\nLop dan xuat Derv2";
    }
};
void main()
{
    Derv1 dv1;                            //doi tuong cua lop dan xuat Derv1
    Derv2 dv2;                            //doi tuong cua lop dan xuat Derv2
    Base* ptr;                            //con tro tro toi lop co so
    ptr=&dv1;                             //dat dia chi cua dv1 vao con tro
    ptr->show();                           //thuc hien show()
    ptr=&dv2;                             //dat dia chi cua dv2 vao con tro
    ptr->show();                           //thuc hien show()
}
```

Các lớp **Derv1** và **Derv2** được rút ra từ lớp cơ sở **Base**. Cả ba lớp có một hàm thành viên **show()**. Trong hàm **main()**, chương trình tạo các đối tượng lớp **Derv1** và **Derv2** và một con trỏ trỏ tới lớp **Base**. Sau đó nó đặt địa chỉ của một đối tượng lớp dẫn xuất vào trong con trỏ lớp cơ sở trong dòng lệnh:

```
ptr=&dv1;    //địa chỉ lớp dẫn xuất trong con trỏ lớp cơ sở
```

Nhớ rằng hoàn toàn có quyền gán một địa chỉ của một kiểu (ví dụ **Derv1**) tới một kiểu khác (**Base**), bởi vì các con trỏ trỏ tới đối tượng của một lớp dẫn xuất có kiểu phù hợp với các con trỏ trỏ tới đối tượng của lớp cơ sở.

Bây giờ có một câu hỏi đặt ra là khi thực hiện lệnh tiếp theo

```
ptr->show();
```

hàm nào được gọi? hàm **Base::show()** hay **Derv1::show()**?

Cũng câu hỏi như vậy sau khi đặt địa chỉ của đối tượng lớp **Derv2** vào con trỏ và lại thử thực hiện hàm **show()** của nó:

```
ptr=&dv2;
```

```
ptr->show();
```

Hàm **show()** nào được gọi ? **Base::show()** hay **Derv2::show()** ?

Kết quả đưa ra từ chương trình sẽ trả lời câu hỏi này:

Lop co so Base

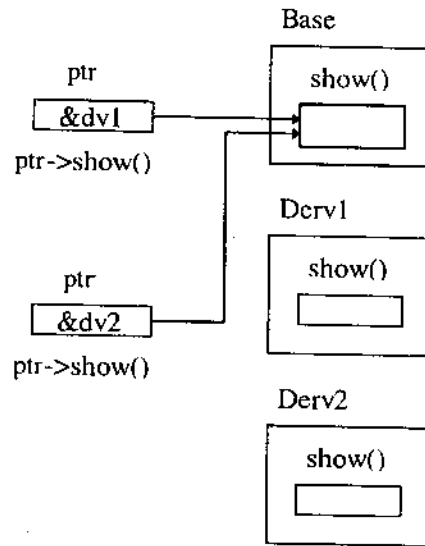
Lop co so Base

Hàm trong lớp cơ sở được thực hiện trong cả hai trường hợp. Trình biên dịch bỏ qua nội dung của con trỏ **ptr** và chọn hàm thành viên phù hợp với kiểu con trỏ, như chỉ ra ở hình 4-1.

Chúng ta cùng làm một sự thay đổi đơn lẻ trong chương trình: đặt từ khóa **virtual** trước khai báo cho hàm **show()** trong lớp cơ sở. Bản 4-2 trình bày chương trình **VIRT**.

Listing 4-2 VIRT

```
//virt.cpp
//cac ham ao duoc truy nhap tu con tro
#include<iostream.h>
class Base          //lop co so
{
public:
    virtual void show()
    {
        cout<<"\nLop co so Base";
    }
};
class Derv1:public Base
{
public:
    void show()
    {
        cout<<"\nLop dan xuat Derv1";
    }
};
class Derv2:public Base
{
public:
    void show()
    {
        cout<<"\nLop dan xuat Derv2";
    }
};
void main()
```



Hình 4-1. Truy nhập con trỏ không ảo.

```

{
    Derv1 dv1;           //doi tuong cua lop dan xuat Derv1
    Derv2 dv2;           //doi tuong cua lop dan xuat Derv2
    Base* ptr;           //con tro tro toi lop co so
    ptr=&dv1;             //dat dia chi cua dv1 vao con tro
    ptr->show();          //thuc hien show()
    ptr=&dv2;             //dat dia chi cua dv2 vao con tro
    ptr->show();          //thuc hien show()
}

```

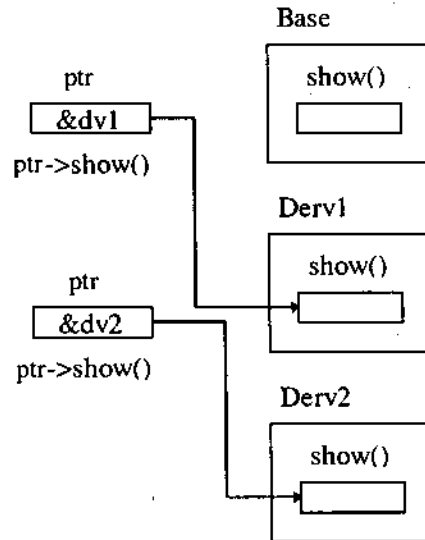
Kết quả chương trình là:

Lop dan xuat Derv1
Lop dan xuat Derv2

Bây giờ là các hàm thành viên của lớp dẫn xuất chứ không phải lớp cơ sở được thực hiện. Thay đổi **ptr** từ địa chỉ của **Derv1** sang địa chỉ của **Derv2** và hàm **show()** cụ thể được thực hiện cũng thay đổi. Bởi vậy, cùng một lời gọi hàm :

`ptr->show();`

thực hiện các hàm khác nhau, tùy thuộc vào nội dung của **ptr**. Trình biên dịch lựa chọn hàm để thực hiện dựa trên nội dung của con trỏ **ptr**, chứ không phải tên kiểu của con trỏ, như trong NOVIRT. Đây là sự đa hình thái. Chúng ta đã làm cho hàm **show()** đa hình thái bằng cách thiết kế nó với **virtual**. Hình 4-2 cho thấy điều này như thế nào.



Hình 4-2. Truy nhập con trỏ không ảo.

4.1.3. Sự liên kết muộn

Chúng ta có thể băn khoăn là trình biên dịch làm thế nào biết được hàm nào cần biên dịch. Trong NOVIRT, trình biên dịch không có vấn đề gì với biểu thức:

`ptr->show();`

Nó luôn luôn biên dịch lời gọi hàm trong lớp cơ sở. Nhưng trong VIRT, trình biên dịch không biết nội dung của con trỏ **ptr** trỏ tới lớp nào. Nó có thể là địa chỉ của một đối tượng của lớp **Derv1** hoặc **Derv2**. Trình biên dịch gọi hàm **show()** nào?

Trong thực tế tại thời điểm biên dịch chương trình, trình biên dịch không biết gọi hàm nào, bởi vậy nó sắp xếp để quyết định hoãn gọi hàm cho đến khi chương trình được chạy.

Tại thời điểm chạy chương trình, khi lời gọi hàm được thực hiện, lệnh mà trình biên dịch đã đặt trong chương trình tìm thấy kiểu của đối tượng mà địa chỉ của nó ở trong **ptr** và gọi hàm **show()** thích hợp: **Derv1::show()** hay **Derv2::show()** tùy thuộc vào lớp của đối tượng.

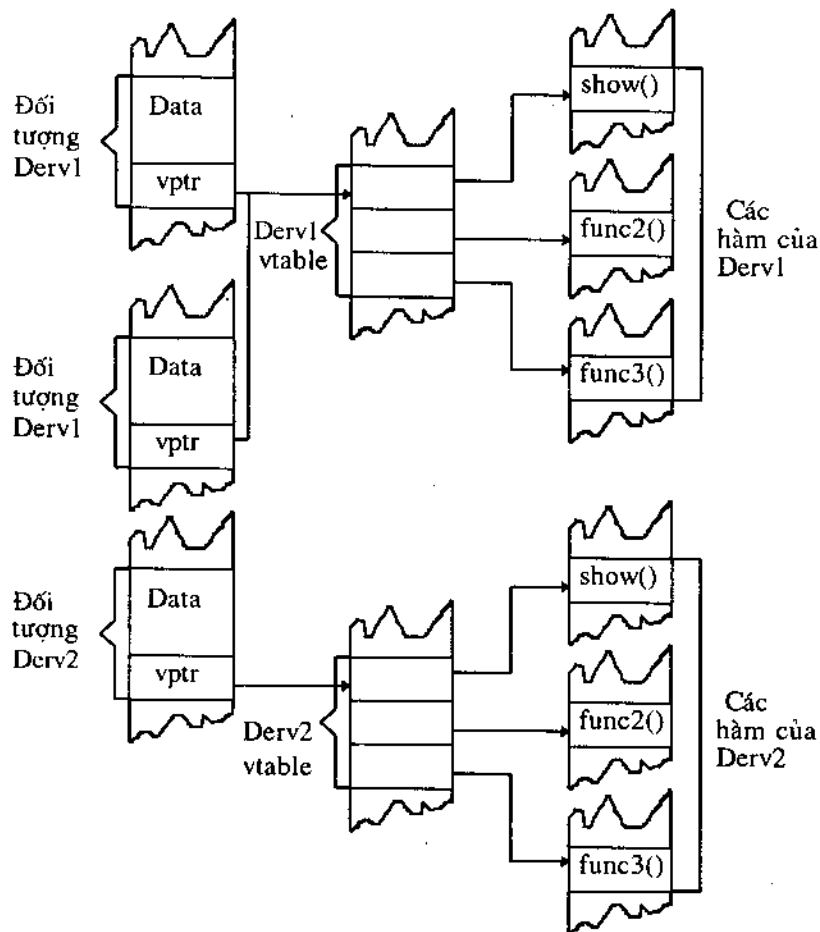
Lựa chọn một hàm tại thời điểm chạy chương trình được gọi là sự liên kết muộn (**late binding**) hay liên kết động (**dynamic binding**), (**Binding** có nghĩa là nối lời gọi hàm tới hàm đó). Nối tới các hàm trong cách thông thường, trong khi biên dịch, được gọi là sự liên kết sớm (**early binding**) hay liên kết tĩnh (**static binding**). Sự liên kết muộn yêu cầu một lượng nhỏ thời gian bỏ trống (lời gọi tới một hàm có thể lâu hơn khoảng 10%) nhưng cho sự tăng vượt bậc khả năng và sự mềm dẻo.

4.1.4. Hàm ảo làm việc như thế nào?

Chúng ta đã biết những vấn đề cơ bản của việc sử dụng hàm ảo, bây giờ chúng ta sẽ tìm hiểu thêm một số điều bí ẩn của nó.

Nên nhớ rằng, đối với một đối tượng thông thường - nghĩa là một đối tượng không có các hàm ảo - trong bộ nhớ chỉ lưu trữ dữ liệu của riêng nó, không có gì khác. Khi một hàm thành viên được gọi cho một đối tượng như vậy, trình biên dịch truyền cho hàm đó địa chỉ của đối tượng gọi nó. Địa chỉ này có sẵn cho hàm đó, nó được đặt ở trong con trỏ **this** và hàm sử dụng địa chỉ này (thường ẩn) để truy nhập dữ liệu của đối tượng. Địa chỉ trong **this** được tạo ra bởi trình biên dịch mỗi khi hàm thành viên được gọi; nó không lưu trữ trong bộ nhớ và không chiếm không gian nhớ. Con trỏ **this** chỉ là sự kết nối cần thiết giữa một đối tượng và các hàm thành viên thông thường của nó.

Với các hàm ảo, những vấn đề này phức tạp hơn. Khi một lớp dẫn xuất được chỉ rõ là có các hàm ảo, trình biên dịch tạo ra một bảng - một mảng - các địa chỉ của các hàm, được gọi bảng ảo (**virtual table**). Bảng ảo được đặt cho một cái tên gì đó chẳng hạn như **vtbl** hay **vtable**, tùy thuộc vào trình biên dịch. Trong ví dụ VIRT, một lớp **Derv1** và **Derv2** có một bảng ảo của riêng nó. Có một mục trong mỗi bảng ảo dành cho tất cả các hàm ảo trong lớp. Bởi vì chúng đều được rút ra từ **Base**, **Derv1** và **Derv2** có các hàm ảo giống nhau, được sắp xếp theo một trật tự giống nhau. Tuy nhiên, các địa chỉ trong bảng là khác nhau đối với hai lớp. Hình 4-3 cho thấy sự sắp xếp này. Nó cũng trình bày các hàm thêm vào trong mỗi lớp, **func2()** và **func3()**, để làm rõ rằng trong mỗi bảng ảo có nhiều mục (các con trỏ trỏ tới các thành viên thích hợp).



Hình 4-3. Liên kết hàm ảo.

Mỗi đối tượng của **Derv1** và **Derv2** khi được tạo chứa một con trỏ phụ ngoài dữ liệu của riêng nó. Điều này cũng đúng với bất kỳ đối tượng của một lớp nào mà có một hoặc nhiều hàm ảo. Con trỏ này có một tên gì đó chẳng hạn như **vptr** hoặc **vpoiner**, chứa địa chỉ của bảng ảo lớp. Do đó, các đối tượng của các lớp có hàm ảo hơi lớn hơn các đối tượng thông thường.

Đặt địa chỉ của một bảng ảo xác định lớp **vtable** trong một con trỏ **vptr** của một đối tượng cho phép một đối tượng biết nó thuộc lớp nào. Trong ví dụ, khi một hàm ảo được gọi cho một đối tượng của **Derv1** hoặc **Derv2**, trình biên dịch, thay vì xác định xem hàm nào sẽ được gọi, nó tạo một lệnh mà sẽ tìm thấy **vptr** của đối tượng đó trước và sau đó sử dụng con trỏ này để truy nhập địa chỉ hàm thành viên thích hợp trong bảng ảo xác định lớp **vtable**. Do đó, đối với các hàm ảo, tự đối tượng xác định việc gọi hàm nào, không phải trình biên dịch xác định.

Ví dụ VIRT có thể sử dụng theo cách khác bởi vì các con trỏ trỏ tới các đối tượng có các hàm ảo thường được lưu trữ trong một mảng chứ không phải trong một biến đơn độc. Ví dụ trong mục tiếp theo cho thấy tình huống điển hình hơn.

4.1.5. Mảng con trỏ trỏ tới các đối tượng

Có lẽ cách thông dụng nhất để sử dụng các hàm ảo là với một mảng con trỏ trỏ tới các đối tượng. Bản 4-3 trình bày một phiên bản đã sửa đổi từ chương trình VIRT để minh họa cấu trúc này. Các lớp trong hai chương trình này giống nhau nhưng hàm **main()** thì khác. Chương trình này gọi là **ARRVIRT**.

Listing 4-3 ARRVIRT

```
//arrvirt.cpp
//mang con tro tro toi cac doi tuong co ham ao
#include<iostream.h>
class Base                                //lop co so
{
    public:
        virtual void show()
        {
            cout<<"\nLop co so Base";
        }
};

class Derv1:public Base
{
    public:
        void show()
        {
            cout<<"\nLop dan xuat Derv1";
        }
};

class Derv2:public Base
{
    public:
        void show()
        {
            cout<<"\nLop dan xuat Derv2";
        }
};

void main()
{
    Base* arrBase[2];                    //mang con tro tro toi cac doi tuong
```

```

Derv1 dv1;           //doi tuong cua lop dan xuat Derv1
Derv2 dv2;           //doi tuong cua lop dan xuat Derv2

arrBase[0]=&dv1;     //dat dia chi cua dv1 vao mang
arrBase[1]=&dv2;     //dat dia chi cua dv2 vao mang
for(int j=0;j<2;j++)  //tat ca cac phan tu cua mang
    arrBase[j]->show(); //thuc hien show()
}

```

Chúng ta đã đặt địa chỉ của đối tượng **dv1** và **dv2** vào trong mảng **arrBase**, nó có kiểu **Base**. Như trong chương trình **VIRT**, trình biên dịch sử dụng kiểu của các con trỏ lưu trữ trong mảng, không phải kiểu của mảng để thực hiện hàm **show()** được gọi. Kết quả là:

```

Lop dan xuat Derv1
Lop dan xuat Derv2

```

Chú ý, sử dụng vòng lặp **for** trong hàm **main()** để gọi hàm **show()** cho bất kỳ số lượng đối tượng nào một cách dễ dàng. Đây là cốt lõi của sự đa hình thái: một hàm đơn độc có thể gọi nhiều hàm khác, tùy thuộc kiểu đối tượng thực hiện lời gọi hàm đó.

Nên biết rằng cơ chế hàm ảo làm việc với các con trỏ trỏ tới các đối tượng (và, như sẽ thấy sau này, với các tham chiếu), không làm việc với chính các đối tượng. Chúng ta không thể viết lại hàm **main()** trong **ARRVIRT** để sử dụng một mảng các đối tượng như thế này:

```

void main()
{
    Base arrobjs[2];    //mang giu cac doi tuong lop co so
    Derv1 dv1;          //doi tuong lop dan xuat 1
    Derv2 dv2;          //doi tuong lop dan xuat 2
    arrobjs[0]=dv1;     //dat doi tuong dv1 vao mang
    arrobjs[1]=dv2;     //dat doi tuong dv2 vao mang
    for(int j=0;j<2;j++) //tat ca cac phan tu cua mang
        arrobjs[j].show(); //thuc hien show()
}

```

Mặc dù trình biên dịch cho phép đặt một đối tượng lớp dẫn xuất vào trong một mảng lớp cơ sở nhưng kết quả sẽ không như chúng ta mong muốn. Kết quả của phiên bản này là:

```

Lop co so Base
Lop co so Base

```

Tại sao các hàm **show()** của lớp dẫn xuất không được thực hiện? Vì các đối tượng của các lớp dẫn xuất có thể không có kích thước giống nhau như các đối tượng của lớp cơ sở, chúng thường lớn hơn (mặc dù trong ví dụ này thì không) bởi vì chúng có dữ liệu thêm vào làm cho chúng trở thành một "loại" đối tượng lớp cơ sở. Bởi vậy, chúng không vừa trong mảng lớp cơ sở. Nếu cố nhét một đối tượng lớp dẫn xuất vào trong một mảng lớp cơ sở thì phần đối tượng dành riêng cho lớp dẫn xuất sẽ bị cắt bỏ, chỉ còn một đối tượng lớp cơ sở. Thậm chí nếu không có sự khác nhau về kích thước giữa đối tượng lớp cơ sở và lớp dẫn xuất thì trình biên dịch vẫn coi các đối tượng lớp dẫn xuất là các đối tượng lớp cơ sở. Hãy luôn nhớ là không bao giờ đặt các đối tượng lớp dẫn xuất vào trong các biến lớp cơ sở; nếu không chúng sẽ quên mất chúng là gì.

Trong khi đó, con trỏ có kích thước giống nhau bất kể chúng trỏ tới lớp nào, bởi vậy sẽ không mất bất cứ thông tin nào khi đặt một địa chỉ lớp dẫn xuất vào trong một mảng con trỏ lớp cơ sở. Hãy sử dụng sự đa hình thái, gọi các hàm ảo với các con trỏ trỏ tới các đối tượng (hay với các tham chiếu-references), không phải với các đối tượng.

4.2. VÍ DỤ VỀ HÀM ẢO

Bây giờ chúng ta đã hiểu các cơ chế của hàm ảo, chúng ta cùng xem một vài tình huống mà ở đó việc sử dụng chúng thật có ý nghĩa. Trong phần này sẽ có hai chương trình.

Trong ví dụ thứ nhất chúng ta sẽ biết được người nào xuất sắc. Ví dụ thứ hai chúng ta sẽ biết cách sử dụng các hàm giống nhau để vẽ các hình khác nhau. Trong cả hai chương trình, việc sử dụng các hàm ảo sẽ đơn giản hóa việc lập trình rất nhiều.

4.2.1. Ví dụ về nhân viên

Ví dụ này mô hình hóa các loại người khác nhau trong một trường đại học hoặc phổ thông. Các loại cơ bản là sinh viên và giáo viên. Lưu ý rằng các loại này có một vài đặc điểm chung. Chúng ta sẽ sắp xếp để lớp **student** và **teacher** được rút ra từ lớp **person**. Ngoài ra, những người khác không phải là sinh viên và giáo viên cũng thấy ở trường đại học như nhân viên quản lý, nhân viên văn phòng v.v... Giả thiết chúng ta không tạo lớp riêng cho những loại nhân viên này mà gộp tất cả chúng vào lớp **person**.

Để đơn giản, dữ liệu duy nhất trong lớp **person** là một tên người. Cũng có thể tưởng tượng ra các biến thích hợp cho một người để thêm vào, chẳng hạn địa chỉ, số điện thoại, mã nhân viên. Lớp sinh viên, **student**, có điểm trung bình chung (TBC) và lớp giáo viên, **teacher**, có sổ sách mà giáo viên đã xuất bản.

Tất cả các lớp chứa các hàm **getData()** và **putData()** để nhập và đưa ra dữ liệu. Chúng cũng chứa một hàm gọi là **isOutstanding()** để người quản lý để tạo một danh sách các sinh viên và giáo viên xuất sắc để trao phần thưởng. Bản 4-4 trình bày chương trình VIRTPERS.

Listing 4-4 VIRTPERS

```
//virtpers.cpp
//cac ham ao voi lop person
#include<iostream.h>
class person
{
protected:
    char name[40];
public:
    virtual void getData()
    {
        cout<<"\nNhap vao ten:"; cin>>name;
    }
    virtual void putData()
    {
        cout<<"\nTen:"<<name;
    }
    virtual void isOutstanding()
    {} //chu y:than ham rong
};

class student:public person //lop sinh vien
{
private:
    float tbc; //diem trung binh chung
public:
    void getData()
    {
        person::getData();
    }
};
```

```

        cout<<" Nhap vao diem trung binh chung cua sinh vien:";
        cin>>tbc;
    }
    void putData()
    {
        person::putData();
        cout<<"\n Diem TBC="<<tbc;
    }

    void isOutstanding()
    {
        if(tbc>=9.0)
            cout<<"(Day la nguoi xuat sac)";
    }
};

class teacher:public person    //lop giao vien
{
private:
    int numPubs;                //so sach xuat ban
public:
    void getData()
    {
        person::getData()
        cout<<" Nhap vao so sach xuat ban cua giao vien:";
        cin>>numPubs;
    }
    void putData()
    {
        person::putData();
        cout<<"\n So sach xuat ban="<<numPubs;
    }
    void isOutstanding()
    {
        if(numPubs>=100)
            cout<<"(Day la nguoi xuat sac)";
    }
};

void main()
{
    person* persPtr[100];        //danh sach con tro tro toi cac
                                //doi tuong person
    int n=0;                     //so nguoi trong danh sach
    char choice;

    do
    {
        cout<<"\nNhap vao mot nguoi binh thuong,sinh vien,hoac giao vien"
            <<"(b,s,g):";
        cin>>choice;
        if(choice=='s')
            persPtr[n]=new student;        //dat mot sinh vien vao danh sach
        else if(choice=='g')
            persPtr[n]=new teacher;        //dat mot giao vien vao danh sach
    }
}

```

```

        else
            persPtr[n]=new person;    //dat mot nguoi binh thuong vao danh sach
            persPtr[n++]->getData();  //nhap du lieu
            cout<<" Co nhap nua khong(c/k)? ";
            cin>>choice;
        }
        while(choice=='c');
        for(int j=0;j<n;j++)
        {
            persPtr[j]->putData();    //dua ten cua moi nguoi ra man hinh
            persPtr[j]->isOutstanding(); //va ghi chu neu xuat sac
        }
    }
}

```

1. Hàm main()

Trong hàm **main()**, đầu tiên chương trình để cho người dùng nhập vào một số tên bất kỳ của người bình thường, sinh viên và giáo viên. Đối với các sinh viên, chương trình còn hỏi điểm TBC; đối với các giáo viên, chương trình hỏi số sách xuất bản. Khi người sử dụng không nhập vào nữa, chương trình đưa tên và các dữ liệu khác cho tất cả mọi người, có ghi chú các sinh viên và giáo viên xuất sắc. Sau đây là vài mẫu tương tác với chương trình:

```

Nhập vào mot nguoi binh thuong,sinh vien hoac giao vien(b,s,g): b
Nhập vào ten: Phuong
    Co nhap nua khong(c/k)? c
Nhập vào mot nguoi binh thuong,sinh vien hoac giao vien(b,s,g): s
Nhập vào ten: Thang
    Nhập vào diem trung binh chung cua sinh vien: 8.2
    Co nhap nua khong(c/k)? c
Nhập vào mot nguoi binh thuong,sinh vien hoac giao vien(b,s,g): s
Nhập vào ten: Binh
    Nhập vào diem trung binh chung cua sinh vien: 9.8
    Co nhap nua khong(c/k)? c
Nhập vào mot nguoi binh thuong,sinh vien hoac giao vien(b,s,g): g
Nhập vào ten: Oanh
    Nhập vào so sach xuat ban cua giao vien: 120
    Co nhap nua khong(c/k)? c
Nhập vào mot nguoi binh thuong,sinh vien hoac giao vien(b,s,g): g
Nhập vào ten: Hoa
    Nhập vào so sach xuat ban cua giao vien: 60
    Co nhap nua khong(c/k)? k

Ten: Phuong
Ten: Thang
    Diem TBC = 8.2
Ten: Binh
    Diem TBC = 9.8 (day la nguoi xuat sac)
Ten: Oanh
    So sach xuat ban = 120 (day la nguoi xuat sac)
Ten: Hoa
    So sach xuat ban = 60

```


2. Các lớp

Lớp **person** chứa một mục dữ liệu là một chuỗi biểu diễn tên người. Lớp **student** và **teacher** thêm các mục dữ liệu mới tới lớp cơ sở **person**. Lớp **student** chứa một biến **tbc** thuộc kiểu **float**, biểu diễn điểm trung bình chung của sinh viên. Lớp **teacher** chứa một biến **numPubs** thuộc kiểu **int**, biểu diễn số sách mà giáo viên đã xuất bản. Sinh viên có điểm TBC trên ≥ 9.0 và giáo viên có số sách xuất bản trên 100 được coi là xuất sắc.

3. Các hàm ảo

Trong lớp **person**, cả ba hàm, **getData()**, **putData()** và **isOutstanding()** được khai báo là hàm ảo. Điều này là cần thiết bởi vì các hàm có các tên này cũng tồn tại trong các lớp dẫn xuất và các đối tượng của các lớp dẫn xuất này sẽ được truy nhập bằng con trỏ. Cách duy nhất để chương trình biết loại đối tượng mà con trỏ trong **persPtr[j]** trỏ tới như trong biểu thức:

```
persPtr[j]->putData();
```

là liệu hàm **putData()** có là hàm ảo không. Nếu nó được khai báo là hàm ảo thì hàm **putData()** thích hợp với lớp của đối tượng được trỏ tới bởi **persPtr[j]** được thực hiện.

4. Hàm **isOutstanding()**

Trong lớp **student**, **isOutstanding()** biểu thị thông báo "đây là người xuất sắc" nếu sinh viên có điểm TBC ≥ 9.0 và trong lớp **teacher**, nó hiển thị thông báo giống như vậy nếu biến **numPubs** ≥ 100 . Tuy nhiên, trong lớp **person** hàm này có thân hàm rỗng (giả thiết không có tiêu chuẩn đánh giá đối với người bình thường).

Hàm **person::isOutstanding()** không làm gì, chúng ta có thể xóa nó đi được không? Câu trả lời là không, nó phải tồn tại do có câu lệnh:

```
persPtr[j]->isOutstanding();
```

trong hàm **main()**, bởi vì **persPtr** giữ các con trỏ thuộc kiểu **person***, cho dù nó cũng có thể giữ các con trỏ thuộc kiểu **student*** hoặc **teacher***.

5. Các hàm ảo trong các lớp khác

Chú ý rằng một hàm thành viên trong một lớp dẫn xuất có thể gọi một hàm thành viên ảo của lớp cơ sở dùng toán tử quy định phạm vi (**scope resolution operator**) giống như cách truy nhập các hàm chồng. Ví dụ, hàm **getData()** trong lớp **student** gọi hàm **getData()** trong lớp **person**:

```
class student:public person
{
    ...
    public:
        void getData()
        {
            person::getData();    //gọi hàm ảo lớp cơ sở
            cout<<" Nhập vào điểm TBC của sinh viên:";
            cin>>tbc;
        }
    ...
};
```

Các hàm ảo khi được truy xuất trực tiếp sẽ được đối xử như các hàm không ảo.

4.2.2. Ví dụ đồ họa

Chúng ta cùng xét một ví dụ khác về các hàm ảo, đó là một ví dụ đồ họa. Các lớp biểu diễn một số hình xác định: **cap** (hình mũ không vành), **bowl** (hình bát), **square** (hình vuông). Các lớp này được rút ra từ một lớp chung là **shape**.

Một cách dễ dàng để hiển thị một hình phức tạp được tạo ra từ các đối tượng như vậy là đặt các con trỏ tới các hình khác nhau trong một mảng và sau đó vẽ tất cả các hình chỉ dùng một vòng lặp **for**:

```
for(int j=0;j<N;j++)
    sharray[j]->draw();
```

Chương trình VIRTSHAP làm được điều này bằng cách khai báo hàm **draw()** là hàm ảo trong lớp **shape**.

Listing 4-5 VIRTSHAP

```
//virtshap.cpp
//ve cac hinh bang cac dau x tren man hinh van ban
//su dung ham ao draw()
#include<iostream.h>
////////////////////////////////////
class shape          //lop co so
{
    private:
        int xco,yco;    //toa do cua hinh
        int size;       //kich thuoc cua hinh
    protected:        //cac ham chi doc
        int getx() const
        { return xco;}
        int gety() const
        { return yco;}
        int getz() const
        { return size;}
    void down() const;  //khai bao ham
    public:
        //ham tao 3 doi so
        shape(int x,int y,int s):xco(x),yco(y),size(s)
        {}
        virtual void draw() const
        { cout<<"Error:Base virtual"<<endl;}
};
void shape::down() const
{
    for(int y=0;y<yco;y++)
        cout<<endl;
}
////////////////////////////////////
class square:public shape    //hinh vuong
{
    public:
        //ham tao 3 doi so
        square(int x, int y,int s):shape(x,y,s)
        {}
        void draw() const;    //khai bao
};
```

```

void square::draw() const    //ve mot hinh vuong
{
    shape::down();          //di chuyen toi dinh cua hinh
    for(int y=0;y<getz();y++) //chuyen y xuong ngang qua hinh
    {
        int x;
        for(x=1;x<getx();x++) //lay khoang trong tren hinh
            cout<<' ';
        for(x=0;x<getz();x++) //ve duong thang bang dau x
            cout<<'x';
        cout<<endl;
    }
}

////////////////////////////////////
class cap:public shape      //hinh mu
{
public:
    //ham tao 3 doi so
    cap(int x, int y,int s):shape(x,y,s)
    {
        void draw() const;    //khai bao
    };
void cap::draw() const      //ve mot hinh mu
{
    shape::down();          //di chuyen toi dinh cua hinh
    for(int y=0;y<getz();y++) //chuyen y xuong ngang qua hinh
    {
        int x;
        for(x=0;x<getx()-y+1;x++) //lay khoang trong tren hinh
            cout<<' ';
        for(x=0;x<2*y+1;x++)      //ve duong thang bang dau x
            cout<<'x';
        cout<<endl;
    }
}

////////////////////////////////////
class bowl:public shape    //hinh cai bat
{
public:
    //ham tao 3 doi so
    bowl(int x, int y,int s):shape(x,y,s)
    {
        void draw() const;    //khai bao
    };
void bowl::draw() const    //ve mot hinh cai bat
{
    shape::down();          //di chuyen toi dinh cua hinh
    for(int y=0;y<getz();y++) //chuyen y xuong ngang qua hinh
    {
        int x;
        for(x=0;x<getx()-(getz()-y)+2;x++) //lay khoang trong tren hinh
            cout<<' ';
        for(x=0;x<2*(getz()-y)-1;x++)      //ve duong thang bang dau x

```

```

        cout<<'x';
        cout<<endl;
    }
}
////////////////////////////////////
void main()
{
    const int N=3;           //so hình
    shape* sharray[N];       //mang con tro tro toi shape
    bowl bw(10,0,3);         //tao mot cai bat
    square sq(20,1,5);       //tao mot hình vuông
    cap cp(30,1,7);          //tao mot cai mu
    sharray[0]=&bw;          //dat dia chi cua chung vao trong mang

    sharray[1]=&sq;
    sharray[2]=&cp;
    cout<<endl<<endl;        //bat dau duoi 2 dong
    for(int j=0;j<N;j++)
        sharray[j]->draw();  //hien thi ca 3 hình
}

```

XXXXX
XXX
X

XXXXX
XXXXX
XXXXX
XXXXX
XXXXX

Trong lớp **shape**, hàm **draw()** là một hàm ảo.

Trong hàm **main()**, chương trình tạo một mảng con trỏ, **sharray**, trỏ tới các hình. Tiếp theo, nó tạo ba đối tượng của các lớp **bowl**, **square**, **cap** và đặt địa chỉ của chúng trong mảng. Bây giờ vẽ ba hình dùng một vòng lặp **for** thật dễ dàng. Hình 4-4 là kết quả của chương trình VIRTSHAP.

Đây là một cách rất mạnh để kết nối các phần tử đồ họa, nhất là khi có một số lớn các đối tượng cần được nhóm vào với nhau và vẽ như một đơn vị.

X
XXX
XXXXX
XXXXXXXXX
XXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXXX

Hình 4-4. Kết quả của chương trình VIRTSHAP.

1. Không có các đối tượng **shape**

Trong chương trình VIRTSHAP, người sử dụng không nên cố tạo một đối tượng **shape**. Lớp **shape** chỉ đóng vai trò là lớp cơ sở cho các lớp **bowl**, **square** và **cap**. Trong trường hợp một người nào đó cố tạo một đối tượng **shape** và vẽ nó thì sẽ nhận được thông báo lỗi mà chúng ta đã đặt trong thân hàm **shape::draw()**.

Tuy nhiên, đây là một cách tìm lỗi không hay. Sẽ tốt hơn nếu sắp xếp các thứ để trình biên dịch tìm ra lỗi, hơn là đợi chúng xuất hiện tại thời điểm chạy chương trình. Ở đây chúng ta muốn trình biên dịch báo cho biết nếu người sử dụng lớp cứ cố tạo một đối tượng lớp cơ sở (**shape**). Chúng ta sẽ thấy cách làm này bằng các hàm ảo thuần túy ở mục 4.4.

2. Khởi tạo mảng

Thay vì sử dụng các lệnh riêng rẽ để tạo các đối tượng và đặt các địa chỉ của chúng trong mảng, chúng ta có thể đơn giản hóa bằng cách khởi tạo mảng dùng các hàm tạo đối tượng:

```
shape* sharray[N]={ &bowl(10,0,3),&square(20,1,5),&cap(30,1,7)};
```

3. Hàm ảo và hàm tạo (*virtual function and constructor*)

Các hàm tạo trong các lớp ở chương trình VIRTSHAP không phải ảo. Các hàm tạo có bao giờ là ảo không? Không, không bao giờ. Các hàm ảo không thể tồn tại cho đến khi hàm tạo đã hoàn thành nhiệm vụ của nó, bởi vậy các hàm tạo không thể là ảo.

Ngoài ra, khi tạo một đối tượng chúng ta thường đã biết loại đối tượng mà chúng ta sẽ tạo ra và có thể chỉ rõ loại này cho trình biên dịch biết. Do đó không cần các hàm tạo ảo như các hàm ảo.

Trái lại, các hàm hủy (**destructor**) có thể và thường nên là ảo. Chúng ta sẽ nói về các hàm hủy ảo trong mục 4.4.

4.3. GỠ KẾT NỐI VỚI SỰ ĐA HÌNH THÁI

Trong phần trước chúng ta đã biết hai ví dụ trong đó sự đa hình thái được sử dụng trong một mảng con trở tới các đối tượng. Tuy nhiên, đây không phải là tình huống duy nhất mà ở đó sự đa hình thái và các hàm ảo đóng vai trò quan trọng. Sự đa hình thái cũng có thể được sử dụng để giúp cho việc phân lập, hay gỡ kết nối một phần của chương trình ra khỏi chương trình khác.

Như chúng ta đã lưu ý trước đây, các chương trình OOP được chia thành hai phần, chúng thường được viết bởi những người lập trình khác nhau tại các thời điểm khác nhau. Đầu tiên, các lớp được viết bởi một nhóm người lập trình (người tạo lớp); sau đó một thời gian, chương trình sử dụng các lớp này được một nhóm người lập trình khác viết (những người sử dụng lớp). Một lợi ích thiết thực của việc sử dụng lại là cùng một tập các lớp có thể được sử dụng nhiều lần bởi các người sử dụng lớp khác nhau.

Trong hầu hết các ví dụ mà chúng ta đã viết, chương trình sử dụng lớp chứa một hàm đơn **main()**. Điều này làm cho các ví dụ dễ hiểu hơn nhưng hoàn toàn không thực tế. Chương trình của người sử dụng trong các chương trình quan trọng thường được chia thành nhiều hàm, chúng sẽ được gọi ra từ hàm **main()** hoặc từ các hàm khác.

Sự đa hình thái xuất hiện ở đâu trong tình huống này? Việc lập trình có thể được đơn giản hóa nếu chương trình của người sử dụng có thể làm việc với một lớp chung hơn là cố gắng biết nhiều lớp khác nhau. Bởi vì các đối tượng (hay các tham chiếu hoặc các con trỏ tới các đối tượng) có thể sẽ được truyền và trả về từ một hàm tới một hàm khác và nếu tất cả các đối số này và các giá trị trả về có thể thuộc một lớp thì việc viết chương trình sẽ đơn giản. Nói cách khác, gỡ kết nối chương trình người sử dụng khỏi các lớp cụ thể là rất hữu ích.

Ví dụ thứ nhất sẽ chỉ ra cách mà sự đa hình thái được sử dụng khi các tham chiếu là các đối số hàm. Một ví dụ khác chỉ ra cách mà sự đa hình thái được sử dụng với con trỏ là đối số hàm. Cuối cùng chúng ta cùng xét một chương trình lớn hơn minh họa một tình huống lập trình thiết thực hơn.

4.3.1. Truyền các tham chiếu

Để có một ví dụ đơn giản về sử dụng sự đa hình thái gỡ kết nối các chương trình của người sử dụng khỏi các lớp, hãy tưởng tượng một lớp cơ sở có nhiều lớp dẫn xuất. Giả sử hàm **main()** gọi một hàm toàn cục **func()** để làm một cái gì đó với các đối tượng của tất cả các lớp dẫn xuất. Nếu hàm **func()** cần phải lo lắng về việc quản lý nhiều đối tượng khác nhau với nhiều cách khác nhau thì chương trình của nó sẽ trở nên rất phức tạp. Tuy nhiên, sử dụng các tham chiếu và các hàm ảo, ta có thể dùng cùng một hàm **func()** cho nhiều lớp khác nhau.

Khi một đối tượng được truyền tới hàm theo tham chiếu, nhìn vào cú pháp thì tưởng đối tượng được truyền nhưng thực ra một con trỏ tới đối tượng được truyền thay thế. Như chúng ta đã biết, các con trỏ luôn luôn có kích thước giống nhau bất kể nó trỏ tới đối tượng nào, bởi vậy sẽ không mất thông tin nếu một tham chiếu tới một đối tượng lớp dẫn xuất được truyền tới một hàm cần một tham chiếu tới một đối tượng lớp cơ sở.

Trong ví dụ, có một lớp cơ sở và hai lớp dẫn xuất, **Derv1** và **Derv2**. Các lệnh trong **main()** gọi **func()** sử dụng các đối số tham chiếu để truyền các đối tượng **Derv1** và **Derv2**. Chú ý rằng cùng một hàm **func()** mà quản lý cả hai đối tượng **Derv1** và **Derv2**. Bản 4-6 trình bày chương trình VIRTREF.

Listing 4-6 VIRTREF

```
//virtref.cpp
//kiem tra cac ham ao va truyen doi so theo tham chieu
#include<iostream.h>
class Base                                //lop co so
{
    public:
    virtual void speak()
    {
        cout<<"\nLop co so Base";
    }
};
class Derv1:public Base
{
    public:
    void speak()
    {
        cout<<"\nLop dan xuat Derv1";
    }
};
class Derv2:public Base
{
    public:
    void speak()
    {
        cout<<"\nLop dan xuat Derv2";
    }
};
void main()
{
    void func(Base&);           //khai bao ham
    Derv1 dv1;                 //doi tuong cua lop dan xuat Derv1
    Derv2 dv2;                 //doi tuong cua lop dan xuat Derv2

    func(dv1);                 //truyen d1 theo tham chieu toi func()
    func(dv2);                 //truyen d2 theo tham chieu toi func()
}
void func(Base& obj)
{
    obj.speak();
}
```

Hàm **func()** bảo đối tượng truyền tới nó gọi hàm thành viên **speak()** của nó để hiển thị đối tượng thuộc lớp nào. Đây là kết quả của chương trình:

```
Lop dan xuat Derv1
Lop dan xuat Derv2
```

Bởi vì đối số của hàm **func()** được truyền theo tham chiếu và bởi vì hàm thành viên **speak()** là hàm ảo, hàm **func()** không cần biết loại đối tượng mà nó sẽ gửi thông điệp **speak()** tới. Nó biết đối tượng được rút ra từ **Base** và đó là tất cả những gì nó cần biết. Đối tượng tự nó phải quan tâm tới việc thực hiện hàm **speak()** nào, như kết quả minh họa.

Ngược lại, giả sử sự đa hình thái không được sử dụng và hàm **func()** phải biết loại đối tượng mà nó đang làm việc với. Hoặc là nó sẽ cần một cái gì đó như một lệnh **switch** để quản lý các lớp khác

nhau (một cách để tìm ra đối tượng ở lớp nào) hoặc hàm `main()` sẽ cần chọn một trong nhiều phiên bản khác nhau của hàm `func()`, tùy thuộc vào lớp. Các giải pháp này phức tạp không cần thiết. Bằng việc để cho đối tượng gửi thông báo tới chính nó chúng ta có thể làm cho chương trình của người sử dụng độc lập với lớp đang được sử dụng, kết quả là đơn giản hóa việc lập trình.

Một thuận lợi nữa của cách này là cùng một chương trình người sử dụng có thể làm việc với các lớp thậm chí chưa được biết. Ví dụ, trong VIRTREF, nếu `Derv3` được thêm vào phân cấp lớp thì hàm `func()` sẽ rất thuận lợi khi làm việc với nó, y như nó làm việc với `Derv1` và `Derv2`.

Tất nhiên tác dụng của việc gỡ kết nối này chỉ có hiệu quả nếu sử dụng các đối số là tham chiếu (hoặc con trỏ). Nếu truyền trực tiếp một đối tượng lớp dẫn xuất tới hàm thì kích thước của nó sẽ bị cắt giảm để bằng kích thước của đối tượng lớp cơ sở, giống như việc gán nó cho một biến lớp cơ sở. Ngoài ra chúng ta phải sử dụng các hàm ảo. Nếu `speak()` không phải là hàm ảo thì kết quả của chương trình sẽ là:

```
Lop co so Base
```

```
Lop co so Base
```

Trong C++, sự đa hình thái phụ thuộc vào các hàm ảo.

Mặc dù không thấy nhiều lợi ích qua việc sử dụng sự đa hình thái trong ví dụ ngắn trên nhưng trong một chương trình lớn hơn, với nhiều lớp dẫn xuất và nhiều hàm trong phần chương trình người sử dụng lớp, nó sẽ có hiệu quả hơn nhiều đối với các hàm viết dưới dạng một lớp cơ sở.

4.3.2. Truyền con trỏ

Tác dụng của việc gỡ kết nối là như nhau dù sử dụng tham chiếu tới các đối tượng hay sử dụng các con trỏ tới các đối tượng. Chúng ta cùng viết lại chương trình VIRTREF để sử dụng con trỏ thay cho tham chiếu.

Listing 4-7 VIRTPTR

```
//virtptr.cpp
//kiem tra cac ham ao va truyen doi so theo con tro
#include<iostream.h>
class Base                                //lop co so
{
public:
    virtual void speak()
    {
        cout<<"\nLop co so Base";
    }
};
class Derv1:public Base
{
public:
    void speak()
    {
        cout<<"\nLop dan xuat Derv1";
    }
};
class Derv2:public Base
{
public:
    void speak()
    {
        cout<<"\nLop dan xuat Derv2";
    }
};
```

```

};
void main()
{
    void func(Base*);           //khai bao ham
    Derv1 dv1;                 //doi tuong cua lop dan xuat Derv1
    Derv2 dv2;                 //doi tuong cua lop dan xuat Derv2

    func(&dv1);                //truyen d1 theo tham chieu toi func()
    func(&dv2);                //truyen d2 theo tham chieu toi func()
}
void func(Base* ptr)          //dinh nghia ham
{
    ptr->speak();
}

```

Các lớp giống các lớp trong chương trình VIRTREF nhưng trong hàm **main()** truyền địa chỉ của các đối tượng hơn là truyền tham chiếu của các đối tượng, tới hàm **func()** và sử dụng toán tử **->** để truy nhập hàm **speak()**. Kết quả đưa ra là:

```

Lop dan xuat Derv1
Lop dan xuat Derv2

```

Một cách điển hình, các tham chiếu thích hợp khi các đối tượng được tạo qua các định nghĩa để tên của chúng được biết, trái lại các con trỏ được sử dụng khi các đối tượng được tạo với **new** và chỉ có các con trỏ trỏ tới chúng mới làm việc. Các tham chiếu an toàn hơn các con trỏ. Giá trị của con trỏ có thể bị thay đổi bởi người lập trình, có thể do vô tình, trái lại một tham chiếu, một khi đã được khởi tạo, không thể thay đổi được.

4.3.3. Ví dụ lớp person

Chúng ta cùng xem xét một ví dụ cung cấp nhiều đặc điểm để thấy được sự đa hình thái làm chương trình độc lập với các lớp cụ thể như thế nào. Ta chỉ cần viết lại phần sử dụng lớp của ví dụ VIRTPERS ở mục 4.2. Mục đích là cho người sử dụng cuối cùng nhiều chức năng điều khiển hơn ở trong chương trình. Chương trình sẽ hiển thị một danh sách các tùy chọn và người sử dụng có thể chọn hoặc thêm một người vào cơ sở dữ liệu hoặc hiển thị tất cả những người trong cơ sở dữ liệu, hoặc là thoát khỏi chương trình. Người sử dụng có thể thêm bao nhiêu người cũng được nếu muốn và hiển thị tất cả chúng tại bất kỳ thời điểm nào.

Để cài đặt khả năng mạnh mẽ này, chúng ta chia phần chương trình sử dụng lớp thành một vài hàm. Một lệnh **switch** trong hàm **main()** sẽ gọi hàm thích hợp tùy thuộc vào ký tự mà người sử dụng gõ vào. Có hai hàm: **getPerson()** lấy dữ liệu từ người sử dụng cho một người được thêm vào cơ sở dữ liệu và hàm **displayPerson()** hiển thị dữ liệu về một người. Các lớp y như trong chương trình VIRTPERS. Bản 4-8 trình bày chương trình PERFUNC.

Listing 4-8 PERFUNC

```

//perfunc.cpp
//truyen con tro tro toi cac doi tuong su dung ham ao
#include<iostream.h>
#include<process.h>           //cho exit()
class person
{
    protected:
        char name[40];
    public:
        virtual void getData()

```



```

        {
            cout<<"\nNhap vao ten:"; cin>>name;
        }
        virtual void putData()
        {
            cout<<"\nTen:"<<name;
        }
        virtual void isOutstanding()
        {
            //chu y:than ham rong
        };

class student:public person //lop sinh vien
{
    private:
        float tbc; //diem trung binh chung
    public:
        void getData()
        {
            person::getData();
            cout<<" Nhap vao diem trung binh chung cua sinh vien:";
            cin>>tbc;
        }
        void putData()
        {
            person::putData();
            cout<<"\n Diem TBC="<<tbc;
        }
        void isOutstanding()
        {
            if(tbc>=9.0)
                cout<<"(Day la nguoi xuat sac)";
        }
};

class teacher:public person //lop giao vien
{
    private:
        int numPubs; //so sach xuat ban
    public:
        void getData()
        {
            person::getData();
            cout<<" Nhap vao so sach xuat ban cua giao vien:";
            cin>>numPubs;
        }
        void putData()
        {
            person::putData();
            cout<<"\n So sach xuat ban="<<numPubs;
        }
        void isOutstanding()
        {
            if(numPubs>=100)
                cout<<"(Day la nguoi xuat sac)";
        }
};

```

```

};

void main()
{
    person* persPtr[100];    //danh sach con tro tro toi cac
                             //doi tuong person
    int n=0;                 //so nguoi trong danh sach
    char choice;
    int j;
    person* getPerson();     //khai bao ham
    void displayPerson(person*);
    while(1)                 //lap lai cho den khi thoat
    {
        cout<<endl
            <<"Go 't' de them mot nguoi moi"<<endl
            <<"Go 'h' de hien thi tat ca moi nguoi"<<endl
            <<"Go 'k' de ket thuc chuong trinh"<<endl
            <<"\nChon gi? ";
        cin>>choice;
        switch(choice)
        {
            case 't':
                persPtr[n++]=getPerson();
                break;
            case 'h':
                for(j=0;j<n;j++)
                    displayPerson(persPtr[j]);
                break;
            case 'k':
                for(j=0;j<n;j++)
                    delete persPtr[j];    //xoa tat ca cac doi tuong
                exit(0);
                break;
            default:
                cout<<"\nKhong chon nhu vay!";
        }    //ket thuc switch
    }    //ket thuc while
}    //ket thuc main()

////////////////////////////////////
//dinh nghia cac ham
person* getPerson()    //ham tra ve mot nguoi
{
    person* tp;    //con tro tro toi person
    char choice;
    cout<<"\nNhap vao mot nguoi binh thuong,sinh vien hoac giao vien"
        <<"(b,s,g).";
    cin>>choice;
    if(choice=='s')
        tp=new student;    //dat mot sinh vien vao danh sach
    else if(choice=='g')
        tp=new teacher;    //dat mot giao vien vao danh sach
    else
        tp=new person;    //dat mot nguoi binh thuong vao
                             //danh sach
    tp->getData();    //lay du lieu
}

```

```

        return tp;
    }
    void displayPerson(person* pp)    //hien thi mot nguoi va
    {                                //chu thich neu xuat sac
        pp->putData();
        pp->isOutstanding();
    }

```

Chú ý là trong hàm **main()** và hàm **displayPerson()** không đề cập chút nào tới các đối tượng ngoài đối tượng **person**. Hàm **displayPerson()** có một đối số là một con trỏ trỏ tới **person**. Nơi duy nhất mà các đối tượng **student** và **teacher** được nói đến rõ ràng là trong hàm **getPerson()**, ở đó các đối tượng mà người sử dụng yêu cầu được tạo với **new**.

Việc gỡ kết nối chương trình người sử dụng khỏi các lớp cụ thể làm cho các hàm này không hề bị ảnh hưởng bởi bất kỳ thay đổi nào trong phân cấp lớp. Dù cho người tạo lớp tạo ra một bản sửa chữa về các lớp, thêm các lớp dẫn xuất mới cho chương trình, chẳng hạn:

```

class administrator:public person
{
};
class football_coach:public person
{
};

```

thì chúng ta cũng không phải thay đổi gì trong hàm **main()** hay hàm **displayPerson()** và chỉ cần thêm một vài dòng trong **getPerson()**.

4.4. LỚP TRỪU TƯỢNG VÀ HÀM HỦY ẢO

Trong phần này chúng ta sẽ nói đến hai chủ đề liên quan tới các hàm ảo. Thứ nhất là các lớp trừu tượng (**abstract class**), chúng là các lớp mà không có đối tượng nào được tạo ra từ chúng, chúng chỉ đóng vai trò là lớp cơ sở cho các lớp dẫn xuất. Các lớp trừu tượng được thực hiện trong C++ bằng các hàm ảo thuần túy. Chủ đề thứ hai là các hàm hủy ảo (**virtual destructor**), chúng là một ý tưởng tốt nếu không muốn các phần của đối tượng cũ nằm trong bộ nhớ.

4.4.1. Lớp trừu tượng

Trong chương trình **VIRTPERS** ở mục 4.2, chúng ta đã tạo các đối tượng lớp cơ sở **person** cũng như các đối tượng lớp dẫn xuất **student** và **teacher**. Trái lại, trong chương trình **VIRTSHAP** chúng ta lại không tạo bất kỳ đối tượng lớp cơ sở **shape** nào, mà chỉ tạo các đối tượng lớp dẫn xuất **bowl**, **cap** và **square**. Trong **VIRTPERS**, chúng ta cho đối tượng chung **person** có ích trong chương trình, nhưng trong chương trình **VIRTSHAP**, lớp **shape** tồn tại chỉ là điểm bắt đầu cho các lớp dẫn xuất khác và không có ý nghĩa gì khi tạo một đối tượng chung **shape**.

Chúng ta nói lớp **shape** trong chương trình **VIRTSHAP** là một lớp trừu tượng (**abstract class**), có nghĩa là không có đối tượng thực nào được rút ra từ nó. Các lớp trừu tượng xuất hiện trong nhiều tình huống. Một nhà máy có thể làm một xe hơi đua, một xe tải hay một xe cứu thương, nhưng không thể làm một xe chung. Nhà máy phải biết các chi tiết về loại xe nào cần có trước khi thực sự làm nó.

Một vài tình huống thì yêu cầu một lớp cơ sở trừu tượng, nhưng một vài tình huống khác lại không. Thường không cần là nhiều hơn. Ví dụ, trong chương trình **VIRTPERS**, có thể là tốt hơn nếu cho **person** là một lớp trừu tượng và rút ra một lớp **miscellaneous** cho tất cả các đối tượng **person** không phải là **student** hoặc **teacher**.

Trên thực tế, lớp **shape** trong ví dụ **VIRTSHAP** là một lớp trừu tượng chỉ trong mắt con người. Trình biên dịch bỏ qua quyết định của chúng ta là bắt nó làm một lớp trừu tượng, nó sẽ không sao nếu chúng ta đặt:

```
shape s3(5,6,7); //hợp lệ nhưng không có nghĩa
```

Chúng ta đã sắp xếp để hàm **draw()** trong lớp **shape** hiển thị một thông báo lỗi:

```
class Base
{
public:
    virtual void draw()
    {cout<<"\nBan ao cua draw() ";}
};
```

Giải pháp này giúp cho ta không cố vẽ một hình shape nhưng nó không phải là một giải pháp hoàn hảo.

4.4.2. Hàm ảo tĩnh khiết

Giả thiết chúng ta đã quyết định tạo một lớp cơ sở trừu tượng, sẽ hơn nếu chúng ta cho trình biên dịch cương quyết ngăn cản bất kỳ người sử dụng lớp nào cố tình tạo một đối tượng của lớp đó. Điều này giúp chúng ta thoải mái hơn trong việc thiết kế lớp cơ sở bởi vì chúng ta không cần trù tính cho các đối tượng thật sự của lớp mà chỉ trù tính cho dữ liệu và các hàm sẽ được sử dụng bởi các lớp dẫn xuất. Để làm được điều này, chúng ta phải định nghĩa ít nhất một hàm ảo tĩnh khiết trong lớp đó.

Một hàm ảo tĩnh khiết (**pure virtual function**) là một hàm ảo không có thân hàm. Thân của hàm ảo trong lớp cơ sở được bỏ đi và thêm vào khai báo hàm ký hiệu = 0.

1. Ví dụ ngắn **VIRT**

Ví dụ thứ nhất về một hàm ảo tĩnh khiết được rút ra từ chương trình **VIRT** trong mục 4.1.

Listing 4-9 **VIRTPURE**

```
//virtpure.cpp
//ham ao tinh khiet
#include<iostream.h>
class Base //lop co so
{
public:
    virtual void show()=0; //ham ao tinh khiet
};
class Derv1:public Base //lop dan xuat 1
{
public:
    void show()
    {
        cout<<"\nLop dan xuat Derv1";
    }
};
class Derv2:public Base //lop dan xuat 2
{
public:
    void show()
    {
```

```

        cout<<"\nLop dan xuat Derv2";
    }
};
void main()
{
    Derv1 dv1;        //doi tuong cua lop dan xuat Derv1
    Derv2 dv2;        //doi tuong cua lop dan xuat Derv2
    //Base ba;        //loi:khong the tao doi tuong cua lop tru tuong
}

```

Bây giờ hàm ảo được khai báo là:

```
virtual void show()=0; //ham ảo tinh khiet
```

Dấu bằng ở đây không phải là toán tử gán; giá trị 0 không được gán cho cái gì. Cú pháp = 0 chỉ đơn giản là cách chỉ cho trình biên dịch biết rằng một hàm sẽ là tinh khiết, nghĩa là sẽ không có thân hàm.

Chúng ta có thể bắn khoản nếu có thể xóa được thân của hàm ảo **show()** trong lớp cơ sở thì tại sao không thể xóa luôn cả hàm. Điều đó thậm chí sạch hơn nhưng không làm việc. Không có hàm ảo **show()** trong lớp cơ sở, các câu lệnh như:

```

Base* list[3];
list[0]= new Derv1;
list[0]->show();           //không thể làm điều này

```

sẽ không hợp lệ bởi vì phiên bản **show()** trong lớp cơ sở sẽ luôn được thực hiện.

2. Ví dụ shape

Nếu chúng ta viết lại lớp **shape** trong chương trình **VIRTSHAP** để sử dụng một hàm ảo tinh khiết cho **show()** thì chúng ta có thể ngăn chặn được việc tạo bất kỳ đối tượng lớp cơ sở nào. Bản 4-10 trình bày chương trình **PURESHAP**.

Listing 4-10 PURESHAP

```

//pureshap.cpp
//ve cac hinh bang cac dau x tren man hinh van ban
//su dung ham ảo tinh khiet draw()
#include<iostream.h>
////////////////////////////////////
class shape           //lop cơ sở
{
private:
    int xco,yco;       //toa do cua hinh
    int size;          //kich thước cua hinh
protected:           //cac ham chỉ doc
    int getx() const
    { return xco;}
    int gety() const
    { return yco;}
    int getz() const
    { return size;}
    void down() const; //khai báo ham
public:
    //ham tạo 3 doi so
    shape(int x,int y,int s):xco(x),yco(y),size(s)
    {}
}

```

```

        virtual void draw() const=0; //ham ao tinh khiet
    };

    void shape::down() const
    {
        for(int y=0;y<yco;y++)
            cout<<endl;
    }
    //////////////////////////////////////
    class square:public shape    //hinh vuong
    {
    public:
        //ham tao 3 doi so
        square(int x, int y,int s):shape(x,y,s)
        {}
        void draw() const;        //khai bao
    };
    void square::draw() const    //ve mot hinh vuong
    {
        shape::down();          //di chuyen toi dinh cua hinh
        for(int y=0;y<getz();y++) //chuyen y xuong ngang qua hinh
        {
            int x;
            for(x=1;x<getx();x++) //lay khoang trong tren hinh
                cout<<' ';
            for(x=0;x<getz();x++) //ve duong thang bang dau x
                cout<<'x';
            cout<<endl;
        }
    }
    //////////////////////////////////////
    class cap:public shape        //hinh cai mu
    {
    public:
        //ham tao 3 doi so
        cap(int x, int y,int s):shape(x,y,s)
        {}
        void draw() const;        //khai bao
    };
    void cap::draw() const        //ve mot hinh cai mu
    {
        shape::down();          //di chuyen toi dinh cua hinh
        for(int y=0;y<getz();y++) //chuyen y xuong ngang qua hinh
        {
            int x;
            for(x=0;x<getx()-y+1;x++)//lay khoang trong tren hinh
                cout<<' ';
            for(x=0;x<2*y+1;x++)//ve duong thang bang dau x
                cout<<'x';
            cout<<endl;
        }
    }
    //////////////////////////////////////
    class bowl:public shape      //hinh cai bat

```

```

{
    public:
        //ham tao 3 doi so
        bowl(int x, int y, int s):shape(x,y,s)
        {}
        void draw() const;           //khai bao
};
void bowl::draw() const           //ve mot hình cái bát
{
    shape::down();                //di chuyen toi dinh cua hình
    for(int y=0;y<getz();y++)      //chuyen y xuong ngang qua hình
    {
        int x;
        for(x=0;x<getx()-(getz()-y)+2;x++) //lay khoang trong tren hình
            cout<<' ';
        for(x=0;x<2*(getz()-y)-1;x++)      //ve duong thang bang dau x
            cout<<'x';
        cout<<endl;
    }
}
////////////////////////////////////
void main()
{
    //shape sh(1,2,3);
    const int N=3;                //so hình
    shape* sharray[N];            //mang con tro tro toi shape
    bowl bw(10,0,3);              //tao mot cái bát
    square sq(20,1,5);            //tao mot hình vuông
    cap cp(30,1,7);               //tao mot cái mũ

    sharray[0]=&bw;               //dat dia chi cua chung vao trong mang
    sharray[1]=&sq;
    sharray[2]=&cp;

    cout<<endl<<endl;            //bat dau duoi 2 dòng
    for(int j=0;j<N;j++)
        sharray[j]->draw();      //hien thi ca 3 hình
}

```

3. Trình biên dịch làm nhiệm vụ

Chúng ta không thể tạo một đối tượng của một lớp chứa một hàm ảo tĩnh khiết và không thể viết một lời gọi hàm truyền hoặc trả về một đối tượng của lớp như vậy theo giá trị. Do đó trong hàm **main()** của chương trình PURESHAP chúng ta không thể khai báo hàm như thế này:

```

void func(shape);    //lời
shape func();        //lời

```

Trình biên dịch biết rằng việc truyền và trả về theo giá trị sẽ tạo một đối tượng và nó sẽ không để điều này xảy ra với một lớp trừu tượng. Truyền hoặc trả về các đối tượng của một lớp trừu tượng theo tham chiếu hoặc con trỏ thì được bởi vì con trỏ sẽ thực sự trỏ tới các đối tượng lớp dẫn xuất.

4. Lớp trừu tượng và hàm ảo tĩnh khiết

Sử dụng các hàm ảo tĩnh khiết làm tín hiệu để trình biên dịch biết một lớp sẽ là ảo đường như là tùy tiện. Tuy nhiên, các hàm ảo tĩnh khiết và các lớp trừu tượng là hai mặt của một vấn đề. Nếu

không bao giờ tạo các đối tượng lớp cơ sở thì chẳng có hại gì khi đặt các hàm ảo tĩnh khiết trong lớp đó. Ngoài ra, một lớp trừu tượng sẽ, bằng cách định nghĩa, có các lớp khác rút ra từ nó. Để sử dụng sự đa hình thái, các lớp dẫn xuất này sẽ yêu cầu các hàm ảo trong lớp cơ sở. Thường thì có ít nhất một hàm như vậy không làm bất cứ việc gì trong lớp cơ sở. Chúng ta gọi các hàm như vậy là các hàm ảo tĩnh khiết và để trình biên dịch biết về chúng với ký hiệu = 0.

5. Hàm ảo tĩnh khiết có thân hàm

Đôi khi chúng ta muốn chuyển một lớp cơ sở thành một lớp trừu tượng, nhưng lớp này cần các hàm thành viên không có thân hàm. Tất cả các hàm trong lớp cơ sở có thể được truy nhập bởi các hàm trong các lớp dẫn xuất. Chúng ta có thể chuyển một hàm có thân hàm thành một hàm ảo tĩnh khiết y như với một hàm không có thân hàm. Đây là một ví dụ về nhân viên, EMPINH. Để chuyển lớp **employee** thành một lớp trừu tượng ta thêm ký hiệu = 0 vào hàm **getData()** trong lớp **employee**, nhưng giữ thân hàm như cũ:

```
virtual void getData()=0
{
    cout<<"\nNhap vao ten:"; cin>>name;
    cout<<" Nhap vao ma so:"; cin>>number;
}
```

4.4.3. Hàm hủy ảo

Nếu chúng ta sử dụng bất kỳ hàm ảo nào, tĩnh khiết hay không tĩnh khiết, trong một lớp thì có thể chúng ta sẽ muốn làm cho hàm hủy của lớp đó là ảo. Tại sao? Để hiểu được điều này chúng ta cùng xét chương trình NOVIDEST, trong đó có một lớp cơ sở và một lớp dẫn xuất, cả hai đều có các hàm hủy không ảo.

Listing 4-11 NOVIDEST

```
//novidest.cpp
//ham huy lop co so khong ao
#include<iostream.h>

class Base
{
public:
    ~Base()
    { cout<<"\nHam huy lop co so Base";}
};
class Derv:public Base
{
public:
    ~Derv()
    { cout<<"\nHam huy lop dan xuat Derv";}
};
void main()
{
    Base* pb=new Derv;
    delete pb;           //dua ra man hinh "Ham huy lop co so Base"
    cout<<"\nChuong trinh ket thuc";
}
```

1. Hàm hủy lớp dẫn xuất có được thực hiện không?

Nhớ lại rằng một đối tượng của lớp dẫn xuất chứa dữ liệu của cả lớp cơ sở và lớp dẫn xuất (mặc dù trong chương trình NOVIDEST thì không có như vậy bởi vì không có bất cứ dữ liệu nào trong cả

hai lớp). Để đảm bảo dữ liệu như vậy hủy bỏ đi một cách thích đáng, cần gọi hàm hủy của lớp cơ sở và lớp dẫn xuất. Kết quả của chương trình NOVIDEST là:

Ham hủy lop co so Base
Chương trình kết thúc

Đây là một vấn đề giống như chúng ta đã thấy trước đây với các hàm thông thường (không phải hàm hủy). Nếu một hàm không phải là ảo thì chỉ có phiên bản của hàm lớp cơ sở được thực hiện khi nó được gọi bằng cách dùng con trỏ lớp cơ sở, cho dù giá trị của con trỏ là địa chỉ của một đối tượng lớp dẫn xuất. Bởi vậy, trong chương trình NOVIDEST, hàm hủy lớp dẫn xuất **Derv** không bao giờ được thực hiện. Đây có thể là một vấn đề nếu hàm này làm một việc gì đó quan trọng.

2. Không được thực hiện trừ khi nó là ảo

Để giải quyết vấn đề này chúng ta làm cho hàm hủy lớp cơ sở là ảo, như chỉ ra trong chương trình VIRTDEST dưới đây:

Listing 4-12 VIRTDEST

```
//virtdest.cpp
//kiem tra ham hủy ảo
#include<iostream.h>
class Base
{
public:
    virtual ~Base()
        { cout<<"\nHam hủy lop co so Base";}
};
class Derv:public Base
{
public:
    ~Derv()
        { cout<<"\nHam hủy lop dan xuất Derv";}
};

void main()
{
    Base* pb=new Derv;
    delete pb;                //đưa ra màn hình "Ham hủy lop co so Base"
    cout<<"\nChương trình kết thúc";
}
```

Kết quả của chương trình là:

Ham hủy lop dan xuất Derv
Ham hủy lop co so Base
Chương trình kết thúc

Bây giờ cả hai hàm được gọi. Tất nhiên trong ví dụ đơn giản này, chẳng có vấn đề gì nếu hàm hủy lớp không được gọi bởi vì không có việc gì để nó làm. Bây giờ chúng ta cùng xem một ví dụ mà ở đó việc gọi hàm hủy lớp dẫn xuất có ý nghĩa hơn nhiều.

3. Ví dụ

Các đối tượng lớp dẫn xuất có thể sử dụng tài nguyên hệ thống cần được giải phóng khi đối tượng bị hủy. Có thể các đối tượng lớp dẫn xuất sử dụng **new** để cấp phát bộ nhớ. Rất cần có một **delete** tương ứng trong hàm hủy lớp dẫn xuất đó và hàm hủy được gọi khi đối tượng lớp dẫn xuất đó bị hủy. Nếu không làm như vậy thì vùng nhớ được cấp phát bởi đối tượng lớp dẫn xuất sẽ trở nên cô

lập, không thể truy nhập nhưng vẫn chiếm không gian nhớ, gây lãng phí tài nguyên hệ thống và là nguyên nhân dẫn đến những vấn đề bộ nhớ nghiêm trọng.

Ví dụ tiếp theo là một chương trình mở rộng của PERSFUNC sẽ cho thấy vấn đề này như thế nào. Nó có lớp **person** và lớp **gradstudent** (sinh viên năm cuối) rút ra từ lớp **person**. Trong chương trình chúng ta sử dụng **new** để cấp phát bộ nhớ cho tên và đề tài của sinh viên năm cuối và các hàm hủy để xóa các vùng nhớ này. Bản 4-13 trình bày chương trình DESTPERS.

Listing 4-13 DESTPERS

```
//destpers.cpp
//ham huy ao va lop person
#include<iostream.h>
#include<string.h>           //cho strlen(),strcpy()
class person
{
protected:
    char* nameptr;           //con tro tro toi ten
public:
    person(char* np)         //ham tao mot doi so
    {
        int length=strlen(np);           //tim do dai cua ten
        nameptr= new char[length+1];     //cap phat bo nho
        strcpy(nameptr,np);              //dat ten vao bo nho
    }
    virtual ~person()          //ham huy ao
    {
        cout<<"\nHam huy lop co so person";
        if(nameptr!=NULL)       //neu duoc su dung thi xoa
            delete[] nameptr;
    }
    virtual void putData()
    {
        cout<<"\nTen:"<<nameptr;
    }
};

class gradstudent:public person    //lop sinh vien nam cuoi
{
private:
    char* topicptr;              //con tro tro toi de tai luan an
public:
    gradstudent(char* n,char* t):person(n),topicptr(NULL)
    {
        int length=strlen(t);           //tim do dai cua ten de tai
        topicptr=new char[length+1];     //cap phat bo nho
        strcpy(topicptr,t);
    }
    ~gradstudent()                //ham huy
    {
        cout<<"\nHam huy lop dan xuat gradstudent";
        if(topicptr!=NULL)           //neu duoc su dung thi xoa
            delete[] topicptr;
    }
    void putData()
    {

```

```

        person::putData();
        cout<<"\n De tai luan an:"<<topicptr;
    }
};
void main()
{
    int j;
    const int total=3;
    person* persPtr[total];    //danh sach con tro tro toi cac
                                //doi tuong person
    char name[40];              //luu tru trung gian
    char topic[80];
    for(j=0;j<total;j++)        //nhan du lieu, tao cac sinh vien nam cuoi
    {
        cout<<"\nNhap vao ten:";
        cin>>name;
        cout<<"Nhap vao de tai luan an:";
        cin>>topic;
        persPtr[j]=new gradstudent(name,topic);
    }
    for(j=0;j<total;j++)        //hien thi cac sinh vien
        persPtr[j]->putData();

    cout<<endl<<endl;

    for(j=0;j<total;j++)        //xoa cac sinh vien
        delete persPtr[j];
}                                //ket thuc main()

```

Các hàm tạo trong **person** và **gradstudent** lấy vùng nhớ bằng toán tử **new** cho tên và đề tài luận án cho sinh viên năm cuối. Các hàm hủy của nó xóa vùng nhớ này. Cả hai hàm hủy đều được gọi cho từng đối tượng, như kết quả đưa ra cho thấy, bởi vậy tất cả các vùng nhớ được cấp phát đều đảm bảo được giải phóng. Đây là vài mẫu tương tác với chương trình:

```

Nhap vao ten: Thang
Nhap vao de tai luan an: NgonNguLapTrinhC++
Nhap vao ten: Binh
Nhap vao de tai luan an: TruyenSoLieu
Nhap vao ten: Hung
Nhap vao de tai luan an: Anten
Ten:Thang
De tai luan an:NgonNguLapTrinhC++
Ten:Binh
De tai luan an:TruyenSoLieu
Ten:Hung
De tai luan an:Anten

```

```

Ham huy lop dan xuat gradstudent
Ham huy lop co so person
Ham huy lop dan xuat gradstudent
Ham huy lop co so person
Ham huy lop dan xuat gradstudent
Ham huy lop co so person

```

4. Khi nào sử dụng các hàm ảo ?

Tạo một hàm hủy ảo trong bất kỳ lớp nào có các hàm ảo là một cách tốt nhất. Điều này thực sự cần thiết chỉ khi các điều kiện sau là đúng: thứ nhất, các lớp sẽ được rút ra từ lớp đang xét; thứ hai, các đối tượng lớp dẫn xuất sẽ được xóa bằng cách sử dụng các con trỏ lớp cơ sở; thứ ba, các hàm hủy trong bất kỳ lớp liên quan nào làm một việc gì đó quan trọng như việc hoàn trả các tài nguyên. Nếu một lớp có các hàm ảo thì có thể cần tất cả các điều kiện này.

Khi một hàm trong một lớp là ảo, không có các điều kiện trên để bắt các hàm khác là ảo bởi vì ngay khi một hàm là ảo, một bảng ảo được thêm vào mọi đối tượng. Do đó, không nhất thiết phải tuân theo các nguyên tắc tạo hàm hủy ảo trong các lớp có hàm ảo.

4.5. ĐỊNH DANH KIỂU TẠI THỜI ĐIỂM CHẠY CHƯƠNG TRÌNH

Đôi khi chúng ta cần tìm lớp của một đối tượng. Giả sử chúng ta có một mảng con trỏ trỏ tới các đối tượng và các con trỏ này có thể trỏ tới vài lớp dẫn xuất khác nhau, như trong chương trình VIRTPEERS ở mục 4.2. Nếu một hàm toàn cục truy nhập tới một trong các con trỏ này, làm thế nào hàm này có thể tìm ra loại đối tượng mà nó trỏ tới?

Chúng ta biết rằng có thể sử dụng một con trỏ như vậy để gọi một hàm ảo cho đối tượng đó và hàm thích hợp sẽ được thực hiện tùy thuộc vào kiểu của đối tượng bởi vì cơ chế hàm ảo biết loại đối tượng nào mà con trỏ đó trỏ tới. Tuy nhiên, những thông tin này không có sẵn đối với người lập trình.

4.5.1. Ví dụ 1

Hầu hết các trình biên dịch C++ mới đây có một hàm đặc biệt là `typeid()`, nó cho phép tìm ra kiểu (hoặc lớp) của đối tượng. Điều này gọi là **Runtime Type Identification** hay RTTI. Bản 4-14, chương trình TYPEID, cho thấy `typeid()` làm việc như thế nào.

Listing 4-14 TYPEID

```
//typeid.cpp
//minh họa hàm typeid()
#include<iostream.h>
#include<typeid.h>

class ClassA
{
};
class ClassB
{
};

void main()
{
    ClassA objA;
    ClassB objB;
    if(typeid(objA)==typeid(ClassA))
        cout<<"\nobjA là một đối tượng của lớp ClassA";
    else
        cout<<"\nobjA không phải là đối tượng của lớp ClassA";

    if(typeid(objB)==typeid(ClassA))
        cout<<"\nobjB là một đối tượng của lớp ClassA";
    else
        cout<<"\nobjB không phải là đối tượng của lớp ClassA";
}
```

Đây là kết quả của chương trình:

objA là một đối tượng của lớp ClassA
objB không phải là đối tượng của lớp ClassA

Cần có file tiêu đề `TYPINFO.H`. Chúng ta có thể sử dụng hoặc là một tên đối tượng hoặc là một tên lớp làm tham số cho hàm `typeid()`, bởi vậy ta dễ biết một đối tượng thuộc một lớp cụ thể nào. Giá trị trả về từ `typeid()` là một con trỏ rất có ích cho mục đích so sánh, như với toán tử `==` và `!=`. Hàm `typeid()` cũng làm việc với các kiểu cơ bản như `int` và `float`.

4.5.2. Ví dụ 2

Chúng ta sẽ cần RTTI khi chúng ta có các con trỏ lớp cơ sở chứa địa chỉ của các đối tượng lớp dẫn xuất. Đây là một tình huống được mô tả ở phần trước nơi mà các hàm ảo làm cho một lớp cơ sở trở thành đa hình thái. Nó có thể xuất hiện khi chúng ta có một mảng con trỏ trỏ tới các đối tượng hoặc khi các hàm có đối số là các tham chiếu hoặc con trỏ trỏ tới lớp cơ sở.

Bây giờ chúng ta cùng sửa chương trình `VIRTPERS` ở mục 4.2 để nó chỉ ra loại đối tượng `person` (`student`, `teacher` hay `person`). Bản 4-15 là chương trình `RTTIPERS`.

Listing 4-15 RTTIPERS

```
//rttipers.cpp
//xác định kiểu tại thời điểm chạy chương trình
#include<iostream.h>
#include<typeinfo.h>

class person
{
protected:
    char name[40];
public:
    virtual void getData()
    {
        cout<<"\nNhập vào tên:"; cin>>name;
    }
    virtual void putData()
    {
        cout<<"\nTên:"<<name;
    }
};

class student:public person //lớp sinh viên
{
private:
    float tbc; //diem trung binh chung
public:
    void getData()
    {
        person::getData();
        cout<<" Nhập vào diem trung binh chung của sinh viên:";
        cin>>tbc;
    }
    void putData()
    {
        person::putData();
        cout<<"\n Diem TBC="<<tbc;
```

```

    }
};

class teacher:public person    //lop giao vien
{
private:
    int numPubs;                //so sach xuat ban
public:
    void getData()
    {
        person::getData();
        cout<<" Nhap vao so sach xuat ban cua giao vien:";
        cin>>numPubs;
    }
    void putData()
    {
        person::putData();
        cout<<"\n So sach xuat ban="<<numPubs;
    }
};

void main()
{
    person* persPtr[100];        //danh sach con tro tro toi cac
                                //doi tuong person
    int n=0;                     //so nguoi trong danh sach
    char choice;

    do
    {
        cout<<"\nNhap vao mot nguoi binh thuong,sinh vien,hoac giao vien"
            <<"(b,s,g):";
        cin>>choice;
        if(choice=='s')
            persPtr[n]=new student;    //dat mot sinh vien vao danh sach
        else if(choice=='g')
            persPtr[n]=new teacher;    //dat mot giao vien vao danh sach
        else
            persPtr[n]=new person; //dat mot nguoi binh thuong vao
                                //danh sach
        persPtr[n++]->getData(); //nhap du lieu
        cout<<" Co nhap nua khong(c/k)? ";
        cin>>choice;
    }
    while(choice!='c');           //lap lai cho den khi khac 'c'

    //hien thi ten lop
    for(int j=0;j<n;j++)
    {
        if(typeid(*persPtr[j])==typeid(student))
            cout<<"\nStudent, ";
        else if(typeid(*persPtr[j])==typeid(teacher))
            cout<<"\nTeacher, ";
        else if(typeid(*persPtr[j])==typeid(person))
            cout<<"\nPerson, ";
    }
}

```

```

else
    cout<<"\nLoi: khong biet kieu. ";

    persPtr[j]->putData();           //dua ten cua moi nguoi ra man hinh
} //ket thuc for

for(int j=0;j<n;j++) //xoa vung nho duoc cap phat trong chuong trinh
    delete persPtr[j];

}                                     //ket thuc ham main()

```

4.6. HÀM BẠN

Một hàm bạn (**friend function**) là một hàm mà không phải là thành viên của một lớp nhưng truy nhập được tới các thành viên **private** và **protected** của lớp đó.

Thông thường trong khi sử dụng sự bao bọc và cất giấu dữ liệu thì các hàm không phải là thành viên sẽ không thể truy nhập tới dữ liệu **private** và **protected** của một đối tượng. Cách giải quyết đó là: nếu không phải là thành viên thì không thể vào được. Tuy nhiên có nhiều tình huống mà ở đó sự phân biệt nghiêm ngặt như vậy sẽ dẫn đến sự bất tiện lớn. Các hàm bạn là một cách giải tỏa sự bất tiện này.

Tình huống này xuất hiện khi không sử dụng được đối tượng ở bên trái một toán tử chồng. Một nơi khác mà ở đó các hàm bạn có ích là khi sử dụng ký hiệu hàm có một đối tượng là đối số. Chúng ta sẽ xem xét các cách sử dụng hàm bạn này.

4.6.1. Vấn đề "bên trái"

Ví dụ, chúng ta chồng toán tử cộng + để cộng hai đối tượng **airtime** với nhau. Chương trình này làm việc tốt chỉ cần các đối tượng được cộng với nhau thực sự là các đối tượng **airtime**.

```
at3=at1+at2;
```

4.6.2. Vấn đề "bên phải"

Ngay từ đầu, chúng ta cần nhớ là: không có vấn đề gì ở bên phải. Cho rằng với tư cách là người tạo, chúng ta muốn cho người sử dụng lớp có thể viết biểu thức cộng một số nguyên, biểu diễn giờ, tới một **airtime**.

```
at2=at1+3;
```

Ví dụ, 2:33 + 5 sẽ là 7:33. Thực hiện điều này hoàn toàn dễ khi số nguyên ở bên phải của toán tử, như chỉ ra ở trên. Nếu có một hàm tạo một đối số trong lớp **airtime**, trình biên dịch sẽ tự động sử dụng nó để chuyển số nguyên sang giá trị **airtime**. Bản 4-16, AIRNOFRI, trình bày khả năng này.

Listing 4-16 AIRNOFRI

```

//airnofri.cpp
//chong toan tu cong + cho lop airtime
//so nguyen khong the dung o ben trai toan tu
#include<iostream.h>

class airtime
{
private:
    int hours; //tu 0 toi 23

```

```

    int minutes;//tu 0 toi 59
public:    //ham tao 1,2,3 doi so
    airtime(int h=0,int m=0):hours(h),minutes(m)
    {}
void display() //dua ra man hinh
    { cout<<hours<<":"<<minutes;}
void get() //nhan vao gio
    {
        char dummy;
        cin>>hours>>dummy>>minutes;
    }
    airtime operator+(airtime right)    //toan tu + duoc chong
    {
        airtime temp;    //tao mot doi tuong trung gian
        temp.hours=hours+right.hours; //cong du lieu
        temp.minutes=minutes+right.minutes;
        if(temp.minutes>=60)
        {
            temp.hours++;
            temp.minutes-=60;
        }
        return temp;
    }
};

void main()
{
    airtime at1,at2;
    cout<<"\nNhap vao mot airtime(9:45):";
    at1.get();
    at2=at1+3;    //cong so nguyen toi airtime
    cout<<"\nairtime + 3 = ";
    at2.display();    //hien thi
    //at2=3 + at1;    //phep toan cau tao khong hop le
}

```

Hàm tạo trong chương trình này quản lý ba tình huống: không đối số, một đối số và hai đối số (cũng có thể sử dụng ba hàm tạo riêng biệt). Hàm tạo được trình biên dịch sử dụng để chuyển số 3 trong hàm main() sang một giá trị airtime (3:0), sau đó nó được cộng với bất kỳ giá trị airtime nào mà người sử dụng nhập vào cho at1. Đây là vài mẫu tương tác:

```

Nhap vao mot airtime: 9:45
airtime + 3 = 12:45

```

Hàm operator+() có một trợ giúp nhỏ từ hàm tạo, chẳng có vấn đề gì khi không phải đối tượng (là số nguyên) xuất hiện bên phải của toán tử cộng +.

4.6.3. Những chú ý khi ở bên trái

Như chú thích trong chương trình AIRNOFRI, trình biên dịch sẽ báo một lỗi khi ta đặt một giá trị nguyên vào bên trái của toán tử + :

```

at2=3+at1;    //illegal structure operation

```

Tại sao lại không làm việc? Sự chống (overloading) che giấu những gì thực tế đang xảy ra. Nhớ rằng toán tử chống + thực ra chỉ là một hàm thành viên được gọi bởi đối tượng bên trái của nó và có

đối số là đối tượng nằm bên phải. Nếu viết ra những gì trình biên dịch thấy thì có thể vấn đề sẽ rõ hơn. Đây là những gì mà một lời gọi hàm **operator+()** thực sự có, khi số nguyên ở bên phải:

```
at2=at1.operator+(at1); //ok
```

Lệnh này hoàn toàn đúng. Nhưng khi số nguyên ở bên trái:

```
at2=3.operator+(at1); //không hợp lệ, '3' không phải là một đối tượng
```

Các hàm thành viên phải được gọi bởi một đối tượng thuộc lớp của chúng, số 3 không phải là đối tượng của **airtime**. Bởi vậy, mặc dù có thể cộng một số nguyên tới một **airtime** nhưng không thể cộng một **airtime** tới một số nguyên. Đây là một sự không nhất quán quan trọng đối với người sử dụng lớp.

1. Hàm bạn tới cứu nguy

Trong tình huống này, chỉ có hàm bạn mới giúp được. Một hàm bạn (**friend function**) có thể được định nghĩa ở ngoài lớp hoặc ở hàm khác như thể nó là một hàm cục bộ thông thường. Sự kết nối của nó tới lớp là ở bên trong lớp nó được khai báo là hàm bạn. Bản 4-17 trình bày chương trình **AIRFRI**.

Listing 4-17 AIRFRI

```
//airfri.cpp
//chong toan tu cong + cho lop airtime
//su dung ham ban de cho phep so nguyen nam ben trai toan tu +
#include<iostream.h>
class airtime
{
private:
    int hours;          //tu 0 toi 23
    int minutes;        //tu 0 toi 59
public:
    //ham tao 1,2,3 doi so
    airtime(int h=0,int m=0):hours(h),minutes(m)
    {}
    void display()      //dua ra man hinh
    { cout<<hours<<":"<<minutes;}
    void get()          //nhan vao gio
    {
        char dummy;
        cin>>hours>>dummy>>minutes;
    }
    friend airtime operator+(airtime,airtime); //khai bao la ham ban
};
//dinh nghia ham ban:toan tu chong+
airtime operator+(airtime left,airtime right)
{
    airtime temp;
    temp.hours=left.hours+right.hours; //tao mot doi tuong trung gian
    temp.minutes=left.minutes+right.minutes; //cong du lieu
    if(temp.minutes>=60)
    {
        temp.hours++;
        temp.minutes-=60;
    }
    return temp;
} //tra ve gia tri trung gian theo gia tri
void main()
```

```

{
    airtime at1,at2;
    cout<<"\nNhap vao mot airtime(9:45):";
    at1.get();
    at2=at1+3;           //cong so nguyen toi airtime
    cout<<"\nairtime + 3 = ";
    at2.display();       //hien thi
    at2=3 + at1;         //cong airtime toi so nguyen
    cout<<"\n3 + airtime = ";
    at2.display();       //hien thi tong
}

```

Một khai báo hàm làm cho operator+() trở thành hàm bạn là:

```
friend airtime operator+(airtime,airtime);
```

Khai báo có thể được đặt ở bất kỳ đâu trong lớp; dù nó đứng ở **public** hay **private** cũng chẳng sao cả, ít nhất là đối với trình biên dịch. Tuy nhiên, về khái niệm nó thuộc phần **public** bởi vì nó là phần giao diện chung (**public interface**) đối với lớp. Nghĩa là bất kỳ người sử dụng lớp nào cũng có thể gọi hàm bạn, chỉ có các thành viên lớp mới không thể truy nhập được nó.

Như chúng ta có thể thấy, hàm **operator+()** không phải là thành viên của **airtime**; nếu nó là thành viên của **airtime** thì nó phải được khai báo là:

```
airtime operator+(airtime,airtime);
```

Tuy nhiên, nó có thể truy nhập **hours** và **minutes** là thành viên **private** của **airtime**. Nó có thể làm được điều này bởi vì nó được khai báo là một hàm bạn ở bên trong lớp.

Hàm bạn **operator+()** có hai đối số gọi là **left** và **right**. Bởi vì nó là một hàm đứng một mình, không phải là thành viên lớp nên nó không được gọi bởi một đối tượng. Nó được gọi đơn giản từ hàm **main()** như bất kỳ hàm toàn cục nào khác. Bởi vậy, cả hai đối số được cộng có thể dùng làm đối số.

Một nguyên tắc chung là phiên bản bạn của một hàm luôn luôn có nhiều hơn phiên bản thành viên một đối số. Xét ở bên trong thì hàm bạn **operator+()** trong chương trình AIRFRI tương tự phiên bản thành viên trong chương trình AIRNOFRI, trừ nó tham chiếu tới dữ liệu trong các đối số **airtime** như **left.hours** và **right.hours**, trong khi đó phiên bản thành viên sử dụng **hours** và **right.hours**. Hàm trả về tổng là giá trị **airtime** thứ ba.

Nếu một trong các đối số, dù ở bên trái hay bên phải, là một số nguyên, trình biên dịch sẽ dùng hàm tạo một đối số để chuyển số nguyên đó sang một **airtime** và sau đó cộng hai giá trị **airtime**. Các câu lệnh trong hàm **main()** trình bày cả hai tình huống và trình biên dịch sẽ không phản ứng gì cả.

2. Chọc thủng các bức tường

Chú ý rằng, trong khi phát triển C++, các hàm bạn đã gây ra tranh luận. Sự xung đột đã trở nên quyết liệt trên đặc điểm đáng mong muốn này. Một mặt nó thêm vào sự linh hoạt cho ngôn ngữ, mặt khác nó không còn giữ nguyên tắc là chỉ có các thành viên mới có thể truy nhập dữ liệu **private** của lớp.

Lỗ thủng của sự toàn vẹn dữ liệu khi các hàm bạn được sử dụng nguy hiểm như thế nào? Một hàm bạn phải được khai báo theo đúng nghĩa của nó là phải nằm ở bên trong lớp mà có dữ liệu bị truy nhập. Do đó, một người lập trình không được phép truy nhập tới chương trình nguồn đối của các lớp không thể chuyển một hàm thành một hàm bạn. Ở khía cạnh này tính toàn vẹn của lớp vẫn còn được giữ. Hiện tại thì không phải là một nguy cơ đe dọa sự toàn vẹn dữ liệu.

Mặc dù vậy, các hàm bạn vẫn là một sự lộn xộn về khái niệm và có thể tiềm ẩn khả năng một tình huống chương trình không tốt nếu số lượng các hàm bạn làm mờ đi ranh giới rõ ràng giữa các lớp. Vì lý do này, các hàm bạn nên được sử dụng một cách tiết kiệm. Luôn sử dụng một hàm thành viên trừ khi có một lý do bắt buộc phải sử dụng hàm bạn.

4.6.4. Hàm bạn cho phép dùng ký hiệu hàm

Đôi khi một hàm bạn cho một cú pháp gọi hàm rõ ràng hơn hàm thành viên. Ví dụ, cho rằng chúng ta muốn một hàm sẽ bình phương một đối tượng của lớp `English` và trả về một kết quả là `feet` bình phương, như một kiểu `float`. Ví dụ MISQ cho thấy điều này có thể được thực hiện như thế nào với một hàm thành viên.

Listing 4-18 MISQ

```
//misq.cpp
//ham thanh vien square()
#include<iostream.h>

class English
{
private:
    int feet;
    float inches;
public:
    English(int ft,float in):feet(ft),inches(in) //ham tao hai doi so
    {
    }
    void getdist() //lay khoang cach tu nguoi su dung
    {
        cout<<"\nNhap feet:"; cin>>feet;
        cout<<"\nNhap inches:"; cin>>inches;
    }
    void showdist()
    {
        cout<<feet<<"'-"<<inches<<"'";
    }
    float square(); //khai bao ham thanh vien
};

float English::square() //tra ve binh phuong cua doi tuong nay
{
    float fltfeet=feet+inches/12; //chuyen sang float
    float feetsqrd=fltfeet*fltfeet; //tinh binh phuong
    return feetsqrd; //tra ve binh phuong
}

void main()
{
    English dist(3,6.0); //con tro tro toi English
    float sqft=dist.square(); //tra ve binh phuong khoang cach

    cout<<"\nKhoang cach="; dist.showdist();
    cout<<"\nBinh phuong="<<sqft<<" feet binh phuong";
}
```

Trong hàm `main()`, chương trình tạo một giá trị khoảng cách `English`, bình phương nó và đưa kết quả ra màn hình. Kết quả đưa ra cho thấy khoảng cách ban đầu và bình phương của nó.

Khoang cach=3'-6"
Binh phuong=12.25 feet binh phuong
Chúng ta đã sử dụng lệnh :

```
sqft=dist.square();
```

để tìm ra bình phương và gán tới **sqft**. Lệnh này làm việc tốt, nhưng nếu muốn làm việc với các đối tượng dùng cú pháp như với các số, chẳng hạn:

```
sqft=square(dist);
```

thì ta phải làm như thế nào? Có thể đạt được điều này bằng cách chuyển hàm **square()** thành hàm bạn của lớp **English**, như trong chương trình FRISQ dưới đây:

Listing 4-19 FRISQ

```
//misq.cpp
//ham thanh vien square()
#include<iostream.h>

class English
{
private:
    int feet;
    float inches;
public:
    English(int ft,float in):feet(ft),inches(in) //ham tao hai doi so
    {}
    void getdist() //lay khoang cach tu nguoi su dung
    {
        cout<<"\nNhap feet:"; cin>>feet;
        cout<<"\nNhap inches:"; cin>>inches;
    }
    void showdist()
    {
        cout<<feet<<"\'-"<<<inches<<"\'";
    }
    friend float square(English); //khai bao ham ban
};

float square(English d) //tra ve binh phuong cua doi tuong nay
{
    float fltfeet=d.feet+d.inches/12; //chuyen sang float
    float feetsqrd=fltfeet*fltfeet; //tinh binh phuong
    return feetsqrd; //tra ve binh phuong
}

void main()
{
    English dist(3,6.0); //con tro tro toi English
    float sqft=square(dist); //tra ve binh phuong khoang cach
    cout<<"\nKhoang cach="; dist.showdist();
    cout<<"\nBinh phuong="<<sqft<<" feet binh phuong";
}
```

Chú ý: phiên bản bạn của hàm luôn luôn có nhiều hơn phiên bản thành viên một đối số.

4.6.5. Hàm bạn như những chiếc cầu nối

Đây là một tình huống khác mà trong đó các hàm bạn có thể có ích. Hãy tưởng tượng chúng ta có một hàm tính toán trên các đối tượng của hai lớp khác nhau. Có thể hàm này có đối số là các đối tượng của hai lớp đó và tính toán trên dữ liệu `private` của chúng. Nếu hai lớp được kế thừa từ cùng một lớp cơ sở thì chúng ta có thể đặt hàm đó vào trong lớp cơ sở. Nhưng làm thế nào nếu hai lớp đó không có liên quan gì với nhau?

Bản 4-20 trình bày một chương trình ngắn, **BRIDGE**, cho thấy các hàm bạn có thể hoạt động như một chiếc cầu nối giữa hai lớp.

Listing 4-20 **BRIDGE**

```
//bridge.cpp
//cac ham ban
#include<iostream.h>
class beta;           //can cho khai bao ham frifunc()
class alpha
{
    private:
        int data;
    public:
        alpha()          //ham tao khong doi so
        { data=3;}
        friend int frifunc(alpha,beta);    //khai bao ham ban
};
class beta
{
    private:
        int data;
    public:
        beta()            //ham tao khong doi so
        { data=7;}
        friend int frifunc(alpha,beta);    //khai bao ham ban
};
int frifunc(alpha a,beta b)
{
    return (a.data+b.data);
}

void main()
{
    alpha aa;
    beta bb;
    cout<<"Ham ban frifunc(aa,bb)="<<frifunc(aa,bb);
}
```

Trong chương trình này, hai lớp đó là **alpha** và **beta**. Các hàm tạo trong các lớp này khởi tạo các mục dữ liệu đơn của chúng bằng các giá trị cố định (3 trong **alpha** và 7 trong **beta**).

Chúng ta muốn hàm **frifunc()** có thể truy nhập tới các thành viên dữ liệu của hai lớp, bởi vậy chúng ta bắt nó là hàm bạn. Nó được khai báo với từ khóa **friend** trong cả hai lớp.

`friend int frifunc(alpha,beta);`

Đối tượng của mỗi lớp được truyền tới hàm **frifunc()** như một đối số và hàm truy nhập tới thành viên dữ liệu **private**, **data**, của cả hai lớp thông qua các đối số này. Hàm **frifunc()** không làm gì nhiều: nó cộng các mục dữ liệu và trả về tổng. Chương trình **main()** gọi hàm này và đưa ra kết quả.

Một điểm khá quan trọng: nhớ rằng một lớp không thể được tham chiếu tới cho tới khi nó được khai báo. Lớp **beta** được tham chiếu trong khai báo hàm **frifunc()** ở lớp **alpha**, bởi vậy **beta** phải được khai báo trước **alpha**. Do đó khai báo:

```
class beta;

đặt ở đầu chương trình.
```

4.7. LỚP BẠN

Các lớp, cũng như các hàm, có thể là các lớp bạn. Lý do thông thường để sử dụng các lớp bạn là làm cho việc liên lạc giữa các lớp dễ dàng hơn. Trong phần này chúng ta sẽ bắt đầu với một vài ví dụ khung về việc liên lạc như vậy, sau đó là một chương trình mô hình hóa một cuộc đua ngựa với các lớp **track** và **horse** cần liên lạc với nhau.

4.7.1. Liên lạc giữa các lớp

Giả sử chúng ta có hai lớp, **alpha** và **beta**, có liên quan gần gũi với nhau. Trên thực tế chúng liên quan gần gũi nhau đến nỗi mà một lớp cần truy nhập trực tiếp tới dữ liệu **private** của lớp khác (không sử dụng hàm truy nhập **public**). Chúng ta không muốn để dữ liệu là **public** bởi vì khi đó một ai đó có thể vô tình thay đổi nó. Ngoài ra không có lớp nào là một "loại" của lớp khác, nên chúng ta không thể liên kết chúng dùng sự kế thừa. Làm thế nào để sắp xếp cho một lớp truy nhập được các thành viên **private** của lớp khác? Câu trả lời là sử dụng các lớp bạn.

Trong liên lạc giữa các lớp, có một sự khác nhau trong việc mô tả lớp nào trước trong chương trình. Chúng ta không thể tham chiếu tới một lớp mà chưa được xác định bởi vì trình biên dịch sẽ không biết gì về chúng. Nếu lớp **alpha** được mô tả trước lớp **beta** thì các hàm thành viên của lớp **beta** dễ dàng truy nhập các thành viên **private** của lớp **alpha**, nhưng sẽ khó hơn cho các hàm trong **alpha** để truy nhập tới dữ liệu **private** trong **beta**. Chúng ta sẽ bắt đầu với trường hợp dễ trước.

4.7.2. Truy nhập các thành viên **private** trong lớp được định nghĩa trước

Bản 4-21, **INTERC1**, cho thấy một hàm thành viên của lớp **beta**, lớp mô tả sau lớp **alpha**, có thể truy nhập thành viên **private** trong lớp **alpha** như thế nào.

Listing 4-21 **INTERC1**

```
//interc1.cpp
//truy nhập du lieu trong mot lop duoc dinh nghia truoc
#include<iostream.h>
class alpha
{
    private:
        friend class beta; //de beta co the truy nhap du lieu alpha
        int adata;
};
class beta
{
    public:
        void bfunc()
        {
            alpha objA; //tao mot doi tuong de truy nhap du lieu private
                        //cua alpha
            objA.adata=3;
        }
}
```

```

};

int frifunc(alpha a,beta b)
{
    return (a.data+b.data);
}

void main()
{
    alpha aa;
    beta bb;
    cout<<"Ham ban frifunc(aa,bb)="<<frifunc(aa,bb);
}

```

Điểm mấu chốt ở đây là khai báo lớp **beta** là một lớp bạn của **alpha**:

```
friend class beta;    //de beta co the truy nhap du lieu alpha
```

Khai báo này cho phép bất cứ hàm thành viên nào của lớp **beta** cũng truy nhập được tới các dữ liệu **private** hay **protected** trong lớp **alpha**. Cả hai từ khóa **friend** và **class** đều cần thiết. Chú ý rằng khai báo được đặt trong phần **private** của **alpha**. Như với các hàm bạn, trình biên dịch không quan tâm đến nơi đặt khai báo, nhưng để chính xác về khái niệm nên đặt nó trong phần **private** bởi vì sự kết bạn giữa hai lớp **alpha** và **beta** chỉ được sử dụng trong hai lớp đó; nó không phải phần giao diện chung (**public interface** - được truy nhập bởi người sử dụng).

Điều quan trọng cần nhận ra là (trừ khi chúng liên kết nhau bằng sự kế thừa) một thành viên lớp không thể truy nhập dữ liệu trong một lớp khác theo cách trừu tượng. Nghĩa là chúng ta không thể nói:

```
alpha::adata=3;    //loi : yeu cau doi tuong
```

Nếu **beta** được rút ra từ **alpha** thì điều này có thể được. Chúng là hai lớp riêng và tình bạn không giống tình gia tộc. Phải có một đối tượng thật (hay một con trỏ tới đối tượng thật, như chúng ta sẽ thấy) trong đó có dữ liệu.

```
alpha objA;
objA.adata=3;
```

4.7.3. Truy nhập các thành viên **private** trong một lớp chưa được định nghĩa

Bây giờ giả sử một hàm thành viên của **alpha** (**afunc()**) muốn truy nhập dữ liệu **private** trong **beta**. Vấn đề này cần được xem xét vì mô tả cho lớp **beta** đứng sau **alpha** và trình biên dịch cần biết một lớp xác định trước khi nó có thể truy nhập các thành viên của lớp.

Trong trường hợp này, phương pháp đặc biệt là di chuyển định nghĩa hàm thành viên **afunc()** ra khỏi mô tả lớp **alpha** và đặt nó sau mô tả lớp **beta**. Bản 4-22 là chương trình INTERC2.

Listing 4-22 INTERC2

```

//interc2.cpp
//truy nhap du lieu trong mot lop chua duoc dinh nghia
#include<iostream.h>
class alpha
{
    public:
        void afunc();    //khai bao ham, dinh nghia phai dat sau beta
};
class beta
{

```

```

private:
    friend class alpha; //de alpha co the truy nhap du lieu beta
    int bdata;          //du lieu cua beta
};
void alpha::afunc()    //ham cua alpha
{
    beta objB;         //tao mot doi tuong de truy nhap du lieu private
                        //cua beta
    objB.bdata=3;
}

```

Chúng ta khai báo **afunc()** trong **alpha** nhưng định nghĩa nó sau mô tả lớp **beta**. Bởi vậy, trình biên dịch biết **beta** như thế nào khi nó biên dịch **afunc()** và bởi vì **alpha** là bạn của **beta** nên nó có thể sử dụng lệnh để truy nhập dữ liệu **private** của **beta**. Ngoài ra, cần chú ý là dữ liệu này phải ở trong một đối tượng thực sự.

4.7.4. Các con trỏ liên lạc giữa các lớp

Trong các chương trình thực tế, khi một lớp truy nhập dữ liệu trong một lớp khác, tham chiếu tới các đối tượng sử dụng con trỏ trở tới chúng thông dụng hơn tham chiếu trực tiếp tới chúng, như trong INTERC1 và INTERC2. Ví dụ tiếp theo không chỉ cho thấy các đối tượng có thể được truy nhập bởi con trỏ như thế nào mà còn minh họa sự liên lạc hai chiều: **alpha** truy nhập dữ liệu **private** của **beta** và **beta** truy nhập dữ liệu **private** của **alpha**. Ý tưởng trong việc sử dụng con trỏ để liên lạc giữa các lớp là mỗi lớp chứa các con trỏ trở tới các đối tượng của lớp khác. Loại liên lạc mà chúng ta mô tả ở đây thường có một số đối tượng khác nhau của hai lớp. Trong ví dụ sau, chúng ta giả thiết mỗi đối tượng **alpha** liên kết với hai đối tượng **beta**. Theo đó, lớp **alpha** có hai con trỏ trở tới **beta** (để **alpha** có thể truy nhập dữ liệu **beta**) và lớp **beta** chứa một con trỏ trở tới **alpha** (để **beta** có thể truy nhập dữ liệu **alpha**). Ngoài ra, mỗi lớp được khai báo là bạn của lớp kia. Bản 4-23 là INTERC3.

Listing 4-23 INTERC3

```

//interc3.cpp
//lien lac giữa các lớp sử dụng các con trỏ và các lớp bạn
#include<iostream.h>

class alpha          //mot alpha liên kết với vài beta
{
private:
    friend class beta; //de beta truy nhập du lieu alpha
    beta* bptr1;       //các con trỏ trở tới beta
    beta* bptr2;
    int adata;
public:
    alpha();            //khai báo hàm tạo, định nghĩa sau
    void afunc();       //khai báo hàm, định nghĩa sau
};
class beta           //mot vài beta liên kết với mot alpha
{
private:
    friend class alpha; //de alpha truy nhập du lieu beta
    alpha* aptr;       //con trỏ trở tới alpha
    int bdata;
    //lưu ý: hàm tạo là private
    beta(alpha* ap):aptr(ap) //hàm tạo một đối số,

```



```

        }
        // khoi tao con tro tro toi alpha
public:
    void bfunc()
    {
        aptr->adata=3; //truy nhap du lieu private cua alpha
    }
};
alpha::alpha() //ham tao alpha, phai dinh nghia sau beta
{
    bptr1=new beta(this); //tao cac beta
    bptr2=new beta(this);
}
void alpha::afunc() //ham cua alpha, phai duoc dinh nghia sau beta
{
    bptr1->bdata=4; //truy nhap du lieu private cua beta
    bptr2->bdata=5;
    bptr2->bfunc(); //truy nhap du lieu alpha qua ham thanh vien cua beta
                    //hien thi cac du lieu trong cac doi tuong
    cout<<"\nadata="<<adata; //du lieu trong alpha
    cout<<"\nbdata1="<<bptr1->bdata; //du lieu trong doi tuong beta 1
    cout<<"\nbdata2="<<bptr2->bdata; //du lieu trong doi tuong beta 2
}
void main()
{
    alpha objA;
    objA.afunc();
}

```

Chúng ta lại sử dụng thủ thuật định nghĩa các hàm của **alpha** sau mô tả lớp **beta** bởi vì các hàm tạo **alpha** và hàm **afunc()** đều yêu cầu truy nhập dữ liệu của **beta**. Chúng ta có thể thấy hàm **afunc()** của **alpha** truy nhập dữ liệu **bdata** của cả hai đối tượng **beta** và hàm **bfunc()** của **beta** truy nhập dữ liệu **adata** trong đối tượng **alpha**. Tất cả các dữ liệu trong các đối tượng được hiển thị thông qua hàm **afunc()** của **alpha**.

Tất cả các đối tượng này được tạo như thế nào? Chúng ta cho rằng người sử dụng lớp chỉ tạo các đối tượng **alpha**. Nếu người sử dụng cố tình tạo các đối tượng **beta** thì có thể trái với sự sắp xếp hai **beta**-một **alpha**. Để đảm bảo rằng người sử dụng không thể tạo các đối tượng **beta**, chúng ta để hàm tạo của **beta** ở **private**, bởi vậy chỉ có các lớp bạn của **beta** mới có thể tạo các đối tượng **beta**.

Khi một đối tượng **alpha** được tạo, hàm tạo của nó tạo hai đối tượng **beta**. Nó làm việc bằng cách gọi hàm tạo **beta** và truyền cho hàm tạo **beta** con trỏ **this** của nó. Do đó, mọi đối tượng **beta** có thể xác định được vị trí của đối tượng **alpha** mà nó liên kết với. Bởi vì hàm tạo của **alpha** sử dụng **new** để tạo hai đối tượng **beta** cho nó nên nó cũng có thể truy nhập chúng bằng con trỏ.

4.7.5. Ví dụ về đua ngựa

Chúng ta cùng đem những gì đã biết về các lớp bạn và sự liên lạc giữa các lớp vào một chương trình trò chơi đua ngựa. Đây là một chương trình bắt chước mà trong đó một số ngựa xuất hiện trên màn hình, bắt đầu từ bên trái và phi tới đường đích ở bên phải. Tốc độ của mỗi con ngựa được xác định một cách ngẫu nhiên để không biết trước được con ngựa nào thắng. Chương trình sử dụng chế độ đồ họa văn bản để các con ngựa được hiển thị một cách dễ dàng (mặc dù hơi thô).

Chương trình này có vài nét đặc biệt: thứ nhất, nó chỉ làm việc được với trình biên dịch Borland bởi vì nó sử dụng một số hàm thư viện riêng của Borland. Những hàm này là **delay()**, **gotoxy()**, **clrscr()**, **random()** và **randomize()**. Thứ hai, đây là nhóm hàm chạy trên môi trường DOS bởi vì

hàm `delay()` không làm việc trong môi trường Windows. Bởi vậy chúng ta phải viết nó như chương trình của DOS.

1. Hoạt động của chương trình FRIHORSE

Khi chương trình bắt đầu, nó yêu cầu người sử dụng cung cấp khoảng cách cuộc đua và số ngựa tham gia. Đơn vị cổ điển của khoảng cách là **furlong** (theo cách nói của người Anh), nó bằng 1/8 dặm. Các cuộc đua điển hình là 6,8,10 hoặc 12 **furlong**. Số ngựa có thể từ 1 tới 10. Chương trình vẽ các đường bắt đầu, các đường kết thúc và các đường thẳng đứng tương đương với số **furlong**. Mỗi con ngựa được biểu diễn bằng một hình chữ nhật có số ở giữa.

2. Thiết kế cuộc đua ngựa

Chúng ta tiếp cận cách thiết kế hướng đối tượng cho cuộc đua ngựa như thế nào? Câu hỏi đầu tiên có thể là có nhóm các thực thể tương tự nhau nào để chúng ta mô hình hóa không? Có, đó là các con ngựa. Đó là lý do để mỗi con ngựa trở thành một đối tượng. Sẽ có một lớp gọi là **horse** mô tả dữ liệu của mỗi con ngựa, chẳng hạn số và chiều dài nó đã chạy được (nó được sử dụng để hiển thị con ngựa ở đúng vị trí trên màn hình).

Tuy nhiên, cũng có dữ liệu cần cho toàn bộ đường đua hơn là từng con ngựa, chẳng hạn chiều dài đường đua, thời gian trôi qua (bắt đầu từ 0 : 00 khi bắt đầu cuộc đua) và tổng số ngựa. Chúng ta phải làm gì với dữ liệu liên quan đến đường đua này? Một khả năng là tạo một lớp gọi là **track** và cài đặt những dữ liệu này vào trong đó. Sau đó chúng ta sẽ có một đối tượng **track** và nhiều đối tượng **horse**.

Track và **horse** liên lạc với nhau như thế nào? Chúng ta có thể sử dụng sự kế thừa để tạo lớp **horse** là con cháu của lớp **track** nhưng điều này hoàn toàn không có ý nghĩa bởi vì các con ngựa không phải là một "loại" đường đua.

Cách mà chúng ta dùng ở đây là tạo lớp **horse** là bạn của lớp **track** để các đối tượng **horse** có thể truy nhập dữ liệu **private** của đối tượng **track**. Ngoài ra, chúng ta không muốn người sử dụng lớp có thể tạo và truy nhập các đối tượng **horse**, nó không thể tồn tại thiếu một đối tượng **track** (mặc dù điều này không giống trong cuộc sống thực). Bởi vậy, chúng ta phải để hàm tạo và các hàm thành viên của lớp **horse** ở **private**. Tuy nhiên, **track** cần truy nhập các thành viên của lớp **horse** này nên ta để **track** là bạn của **horse**.

Người sử dụng tạo một đối tượng **track** và đối tượng **track** đó tạo các đối tượng **horse**. Đối tượng **track** và các đối tượng **horse** liên lạc với nhau dùng con trỏ và mối quan hệ bạn, như minh họa trong ví dụ INTERC3. Bản 4-24 trình bày chương trình FRIHORSE.

Listing 4-24 FRIHORSE

```
//frihorse.cpp
//mô hình hóa cuộc đua ngựa, sử dụng lớp bạn
#include<iostream.h>
#include<dos.h>           //cho delay()
#include<conio.h>          //cho gotoxy()
#include<stdlib.h>         //cho random()
#include<time.h>           //cho randomize()

const int CPF=5;           //số cột màn hình/furlong
////////////////////////
class track
{
private:
    friend class horse; //để horse có thể truy nhập dữ liệu
    horse* hptr;        //con trỏ trỏ tới horse
```

```

    int total;           //tong so ngua
    int count;           //so ngua da tao tu truoc den nay
    int track_length;    //chieu dai duong dua tinh bang furlong
    float elapse_time;   //thoi gian tu khi bat dau cuoc dua
public:
    track(float,int);    //ham tao hai doi so
    void track_tick();   //hien thi thoi gian toan bo cuoc dua
    ~track();            //ham huy
};

class horse
{
private:
    friend class track;    //de track truy nhap du lieu cua horse
    track* ptr_track;      //con tro tro toi track
    int horse_number;      //so cua con ngua nay
    float finish_time;     //thoi gian toi dich cua con ngua nay
    float distance_run;    //quang duong di duoc tu khi bat dau
                        //chu y: cac ham thanh vien la private
    horse():distance_run(0.0) //tao mot con ngua
    {
    void horse_init(track* pt)//khoi tao mot con ngua
    {
        ptr_track=pt;
        horse_number=(ptr_track->count)++;
    }
    void horse_tick();      //hien thi thoi gian cho mot con ngua
};

////////////////////////////////////
void horse::horse_tick()   //cho tung con ngua
{
    gotoxy(1+int(distance_run*CPF),2+horse_number*2);
    if(distance_run<ptr_track->track_length + 1.0/CPF)
    {
        if(random(3)%3)
            distance_run+=0.2;           //tang len 0.2
        finish_time=ptr_track->elapse_time; //cap nhap thoi gian toi dich
    }
else
{
    int mins=int(finish_time/60);
    int secs=int(finish_time) - mins*60;
    cout<<"Thoi gian:"<<mins<<":"<<secs;
}
}

//-----
//ham tao hai doi so
track::track(float l,int t):track_length(l),total(t),count(0),
                        elapse_time(0.0)
{
    randomize();           //khoi tao bo ngau nhien
    clrscr();
    for(int f=0;f<=track_length;f++) //cho tung furlong
        for(int r=1;r<=total*2+1;r++) //cho tung cot man hinh

```

```

        gotoxy(f*CPF+5,r);
        if(f==0 || f==track_length)
            cout<<'\xDE';        //ve duong bat dau hoac ket thuc
        else
            cout<<'\xB3';        //ve vat danh dau furlong
    }
    hptr=new horse[total];        //tao cac horse
    for(int j=0;j<total;j++)
        (hptr+j)->horse_init(this);
}
void track::track_tick()
{
    elapse_time+=1.75;            //cap nhat thoi gian
    for(int j=0;j<total;j++)      //cho tung con ngua
        (hptr+j)->horse_tick();  //cap nhat horse
}
track::~track()                  //ham huy
{ delete[] hptr; }               //xoa bo nho cho tat ca horse
void main()
{
    float length;
    int nhorse;
    cout<<"\nNhap vao chieu dai duong dua (don vi furlong,6-12):";
    cin>>length;
    cout<<"\nNhap vao so ngua(1-10):";
    cin>>nhorse;
    track t(length,nhorse);        //tao doi tuong track va cac doi tuong horse
    while(!kbhit())                //thoat khi bam mot phim
    {
        t.track_tick();            //di chuyen va hien thi tat ca ngua
        delay(500);                //doi 1/2 giay
    }
    t::~track();                  //huy cac doi tuong track va horse
}

```

CHƯƠNG 5

STREAM VÀ FILE

5.1. LỚP STREAM

Stream là một tên chung cho luồng dữ liệu vào/ra. Vì lý do này mà các **stream** trong C++ thường được gọi là các **iostream** (các dòng vào ra). Một **iostream** có thể được biểu diễn bởi một đối tượng cụ thể. Ví dụ, chúng ta đã biết nhiều ví dụ về các đối tượng **cin** và **cout** được dùng cho vào ra dữ liệu.

5.1.1. Thuận tiện của các stream

Những người lập trình C có thể sẽ băn khoăn không biết sử dụng các lớp **stream** cho vào/ra thay thế các hàm C truyền thống như **printf()**, **scanf()** và cho file như **fprintf()**, **fscanf()** v.v... có những thuận lợi gì?

Một trong những lý do là các **stream** ít sinh ra lỗi. Chúng ta có thể thấu hiểu được điều này nếu so sánh với các hàm vào/ra C truyền thống mà luôn cần các ký tự định dạng như **%f**, **%d**, **%c**... Không có các ký tự như vậy trong các **stream**, bởi vì mỗi đối tượng đã biết cách hiển thị chính nó. Điều này đã xóa bỏ được một nguồn chính tạo ra lỗi chương trình.

Lý do thứ hai là chúng ta có thể chồng (overload) các toán tử đã có vào các hàm, chẳng hạn toán tử chèn vào **<<** (**insertion operator**) và toán tử lấy ra **>>** (**extraction operator**), để làm việc với các lớp mà chúng ta tạo ra. Điều này làm cho các lớp của chúng ta làm việc như các kiểu có sẵn, ngoài ra nó còn làm cho việc lập trình dễ dàng hơn và ít sinh lỗi hơn (chưa nói là thỏa mãn về thẩm mỹ).

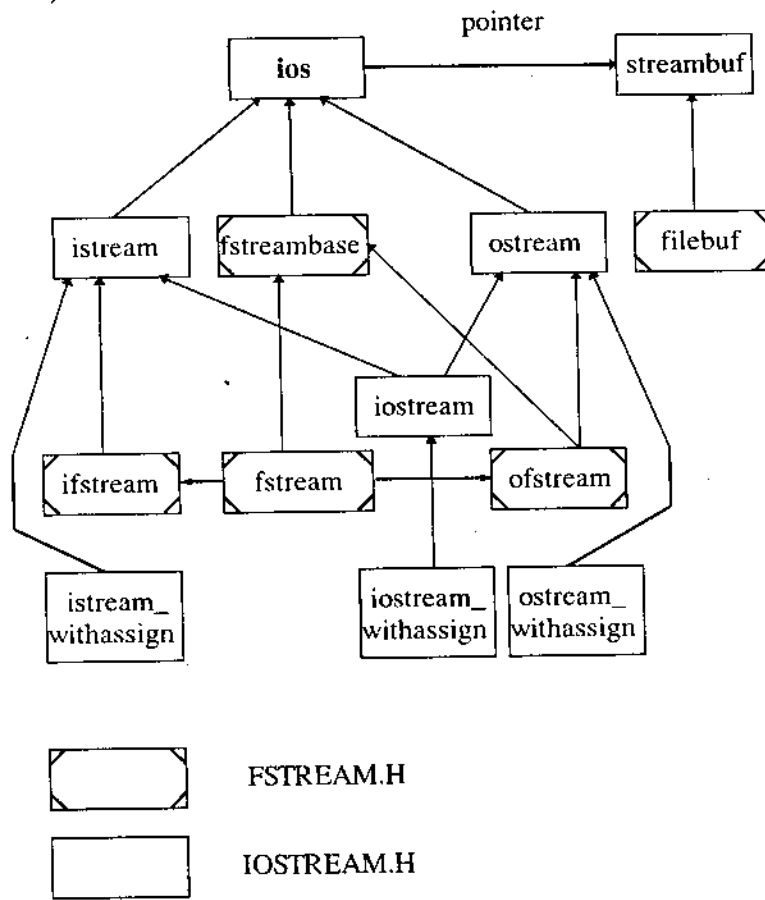
Chúng ta có thể băn khoăn là liệu các dòng (stream) I/O của C++ có còn quan trọng không khi lập trình trong một môi trường **Graphics User Interface** (GUI) chẳng hạn như Windows, ở đó việc đưa trực tiếp văn bản ra màn hình không thực hiện được. Câu trả lời là vẫn còn quan trọng bởi vì chúng là cách tốt nhất để đưa dữ liệu ra file và định dạng dữ liệu trong bộ nhớ để sau đó sử dụng các hộp hội thoại và các phần tử GUI.

5.1.2. Phân cấp lớp stream

Các lớp **stream** được sắp xếp trong một phân cấp lớp khá phức tạp. Chúng ta không cần biết tất cả về phân cấp lớp này để thực hiện vào/ra, nhưng có thể thấy một tổng quan hữu ích. Hình 5-1 trình bày sự sắp xếp quan trọng nhất của các lớp này.

Chúng ta đã sử dụng rất nhiều một số lớp **stream**. Toán tử lấy ra **>>** (**extraction operator**) là một thành viên của lớp **istream** và toán tử chèn vào **<<** (**insertion operator**) là một thành viên của lớp **ostream**. Cả hai lớp này đều rút ra từ lớp **ios**. Đối tượng **cout** biểu diễn dòng ra chuẩn, nó thường hướng tới màn hình video, là một đối tượng được định nghĩa của lớp **ostream_withassign**, lớp này rút ra từ **ostream**. Tương tự, **cin** là một đối tượng của lớp **istream_withassign**, nó được rút ra từ lớp **istream**.

Các lớp được dùng cho vào/ra màn hình video và bàn phím được khai báo trong file tiêu đề **IOSTREAM.H** mà chúng ta đã biết trong các ví dụ từ trước đến nay. Các lớp được dùng riêng cho vào/ra file được khai báo trong file tiêu đề **FSTREAM.H**. Hình 5-1 cho thấy những lớp ở trong hai file tiêu đề này. Ngoài ra, một số tác tử được khai báo trong **IOMANIP.H** và các lớp bộ nhớ được khai báo trong **STRSTREA.H**.



Hình 5-1. Phân cấp lớp stream.

Chúng ta có thể thấy trong hình 5-1, lớp **ios** là lớp cơ sở cho phân cấp lớp **stream**. Nó chứa nhiều hằng và nhiều hàm thành viên chung cho các hoạt động vào/ra của tất cả các loại. Một vài hàm trong số này như **showpoint** và **fixed**, định dạng cờ (flag) mà chúng ta đã gặp. Lớp **ios** cũng chứa một con trỏ tới lớp **streambuf**, lớp này chứa bộ đệm của bộ nhớ thực mà dữ liệu được đọc, ghi vào đó và các chương trình mức thấp quản lý dữ liệu này. Bình thường, chúng ta không cần lo lắng về lớp **streambuf**, nó được tham chiếu một cách tự động bởi các lớp khác, nhưng đôi khi truy cập trực tiếp bộ đệm này cũng có ích.

Các lớp **istream**, **ostream** được rút ra từ **ios** và được dành cho vào/ra dữ liệu. Lớp **istream** chứa các hàm thành viên như **get()**, **getline()**, **read()** và toán tử lấy ra (**>>**), trái lại **ostream** chứa các hàm **put()**, **write()** và toán tử chèn vào (**<<**).

Lớp **iostream** được rút ra từ cả hai lớp **istream** và **ostream** bằng sự kế thừa bội. Các lớp rút ra từ **iostream** có thể được dùng với các thiết bị, chẳng hạn file đĩa, mà có thể vào/ra cùng một lúc. Ba lớp, **istream_withassign**, **ostream_withassign**, **iostream_withassign** được kế thừa từ **istream**, **ostream** và **iostream** tương ứng. Các lớp này có thêm toán tử gán để **cout** và **cin** có thể được gán cho các **stream** khác.

5.1.3. Lớp ios

Lớp **ios** là lớp ông của tất cả các lớp **stream** và chứa các đặc điểm chính cần cho hoạt động của các **stream C++**. Ba đặc điểm quan trọng nhất là cờ định dạng, bit trạng thái lỗi và chế độ hoạt động file. Bây giờ chúng ta cùng xét các cờ định dạng và các bit trạng thái lỗi. Còn chế độ hoạt động file để đến khi nói về các file đĩa.

1. Cờ định dạng

Các cờ định dạng là một tập các định nghĩa **enum** trong lớp **ios**. Chúng hoạt động như các công tắc bật/tắt để lựa chọn các khía cạnh khác nhau của hoạt động và định dạng vào/ra. Bảng 5-1 trình bày một danh sách đầy đủ về các cờ định dạng.

Bảng 5-1. Các cờ định dạng ios

Các cờ (Flags)	Ý nghĩa
skipws	Bỏ qua khoảng trống ở đầu vào
left	Đưa ra căn lề trái
right	Đưa ra căn lề phải
internal	Sử dụng cả dấu (hoặc chỉ thị cơ số) và số [+12.34]
dec	Chuyển sang dạng thập phân
oct	Chuyển sang dạng bát phân
hex	Chuyển sang dạng hệ cơ số 16
showbase	Sử dụng chỉ thị cơ số ở đầu ra
showpoint	Hiển thị dấu chấm thập phân ở đầu ra
uppercase	Sử dụng chữ hoa và các ký tự hệ hex là hoa
showpos	Hiển thị dấu cộng trước các số nguyên dương
scientific	Sử dụng dạng mũ khi đưa ra số phẩy động [9.1234e2]
fixed	Sử dụng dạng cố định khi đưa ra số dấu phẩy động
unitbuf	Làm sạch tất cả stream sau khi chèn vào >>
stdio	Làm sạch stdout, stderr sau khi chèn vào >>

Có vài cách để thiết lập các cờ định dạng, các cờ khác nhau có thể được thiết lập theo những cách khác nhau. Bởi vì chúng là thành viên của lớp **ios**, các cờ thường phải đứng sau tên **ios** và toán tử quy định phạm vi (**scope resolution operator**), ví dụ **ios::skipws**. Tất cả các cờ có thể được thiết lập bằng cách dùng hàm thành viên của lớp **ios** là **setf()** và **unsetf()**. Ví dụ:

```
cout.setf(ios::left);    //van ban dua ra can le trai
cout<<"Day la van ban duoc can le trai";
cout.unsetf(ios::left);  //tra ve mac dinh (can le phai)
```

Nhiều cờ định dạng có thể thiết lập bằng cách sử dụng các tác tử (manipulator).

2. Các tác tử (manipulators)

Các tác tử là các lệnh định dạng được chèn trực tiếp vào một **stream**. Ví dụ như tác tử **endl** mà chúng ta đã sử dụng nhiều, nó gửi một dòng mới vào **stream** và làm sạch nó:

```
cout<<"To each his own"<<endl;
```

Chúng ta cũng đã sử dụng tác tử **setioflags()**:

```
cout<<setioflags(ios::fixed) //su dung dau cham thap phan co dinh
<<setioflags(ios::showpoint)//luon luon su dung dau cham thap phan
```

<<var;

Như ba ví dụ đã minh họa, ta thấy các tác tử có ba dạng: các tác tử có một đối số và các tác tử không có đối số. Bảng 5-2 tóm tắt các tác tử không có đối số.

Bảng 5-2. Các tác tử ios không đối số

Tác tử	Chức năng
ws	Bật chế độ bỏ qua ký tự trắng ở đầu vào
dec	Chuyển sang dạng thập phân
oct	Chuyển sang dạng bát phân
hex	Chuyển sang dạng cơ số 16
endl	Chèn dòng mới và làm sạch dòng đưa ra
ends	Chèn ký tự null để kết thúc một chuỗi đưa ra
flush	Làm sạch dòng đưa ra (output stream)
lock	Khóa việc quản lý file
unlock	Mở khóa việc quản lý file

Chúng ta có thể chèn trực tiếp các tác tử này vào **stream**. Ví dụ, để đưa ra giá trị biến **var** ở dạng cơ số 16, chúng ta có thể nói:

```
cout<<hex<<var;
```

Các trạng thái thiết lập bởi các tác tử không đối số vẫn còn tác dụng cho đến khi **stream** bị phá hủy. Bởi vậy, chúng ta có thể đưa ra nhiều số dạng cơ số 16 chỉ cần chèn một tác tử **hex**.

Bảng 5-3 tóm tắt các tác tử có đối số. Với các tác tử này chúng ta cần có file tiêu đề **IOMANIP.H**.

Bảng 5-3. Các tác tử ios có đối số

Tác tử	Đối số	Chức năng
setw()	Độ rộng trường (int)	Thiết lập độ rộng trường đưa ra
setfill()	Ký tự điền đầy (int)	Thiết lập ký tự điền đầy cho kết quả đưa ra (mặc định là space)
setprecision()	Độ chính xác (int)	Thiết lập độ chính xác (số chữ số hiển thị sau dấu chấm)
setioflags()	Các cờ định dạng (long)	Thiết lập các cờ xác định
resetioflags()	Các cờ định dạng (long)	Xóa các cờ định dạng

Các tác tử có đối số chỉ ảnh hưởng tới mục tiếp theo trong **stream**. Ví dụ, nếu dùng **setw()** để thiết lập độ rộng trường trong đó một số được hiển thị thì chúng ta phải dùng nó cho các số tiếp theo.

3. Các hàm

Lớp **ios** chứa một số hàm có thể dùng để thiết lập các cờ định dạng và thực hiện các nhiệm vụ khác. Bảng 5-4 cho thấy các hàm này, trừ các hàm làm việc với các lỗi được nói đến ở phần sau.

Bảng 5-4. Các tác tử ios không đối số

Hàm	Chức năng
<code>ch=fill()</code>	Trả về ký tự điền đầy (điền đầy các phần không sử dụng của trường, mặc định là dấu cách)
<code>fill(ch)</code>	Thiết lập ký tự điền đầy
<code>p=precision()</code>	Trả về độ chính xác (số chữ số hiển thị sau dấu chấm)
<code>precision(p)</code>	Thiết lập độ chính xác
<code>w=width()</code>	Trả về độ rộng trường hiện tại
<code>width(w)</code>	Thiết lập độ rộng trường
<code>setf(flags)</code>	Thiết lập cờ định dạng xác định (vd: <code>ios::left</code>)
<code>unsetf(flags)</code>	Hủy các cờ định dạng xác định
<code>setf(flags,field)</code>	Đầu tiên xóa field, sau đó thiết lập flags

Các hàm này được gọi cho các đối tượng cụ thể sử dụng toán tử chấm thông thường. Ví dụ, để thiết lập độ rộng trường bằng 14 ta có thể nói:

```
cout.width(14);
```

Tương tự, lệnh sau thiết lập ký tự điền đầy là một dấu sao (*):

```
cout.fill('*');
```

Chúng ta cũng có thể dùng một vài hàm để tác động trực tiếp tới các cờ định dạng. Ví dụ, để thiết lập căn lề trái dùng :

```
cout.setf(ios::left);
```

Một phiên bản hai đối số của hàm `setf()` dùng đối số thứ hai để xóa tất cả các cờ của một loại hoặc một trường cụ thể, sau đó thiết lập cờ được xác định trong đối số thứ nhất. Cách này làm cho dễ dàng hơn trong việc xóa các cờ có liên quan trước khi thiết lập một cờ mới. Bảng 5-5 chỉ ra sự sắp xếp này.

Bảng 5-5. Phiên bản hai đối số của hàm `setf()`

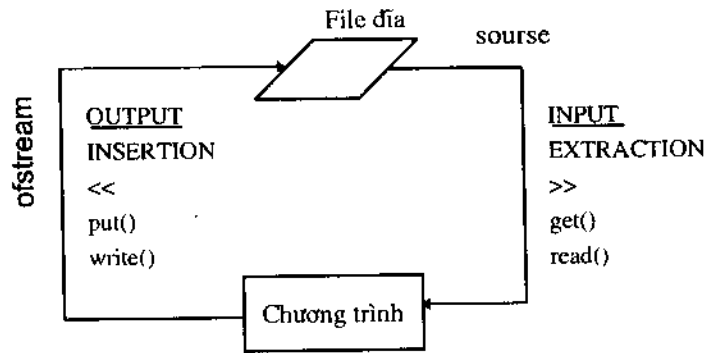
Đối số thứ nhất: cờ để thiết lập	Đối số thứ hai: trường cần xóa
<code>dec, oct, hex</code>	<code>basefield</code>
<code>left, right, internal</code>	<code>adjustfield</code>
<code>scientific, fixed</code>	<code>floatfield</code>

Ví dụ:

```
cout.setf(ios::left,ios::adjustfield);
```

Sẽ xóa tất cả các cờ liên quan đến căn lề văn bản và sau đó thiết lập cờ `left` để kết quả đưa ra căn lề trái.

Bằng việc sử dụng các kỹ thuật chỉ ra ở đây với các cờ định dạng, chúng ta cũng có thể tìm ra một cách để định dạng vào/ra không chỉ cho bàn phím mà còn cho các file.



Hình 5-2. Vào và ra.

4. Lớp istream

Lớp `istream`, được rút ra từ lớp `ios`, thực hiện các hoạt động dành riêng cho đầu vào, hay lấy ra (extraction). Nó dễ nhầm với hoạt động đưa ra, chèn vào (insertion).

Bảng 5-6 liệt kê các hàm thường được sử dụng của lớp `istream`.

Bảng 5-6. Các hàm `istream`

Hàm	Chức năng
>>	Định dạng lấy ra cho tất cả các kiểu cơ bản và kiểu chồng.
<code>get(ch)</code>	Lấy ra một ký tự đặt vào <code>ch</code> .
<code>get(str)</code>	Lấy ra các ký tự đặt vào <code>str</code> cho đến khi gặp '\0'.
<code>get(str,MAX)</code>	Lấy ra tới MAX ký tự đặt vào <code>str</code> .
<code>get(str,DELIM)</code>	Lấy ra các ký tự đặt vào <code>str</code> cho đến khi gặp một ký tự giới hạn (diễn hình là '\n'). Để lại ký tự giới hạn trong stream.
<code>get(str,MAX,DELIM)</code>	Lấy ra các ký tự đặt vào mảng <code>str</code> cho đến MAX ký tự hoặc gặp ký tự giới hạn <code>DELIM</code> . Để lại ký tự giới hạn trong stream.
<code>getline(str,MAX,DELIM)</code>	Lấy ra các ký tự đưa vào mảng <code>str</code> cho đến MAX ký tự hoặc gặp ký tự giới hạn <code>DELIM</code> . Lấy ra cả ký tự giới hạn.
<code>putback(ch)</code>	Chèn ký tự đọc lần cuối cùng quay trở lại stream vào (input).
<code>ignore(MAX,DELIM)</code>	Lấy ra và hủy bỏ tới MAX ký tự cho đến khi (và gồm cả) gặp ký tự giới hạn (diễn hình là '\n').
<code>peek(ch)</code>	Đọc một ký tự, để nó trong stream.
<code>count=gcount()</code>	Trả lại số ký tự được đọc bởi một lời gọi hàm (đứng ngay trước) <code>get()</code> , <code>getline()</code> hoặc <code>read()</code> .
<code>read(str,MAX)</code>	Cho các file. Lấy ra tới MAX ký tự đưa vào <code>str</code> cho đến khi gặp cuối file EOF.
<code>seekg(position)</code>	Thiết lập khoảng cách (tính theo byte) cho con trỏ file tính từ đầu file.
<code>seekg(position,seek_dir)</code>	Thiết lập khoảng cách (tính theo byte) cho con trỏ file tính từ một nơi xác định trong file: <code>seek_dir</code> có thể là <code>ios::beg</code> , <code>ios::cur</code> , <code>ios::end</code> .
<code>position=tellg()</code>	Trả về vị trí tính theo byte của con trỏ file tính từ đầu file.

Chúng ta đã biết một vài hàm trong số các hàm này, chẳng hạn `get()`, hầu hết chúng đều hoạt động trên đối tượng `cin`, biểu diễn bàn phím cũng như các file đĩa. Tuy nhiên, bốn hàm cuối chỉ làm việc riêng với các file đĩa. Chúng ta sẽ thấy chúng làm việc như thế nào trong phần sau.

5. Lớp `ostream`

Lớp `ostream` quản lý các hoạt động đưa ra hoặc chèn vào - **insertion**. Bảng 5-7 là các hàm thành viên thường được sử dụng của lớp này.

Bảng 5-7. Các hàm `ostream`

Hàm	Chức năng
<<	Định dạng chèn vào cho tất cả các kiểu cơ bản và kiểu chồng.
<code>put(ch)</code>	Chèn một ký tự <code>ch</code> vào stream.
<code>flush()</code>	Làm sạch nội dung vùng đệm và chèn một dòng mới.
<code>write(str,SIZE)</code>	Chèn <code>SIZE</code> ký tự từ mảng <code>str</code> vào file.
<code>seekp(position)</code>	Thiết lập khoảng cách (tính theo byte) cho con trỏ file tính từ đầu file.
<code>seekp(position,seek_dir)</code>	Thiết lập khoảng cách (tính theo byte) cho con trỏ file tính từ vị trí cd trong file. <code>seek_dir</code> có thể là <code>ios::beg</code> , <code>ios::cur</code> , <code>ios::end</code> .
<code>position=tellp()</code>	Trả về vị trí con trỏ file, tính theo byte.

Bốn hàm cuối chỉ làm riêng với các file.

6. Lớp `iostream` và `_withassign`

Lớp `iostream`, được rút ra từ `istream` và `ostream`, chỉ đóng vai trò là một lớp cơ sở cho các lớp khác, nhất là `iostream_withassign`. Nó không có các hàm của riêng nó (trừ hàm tạo và hàm hủy). Các lớp rút ra từ `iostream` có thể thực hiện cả hoạt động vào và ra.

Có ba lớp `_withassign`:

`istream_withassign`, rút ra từ `istream`
`ostream_withassign`, rút ra từ `ostream`
`iostream_withassign`, rút ra từ `iostream`

Các lớp `_withassign` rất giống các lớp mà chúng rút ra, trừ là chúng có các toán tử gán (=) được chồng để các đối tượng của chúng có thể được copy.

Tại sao cần chia ra các lớp `stream` có thể copy và các lớp `stream` không thể copy? Nói chung, copy các đối tượng `stream` không phải là một ý tưởng tốt. Mỗi đối tượng như vậy liên kết với một đối tượng `streambuf` cụ thể, nó có một vùng trong bộ nhớ để nhớ dữ liệu thật sự của đối tượng. Nếu copy đối tượng `stream`, có thể gây ra nhầm lẫn nếu chúng ta cũng copy các đối tượng `streambuf`. Tuy nhiên trong một vài trường hợp có thể là quan trọng để copy các đối tượng `stream`, như trong trường hợp định hướng mới cho các đối tượng định nghĩa trước như `cin` và `cout`.

Vì vậy, các lớp **istream**, **ostream** và **iostream** được làm không thể copy (bằng cách để các hàm tạo copy và các toán tử gán bị chồng ở phần **private**), trái lại, các lớp **_withassign** rút ra từ chúng có thể copy được.

7. Các đối tượng stream định nghĩa trước

Chúng ta đã sử dụng rất nhiều hai đối tượng được định nghĩa trước rút ra từ các lớp **_withassign** là **cin** và **cout**. Hai đối tượng này thường được nối tới bàn phím và màn hình. Ngoài ra còn có hai đối tượng được định nghĩa trước khác là **cerr** và **clog**. Bảng 5-8 liệt kê cả bốn đối tượng này.

Bảng 5-8. Các đối tượng stream định nghĩa trước

Tên	Lớp	Dùng cho
cin	Istream_withassign	Nhập vào từ bàn phím
cout	Ostream_withassign	Đưa ra màn hình thông thường
cerr	Ostream_withassign	Đưa ra lỗi
clog	Ostream_withassign	Đưa ra log

Đối tượng **cerr** thường được dùng cho thông báo lỗi và chẩn đoán chương trình. Kết quả gửi tới **cerr** được hiển thị ngay lập tức, không qua vùng đệm như với **cout**. Ngoài ra đưa tới **cerr** không thể định hướng mới được. Vì những lý do này, chúng ta có một cơ hội tốt hơn để nhìn thấy một thông báo đưa ra cuối cùng nếu chương trình của ta chết yểu. Một đối tượng khác, **clog**, tương tự **cerr** trong đó không được định nghĩa mới, nhưng kết quả đưa ra của nó qua bộ đệm còn **cerr** thì không.

5.2. CÁC LỖI STREAM

Từ trước đến giờ chúng ta đã sử dụng các đối tượng **cin** và **cout** để nhập vào và đưa ra mà không cần biết gì thêm, chẳng hạn:

```
cout<<"Good morning";  
và  
cin>>var;
```

Tuy nhiên, cách sử dụng này cho rằng không có lỗi gì xảy ra trong quá trình vào/ra. Điều này không phải lúc nào cũng đúng. Điều gì sẽ xảy ra nếu người sử dụng nhập vào chuỗi "chin" thay vì một số nguyên 9, hoặc ấn ENTER mà không nhập gì cả? Điều gì sẽ xảy ra nếu có một phần cứng bị lỗi? Chúng ta sẽ khảo sát những vấn đề như vậy trong phần này. Nhiều kỹ thuật cũng thích hợp với vào/ra file.

5.2.1. Các bit trạng thái lỗi (error-status bits)

Các bit trạng thái - lỗi là một số **enum ios** báo cáo các lỗi xảy ra trong một hoạt động vào hoặc ra. Chúng được tóm tắt trong bảng 5-9.

Bảng 5-9. Các bit trạng thái - lỗi

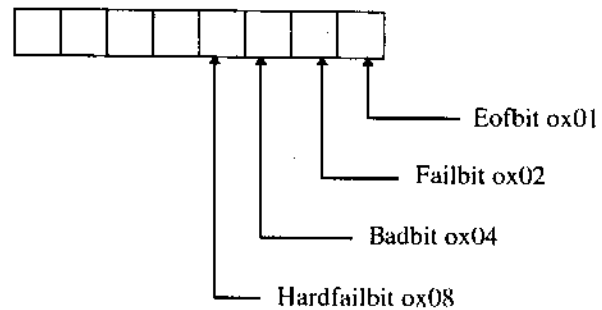
Tên	Ý nghĩa
goodbit	Không có lỗi (không có bit nào được thiết lập).
eofbit	Đã tới cuối file.
failbit	Hoạt động bị lỗi (lỗi người sử dụng, EOF sớm).
badbit	Hoạt động không hợp lệ.
hardfail	Lỗi không thể khôi phục.

Hình 5-3 cho thấy vị trí của các bit này trong byte trạng thái lỗi.

Nhiều hàm `ios` có thể được dùng để đọc (và thậm chí thiết lập) các bit lỗi này, như tóm tắt trong bảng 5-10.

5.2.2. Nhập vào các số

Chúng ta cùng xem cách kiểm soát lỗi khi nhập vào các số. Cách này áp dụng cho các đối tượng đọc vào từ bàn phím và đĩa. Ý tưởng là kiểm tra giá trị của **goodbit**, báo hiệu một lỗi nếu nó không phải là **true** và cho người sử dụng một cơ hội khác để nhập vào cho đúng.



Hình 5-3. Byte trạng thái - lỗi.

Bảng 5-10. Các hàm cho các bit lỗi

Hàm	Chức năng
<code>int=eof()</code>	Trả lại true nếu EOF được thiết lập.
<code>int=fail()</code>	Trả lại true nếu failbit, hoặc badbit, hoặc hardfail được thiết lập.
<code>int=bad()</code>	Trả lại true nếu badbit hoặc hardfail được thiết lập.
<code>int=good()</code>	Trả lại true nếu mọi thứ đều tốt, không có bit nào thiết lập.
<code>clear(int=0)</code>	Không có đối số xóa tất cả các bit lỗi; nếu có đối số sẽ thiết lập các bit xác định, như trong <code>clear (ios::failbit)</code> .

```
#include<iostream.h>
void main()
{
    int var;
    while(1)
    {
        cout<<"\nNhap vao not so nguyen:";
        cin>>var;
```

```

        if(cin.good())           //neu khong co loi
        {
            cin.ignore(10,'\n'); //xoa bo mot dong moi
            break;               //thoat khoi vong lap
        }
        cin.clear();             //xoa cac bit loi
        cout<<"\nNhap sai!Yeu cau nhap lai!";
        cin.ignore(10,'\n');
    }
    cout<<"\nSo nguyen var la:"<<var;
}

```

Lỗi thông dụng nhất mà hệ thống này tìm kiếm được khi đọc vào từ bàn phím là người sử dụng gõ vào không phải chữ số (chẳng hạn như "chin" thay vì 9). Điều này làm cho **failbit** được thiết lập. Tuy nhiên, nó cũng tìm kiếm những hư hỏng liên quan đến hệ thống mà thông dụng là file đĩa.

Các số dấu phẩy động (**float**, **double**, **long double**) có thể được phân tích để tìm lỗi như đối với các số nguyên.

5.2.3. Nhập quá nhiều ký tự

Các ký tự phụ có thể là một vấn đề khi đọc vào từ các **stream** vào. Điều này đặc biệt đúng khi có lỗi. Điển hình là các ký tự phụ để lại trong **stream** sau khi việc nhập vào được coi như hoàn tất. Lúc đó chúng được truyền tới hoạt động vào tiếp theo mặc dù chúng không được dùng cho hoạt động tiếp theo này. Thường thì một '\n' vẫn còn ở lại, nhưng đôi khi các ký tự khác cũng được để lại. Để giữ sạch những ký tự phụ này, hàm thành viên **ignore(MAX,DELIM)** của **istream** được sử dụng. Nó đọc và ném đi lên tới MAX ký tự, bao gồm cả ký tự giới hạn được chỉ rõ ở đối số thứ hai. Trong ví dụ trên, dòng lệnh:

```
cin.ignore(10,'\n');
```

làm cho **cin** đọc tới 10 ký tự, bao gồm cả ký tự '\n' và hủy bỏ chúng khỏi **stream** vào.

5.2.4. Không nhập gì

Các ký tự trắng, chẳng hạn như TAB, SPACE và '\n', thường được bỏ qua khi nhập vào các số. Điều này có thể gây ra vài tác động phụ không mong muốn. Ví dụ, người sử dụng, được thông báo nhập vào một số, có thể chỉ ấn Enter mà không gõ bất kỳ chữ số nào (có thể họ nghĩ rằng như thế sẽ nhập vào 0, hoặc có thể họ bối rối không biết làm gì). Trong chương trình trên, câu lệnh đơn giản:

```
cin>>var;
```

nếu ấn Enter cũng làm cho con trỏ rơi xuống dòng tiếp theo trong khi **stream** vẫn đợi nhập vào một số. Có vấn đề gì với việc con trỏ rơi xuống dòng tiếp theo? Thứ nhất, những người lập trình không có kinh nghiệm, thấy chương trình không chấp nhận khi ấn Enter có thể cho rằng máy tính bị hỏng. Thứ hai, ấn Enter lặp đi lặp lại làm cho con trỏ càng ngày càng rơi xuống thấp cho đến khi toàn bộ màn hình bắt đầu cuộn lên. Việc này chẳng có vấn đề gì trong tương tác kiểu teletype, ở đó chương trình và người sử dụng chỉ đơn giản gõ phím với nhau. Tuy nhiên, trong các chương trình đồ họa văn bản (text-based graphics), việc cuộn màn hình làm đảo lộn màn hình và cuối cùng phá hủy hoàn toàn màn hình.

Bởi vậy, có thể bảo **stream** vào không bỏ qua các ký tự trắng (**whitespace**) là rất quan trọng. Điều này được thực hiện bằng cách xóa cờ **skipws**:

```

    cout<<"\nNhap vao mot so nguyen:";
    cin.unsetf(ios::skipws);

```

```

cin>>var;
if(cin.good())
{
    //không có lỗi
}
    //có lỗi

```

Bây giờ nếu người sử dụng chỉ ấn Enter mà không gõ bất kỳ chữ số nào, **failbit** sẽ được thiết lập và một lỗi sẽ được tạo ra. Lúc đó chương trình có thể báo người sử dụng phải làm gì hoặc đi chuyển con trỏ để màn hình không bị cuộn.

5.2.5. Nhập vào chuỗi (strings) và các ký tự (characters)

Người sử dụng không thể nhầm lẫn khi nhập vào các chuỗi và các ký tự bởi vì tất cả các việc nhập vào, thậm chí là các số, đều được giải thích như một chuỗi. Tuy nhiên, nếu đến từ một file đĩa thì các chuỗi và các ký tự vẫn nên được kiểm tra để tìm các lỗi trong trường hợp gặp EOF hoặc xảy ra cái gì đó không đúng. Không giống như các số, chúng ta thường muốn bỏ đi các ký tự trắng khi nhập vào các chuỗi và các ký tự.

1. Nhập khoảng cách không có lỗi

Chúng ta cùng xem một chương trình trong đó việc nhập vào của người sử dụng cho lớp **Distance** được kiểm tra để tìm lỗi. Chương trình này nhận các giá trị ở dạng **feet** và **inches** từ người sử dụng và hiển thị chúng. Nếu người sử dụng phạm phải một lỗi, chương trình sẽ không chấp nhận việc nhập vào đó với một lời giải thích thích đáng cho người sử dụng và hiển thị thông báo để nhập vào mới.

Chương trình rất đơn giản chỉ trừ hàm thành viên **getdist()** đã được mở rộng để kiểm soát lỗi. Cũng có vài lệnh để đảm bảo người sử dụng không nhập vào số dấu phẩy động cho **feet**. Điều này là quan trọng bởi vì, khác với giá trị **feet** là một số nguyên, các giá trị **inches** là một số dấu phẩy động và người sử dụng có thể rất dễ nhầm.

Thông thường nếu mong đợi một số nguyên, toán tử lấy ra (**extraction operator**) chỉ đơn giản kết thúc không thông báo lỗi khi nó thấy một dấu chấm thập phân. Chương trình muốn biết một lỗi như vậy nên nó đọc giá trị **feet** như một chuỗi thay vì một số nguyên. Sau đó nó kiểm tra chuỗi bằng một hàm tự tạo **isint()**, nó trả về **true** nếu chuỗi đó là một số nguyên. Để kiểm tra được số **int**, chuỗi đó chỉ được nhận các chữ số và chúng phải được đánh giá là một số nằm trong khoảng -32768 tới 32767 (khoảng của kiểu **int**). Nếu chuỗi nhập vào đã qua kiểm tra số **int**, chương trình chuyển nó thành một số nguyên thực sự bằng hàm thư viện **atoi()**.

Các giá trị **inches** là một số dấu phẩy động. Chương trình kiểm tra khoảng của nó ($0 \leq \text{inches} \leq 12.0$). Chương trình cũng kiểm tra nó để tìm các bit lỗi **ios**. Thông thường nhất, **failbit** sẽ được thiết lập bởi vì người sử dụng thường gõ vào không phải chữ số thay vì một số dấu phẩy động. Bản 5-1 trình bày chương trình ENGLERR.

Listing 5-1 ENGLERR

```

//englerr.cpp
//kiểm tra nhập vào voi lop khoang cach Anh
#include<iostream.h>
#include<string.h>    //cho strlen()
#include<stdlib.h>    //cho atoi()

int isint(char*);      //khai bao ham
const int IGN=10;      //so ky tu bo qua

```

```

class Distance                                //lop khoang cach Anh
{
private:
    int feet;
    float inches;
public:
    Distance()                                //ham tao khong doi so
        { feet=0;inches=0.0;}
    Distance(int ft,float in)                 //ham tao hai doi so
        {feet=ft;inches=in;}
    void showDist()                           //hien thi khoang cach
        {
            cout<<feet<<"\t"<<inches<<"\n";
        }
    void getDist();                            //khai bao
};
void Distance::getDist()                     //lay chieu dai cua nguoi su dung
{
    char intstr[80];                          //de nhap vao chuoi
    while(1)
    {
        cout<<"\n\nNhap vao feet:";
        cin.unsetf(ios::skipws);              //khong bo qua cac ky tu trang
        cin>>intstr;                          //nhap vao feet nhu mot chuoi
        if(isint(intstr))                     //neu no la mot so nguyen
        {
            cin.ignore(IGN,'\n');              //huy cac ky tu gom ca dong moi
            feet=atoi(intstr);                //chuyen sang cac so nguyen
            break;
        }
        cin.ignore(IGN,'\n');                 //neu khong phai la mot so nguyen
        cout<<"Feet phai la mot so nguyen\n";
    } //ket thuc vong lap feet
    while(1)
    {
        cout<<"\n\nNhap vao inches:";
        cin.unsetf(ios::skipws);              //khong bo qua cac ky tu trang
        cin>>inches;                          //nhap vao inches, kieu float
        if(inches>=12.0 || inches<0.0)
        {
            cout<<"Inches phai nam trong khoang 0.0 toi 11.99";
            cin.clear(ios::failbit);           //thiet lap failbit
        }
        if(cin.good())                        //kiem tra de tim loi cin,
        {                                     //thuong la khong phai chu so
            cin.ignore(IGN,'\n');              //huy dong moi
            break;                             //nhap vao tot, thoat khoi while
        }
        cin.clear();                          //loi, xoa cac bit loi
        cin.ignore(IGN,'\n');                  //huy cac ky tu, gom ca dong moi
        cout<<"\nNhap vao inches khong dung\n"; //bat dau nhap lai
    } //ket thuc vong lap inches
}

int isint(char* str)                          //ham tra ve true neu chuoi bieu dien so nguyen

```



```

{
    int len=strlen(str); //lay do dai
    if(len==0 || len>5) //neu khong nhap hoac nhap qua dai
        return 0;
    for(int j=0;j<len;j++) //kiem tra tung ky tu
        if((str[j]<'0' || str[j]>'9') && str[j]!='.') //neu khong phai chu so
            return 0; //hoac dau tru
    long n=atol(str); //chuyen sang so nguyen long
    if(n<-32768L || n>32767L) //neu vuot ra ngoai khoang so nguyen
        return 0;
    return 1; //chuoi la mot so nguyen
}

void main()
{
    Distance d; //tao mot doi tuong khoang cach
    char ans;

    do
    {
        d.getDist(); //lay gia tri khoang cach tu nguoi dung
        cout<<"\nKhoang cach=";
        d.showDist(); //hien thi no
        cout<<"\nCo tiep tục nua khong(c/k)";
        cin>>ans;
        cin.ignore(IGN, '\n'); //huy cac ky tu, gom ca dong moi
    }
    while(ans!='k'); //lap lai cho den khi gap 'k'
}

```

Sau đây là một vài tương tác với chương trình:

```

Nhap vao feet:12
Nhap vao inches:3.4
Khoang cach=12'-3.4"
Co tiep tục nua khong(c/k)c
Nhap vao feet:5.6
Feet phai la mot so nguyen
Nhap vao feet:34567
Feet phai la mot so nguyen
Nhap vao feet:4
Nhap vao inches:13
Inches phai nam trong khoang 0.0 toi 11.99
Nhap vao inches khong dung
Nhap vao inches:2.4
Khoang cach=4'-2.4"
Co tiep tục nua khong(c/k)k

```

Chúng ta đã sử dụng một mẹo ở chương trình này: thiết lập cờ trạng thái lỗi bằng tay. Sở dĩ phải làm như vậy bởi vì chúng ta muốn đảm bảo rằng giá trị **inches** lớn hơn 0 nhưng nhỏ hơn 12.0. Nếu không đúng như vậy, chúng ta bật cờ **failbit** bằng câu lệnh:

```
cin.clear(ios::failbit);
```

Khi chương trình kiểm tra để tìm lỗi với câu lệnh **cin.good()**, nó sẽ phát hiện thấy cờ **failbit** được thiết lập và thông báo rằng việc nhập vào không đúng.

2. Nhập tất cả vào bằng ký tự

Một cách khác để kiểm soát lỗi là đọc tất cả vào, thậm chí là các số, như các chuỗi ký tự. Sau đó chương trình có thể phân tích chuỗi, từng ký tự một, để xác định xem liệu người sử dụng có gõ vào những gì có nghĩa không. Nếu là những gì có nghĩa, nó sẽ chuyển sang một kiểu số thích hợp. Chi tiết về cách này tùy thuộc vào việc nhập vào cụ thể. Nếu chúng ta đang cố gắng viết một chương trình sao cho bảo vệ tốt nhất thì đây sẽ là cách tốt nhất.

Tuy nhiên, ngay cả khi nhập vào các ký tự, chúng ta cũng nên kiểm tra lỗi dùng các bit trạng thái lỗi.

5.3. VÀO/RA FILE ĐĨA VỚI CÁC STREAM

Các file đĩa yêu cầu một tập các lớp khác với các lớp được sử dụng cho bàn phím và màn hình. Các lớp này là **istream** dùng cho vào từ file, **fstream** dùng cho cả vào và ra file và **ofstream** dùng cho đưa ra file. Các đối tượng của các lớp này có thể liên kết với các file đĩa và chúng ta có thể sử dụng các hàm thành viên của chúng để đọc và ghi file.

Xem lại hình 5-1, chúng ta có thể thấy rằng **istream** được rút ra từ **istream**, **fstream** được rút ra từ **istream** và **ofstream** được rút ra từ **ostream**. Những lớp con cháu này lần lượt được rút ra từ **ios**. Bởi vậy các lớp hướng file kế thừa nhiều hàm thành viên của các lớp chung hơn. Các lớp hướng file cũng được rút ra từ lớp cơ sở **fstreambase** (bằng sự kế thừa bội). Lớp **fstreambase** chứa một đối tượng của lớp **filebuf**, nó là một bộ đệm hướng file có liên quan tới các hàm thành viên được rút ra từ lớp chung hơn là **streambuf**. Đối với nhiều thao tác file, **streambuf** được truy nhập một cách tự động, nhưng chúng ta sẽ thấy các trường hợp mà ở đó chúng ta có thể truy nhập nó một cách trực tiếp.

Các lớp **istream**, **fstream** và **ofstream** được khai báo trong file tiêu đề **FSTREAM.H**. File này cũng gồm có file tiêu đề **IOSTREAM.H**, bởi vậy không cần có nó khi có file tiêu đề **FSTREAM.H**, **FSTREAM.H** sẽ quan tâm tới tất cả các dòng (**stream**) vào/ra.

Những người lập trình C sẽ thấy cách vào/ra đĩa trong C++ hơi khác cách được dùng trong C. Các hàm C cũ, chẳng hạn như **fread()**, **fwrite()**, sẽ vẫn làm việc trong C++, nhưng chúng không còn thích hợp với môi trường hướng đối tượng. Như với **printf()** và **scanf()**, chúng không mở rộng được, trái lại chúng ta có thể mở rộng các **istream** C++ để chúng làm việc với các lớp của riêng chúng ta. Cũng nên lưu ý khi sử dụng hỗn hợp cả các hàm C cũ và các **stream** C++, không phải lúc nào chúng cũng làm việc với nhau một cách hài hòa.

5.3.1. Vào/ra file định dạng (formatted file I/O)

Có hai loại vào/ra file đĩa cơ bản trong C++ là: định dạng (**formatted**) và nhị phân (**binary**). Trong vào/ra định dạng, các số được lưu trữ trên đĩa như một xâu ký tự. Vì vậy, 6.02, nhẽ ra lưu trữ dưới dạng 4 byte kiểu **float** hoặc 8 byte kiểu **double**, nó được lưu trữ như các ký tự '6', '.', '0', '2'. Điều này có thể không hiệu quả đối với các số có nhiều chữ số nhưng nó thích hợp trong nhiều tình huống và dễ cài đặt. Ít nhiều thì các ký tự và các chuỗi cũng thường được lưu trữ.

1. Ghi dữ liệu

Chương trình sau ghi một ký tự, một số nguyên, một số kiểu **double** và hai chuỗi ra đĩa có tên là **FDATA.TXT**, không đưa ra màn hình. Bản 5-2 trình bày chương trình **FORMATO**.

Listing 5-2 FORMATO

```
//formato.cpp
//ghi ket qua duoc dinh dang ra mot file, dung toan tu <<
#include<fstream.h>      //cho vào/ra file
```

```

void main()
{
    char ch='x';           //ky tu
    int j=77;              //so nguyen
    double d=6.02;         //so dau phay dong
    char st1[]="Thang";    //cac chuoi khong co dau cach
    char st2[]="Hung";

    ofstream outfile("fdata.txt");//tao doi tuong ofstream de dua ra
                                //file

    outfile<<ch
        <<j
        <<'
        <<d
        <<st1
        <<'
    <<st2;
}

```

Ở đây chương trình định nghĩa một đối tượng gọi là **outfile** thuộc lớp **ofstream**. Cùng lúc đó, nó khởi tạo đối tượng bằng tên file **FDATA.TXT**. Sự khởi tạo là để dành các tài nguyên cho file và truy nhập hoặc mở file có tên trên đĩa. Nếu file không tồn tại, nó được tạo. Nếu file đã tồn tại, nó bị hủy bỏ và dữ liệu mới sẽ thay thế dữ liệu cũ. Đối tượng **outfile** hoạt động y như **cout** trong chương trình trước, bởi vậy toán tử chèn vào được sử dụng để đưa các biến thuộc bất kỳ kiểu cơ bản nào ra file. Điều này làm việc bởi vì toán tử **<<** được chồng một cách thích hợp trong **ostream**, **ofstream** được rút ra từ lớp này.

Khi chương trình kết thúc, đối tượng **outfile** ra khỏi phạm vi hoạt động, hàm hủy của nó được gọi để đóng file đó. Bởi vậy, chúng ta không cần đóng file.

Có vài nhiễu định dạng tiềm ẩn trong vào/ra file định dạng. Thứ nhất, phải tách riêng các số (chẳng hạn 77 và 6.02) với các ký tự không phải số bằng các ký tự trắng. Bởi vì các số được lưu trữ như một dãy liên tiếp các ký tự hơn là như một trường có chiều dài cố định, đây là cách duy nhất để toán tử lấy ra có thể hiểu được khi dữ liệu được đọc trở lại từ file, ở đó một số kết thúc và số tiếp theo bắt đầu. Thứ hai, các chuỗi cũng phải được phân tách nhau bằng các ký tự trắng cũng vì lý do như vậy. Điều này ám chỉ rằng các chuỗi không được có khoảng trống. Trong ví dụ này chúng ta sử dụng ký tự trắng là (" ") cho cả hai loại ký tự giới hạn. Các ký tự không cần ký tự giới hạn bởi vì chúng có chiều dài cố định.

Chúng ta có thể kiểm tra xem chương trình **FORMATO** đã thực sự ghi dữ liệu ra đĩa chưa bằng cách kiểm tra file **FDATA.TXT** với bất kỳ hệ soạn thảo nào.

2. Đọc dữ liệu

Bất kỳ chương trình nào cũng có thể đọc dữ liệu từ file tạo bởi chương trình **FORMATO** bằng cách dùng một đối tượng **ifstream** được khởi tạo với tên file đó. File tự động mở khi đối tượng được tạo. Sau đó chương trình có thể đọc nó dùng toán tử lấy ra **>>**.

Bản 5-3 trình bày chương trình **FORMATI**, nó đọc dữ liệu trở lại từ file **FDATA.TXT**.

Listing 5-3 FORMATI

```

//formati.cpp
//doc ket qua dinh dang tu mot file, dung toan tu >>
#include<fstream.h>           //cho vao/ra file
const int MAX=80;

```

```

void main()
{
    char ch;                //cac bien rong
    int j;
    double d;
    char st1[MAX];
    char st2[MAX];

    ifstream infile("fdata.txt"); //tao doi tuong ifstream de doc
                                // du lieu vao tu file
    infile>>ch>>j>>d>>st1>>st2; //lay du lieu ra tu no
    cout<<ch<<endl
        <<j<<endl
        <<d<<endl
        <<st1<<endl
        <<st2;
}

```

Ở đây đối tượng **ifstream**, chúng ta đặt tên là **infile**, hoạt động như **cin** trong các chương trình trước. Đó là, 77 được lưu trữ trong biến **j** như một kiểu **int**, không phải như hai ký tự và 6.02 được lưu trữ như một số **double**.

3. Chuỗi có khoảng trống

Ví dụ trước sẽ không làm việc với các chuỗi có khoảng trống. Để quản lý các chuỗi như vậy, chúng ta cần ghi một ký tự giới hạn riêng sau mỗi chuỗi và sử dụng hàm **getline()**, hơn là toán tử lấy ra << (**extraction operator**), để đọc chúng vào. Chương trình sau, **OLINE**, đưa ra vài chuỗi có khoảng trống trong đó.

Listing 5-4 OLINE

```

//oline.cpp
//dua ra file cac chuoi, dung toan tu <<
#include<fstream.h>        //cho vao/ra file
void main()
{
    ofstream outfile("test.txt"); //tao doi tuong ofstream de dua ra
                                //file
    outfile<<"Truong Dai hoc Bach Khoa Ha Noi\n"
        <<"Khoa Dien tu - Vien thong\n"
        <<"Lop Dien tu 3 - k39\n"
        <<"Sinh vien: Ngo Cong Thang\n ";
}

```

Khi chương trình được chạy, các dòng văn bản được ghi ra một file. Mỗi dòng kết thúc bằng một ký tự **newline('\n')**.

Để lấy ra các chuỗi từ file đó, chương trình cần tạo một đối tượng **ifstream** và đọc từ nó mỗi lần một dòng dùng hàm **getline()**, nó là thành viên của **istream**. Hàm này đọc các ký tự, gồm cả ký tự trắng, cho đến khi gặp ký tự '\n' và đặt chuỗi lấy được trong đối số **buffer**. Kích thước cực đại của **buffer** được cho bởi đối số thứ hai. Nội dung của **buffer** được hiển thị sau mỗi dòng. Bản 5-5 trình bày chương trình này, **ILINE**.

Listing 5-5 ILINE

```

//iline.cpp
//nhap vao file cac chuoi, dung ham getline()

```

```

#include<fstream.h>          //cho vào/ra file
void main()
{
    const int MAX=80;        //kich thước của buffer
    char buffer[MAX];        //bỏ dấu ký tự
    ifstream infile("test.txt"); //tạo file để nhập vào
    while(infile)             //cho đến cuối file
    {
        infile.getline(buffer,MAX); //đọc một dòng văn bản
        cout<<buffer<<endl;
    }
}

```

Kết quả mà chương trình ILINE đưa ra màn hình giống như dữ liệu được ghi vào file TEST.TXT bởi chương trình OLINE. Chương trình không có cách nào biết được có bao nhiêu chuỗi ở trong file, bởi vậy nó tiếp tục đọc mỗi lần một chuỗi cho đến khi gặp một EOF. Lưu ý là đừng sử dụng chương trình này để đọc một file văn bản bất kỳ. Nó yêu cầu tất cả các dòng văn bản phải kết thúc bằng ký tự '\n' và nếu gặp một file không như thế thì chương trình sẽ treo.

4. Tìm kiếm cuối file

Như chúng ta đã biết, các đối tượng rút ra từ ios chứa các bit trạng thái lỗi mà có thể kiểm tra để xác định kết quả của các hoạt động. Khi đọc một file, từng ít một như trong chương trình ILINE, cuối cùng thì chúng ta cũng gặp dấu hiệu kết thúc file. EOF là một tín hiệu gửi tới chương trình từ phần cứng khi không còn dữ liệu để đọc. Trong ILINE, chúng ta có thể sử dụng cấu trúc sau để kiểm tra kết thúc file:

```
while(!infile.eof())          //cho đến khi gặp cuối file
```

Tuy nhiên, kiểm tra riêng một eofbit có nghĩa là sẽ không kiểm tra các bit lỗi khác, chẳng hạn failbit và badbit, nó cũng có thể xảy ra, mặc dù rất hiếm. Để kiểm tra tất cả các bit lỗi, chúng ta thay đổi điều kiện vòng lặp:

```
while(infile.good())          //cho đến khi gặp bất kỳ lỗi nào
```

Nhưng đơn giản hơn chúng ta kiểm tra trực tiếp stream, như trong chương trình ILINE:

```
while (infile)                //cho đến khi gặp bất kỳ lỗi nào
```

Bất kỳ đối tượng stream nào cũng có một giá trị mà có thể được kiểm tra để tìm các điều kiện lỗi thông thường, gồm có cả EOF. Nếu bất kỳ một điều kiện như vậy đúng, đối tượng trả về giá trị 0. Nếu mọi thứ đều tốt, đối tượng trả về giá trị khác 0. Giá trị này thực chất là một con trỏ, nhưng địa chỉ được trả về không có ý nghĩa trừ được kiểm tra để xác định là giá trị 0 hay khác 0.

5. Vào/ra ký tự

Các hàm put() và get(), chúng tương ứng là thành viên của ostream và istream, có thể được sử dụng để đưa ra và nhận vào các ký tự. Bản 5-6 trình bày chương trình OCHAR, nó đưa ra từng ký tự một.

Listing 5-6 OCHAR

```

//ochar.cpp
//đưa ra file các ký tự
#include<fstream.h>          //cho vào/ra file
#include<string.h>           //cho strlen()
void main()
{

```

```

char str[]="Thoi gian la mot nguoi thay, nhưng that khong may\n"
           "no giết chết tất cả những học trò của nó Berlioz";

ofstream outfile("test.txt"); //tao file để đưa ra
for(int j=0;j<strlen(str);j++)//cho từng ký tự
    outfile.put(str[j]);      //ghi nó ra file
}

```

Trong chương trình này, đối tượng **ofstream** được tạo như trong chương trình OLINE. Chiều dài của chuỗi được tìm thấy bằng hàm **strlen()** và các ký tự được đưa ra màn hình bằng hàm **put()** trong một vòng lặp **for**. Chuỗi **str** được ghi ra file **TEST.TXT**. Chúng ta có thể đọc lại file này và hiển thị nó bằng chương trình **ICHAR** sau:

Listing 5-7 ICHAR

```

//ichar.cpp
//doc vào từ file với các ký tự
#include<fstream.h>      //cho vào/ra file
void main()
{
    char ch;              //ký tự để đọc
    ifstream infile("test.txt"); //tao file để đọc vào
    while(infile)          //cho đến khi gặp cuối file
    {
        infile.get(ch);    //đọc ký tự
        cout<<ch;         //hiển thị nó
    }
}

```

Chương trình này sử dụng hàm **get()** và tiếp tục đọc cho đến khi gặp EOF. Mỗi ký tự đọc được từ file được hiển thị bởi **cout**, bởi vậy toàn bộ chuỗi được hiển thị trên màn hình.

6. Truy nhập trực tiếp đối tượng *streambuf*

Một cách khác để đọc các ký tự từ một file là hàm **rdbuf()**, một thành viên của lớp **ios**. Hàm này trả về một con trỏ tới đối tượng **streambuf** (hay **filebuf**) liên kết với đối tượng **stream**. Đối tượng này chứa một bộ đệm giữ các ký tự đọc từ **stream**, bởi vậy chúng ta có thể sử dụng con trỏ tới nó như một đối tượng dữ liệu ở bên phải các đối tượng **stream**. Bản 5-8 là chương trình **ICHAR2**.

Listing 5-8 ICHAR2

```

//ichar2.cpp
//doc vào từ file với các ký tự
#include<fstream.h>      //cho vào/ra file
void main()
{
    ifstream infile("test.txt"); //tao file để đọc vào
    cout<<infile.rdbuf();        //gửi bộ đệm của nó tới cout
}

```

Chương trình này có tác dụng như **ICHAR**. Nó là một chương trình hướng file ngắn nhất.

5.3.2. Vào/ra nhị phân (binary I/O)

Chúng ta có thể ghi một vài số ra đĩa sử dụng vào ra định dạng, nhưng nếu chúng ta cần lưu trữ một số lượng lớn dữ liệu số thì hiệu quả hơn là sử dụng vào/ra nhị phân, ở đó các số được lưu trữ

như chúng ở trong bộ nhớ RAM của máy tính hơn là các ký tự. Trong vào/ra nhị phân, một số nguyên được luôn lưu trữ trong 2 byte, trong khi đó dạng văn bản của nó, chẳng hạn 12345 yêu cầu 5 byte. Tương tự, kiểu float luôn luôn được lưu trữ trong 4 byte, trong khi đó phiên bản định dạng của 6.0214e13 phải cần 10 byte.

Ví dụ tiếp theo cho thấy một mảng số nguyên được ghi ra đĩa và đọc trở lại bộ nhớ sử dụng dạng nhị phân như thế nào. Chúng ta sử dụng hai hàm mới: **write()**, một thành viên của **ofstream** và hàm **read()**, một thành viên của **ifstream**. Các hàm này nghĩ về dữ liệu trong thuật ngữ byte (kiểu **char**). Chúng không quan tâm đến dữ liệu được định dạng như thế nào, chúng chỉ đơn giản chuyển từ một bộ đệm đầy file tới file đĩa và từ file đĩa vào bộ đệm. Các tham số cho **write()** và **read()** là địa chỉ của bộ đệm dữ liệu và kích thước của chúng. Địa chỉ phải ép thành kiểu **char** và kích thước tính bằng byte (ký tự) chứ không phải là số mục dữ liệu trong bộ đệm. Bản 5-9 trình bày chương trình BINIO.

Listing 5-9 BINIO

```
//binio.cpp
//vao va ra nhi phan voi cac so nguyen
#include<fstream.h>           //cho vao ra file
const int MAX=100;           //so luong so nguyen
int buff[MAX];                //bo dem cho so nguyen
void main()
{
    int j;
    for(j=0;j<MAX;j++)        //dua vao day bo dem cac du lieu
        buff[j]=j;           //(0,1,2,...)

    ofstream os("edata.dat",ios::binary);
    os.write((char*)buff,MAX*sizeof(int)); //ghi ra dia
    os.close();                //dong file
    for(j=0;j<MAX;j++)         //xoa bo dem
        buff[j]=0;

    ifstream is("edata.dat",ios::binary); //tao mot stream file vao
    is.read((char*)buff,MAX*sizeof(int)); //doc tu file

    for(j=0;j<MAX;j++)
        if(buff[j]!=j)
            { cerr<<"\nDu lieu khong dung";return;}
    cout<<"\nDu lieu dung";
}
```

Phải sử dụng đối số **ios::binary** trong tham số thứ hai cho hàm **write()** và **read()** khi làm việc với dữ liệu nhị phân. Bởi vì mặc định là chế độ văn bản, nó ít quan tâm đến dữ liệu. Ví dụ, trong chế độ văn bản, ký tự '\n' được mở rộng thành hai byte - một xuống dòng và một về đầu dòng - trước khi được lưu ra đĩa. Điều này làm cho một file văn bản dễ đọc hơn đối với hệ soạn thảo nhưng nó gây ra sự nhầm lẫn khi áp dụng cho dữ liệu nhị phân bởi vì mỗi byte cho giá trị ASCII được chuyển sang 2 byte.

5.3.3. Vào/ra đối tượng (object I/O)

Bởi vì C++ là ngôn ngữ hướng đối tượng nên việc xem xét các đối tượng được ghi ra đĩa và đọc vào từ đĩa như thế nào là rất có nghĩa. Ví dụ tiếp theo cho thấy quá trình này. Lớp **person**, được sử dụng trong vài ví dụ trước, cung cấp các đối tượng.

1. Ghi một đối tượng ra đĩa

Khi ghi một đối tượng ra đĩa, nói chung nên sử dụng chế độ nhị phân. Chế độ này ghi cấu hình bit ra đĩa như nó được lưu trong bộ nhớ và đảm bảo rằng dữ liệu số chứa trong các đối tượng được quản lý đúng. Bản 5-10 là chương trình OPERS, nó yêu cầu người sử dụng nhập thông tin về một đối tượng lớp **person** và sau đó ghi đối tượng này ra file đĩa **PERSON.DAT**.

Listing 5-10 OPERS

```
//opers.cpp
//ghi doi tuong person ra dia
#include<fstream.h> //cho vao ra file
class person       //lop person
{
    protected:
        char name[40]; //ten nguoi
        int age;       //tuoi
    public:
        void getData() //lay du lieu
        {
            cout<<"\nNhap vao ten:"; cin>>name;
            cout<<"\nNhap vao tuoi:"; cin>>age;
        }
};
void main()
{
    person pers; //tao mot doi tuong person

    pers.getData(); //lay du lieu cho nguoi
    ofstream outfile("PERSON.DAT",ios::binary); //tao mot doi tuong ofstream
    outfile.write((char*)&pers,sizeof(pers)); //ghi ra no
}
```

Hàm thành viên **getData()** của **person** được gọi để thông báo cho người sử dụng nhập thông tin cho một người. Đây là mẫu tương tác với chương trình:

Nhap vao ten:Thang

Tuoi:25

Sau đó nội dung của đối tượng **pers** được ghi ra đĩa bằng hàm **write()**. Chúng ta sử dụng toán tử **sizeof()** để tìm kích thước của đối tượng **pers**.

2. Đọc một đối tượng từ đĩa

Đọc trở lại một đối tượng từ file **PERSON.DAT** yêu cầu hàm thành viên **read()**. Bản 5-11 trình bày chương trình IPERS.

Listing 5-11 IPERS

```
//ipers.cpp
//doc doi tuong person tu dia
#include<fstream.h> //cho vao ra file
class person       //lop person
{
    protected:
        char name[40]; //ten nguoi
        int age;       //tuoi
    public:
        void readData() //lay du lieu
        {
            cout<<"\nNhap vao ten:"; cin>>name;
            cout<<"\nNhap vao tuoi:"; cin>>age;
        }
};
void main()
{
    person pers; //tao mot doi tuong person

    pers.readData(); //lay du lieu cho nguoi
    ifstream infile("PERSON.DAT",ios::binary); //tao mot doi tuong ifstream
    infile.read((char*)&pers,sizeof(pers)); //doc ra no
}
```



```

public:
    void showData()           //hien thi du lieu
    {
        cout<<"\nTen:"<<name;
        cout<<"\nTuoi:"<<age;
    }
};
void main()
{
    person pers;                //tao mot bien person
    ifstream infile("PERSON.DAT",ios::binary); //tao mot doi tuong ifstream
    infile.read((char*)&pers,sizeof(pers));    //doc no
    pers.showData();             //hien thi du lieu
}

```

Kết quả từ chương trình IPERS phản ánh bất kỳ dữ liệu gì mà chương trình OPERS đặt vào file PERSON.DAT.

```

Ten:Thang
Tuoi:25

```

3. Cấu trúc dữ liệu thích hợp

Để làm việc đúng, các chương trình đọc và ghi các đối tượng tới file, như IPERS và OPERS, phải có cùng một lớp các đối tượng. Các đối tượng **person** trong các chương trình này dài chính xác là 42 byte, 40 byte đầu chiếm bởi chuỗi biểu diễn tên người và hai byte sau chứa một số nguyên biểu diễn tuổi. Nếu hai chương trình hiểu trường tên có chiều dài khác nhau thì không chương trình nào có thể đọc chính xác một file tạo bởi chương trình kia.

Tuy nhiên, mặc dù các lớp **person** trong OPERS và IPERS có dữ liệu giống nhau nhưng chúng có thể có các hàm thành viên khác nhau. Chương trình thứ nhất có một hàm **getData()**, trái lại chương trình thứ hai chỉ có hàm thành viên **showData()**. Chẳng có vấn đề gì trong việc chúng sử dụng các hàm thành viên khác nhau bởi vì các hàm thành viên không được ghi ra đĩa cùng với dữ liệu của đối tượng. Dữ liệu phải có dạng giống nhau, còn sự không nhất quán của các hàm thành viên không ảnh hưởng gì. Điều này chỉ đúng trong các lớp không sử dụng hàm ảo.

Nếu đọc và ghi các đối tượng của các lớp dẫn xuất tới một file thì chúng ta phải cẩn thận hơn. Các đối tượng của các lớp dẫn xuất sử dụng các hàm ảo có chứa **vptr** (nói trong mục 4.1). Con trỏ này giữ địa chỉ của bảng các hàm ảo dùng trong lớp. Khi ghi một đối tượng ra đĩa, số này được ghi cùng với dữ liệu khác của đối tượng. Nếu thay đổi các hàm thành viên của một lớp, số này cũng thay đổi theo. Nếu ghi một đối tượng của một lớp tới một file và sau đó đọc lại nó vào một đối tượng của một lớp có dữ liệu giống với dữ liệu của lớp khi ghi nhưng các hàm thành viên khác thì chúng ta sẽ gặp rắc rối lớn nếu cố dùng các hàm ảo trên đối tượng đó. Phải chắc chắn là một lớp để đọc một đối tượng phải giống lớp đã ghi ra đĩa.

4. Vào/ra nhiều đối tượng

Các chương trình OPERS và IPERS đã ghi và đọc chỉ một đối tượng mỗi lần. Ví dụ tiếp theo mở ra một file và ghi số lượng đối tượng mà người sử dụng muốn. Sau đó nó đọc và hiển thị toàn bộ nội dung của file đó. Bản 5-12 trình bày chương trình này, DISKFUN.

Listing 5-12 DISKFUN

```

//diskfun.cpp
//doc va ghi vai doi tuong person ra dia
#include<fstream.h>           //cho vao ra file
class person                 //lop person

```

```

{
protected:
    char name[40];          //ten nguoi
    int age;                //tuoi
public:
    void getData()          //lay du lieu
    {
        cout<<"\nNhap vao ten:";cin>>name;
        cout<<"\nNhap vao tuoi:";cin>>age;
    }
    void showData()         //hien thi du lieu
    {
        cout<<"\nTen:"<<name;
        cout<<"\nTuoi:"<<age;
    }
};

void main()
{
    person pers;            //tao mot bien person
    char ch;

    fstream file;           //tao mot file vao va ra
                             //mo de ghi them vao
    file.open("PERSON.DAT",ios::app | ios::out | ios::in | ios::binary);
                             //lay du lieu tu nguoi su dung de ghi ra
                             //file

    do
    {
        cout<<"\nNhap du lieu cho nguoi:";
        pers.getData();      //lay du lieu cua mot nguoi
        file.write((char*)&pers,sizeof(pers)); //ghi no ra dia
        cout<<"Co nhap nua khong(c/k)?";
        cin>>ch;
    }
    while(ch!='c');
    file.seekg(0);           //thiet lap lai de bat dau doc
    file.read((char*)&pers,sizeof(pers));
    while(!file.eof())       //thoat khi gap cuoi file
    {
        cout<<"\nNguoi:";    //hien thi
        pers.showData();
        file.read((char*)&pers,sizeof(pers)); //doc nguoi tiep theo
    }
}

```

Đây là vài mẫu tương tác với chương trình DISKFUN với giả thiết là chương trình đã chạy trước đây và ba đối tượng đã được ghi ra đĩa.

```

Nhap du lieu cho nguoi:
Nhap vao ten:Oanh
Nhap vao tuoi:22
Co nhap nua khong(c/k)?k
Nguoi:
Ten:Thang

```

Tuoi:25
 Nguoi:
 Ten:Hùng
 Tuoi:23
 Nguoi:
 Ten:Hoà
 Tuoi:25
 Nguoi:
 Ten:Oanh
 Tuoi:22

5.3.4. Lớp fstream

Từ trước đến giờ trong chương này, các đối tượng file mà chúng ta đã tạo hoặc là để nhập vào hoặc là để đưa ra. Trong chương trình DISKFUN, chúng ta đã tạo một file mà có thể dùng cho cả vào và ra. Nó yêu cầu một đối tượng của lớp **fstream**, được rút ra từ **iostream**, **iostream** lại được rút ra từ **istream** và **ostream**, bởi vậy nó có thể quản lý cả vào và ra.

1. Hàm open()

Trong các ví dụ trước đây, các đối tượng file được tạo ra và khởi tạo trong cùng một câu lệnh:

```
ofstream outfile("TEST.TXT");
```

Trong DISKFUN, chúng ta đã sử dụng một cách khác: tạo file trong một lệnh và mở nó trong một lệnh khác với hàm **open()**, nó là thành viên của lớp **fstream**. Đây là cách hữu ích trong các tình huống mà ở đó việc mở file có thể lỗi. Chúng ta có thể tạo một đối tượng chỉ một lần rồi sau đó mở nó nhiều lần mà không cần tạo một đối tượng **stream** trong mỗi lần mở.

2. Bit chế độ

Chúng ta đã biết bit chế độ **ios::binary** trong các ví dụ trước đây. Trong hàm **open()** có một vài bit chế độ mới. Các bit chế độ được định nghĩa trong **ios**, chỉ ra nhiều cách mở một đối tượng **stream**. Bảng 5-11 cho thấy các khả năng này.

Trong chương trình DISKFUN, chúng ta sử dụng **app** bởi vì muốn giữ lại tất cả những gì đã có trong file trước đây; bất kể những gì chúng ta muốn ghi ra file sẽ được thêm vào cuối nội dung đã tồn tại. Sử dụng **in** và **out** để thực hiện cả vào và ra trên cùng file đó và sử dụng **binary** để ghi ra các đối tượng nhị phân. Các gạch thẳng đứng ở giữa các cờ làm cho các bit biểu diễn các cờ này được kết hợp (ORed) với nhau thành một số nguyên để vài cờ có tác dụng đồng thời.

Bảng 5-11. Các bit chế độ cho hàm **open()**

Bit chế độ	Tác dụng
In	Mở để đọc (mặc định cho ifstream).
Out	Mở để ghi (mặc định cho ofstream).
Ate	Bắt đầu đọc hoặc ghi ở cuối file (ATEnd).
App	Bắt đầu ghi tại cuối file (APPend).
Trunc	Cắt bỏ file tới chiều dài bằng 0 (TRUNCate).
Nocreat	Lỗi khi mở nếu file không tồn tại.
Noreplace	Lỗi khi mở để ghi ra nếu file đã tồn tại, trừ khi ate hoặc app được thiết lập.
Binary	Mở file trong chế độ nhị phân (mặc định là văn bản).

Mỗi lần chúng ta ghi một đối tượng **person** ra file dùng hàm **write()**. Sau khi kết thúc ghi, chúng ta đọc lại toàn bộ file đó. Trước khi làm điều này, chúng ta phải thiết lập lại vị trí hiện tại của file bằng hàm **seekg()**. Làm điều này để đảm bảo bắt đầu đọc tại đầu file. Sau đó, trong vòng lặp **while**, lặp lại việc đọc một đối tượng **person** từ file và hiển thị nó trên màn hình.

Cứ thế tiếp tục cho đến khi đọc tất cả các đối tượng **person** - trạng thái này được phát hiện nhờ hàm **eof()**, nó trả về trạng thái của **ios::eofbit**.

5.4. CÁC LỖI FILE VÀ CON TRỎ FILE

Phần này gồm hai chủ đề ngắn: cách quản lý lỗi vào ra file và cách xác định chính xác một vị trí trong file để đọc và ghi dữ liệu.

5.4.1. Quản lý lỗi vào/ra file

Trong các ví dụ về file từ trước đến giờ chúng ta không quan tâm đến các tình huống lỗi. Trong các ví dụ đó chúng ta cho rằng các file mở để đọc đã tồn tại và các file mở để ghi có thể được tạo và ghi thêm dữ liệu vào. Chúng ta cũng cho rằng không có lỗi gì xảy ra trong quá trình đọc và ghi. Trong một chương trình thực tế, việc kiểm tra các giả thiết như vậy và các thao tác thích hợp nếu chúng diễn ra không đúng là rất quan trọng. Một file mà chúng ta nghĩ là đã tồn tại có thể không tồn tại hoặc một tên mà chúng ta định dùng cho một file mới có thể đã gắn với một file đã tồn tại hoặc có thể không còn nhiều chỗ trống trên đĩa hoặc có một cửa ổ đĩa được mở.

1. Xử lý các lỗi

Chương trình sau cho thấy cách quản lý các lỗi như vậy một cách thuận lợi nhất. Tất cả các thao tác liên quan đến đĩa được kiểm tra ngay sau khi chúng được thực hiện. Nếu một lỗi đã xảy ra, một thông báo được đưa ra và chương trình kết thúc. Chúng ta sử dụng kỹ thuật đã được đề cập tới trước đây để kiểm tra giá trị trả về từ chính đối tượng đó, qua đó xác định được các trạng thái lỗi của nó. Chương trình mở một **stream** đưa ra, ghi một mảng các số nguyên tới nó với một lời gọi hàm **write()** và đóng đối tượng. Sau đó nó mở một đối tượng **stream** vào và đọc mảng số nguyên đó với một lời gọi hàm **read()**. Bản 5-13 trình bày chương trình **REWERR**.

Listing 5-13 REWERR

```
//rewerr.cpp
//quan ly loi trong khi vao ra
#include<fstream.h>           //cho cac stream file
#include<process.h>           //cho exit()

const int MAX=1000;
int buff[MAX];
void main()
{
    int j;
    for(j=0;j<MAX;j++)        //dua vao day bo dem cac du lieu
        buff[j]=j;           //(0,1,2,...)

    ofstream os;               //tao mot stream dua ra
    os.open("edata.dat",ios::trunc | ios::binary);
    if(!os)
    { cerr<<"\nCould not open output file";exit(1);}
    cout<<"\nWritting...";

    os.write((char*)buff,MAX*sizeof(int)); //ghi ra dia
```

```

if(!os)
{ cerr<<"\nCould not write to file";exit(1);}
os.close(); //dong file
for(j=0;j<MAX;j++) //xoa bo dem
buff[j]=0;
ifstream is; //tao mot stream file vao
is.open("edata.dat",ios::binary);
if(!os)
{ cerr<<"\nCould not open input file";exit(1);}
cout<<"\nReading...";
is.read((char*)buff,MAX*sizeof(int)); //doc tu file
if(!os)
{ cerr<<"\nCould not read from file";exit(1);}
//kiem tra du lieu

for(j=0;j<MAX;j++)
if(buff[j]!=j)
{ cerr<<"\nData is incorrect";exit(1);}
cout<<"\nData is correct";
}

```

2. Phân tích các lỗi

Trong ví dụ REWERR, để xác định xem có lỗi xảy ra trong một hoạt động vào ra file không, chúng ta kiểm tra giá trị trả về của toàn thể đối tượng **stream**:

```

if(!is)
//lỗi xảy ra

```

Ở đây là trả về một giá trị con trỏ nếu mọi thứ diễn ra tốt, trả về 0 nếu không diễn ra tốt đẹp. Đây là cách ngắn nhất để tìm lỗi: dù là lỗi nào đi nữa, nó cũng được tìm kiếm bằng cách giống nhau. Tuy nhiên, cũng có thể sử dụng các bit trạng thái lỗi để tìm ra những thông tin cụ thể về một lỗi vào ra file. Chúng ta đã biết một số bit trạng thái này làm việc với màn hình và bàn phím. Ví dụ tiếp theo, **FERRORS**, cho thấy cách sử dụng chúng trong I/O. Bản 5-14 trình bày ví dụ này.

Listing 5-14 FERRORS

```

//ferrors.cpp
//kiem tra loi mo file
#include<fstream.h> //cho cac stream file
void main()
{
    ifstream file;
    file.open("GROUP.DAT",ios::nocreate);
    if(!file)
        cout<<"\nCan't open GROUP.DAT";
    else
        cout<<"\nFile open successfully";
    cout<<"\nFile="<<file;
    cout<<"\nError state="<<file.rdstate();
    cout<<"\ngood()="<<file.good();
    cout<<"\neof()="<<file.eof();
    cout<<"\nfail()="<<file.fail();
    cout<<"\nbad()="<<file.bad();
    file.close();
}

```

Đầu tiên chương trình kiểm tra giá trị của đối tượng file. Nếu giá trị của nó là 0 thì file có thể không mở được bởi vì nó không tồn tại. Đây là kết quả từ chương trình FERRORS:

```
Can't open GROUP.DAT
File=0x0000
Error state=4
good()=0
eof()=0
fail()=4
bad()=4
```

Trạng thái lỗi trả về bởi **rdstate()** là 4. Đây là bit chỉ rằng file không tồn tại; nó được thiết lập tới **true** (giá trị khác 0). Các bit khác đều được thiết lập bằng 0. Hàm **good()** trả về 1 (true) chỉ khi không có bit nào được thiết lập, bởi vậy nó trả về 0. Vì không ở cuối file nên hàm **eof()** trả về 0. Các hàm **fail()** và **bad()** trả về giá trị khác 0 bởi vì có lỗi xảy ra.

Trong một chương trình quan trọng, một vài hoặc tất cả các hàm này được sử dụng sau mỗi thao tác vào/ra để đảm bảo rằng mọi thứ đều diễn ra như mong muốn.

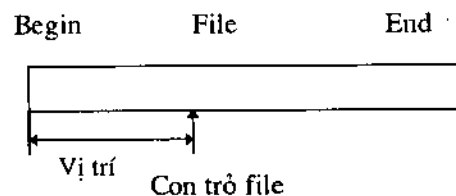
5.4.2. Con trỏ file

Mỗi đối tượng file kết hợp với nó hai giá trị nguyên gọi là **get pointer** (con trỏ đọc) và **put pointer** (con trỏ ghi). Hai giá trị này cũng được gọi là **current get position** (vị trí đọc hiện tại) và **current put position** (vị trí ghi hiện tại), hoặc nếu biết rõ giá trị nào rồi thì chỉ cần nói **current position** (vị trí hiện tại). Các giá trị này xác định số byte trong file mà ở đó sẽ diễn ra việc đọc và ghi (thuật ngữ **pointer** trong ngữ cảnh này không nên nhầm với các pointer C++ (các con trỏ) được sử dụng như các biến địa chỉ).

Chúng ta thường muốn bắt đầu đọc tại đầu file và tiếp tục đọc cho đến cuối file. Khi ghi, có thể chúng ta muốn bắt đầu ở đầu file, xóa tất cả nội dung đã có, hoặc ở cuối file nếu chúng ta muốn ghi thêm vào file. Đây là các hoạt động mặc định, bởi vậy không có hoạt động nào của con trỏ file là cần thiết ở đây. Tuy nhiên, đôi khi chúng ta phải điều khiển con trỏ file để có thể đọc và ghi tới một vị trí tùy ý trên file. Các hàm **seekg()** và **tellg()** cho phép thiết lập và kiểm tra **get pointer**, các hàm **seekp()** và **tellp()** cho phép thực hiện hành động như vậy đối với **put pointer**.

1. Xác định vị trí

Chúng ta đã có một ví dụ về **get pointer** trong chương trình DISKFUN, đó là hàm **seekg()** thiết lập tới đầu file để việc đọc bắt đầu tại đó. Hàm **seekg()** này có một đối số, nó biểu diễn vị trí tuyệt đối trong file. Bắt đầu của file là byte 0, đó là những gì mà chúng ta đã dùng trong DISKFUN. Hình 5-4 cho thấy điều này trông như thế nào.



Hình 5-4. Hàm **seekg()** có một đối số.

2. Xác định độ lệch (offset)

Hàm **seekg()** có thể được sử dụng trong hai cách. Chúng ta đã thấy cách thứ nhất, ở đó đối số đơn biểu diễn vị trí tính từ đầu file. Chúng ta cũng có thể sử dụng nó với hai đối số, đối số thứ nhất biểu diễn độ lệch (**offset**), tính từ vị trí cụ thể trong file và đối số thứ hai xác định nơi mà độ lệch tính từ đó. Có ba khả năng cho đối số thứ hai: **beg** là đầu file, **cur** là vị trí con trỏ hiện tại và **end** là cuối file. Câu lệnh:

```
seekg(-10,ios::end);
```

sẽ thiết lập **get pointer** (con trỏ đọc) tới vị trí 10 byte trước cuối file. Hình 5-5 cho thấy sự sắp xếp này.

Bản 5-15 trình bày một ví dụ sử dụng phiên bản hai đối số của hàm **seekg()** để tìm ra một đối tượng **person** cụ thể trong file **PERSON.DAT** đã được tạo bởi chương trình **DISKFUN** và hiển thị dữ liệu cho người tìm được này.

Listing 5-15 SEEKG

```
//seekg.cpp
//tìm người cụ thể trong file
#include<fstream.h>           //cho vào ra file
class person                  //lớp person
{
protected:
    char name[40];            //ten người
    int age;                  //tuoi
public:
    void showData()           //hiển thị dữ liệu
    {
        cout<<"\nTen:"<<name;
        cout<<"\nTuoi:"<<age;
    }
};
void main()
{
    person pers;              //tạo một biến person

    ifstream infile;          //tạo một file vào
    infile.open("PERSON.DAT",ios::binary); //mở file

    infile.seekg(0,ios::end);  //chuyển tới byte 0 tính từ cuối file
    int endposition=infile.tellg();//tìm vị trí hiện tại của get
                                //pointer
    int n=endposition/sizeof(pers);//số người trong file
    cout<<"\nCó "<<n<<" người trong file";
    cout<<"\nTìm người số:";
    cin>>n;
    int position=(n-1)*sizeof(pers); //tính số byte tính từ đầu file
                                        //tới vị trí cần đọc
    infile.seekg(position);        //chuyển tới vị trí đó
    infile.read((char*)&pers,sizeof(pers));//đọc một người
    pers.showData();               //hiển thị người này
}
```

Đây là kết quả đưa ra từ chương trình, giả sử file **PERSON.DAT** giống file trong ví dụ **DISKFUN**.

Có 3 người trong file
Tìm người số:2
Ten:Thang
Tuoi:25

Đối với người sử dụng, chúng ta đánh số bắt đầu từ 1, mặc dù chương trình đánh số từ 0, bởi vậy người số 2 là người thứ ba trong file.

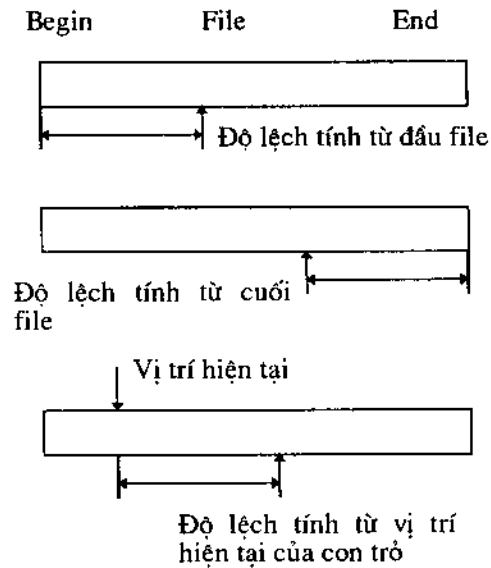
Việc đầu tiên chương trình làm là tìm ra bao nhiêu người có trong file bằng cách chuyển con trỏ đọc (get pointer) xuống cuối file với lệnh:

```
seekg(0,ios::end);
```

3. Hàm tellg()

Hàm **tellg()** trả về vị trí hiện tại của **get pointer**. Sau khi đưa **get pointer** về cuối file, chương trình sử dụng hàm **tellg()** để trả về vị trí của **get pointer**; đây là chiều dài file tính bằng byte. Tiếp theo, chương trình tính toán xem có bao nhiêu đối tượng **person** trong file bằng cách chia kích thước của file cho kích thước của một **person**; sau đó nó hiển thị kết quả.

Trong kết quả đưa ra ở trên, người sử dụng xác định đối tượng thứ hai trong file và chương trình tính toán xem vị trí này hết bao nhiêu byte tính từ đầu file rồi dùng hàm **seekg()** để đến vị trí này. Sau đó sử dụng hàm **read()** để đọc dữ liệu của người ở vị trí đó. Cuối cùng nó hiển thị dữ liệu đọc được của đối tượng với hàm thành viên **showData()**.



Hình 5-5. Hàm seekg() có hai đối số.

5.5. VÀO/RA FILE DÙNG HÀM THÀNH VIÊN

Từ trước đến giờ chúng ta đã để hàm **main()** điều hành chi tiết việc vào/ra file. Điều này thích hợp cho việc minh họa, nhưng trong các chương trình hướng đối tượng thực tế việc có các hoạt động vào ra file như các hàm thành viên của một lớp là đương nhiên. Trong phần này chúng ta có hai chương trình làm việc này. Chương trình thứ nhất sử dụng các hàm thành viên thông thường trong đó mỗi đối tượng đều có trách nhiệm đọc và ghi chính nó ra file. Chương trình thứ hai cho thấy các hàm thành viên tĩnh (**static member function**) có thể đọc và ghi tất cả các đối tượng của một lớp cùng một lúc như thế nào.

5.5.1. Các đối tượng đọc và ghi chính nó

Thông thường, để từng thành viên của một lớp đọc và ghi chính nó ra một file là có ý nghĩa. Đây là một cách đơn giản và làm việc tốt nếu không có nhiều đối tượng được đọc và ghi cùng một lúc. Trong ví dụ này, chúng ta thêm vào các hàm thành viên - **diskOut()** và **diskIn()** - vào lớp **person**. Các hàm này cho phép một đối tượng **person** ghi chính nó ra đĩa và đọc chính nó trở lại từ đĩa.

Để đơn giản, chúng ta có vài giả thiết: thứ nhất, tất cả các đối tượng của lớp được lưu trữ trong cùng một file, có tên là **PERSON.DAT**. Thứ hai, các đối tượng mới luôn luôn được thêm vào cuối file. Một đối tượng của hàm **diskIn()** cho phép ta đọc dữ liệu cho bất kỳ người nào trong file. Để ngăn cản việc đọc dữ liệu vượt quá cuối file, chúng ta đưa vào một hàm thành viên tĩnh, **diskOut()**, nó trả lại số người được lưu trữ trong file. Bản 5-16 trình bày chương trình này, **REWOBJ**.

Listing 5-16 REWOBJ

```
//rewobj.cpp
//cac doi tuong person thuc hien vào ra đĩa
```



```

#include<fstream.h>
class person
{
    protected:
        char name[40];           //ten nguoi
        int age;                  //tuoi
    public:
        void getData()           //lay du lieu
        {
            cout<<"\n Nhap vao ten:"; cin>>name;
            cout<<" Nhap vao tuoi:"; cin>>age;
        }
        void showData()         //hien thi du lieu
        {
            cout<<"\n Ten:"<<name;
            cout<<"\n Tuoi:"<<age;
        }
        void diskIn(int);        //doc tu file
        void diskOut();          //ghi ra file
        static int diskCount();  //tra ve so nguoi trong file
};

void person::diskIn(int pn)     //doc mot nguoi co so la pn tu file
{
    ifstream infile;            //tao mot stream
    infile.open("PERSON.DAT");   //mo no
    infile.seekg(pn*sizeof(person)); //di chuyen con tro file
    infile.read((char*)this,sizeof(*this)); //doc mot nguoi tu file
}

void person::diskOut()          //ghi mot nguoi vao cuoi file
{
    ofstream outfile;
    outfile.open("PERSON.DAT",ios::app | ios::binary);
    outfile.write((char*)this,sizeof(*this)); //ghi ra file
}

int person::diskCount()         //tra ve so nguoi trong file
{
    ifstream infile;
    infile.open("PERSON.DAT",ios::binary);
    infile.seekg(0,ios::end);    //dua con tro ve cuoi file
    return infile.tellg()/sizeof(person);
}
//-----
void main()
{
    person p;                   //tao mot doi tuong person rong
    char ch;
    do
    {
        cout<<"\nNhap du lieu cho nguoi:";
        p.getData(); //lay du lieu
        p.diskOut(); //ghi ra dia
        cout<<"\n\nCo tiep khong(c/k)?";
    }
}

```

```

        cin>>ch;
    }
    while(ch=='c');           //cho den khi nguoi su dung go 'k'
    int n=person::diskCount(); //co bao nhieu nguoi trong file
    cout<<"\nCo "<<n<<" nguoi trong file";
    for(int j=0;j<n;j++)
    {
        cout<<"\nNguoi thu "<<(j+1);
        p.diskIn(j);          //doc mot nguoi tu dia
        p.showData();          //hien thi
    }
}

```

Không có quá nhiều cái mới ở đây, hầu hết các thành phần của chương trình này chúng ta đã gặp. Nó hoạt động giống như chương trình DISKFUN. Tuy nhiên, chú ý rằng tất cả các chi tiết về hoạt động vào/ra file đĩa là ẩn đối với hàm **main()**, nó được giấu trong lớp **person**.

Chúng ta không biết trước dữ liệu ở đâu mà đọc và ghi bởi vì mỗi đối tượng ở một nơi khác nhau trong bộ nhớ. Tuy nhiên, con trỏ **this** luôn bảo cho chúng ta biết chúng ta đang ở đâu khi chúng ta ở trong một hàm thành viên. Trong các hàm **read()** và **write()**, địa chỉ của đối tượng được đọc và ghi là **this** và kích thước của nó là **sizeof(*this)**.

Đây là một vài mẫu tương tác, cho rằng đã có hai người trong file khi chương trình bắt đầu:

```

Nhập dữ liệu cho người:
Nhập vào tên:Hung
Nhập vào tuổi:23

```

```

Có tiếp không(c/k)?c
Nhập vào tên:Hoa
Nhập vào tuổi:24
Có tiếp không(c/k)?k

```

```

Có 4 người trong file
Người thu 1
Tên:Binh
Tuổi:24
Người thu 2
Tên:Thang
Tuổi:24
Người thu 3
Tên:Hung
Tuổi:23
Người thu 4
Tên:Hoa
Tuổi:24

```

Nếu muốn người sử dụng có thể xác định tên file sử dụng bởi lớp, thay vì cố định nó trong các hàm thành viên như trong chương trình trên, chúng ta có thể tạo một biến thành viên tĩnh (ví dụ **static char filename[15]**) và một hàm tĩnh để thiết lập nó. Hoặc nếu muốn mỗi đối tượng ghi ra một tên file khác nhau thì sử dụng hàm không tĩnh.

5.5.2. Các lớp đọc và ghi chính nó

Chúng ta cho rằng có nhiều đối tượng trong bộ nhớ và muốn ghi tất cả chúng ra file. Để một hàm thành viên cho mỗi đối tượng mở file, ghi đối tượng ra file và sau đó đóng file, như trong ví dụ REWOBJ là không hiệu suất. Mở file một lần, ghi tất cả các đối tượng ra file và sau đó đóng nó sẽ nhanh hơn rất nhiều - có nhiều đối tượng hơn và đúng hơn.

1. Hàm tĩnh (static function)

Có một cách ghi đồng thời nhiều đối tượng là sử dụng một hàm thành viên tĩnh. Nó dùng cho toàn bộ lớp hơn là cho một đối tượng. Hàm này có thể ghi tất cả các đối tượng một lúc. Một hàm như vậy làm thế nào biết được tất cả các đối tượng ở đâu? Nó có thể truy nhập một mảng con trỏ tới các đối tượng, mà có thể được lưu trữ như dữ liệu tĩnh (**static data**). Khi một đối tượng được tạo, một con trỏ tới nó được lưu trong mảng này. Một thành viên dữ liệu tĩnh có thể biết được bao nhiêu đối tượng đã được tạo. Hàm ghi tĩnh có thể mở file; sau đó, trong một vòng lặp, có thể lướt qua mảng, ghi lần lượt từng đối tượng; cuối cùng nó có thể đóng file.

2. Kích thước của các đối tượng dẫn xuất

Để làm cho mọi cái trở nên thú vị, chúng ta cùng làm một giả định chi tiết hơn: các đối tượng lưu trong bộ nhớ có kích thước khác nhau. Tại sao có tình huống này? Tình huống này xuất hiện điển hình khi có vài lớp được rút ra từ một lớp cơ sở. Ví dụ, xét chương trình EMPINH ở mục 2.2. Trong chương trình này chúng ta có một lớp **employee** đóng vai trò là lớp cơ sở cho các lớp **manager**, **scientist** và **laborer**. Các đối tượng của các lớp dẫn xuất này có kích thước khác nhau bởi vì chúng chứa số lượng dữ liệu khác nhau. Do đó, ngoài tên và mã nhân viên, dùng cho tất cả nhân viên, còn có chức vụ cho người quản lý (**manager**) và số sách xuất bản cho nhà khoa học (**scientist**). Không có dữ liệu thêm vào cho công nhân (**laborer**).

Chúng ta muốn ghi dữ liệu từ một danh sách chứa cả ba đối tượng dẫn xuất (**manager**, **scientist** và **laborer**) chỉ dùng một vòng lặp và một hàm thành viên **write()** của **ofstream**. Nhưng để sử dụng hàm này chúng ta cần biết kích thước của đối tượng cho đối số thứ hai của hàm **write()**.

Giả sử chúng ta có một mảng con trỏ (có tên là **arrap[j]**) trỏ tới các đối tượng kiểu **employee**. Các con trỏ này có thể trỏ tới các đối tượng của ba lớp dẫn xuất (xem chương trình VIRTPERS ở mục 4.2). Chúng ta biết rằng nếu sử dụng hàm ảo chúng ta có thể tạo các câu lệnh như:

```
arrap[j]->putData();
```

Phiên bản hàm **putData()** phù hợp với đối tượng được trỏ tới bởi con trỏ sẽ được sử dụng hơn là các hàm lớp cơ sở. Nhưng chúng ta có thể sử dụng hàm **sizeof()** để trả về kích thước của một đối số con trỏ được không? Nghĩa là chúng ta có thể viết như thế này được không?

```
outfile.write((char*)arrap[j],sizeof(*arrap[j])); //không tốt
```

Không được, bởi vì **sizeof()** không phải là một hàm ảo. Nó không biết rằng nó cần xem xét kiểu của đối tượng được trỏ tới hơn là kiểu của con trỏ. Nó sẽ luôn luôn trả về kích thước của một đối tượng lớp cơ sở.

3. Sử dụng hàm **typeid()**

Làm thế nào biết được kích thước của một con trỏ nếu tất cả những gì chúng ta biết là một con trỏ trỏ tới nó. Câu trả lời là hàm **typeid()**, đã nói ở mục 4.5. Chúng ta có thể sử dụng hàm này để tìm ra lớp của một đối tượng và sau đó dùng tên lớp này trong hàm **sizeof()**. Ví dụ tiếp theo cho thấy nó làm việc như thế nào. Một khi chúng ta đã biết kích thước của một đối tượng, chúng ta có thể dùng hàm **write()** để ghi đối tượng ra đĩa.

Chúng ta thêm vào một giao diện người sử dụng đơn giản cho chương trình EMPLOY và cho các hàm thành viên là ảo để có thể sử dụng mảng con trỏ tới các đối tượng. Trong chương trình này chúng ta cũng đưa vào một vài kỹ thuật tìm kiếm lỗi ở phần trước. Nó minh họa nhiều kỹ thuật có thể dùng trong một chương trình ứng dụng cơ sở dữ liệu. Nó cũng cho thấy thế mạnh của OPP. Bản 5-17 trình bày chương trình này, EMPL_IO.

Listing 5-17 EMPL_IO

```
//empl_io.cpp
//thuc hien vao ra file tren cac doi tuong employee
//quan ly cac doi tuong co kich thuoc khac nhau

#include<fstream.h>      //cho cac ham file stream
#include<conio.h>         //cho getch()
#include<process.h>       //cho exit()
#include<typeinfo.h>      //cho typeid()
const int LEN=32;        //do dai cuc dai cua ten
const int MAXEM=100;     //so nhan vien cuc dai
enum employee_type {tmanager,tscientist,tlaborer};

class employee           //lop nhan vien
{ private:
    char name[LEN];      //ten nhan cong
    unsigned long number; //ma nhan cong
    static int n;         //so nhan vien hien co
    static employee* arrap[MAXEM]; //mang cac con tro tro toi cac doi tuong
                                //employee
public:
    virtual void getdata()
    {
        cout<<"\n Nhap vao ten:"<<cin>>name;
        cout<<"\n Nhap vao ma so:"<<cin>>number;
    }
    virtual void putdata()
    {
        cout<<"\n Name="<<name;
        cout<<"\n Ma so="<<number;
    }
    virtual employee_type get_type(); //lay kieu
    static void add();                //them mot nhan vien
    static void display();            //hien thi tat ca nhan vien
    static void read();               //doc tu file dia
    static void write();              //ghi toi file dia
    static void destroy();            //xoa cac doi tuong khoi bo nho
};
//cac bien tinh
int employee::n=0;                   //so nhan vien hien co
employee* employee::arrap[MAXEM]; //mang cac con tro tro toi cac nhan vien
////////////////////////////////////
class manager:public employee        //lop nguoi quan ly
{
private:
    char title[LEN];                 //chuc vu:giam doc, pho giam doc...
public:
    void getdata() .
```

```

    {
        employee::getdata();
        cout<<"\n Chuc vu:";cin>>title;
    }
    void putdata()
    {
        employee::putdata();
        cout<<"\n Chuc vu="<<title;
    }
};
////////////////////////////////////
class scientist:public employee    //lop nha khoa hoc
{
    private:
        int pubs;                //so an ban
    public:
        void getdata()
        {
            employee::getdata();
            cout<<"\n So an ban da co:";cin>>pubs;
        }
    void putdata()
    {
        employee::putdata();
        cout<<"\n So an ban="<<pubs;
    }
};
////////////////////////////////////
class laborer:public employee    //lop cong nhan
{
};
////////////////////////////////////
//dinh nghia cac ham thanh vien
//tra ve kieu cua doi tuong nay
employee_type employee::get_type()
{
    if(typeid(*this)==typeid(manager))
        return tmanager;
    else if(typeid(*this)==typeid(scientist))
        return tscientist;
    else if(typeid(*this)==typeid(laborer))
        return tlaborer;
    else
        { cout<<"\nKieu employee sai";exit(1);}
}
//them mot nhan vien vao danh sach trong bo nho
void employee::add()
{
    cout<<"\n Go 'm' de them vao mot nguoi quan ly";
    cout<<"\n Go 's' de them vao mot nha khoa hoc";
    cout<<"\n Go 'l' de them vao mot cong nhan";
    cout<<"\n\n Lua chon:";
    switch(getch())    //tao kieu employee cu the
    {
        case 'm':arrap[n]=new manager; break;
        case 's':arrap[n]=new scientist; break;
    }
}

```

```

        case 'l':arrap[n]=new laborer; break;
        default:cout<<"\nKhong biet kieu nhan vien nay";return;
    }
    arrap[n++]>getData(); //lay du lieu tu nguoi su dung
}
//hien thi tat ca nhan vien
void employee::display()
{
    for(int j=0;j<n;j++)
    {
        cout<<"\n"<<(j+1);           //hien thi so thu tu
        switch(arrap[j]>get_type())    //hien thi kieu
        {
            case tmanager:cout<<"Kieu:Nguoi quan ly";break;
            case tscientist:cout<<"Kieu:Nha khoa hoc";break;
            case tlaborer:cout<<"Kieu:Cong nhan";break;
            default:cout<<"Khong biet kieu nay.";return;
        }
        arrap[j]>putData();
    }
}
//ghi tat ca cac doi tuong hien co ra file
void employee::write()
{
    int size;
    cout<<"Dang ghi..."<<n<<" nhan vien.";
    ofstream outfile;           //tao mot stream file de ghi
    employee_type etype;        //kieu cua doi tuong employee
                                //mo file o che do nhi phan
    outfile.open("EMPLOY.DAT",ios::trunc | ios::binary);
    if(!outfile)
    {cout<<"\nKhong mo duoc file";return;}
    for(int j=0;j<n;j++)
    {
        etype=arrap[j]>get_type();
        outfile.write((char*)&etype,sizeof(etype));//ghi kieu employee
        switch(etype)
        {
            case tmanager:size=sizeof(manager);break;
            case tscientist:size=sizeof(scientist);break;
            case tlaborer:size=sizeof(laborer);break;
        }
        outfile.write((char*)(arr[j]),size);
        if(!outfile)
        {cout<<"\nKhong ghi duoc ra file";return;}
    }
}
//doc du lieu cho tat ca cac nhan vien tu file dua vao bo nho
void employee::read()
{
    int size;           //kich thuoc cua mot doi tuong employee cu the
    employee_type etype;//kieu employee
    ifstream infile;    //tao mot stream file doc vao
    infile.open("EMPLOY.DAT",ios::binary);//mo o che do nhi phan

```

```

if(!infile)
{cout<<"\nKhong mo duoc file";return;}
n=0; //chua co nhan vien nao trong bo nho
while(1)
{
    infile.read((char*)&etype,sizeof(etype));
    if(infile.eof()) //thoat khoi vong lap khi gap EOF
        break;
    if(!infile)
    {cout<<"\nKhong doc duoc kieu nhan vien tu file";return;}
    switch(etype)
    {
        case tmanager:
            arrap[n]=new manager;
            size=sizeof(manager);
            break;
        case tscientist:
            arrap[n]=new scientist;
            size=sizeof(scientist);
            break;
        case tlaborer:
            arrap[n]=new laborer;
            size=sizeof(laborer);
            break;
        default:cout<<"\nKhong biet kieu trong file";return;
    }
    //doc du lieu tu file vao no
    infile.read((char*)arrap[n],size);
    if(!infile)
    {cout<<"\nKhong doc duoc du lieu tu file";return;}
    n++;
} //ket thuc vong lap while
cout<<"\nDa doc duoc "<<n<<" nhan vien";
}
//xoa bo nho duoc cap phat cho cac nhan vien
void employee::destroy()
{
    for(int j=0;j<n;j++)
        delete arrap[j];
}
////////////////////////////////////
void main()
{
    while(1)
    {
        cout<<"\n Go 'a' -- Them du lieu cho mot nhan vien"
        <<"\n Go 'd' -- Hien thi du lieu cho tat ca nhan vien"
        <<"\n Go 'w' -- Ghi tat ca nhan vien ra file"
        <<"\n Go 'r' -- Doc tat ca nhan vien tu file"
        <<"\n Go 'x' -- Thoat khoi chuong trinh"
        <<"\n\n Lua chon:";
        switch(getch())
        {
            case 'a':
                employee::add();

```

```

        break;
        case 'd':
            employee::display();
        break;
        case 'w':
            employee::write();
        break;
        case 'r':
            employee::read();
        break;
        case 'x':
            employee::destroy();
        break;
        default:cout<<"\nKhong biet lenh nay";
    }
}
}

```

4. Mã hóa số cho kiểu đối tượng

Chúng ta đã biết cách tìm ra lớp của một đối tượng trong bộ nhớ, nhưng làm thế nào biết được lớp của đối tượng mà dữ liệu của nó phải đọc vào từ đĩa. Không có hàm nào giúp chúng ta làm việc này. Bởi vậy, khi ghi dữ liệu của một đối tượng ra đĩa, chúng ta cần ghi một mã số (biến **enum employee_type**) trực tiếp ra đĩa ngay trước dữ liệu của đối tượng. Sau đó, khi đọc một đối tượng trở lại từ file vào bộ nhớ, chúng ta đọc giá trị này và tạo một đối tượng mới của kiểu được chỉ ra. Cuối cùng copy dữ liệu từ file vào đối tượng mới này.

5.6. CÁC TOÁN TỬ CHỒNG << VÀ >>

Phần này chúng ta sẽ nói cách chồng các toán tử chèn (>>) và toán tử lấy ra (<<). Đây là một đặc điểm mạnh của C++. Nó cho phép ta vào ra với các kiểu dữ liệu được định nghĩa bởi người sử dụng giống như các kiểu dữ liệu cơ bản như **int** và **double**. Ví dụ, nếu có một đối tượng của lớp **bow1** là **b1** thì ta có thể đưa nó ra màn hình với lệnh :

```
cout<<b1;
```

y như thể nó là một kiểu dữ liệu cơ bản.

Chúng ta có thể chồng các toán tử chèn và lấy ra để chúng làm việc với màn hình và bàn phím (**cout** và **cin**). Chúng ta có thể chồng chúng để chúng làm việc với các file đĩa.

5.6.1. Chồng cho cout và cin

Bản 5-18 trình bày một ví dụ, ENGLIO, chồng toán tử chèn và lấy ra cho lớp **English** để chúng làm việc với **cout** và **cin**.

Listing 5-18 ENGLIO

```

//englio.cpp
//toan tu chong << va >>
#include<iostream.h>
class English
{
    private:
        int feet;
        float inches;

```



```

public:
    English()
        {feet=0;inches=0.0;}           //ham tao khong doi so
    English(int ft,float trong)         //ham tao hai doi so
        {feet=ft;inches=in;}
    friend istream& operator>>(istream& s,English& d);
    friend ostream& operator<<(ostream& s,const English& d);
};

istream& operator>>(istream& s,English& d)//lay khoang cach tu nguoi su dung
{
    cout<<"\nNhap vao feet:"<<s>>d.feet;           //su dung toan tu chong >>
    cout<<"\nNhap vao inches:"<<s>>d.inches;
    return s;
}
//hien thi khoang cach
ostream& operator<<(ostream& s,const English& d)
{
    s<<d.feet<<"'-"<<d.inches<<"'"; //su dung toan tu chong <<
    return s;
}
////////////////////////////////////
void main()
{
    English dist1,dist2;           //dinh nghĩa các đối tượng English
    cout<<"\nNhap vao hai gia tri English:";
    cin>>dist1>>dist2;           //lay du lieu tu nguoi su dung
    English dist3(11,6.25);       //dinh nghĩa và khởi tạo dist3
    cout<<"\ndist1="<<dist1<<"\ndist2="<<dist2;
    cout<<"\ndist3="<<dist3;
}

```

Chương trình này yêu cầu người sử dụng nhập vào hai giá trị khoảng cách và sau đó hiển thị các giá trị này và một giá trị khác (11'-6.25") được khởi tạo trong chương trình. Đây là vài mẫu tương tác:

```

Nhap vao hai gia tri English:
Nhap vao feet:4
Nhap vao inches:2.25
Nhap vao feet:6
Nhap vao inches:6.75
dist1=4'-2.25"
dist2=6'-6.75"
dist3=11'-6.25"

```

Chú ý rằng trong hàm **main()** các đối tượng **English** được đối xử thuận tiện và tự nhiên, giống như bất kỳ kiểu nào khác, sử dụng lệnh chẳng hạn như:

```
cin>>dist1>>dist2;
```

và

```
cout<<"\ndist1="<<dist1<<"\ndist2="<<dist2;
```

Các hàm **operator<<()** và **operator>>()** phải là bạn của lớp **English**, bởi vì các đối tượng **istream** và **ostream** xuất hiện bên trái toán tử (xem các hàm bạn ở mục 4.6). Chúng trả về, theo

tham chiếu, một đối tượng của **istream** (cho >>) hoặc **ostream** (cho <<). Giá trị trả về này cho phép móc nối để nhiều giá trị có thể nhập vào hoặc đưa ra trong một câu lệnh.

Các toán tử có hai đối số, cả hai được truyền theo tham chiếu. Đối số thứ nhất là một đối tượng của **istream** (cho >>, thường là **cin**) hoặc **ostream** (cho <<, thường là **cout**). Đối số thứ hai là đối tượng được hiển thị, trong ví dụ này là đối tượng **English**. Trên thực tế, đối tượng **stream** và **English** được truyền theo tham chiếu cho phép hàm thay đổi chúng. Toán tử >> lấy kết quả nhập vào từ **stream** được chỉ rõ trong đối số thứ nhất và copy nó vào dữ liệu thành viên của đối tượng được chỉ rõ trong đối số thứ hai. Toán tử << copy dữ liệu từ đối tượng được chỉ rõ trong đối số thứ hai và gửi nó vào **stream** được chỉ rõ trong đối số thứ nhất. Chúng ta có thể chống toán tử chèn và toán tử lấy ra cho các lớp khác theo cách này.

Chú ý rằng **const** đứng trước đối số thứ hai của toán tử <<. Nó đảm bảo rằng việc đưa ra một giá trị **English** không làm thay đổi nó.

5.6.2. Chống cho các file

Ví dụ tiếp theo cho thấy có thể chống các toán tử << và >> trong lớp **English** để chúng làm việc với cả vào/ra file và **cout, cin**. Bản 5-19 trình bày chương trình ENGLIO2.

Listing 5-19 ENGLIO2

```
//englio2.cpp
//toan tu chong << va >> lam viec voi file
#include<fstream.h>
class English
{
private:
    int feet;
    float inches;
public:
    English()
        {feet=0;inches=0.0;} //ham tao khong doi so
    English(int ft,float in) //ham tao hai doi so
        {feet=ft;inches=in;}
    friend istream& operator>>(istream& s,English& d);
    friend ostream& operator<<(ostream& s,const English& d);
};

//lay khoang cach tu file hoac tu ban phim
istream& operator>>(istream& s,English& d)
{
    char dummy; //cho ', -, va "
    s>>d.feet>>dummy>>dummy>>d.inches>>dummy; //chong toan tu >>
    return s;
}

//gui toi file hoac man hinh voi toan tu chong <<
ostream& operator<<(ostream& s,const English& d)
{
    s<<d.feet<<"\n"<<d.inches<<"\n"; //su dung toan tu chong <<
    return s;
}
////////////////////////////////////
void main()
{
    char ch;
```

```

English dist1;
ofstream ofile;           //tao va mo mot file de dua ra
ofile.open("DISK.DAT");
do
{
    cout<<"\nNhap vao khoang cach Anh:";
    cin>>dist1;           //lay khoang cach tu nguoi su dung
    ofile<<dist1;         //ghi no ra file
    cout<<"\nCo tiep tục nua không(c/k)?";
    cin>>ch;
}
while(ch!='k');
ofile.close();           //dong file lai
ifstream ifile;          //tao va mo mot stream file vao
ifile.open("DISK.DAT");
cout<<"\nNội dung của file đĩa là:";
while(1)
{
    ifile>>dist1;
    if(ifile.eof()) break;
    cout<<"\nKhoang cach = "<<dist1; //hien thi khoang cach
}
}

```

Các toán tử chồng thay đổi rất ít so với chương trình ENGLIO. Toán tử >> không còn thông báo để nhập vào bởi vì không có ý nghĩa khi thông báo cho một file. Khi nhận vào từ bàn phím, chúng ta cho rằng người sử dụng biết cách nhập vào giá trị **feet** và **inches**, bao gồm cả dấu phẩy trên, dấu gạch ngang và dấu nháy kép, chẳng hạn nhập vào 19'-2.5". Toán tử << không thay đổi. Chương trình yêu cầu người sử dụng nhập vào, ghi mỗi giá trị **English** ra file như nó có. Khi người sử dụng kết thúc việc nhập, chương trình đọc và hiển thị tất cả các giá trị từ file. Đây là một vài mẫu tương tác với chương trình:

```

Nhap vao khoang cach Anh:2'-4.5"
Co tiep tục nua không(c/k)?c
Nhap vao khoang cach Anh:6'-2.65"
Co tiep tục nua không(c/k)?c
Nhap vao khoang cach Anh:5'-0.5"
Co tiep tục nua không(c/k)?k
Nội dung của file là:
Khoang cach = 2'-4.5"
Khoang cach = 6'-2.65"
Khoang cach = 5'-0.5"

```

Nếu người sử dụng bị lỗi nhập vào khoảng cách với dấu câu không đúng thì khoảng cách sẽ không được ghi ra file và file sẽ không được đọc bằng toán tử <<. Trong chương trình thực tế, việc kiểm tra lỗi là cần thiết.

5.6.3. Chồng cho vào/ra nhị phân

Từ trước đến giờ chúng ta mới nói đến việc chồng các toán tử << và >> cho vào/ra định dạng. Chúng có thể được chồng cho vào ra nhị phân không? Đây có thể là một cách có hiệu quả để lưu trữ thông tin, đặc biệt nếu đối tượng chứa nhiều dữ liệu số.

Để có ví dụ, chúng ta lưu các đối tượng **person** ra các file đĩa ở dạng nhị phân (xem ví dụ **VIRTPERS** ở mục 4.2). Dạng nhị phân có nghĩa là dữ liệu trên đĩa sẽ có dạng giống như trong bộ

nhớ. Dữ liệu này bao gồm một trường tên 40 ký tự và một trường tuổi là một số nguyên. Các hàm thành viên của chúng được dùng để lấy dữ liệu này từ bàn phím và hiển thị chúng. Các toán tử << và >> được chồng để ghi dữ liệu tới một file đĩa và đọc nó trở lại, sử dụng **read()** và **write()**. Bản 5-20 trình bày chương trình này PERSIO.

Listing 5-20 PERSIO

```
//persio.cpp
//chong toan tu >> va << cho cac doi tuong
//luu tru du lieu nhi phan trong file
#include<fstream.h>

class person
{
protected:
    char name[40];          //ten nguoi
    int age;                //tuoi
public:
    void getData()          //lay du lieu
    {
        cout<<"\n Nhap vao ten:"; cin>>name;
        cout<<" Nhap vao tuoi:"; cin>>age;
    }
    void showData()         //hien thi du lieu
    {
        cout<<"\n Ten:"<<name;
        cout<<"\n Tuoi:"<<age;
    }
    void persin(istream& s)  //doc vao tu file
    {
        s.read((char*)this,sizeof(*this));
    }
    void persout(ostream& s) //doc ra khoi file
    {
        s.write((char*)this,sizeof(*this));
    }
    friend istream& operator>>(istream& s,person& p);
    friend ostream& operator<<(ostream& s,person& p);
};
//dinh nghĩa các hàm bạn
istream& operator>>(istream& s,person& p)
{
    p.persin(s);
    return s;
}

ostream& operator<<(ostream& s,person& p)
{
    p.persout(s);
    return s;
}
//-----
void main()
{
    person pers1,pers2,pers3,pers4;//tao 4 doi tuong person rong
```

```

        cout<<"\nNguoi thu 1:";    //lay du lieu cho nguoi thu nhat
        pers1.getData();
        cout<<"\nNguoi thu 2:";    //lay du lieu cho nguoi thu hai
        pers2.getData();

                                //tao mot file de dua ra
        ofstream outfile("PERSON.DAT",ios::binary);

        outfile<<pers1<<pers2;    //ghi ra file
        outfile.close();           //dong lai
        ifstream infile("PERSON.DAT",ios::binary); //tao mot stream file vao
        infile>>pers3>>pers4;     //doc tu file vao pers3 va pers4

        cout<<"\nNguoi thu 3:";    //hien thi cac doi tuong moi
        pers3.showData();
        cout<<"\nNguoi thu 4:";
        pers4.showData();
    }

```

Chúng ta đã phải vượt qua một rắc rối trong các toán tử chồng >> và <<. Các toán tử này phải là các hàm bạn bởi vì một đối tượng **stream** được dùng làm đối số bên trái. Tuy nhiên, bởi vì chúng ta sử dụng dạng nhị phân nên truy nhập trực tiếp dữ liệu của đối tượng **person** trong bộ nhớ là thuận tiện nhất, nó được quản lý với con trỏ **this**. Nhưng con trỏ **this** không thể dùng được trong các hàm bạn. Bởi vậy, từ trong các toán tử chồng chúng ta phải gọi các hàm thành viên của **person** là **persin()** và **persout()** để thực hiện việc đọc và ghi.

Trong hàm **main()**, chương trình lấy dữ liệu cho hai đối tượng **person** từ người sử dụng, ghi ra đĩa, đọc trở lại vào hai đối tượng khác và hiển thị nội dung hai đối tượng mới này. Sau đây là vài mẫu tương tác:

```

Nguoi thu 1:
  Nhap vao ten: Binh
  Nhap vao tuoi: 23
Nguoi thu 2:
  Nhap vao ten: Thang
  Nhap vao tuoi: 24
Nguoi thu 3
  Ten:Binh
  Tuoi:23
Nguoi thu 4
  Ten:Thang
  Tuoi:24

```

Chúng ta thấy sự móc nối các toán tử cho phép một lệnh trong hàm **main()** ghi hai đối tượng **person** ra đĩa:

```
outfile<<pers1<<pers2;
```

và một lệnh đọc dữ liệu từ file vào hai đối tượng **person**:

```
infile>>pers3>>pers4;
```

5.7. BỘ NHỚ NHƯ MỘT ĐỐI TƯỢNG STREAM

Chúng ta có thể đối xử với một phần của bộ nhớ như là một đối tượng **stream**, chèn dữ liệu vào nó y như chèn vào một file. Đây được gọi là định dạng bộ nhớ trong. Nó rất hữu ích trong nhiều

tình huống, ví dụ, với các hàm trong môi trường GUI (chẳng hạn như Window) mà ở đó việc ghi dữ liệu tới một hộp hội thoại hay một cửa sổ yêu cầu dữ liệu phải là một chuỗi, cho dù là số. Đưa chuỗi này trong một phân bộ nhớ, sử dụng các **iostream** với vào/ra định dạng, sau đó gọi các hàm GUI với đối số là chuỗi đó. Tất nhiên còn có nhiều tình huống mà ở đó định dạng bộ nhớ trong rất thuận tiện.

5.7.1. Kích thước bộ đệm cố định

Có một họ lớp **stream** cài đặt cho định dạng bộ nhớ trong như vậy. Để đưa ra bộ nhớ, có **ostream**, nó được rút ra từ lớp **ostream**. Để đọc vào từ bộ nhớ có **istream**, nó được rút ra từ lớp **istream** và để đọc ghi các đối tượng bộ nhớ có **stringstream**, nó được rút ra từ **istream**. Để sử dụng các đối tượng bộ nhớ chúng ta cần có file tiêu đề **STRSTREA.H**. Ví dụ sau cho thấy cách sử dụng cả các đối tượng **ostream** và **istream**. Bản 5-21 trình bày chương trình **STRSTR**.

Listing 5-21 STRSTR

```
//sstr.cpp
//sử dụng đối tượng bộ nhớ
#include<strstrea.h>

const int SIZE=80;
void main()
{
    char membuff[SIZE];           //bộ nhớ đệm
    ostream omem(membuff,SIZE); //tạo một đối tượng bộ nhớ để đưa ra

    int oj=77;
    double od=890.12;
    char ostr1[]="Thang";
    char ostr2[]="Binh";

    omem<<"oj= "<<oj<<endl
        <<"od= "<<od<<endl
        <<"ostr1= "<<ostr1<<endl
        <<"ostr2= "<<ostr2<<endl
        <<ends;           //kết thúc bộ đệm với '\0'
    cout<<membuff;        //hiển thị bộ đệm

    char dummy[2^];        //khái báo các biến mới để đọc vào từ bộ nhớ
    int ij;
    double id;
    char istr1[20];
    char istr2[20];

    istream imem(membuff,SIZE); //tạo một đối tượng bộ nhớ để đọc vào
    imem>>dummy>>ij>>dummy>>id>>dummy>>istr1>>dummy>>istr2;

    cout<<"\nij= "<<ij<<"\nid= "<<id
        <<"\nistr1= "<<istr1<<"\nistr2= "<<istr2;
}
```

1. Đối tượng ostream

Một cách sử dụng đối tượng **ostream** là bắt đầu với bộ đệm dữ liệu kiểu **char***. Sau đó chúng ta tạo một đối tượng **ostream**, dùng bộ nhớ đệm và kích thước của nó làm đối số cho hàm

tạo của đối tượng. Sau đó có thể gửi văn bản định dạng tới đối tượng **ostrstream**, dùng toán tử <<, y như gửi văn bản tới **cout** hoặc một file đĩa.

Khi chạy chương trình, **membuff** sẽ được điền đầy văn bản định dạng:

```
oj=77\nod=890.12\nostr1=Thang\nostr2=Binh\n\0
```

Chúng ta có thể định dạng văn bản bằng các tác tử, y như làm với **cout**. Tác tử **ends** chèn một ký tự '\0' vào cuối chuỗi (đừng quên tác tử này; chuỗi phải được kết thúc bằng ký tự '\0' mà nó lại không được tự động thêm vào). Chương trình bây giờ hiển thị nội dung của bộ đệm như một chuỗi kiểu **char**:

```
cout<<membuff;
```

Đây là kết quả đưa ra:

```
oj= 77
od= 890.12
ostr1= Thang
ostr2= Binh
```

Trong ví dụ này chương trình hiển thị nội dung của bộ đệm chỉ để cho thấy bộ đệm trông như thế nào. Thông thường chúng ta sẽ có một ứng dụng ưu việt hơn cho văn bản định dạng này, chẳng hạn như chuyển địa chỉ của bộ đệm tới các hàm GUI để hiển thị trong hộp hội thoại.

2. Dòng vào bộ nhớ (*input memory streams*)

Ứng dụng thông thường nhất cho định dạng bộ nhớ trong là lưu trữ văn bản trong bộ nhớ sử dụng một đối tượng **ostrstream**. Tuy nhiên, chúng ta cũng có thể đọc dữ liệu định dạng ra khỏi bộ nhớ và lưu nó trong các biến. Điều này cho phép ta chuyển nhiều giá trị ngay từ dạng ký tự số sang các giá trị số thay vì phải dùng các hàm chuyển đổi của riêng ngôn ngữ C như **atof()** và **atoi()**.

Phần thứ hai của chương trình **STRSTR** là tạo một đối tượng **istrstream**, **imem** và nối nó với cùng bộ đệm, **membuff**, mà được sử dụng cho đối tượng **ostrstream** trước đây. Sau đó chương trình đọc dữ liệu ra khỏi **imem** và đưa vào các biến thích hợp. Ở đây tự động chuyển đổi dữ liệu **char*** định dạng sang các biến kiểu **int**, **double** và **char**. Cuối cùng chương trình hiển thị giá trị của các biến này để thấy quá trình chuyển đổi cho kết quả đúng. Đây là phần thứ hai của kết quả đưa ra:

```
ij= 77
id= 890.12
istr1= Thang
istr2= Binh
```

Chú ý rằng, để quá trình chuyển đổi này làm việc, các biến trong **membuff** phải được giới hạn bởi các ký tự trắng. Đó là lý do tại sao chúng ta phải để các khoảng trắng sau dấu bằng khi ghi dữ liệu vào **omem** trong phần thứ nhất của chương trình.

Cũng cần lưu ý là khi đọc dữ liệu ra khỏi **imem** đưa vào các biến riêng biệt, để không có các văn bản trong các biến, chúng ta đọc các chuỗi ngắn này vào một bộ đệm gọi là **dummy** và sau đó bỏ chúng đi.

3. Phổ dụng (*universality*)

Để tập trung sự chú ý vào kỹ thuật định dạng bộ nhớ, trong ví dụ trên chúng ta chỉ trình bày một mình hàm **main()**. Nếu chúng ta viết một chương trình hướng đối tượng, có thể muốn đưa các chương trình để quản lý vào/ra bộ nhớ trong vào các hàm thành viên để các đối tượng có thể đọc ghi chính nó tới bộ nhớ.

Nếu quản lý mọi thứ tốt, chúng ta có thể sử dụng cùng một hàm thành viên để đọc và ghi đối tượng tới bộ nhớ cũng như đọc và ghi chúng tới file đĩa. Chồng toán tử << và >> để làm việc với các file đĩa. Sau đó để tên của đối tượng file (hoặc là **istream** hoặc **istrstream** hoặc **ostream** hoặc **ostrstream**) được truyền như đối số tới hàm, như trong chương trình REWOBJ.

4. Con trỏ file

Các con trỏ file, chẳng hạn các con trỏ file được quản lý bởi **seekg()** và **tellg()** có thể được dùng với các đối tượng bộ nhớ y như với các đối tượng file.

5.7.2. Kích thước bộ nhớ động

Có một loại thứ hai của đối tượng **ostrstream**. Thay vì phải gán đối tượng bộ nhớ tới một bộ đệm được định nghĩa bởi người dùng, như trong chương trình **STRSTR**, chúng ta có thể tạo một đối tượng **ostrstream** mà có thể tự cấp phát bộ nhớ cho riêng nó và điều chỉnh kích thước của nó một cách động để giữ bất cứ cái gì chúng ta đặt vào nó. Để làm được điều này, chúng ta chỉ cần định nghĩa đối tượng không có đối số. Một khi nó được định nghĩa, được điền dữ liệu vào, thì nó sẽ mở rộng để giữ bất cứ gì đặt vào nó. Nếu tiếp tục thêm dữ liệu vào, đối tượng sẽ tiếp tục mở rộng. Bản 5-22 trình bày chương trình **AUTOSTR**.

Listing 5-22 AUTOSTR

```
//autostr.cpp
//ghi du lieu dinh dang vao bo nho
#include<strstream.h>          //cho lop ostrstream
void main()
{
    ostrstream omem;          //tao mot doi tuong stream dong

    omem<<"j="<<111<<endl //chen cac so vao ostrstream
        <<"k="<<2.3456<<endl;
    omem<<ends;              //ket thuc voi '\0'

    char* p=omem.str();       //lay dia chi cua bo dem
    cout<<p;                  //hien thi no
}
```

Trong chương trình chúng ta tạo một đối tượng **ostrstream** (chú ý là không có đối số) và đặt dữ liệu vào nó: số nguyên 111 và số dấu phẩy động 2.3456. Để hiển thị nội dung của nó, chúng ta sử dụng hàm thành viên **str()**, nó trả về một con trỏ kiểu **char*** trỏ tới vùng nhớ chứa dữ liệu:

```
char* p=omem.str();
```

Sau đó chúng ta có thể sử dụng con trỏ này gửi tới một hàm khác để hiển thị dữ liệu và truy nhập nó cho các mục đích khác. Trong chương trình, chúng ta chèn con trỏ tới **cout**:

```
cout<<p;
```

Kết quả đưa ra như sau:

```
j= 111
k= 2.3456
```

Diễn hình, giá trị con trỏ được gửi tới một hàm GUI để hiển thị dữ liệu. Cách sử dụng **ostrstream** này làm việc tốt, nó giảm bớt công việc cài đặt bộ đệm và sự lo lắng về độ lớn của nó.

Nếu sử dụng cách chỉ ra ở trên, đặt dữ liệu vào đối tượng dùng một lệnh, sau đó lấy giá trị của con trỏ tới dữ liệu và không bao giờ thay đổi nội dung hoặc truy nhập lại dữ liệu thì mọi thứ làm việc như mong đợi.

Tuy nhiên, đi vượt ra ngoài cách này nhiều sẽ nhanh chóng rơi vào rắc rối. Tại sao vậy? Nhớ rằng các đối tượng **ostream** động không trông mong vào bộ đệm được cung cấp bởi người sử dụng, chúng phải tự động cấp phát bộ nhớ để lưu trữ dữ liệu của chúng. Nếu đặt thêm dữ liệu vào, chúng phải đi tìm vùng nhớ phù hợp với số lượng dữ liệu của chúng. Nếu cứ tiếp tục thêm dữ liệu nữa thì chúng có thể phải chuyển dữ liệu tới một nơi khác trong bộ nhớ để kích thước vùng nhớ của chúng tăng lên. Nhưng nếu đã tìm ra nơi đặt dữ liệu bằng `str()` thì đối tượng sẽ không di chuyển dữ liệu của nó được nữa. Ngoài ra, chúng ta phải có trách nhiệm hủy bỏ vùng nhớ của nó bằng `delete`.

CHƯƠNG 6

CÁC BẢN MẪU

Các bản mẫu (**templates**) là cơ chế bên trong cho các lớp côngtenơ hiện đại, chẳng hạn như thư viện mẫu chuẩn (STL) mà chúng ta sẽ nói trong phần cấu trúc dữ liệu. Bởi vậy trong chương này chúng ta sẽ tìm hiểu cách tạo và sử dụng các bản mẫu (cả mẫu hàm và lớp). Trong chương này chúng ta cũng tìm hiểu về lớp **string** chuẩn, đó là một lớp quản lý các mảng ký tự rất hữu hiệu, thuận lợi hơn chuỗi ký tự **char*** của C rất nhiều. Chúng ta cũng nói về cách tạo các chương trình nhiều file.

6.1. MẪU HÀM

Giả sử chúng ta muốn viết một hàm (**function templates**) trả về giá trị tuyệt đối của hai số. Thông thường hàm này được viết cho một kiểu dữ liệu cụ thể:

```
int abs(int n)           //giá trị tuyệt đối của các số nguyên
{ return (n < 0)? -n : n; } //nếu n âm trả về -n
```

Ở đây hàm được định nghĩa có một đối số kiểu **int** và trả về một giá trị cùng kiểu. Nhưng bây giờ nếu chúng ta muốn viết một hàm tìm giá trị tuyệt đối của một giá trị kiểu **long**, chúng ta phải viết một hàm hoàn toàn mới:

```
long abs(long n)         //giá trị tuyệt đối của các số long
{ return (n < 0)? -n : n; }
```

Và, để cho kiểu **float**:

```
float abs(float n)       //giá trị tuyệt đối của các số float
{ return (n < 0)? -n : n; }
```

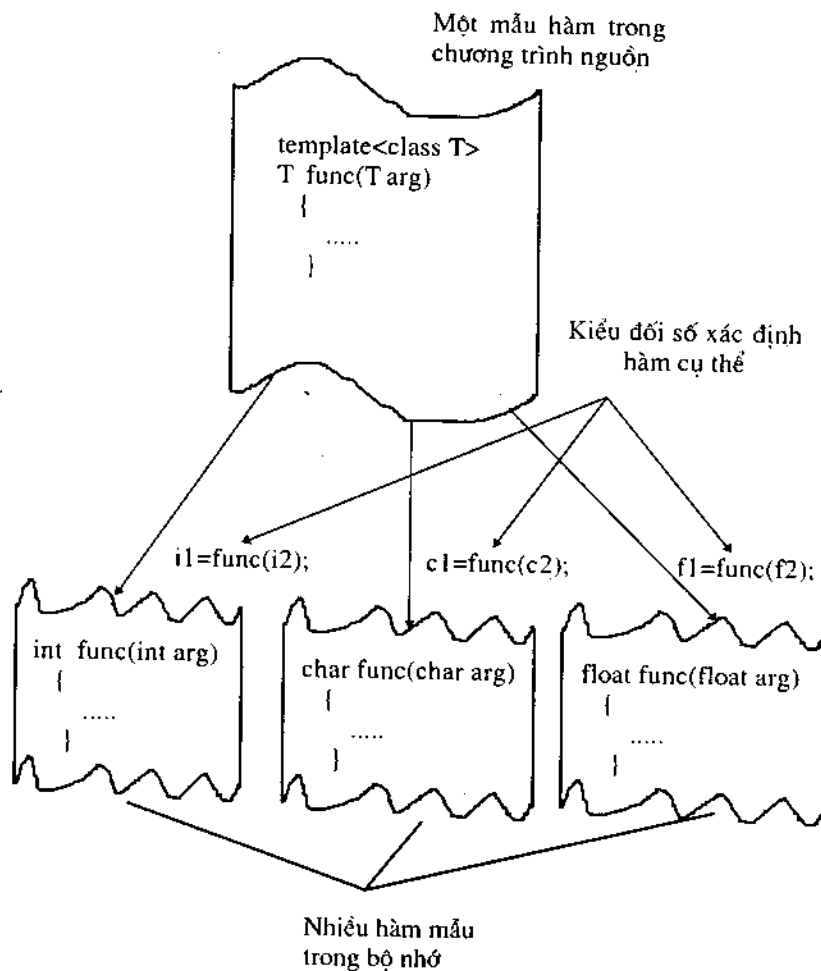
Các trường hợp đều có thân hàm giống nhau, nhưng phải tách thành các hàm riêng biệt bởi vì chúng quản lý các biến thuộc các kiểu dữ liệu khác nhau. C++ cho phép chồng các hàm này để có cùng tên, cho dù vậy chúng ta vẫn phải viết các định nghĩa hàm cho mỗi trường hợp. Trong ngôn ngữ C, không trợ giúp việc chồng hàm nên các hàm cho các kiểu khác nhau không thể có cùng tên. Chính vì vậy, trong thư viện hàm C có những họ hàm có tên tương tự nhau, chẳng hạn **abs()**, **fabs()**, **labs()**, **cabs()**...

Viết đi viết lại cùng một thân hàm cho các kiểu dữ liệu khác nhau lãng phí thời gian cũng như khoảng trống trong chương trình. Ngoài ra, nếu tìm thấy một lỗi trong một hàm như vậy thì chúng ta phải nhớ sửa nó trong từng thân hàm.

Sẽ tốt hơn nếu có một cách để viết một thân hàm như vậy chỉ một lần và bảo nó làm việc cho nhiều kiểu dữ liệu khác nhau. Đây là tất cả những gì mà mẫu hàm làm giúp chúng ta. Ý tưởng đó được chỉ ra bằng sơ đồ trong hình 6-1.

6.1.1. Mẫu hàm đơn giản

Ví dụ đầu tiên cho thấy cách viết một hàm giá trị tuyệt đối như một mẫu hàm để nó làm việc với bất kỳ kiểu dữ liệu số cơ bản nào. Chương trình này định nghĩa một bản mẫu của **abs()** và sau đó, trong hàm **main()** gọi hàm này 6 lần với các kiểu dữ liệu khác nhau để minh họa rằng nó làm việc. Bản 6-1 trình bày chương trình **TEMPABS**.



Hình 6-1. Một mẫu hàm.

Listing 6-1 TEMPABS

```

//tempabs.cpp
//ham mau duoc dung cho ham gia tri tuyet doi
#include<iostream.h>
template <class T>                                //ham mau
T abs(T n)
{
    return (n<0)? -n : n;
}
void main()
{
    int int1=5;
    int int2=-6;
    long lon1=7000L;
    long lon2=-8000L;
    double dub1=9.95;
    double dub2=-10.15;

    cout<<"\nabs("<<int1<<")= "<<abs(int1);    //abs(int1)
    cout<<"\nabs("<<int2<<")= "<<abs(int2);    //abs(int2)

```

```

cout<<"\nabs("<<lon1<<")="<<abs(lon1);    //abs(lon1)
cout<<"\nabs("<<lon2<<")="<<abs(lon2);    //abs(lon2)
cout<<"\nabs("<<dub1<<")="<<abs(dub1);    //abs(dub1)
cout<<"\nabs("<<dub2<<")="<<abs(dub2);    //abs(dub2)
}

```

Đây là kết quả đưa ra của chương trình:

```

abs(5)= 5
abs(-6)= 6
abs(7000)= 7000
abs(-8000)= 8000
abs(9.95)= 9.95
abs(-10.15)= 10.15

```

Chúng ta có thể thấy, hàm **abs()** bây giờ làm việc với cả ba kiểu dữ liệu (**int**, **long** và **double**) mà chúng ta sử dụng làm đối số. Nó cũng làm việc với các kiểu dữ liệu cơ bản khác và thậm chí sẽ làm việc với các kiểu dữ liệu được định nghĩa bởi người sử dụng, với điều kiện toán tử **<** và dấu **-** được chống thích hợp.

Đây là cách xác định hàm **abs()** để nó làm việc với nhiều kiểu dữ liệu:

```

template<class T>
T abs(T n)
{
    return ( n < 0)? -n : n;
}

```

Đây là toàn bộ cú pháp một mẫu hàm (function template), dòng đầu tiên bắt đầu với từ khóa **template** và theo sau là định nghĩa hàm. Cách viết hàm **abs()** mới này cho sự linh hoạt đáng kinh ngạc như thế nào?

6.1.2. Cú pháp mẫu hàm

Điều đổi mới chính ở trong các mẫu hàm là biểu diễn kiểu dữ liệu được hàm sử dụng không như kiểu cụ thể chẳng hạn như **int**, mà bởi một tên có thể đại diện cho bất kỳ kiểu nào. Trong mẫu hàm ở trên, tên này là **T** (không có gì ghê gớm về tên này, có thể dùng bất cứ tên gì nếu muốn, như **Type**, hoặc **anyType** hoặc **FooBar**, ...). Từ khóa **template** báo cho trình biên dịch biết là chúng ta định nghĩa một mẫu hàm. Từ khóa **class**, ở trong hai dấu ngoặc nhọn **< >**, cũng có thể gọi là **Type** bởi vì chúng ta có thể định nghĩa các kiểu dữ liệu của riêng chúng ta nên thực sự không có gì phân biệt giữa các kiểu và các lớp. Biến theo sau từ khóa **class** (là **T** trong ví dụ này) được gọi là đối số mẫu (**template argument**).

Suốt định nghĩa hàm, bất kỳ chỗ nào một kiểu dữ liệu cụ thể, chẳng hạn như **int**, được viết như thông thường, chúng ta thay bằng đối số mẫu, **T**. Trong hàm **abs()**, tên này xuất hiện chỉ hai lần, cả hai lần đều ở dòng đầu tiên (khai báo hàm) là kiểu đối số và kiểu trả về. Trong các hàm phức tạp hơn, nó có thể xuất hiện nhiều lần trong chương trình.

6.1.3. Trình biên dịch làm gì ?

Trình biên dịch làm gì khi nó thấy từ khóa **template** và định nghĩa hàm theo sau đó? Nó không làm gì cả. Mẫu hàm tự nó không làm cho trình biên dịch phải tạo ra bất kỳ mã lệnh nào. Trình biên dịch không thể tạo ra mã bởi vì nó chưa biết hàm sẽ làm việc với kiểu dữ liệu nào. Nó chỉ đơn giản nhớ rằng mẫu hàm có thể sử dụng trong tương lai.

Việc tạo ra mã lệnh không xảy ra cho đến khi hàm thực sự được gọi bằng một lệnh trong chương trình. Trong chương trình TEMPABS, nó xảy ra trong biểu thức chẳng hạn như `abs(int1)` ở lệnh:

```
cout<<"\nabs("<<int1<<" )= "<<abs(int1);
```

Khi trình biên dịch thấy một lời gọi hàm như vậy, nó biết rằng kiểu sử dụng là `int`, bởi vì đó là kiểu của đối số `int1`. Bởi vậy nó tạo ra một phiên bản cụ thể của hàm `abs()` cho kiểu `int` và thay thế `int` bất kỳ đâu nó thấy tên `T` trong mẫu hàm. Đây gọi là cụ thể mẫu hàm (**instantiating the function template**) và mỗi phiên bản cụ thể của hàm được gọi là một hàm mẫu (nghĩa là, một hàm mẫu là một cụ thể rõ ràng của một mẫu hàm).

Trình biên dịch cũng tạo ra một lời gọi tới hàm mới được cụ thể và chèn nó vào chương trình nơi có hàm `abs(int1)`. Tương tự, biểu thức `abs(long)` làm cho trình biên dịch tạo ra một phiên bản của `abs()` làm việc với kiểu `long`, cũng như lời gọi tới hàm này, trái lại lời gọi `abs(double)` tạo ra một hàm làm việc trên kiểu `double`. Tất nhiên trình biên dịch đủ thông minh để tạo ra duy nhất một phiên bản `abs()` cho mỗi kiểu dữ liệu. Do đó, mặc dù có hai lời gọi tới phiên bản `int` của hàm nhưng chỉ có một mã lệnh thực hiện cho phiên bản này.

Chú ý rằng số lượng bộ nhớ RAM mà chương trình sử dụng là bằng nhau dù sử dụng dạng mẫu hay viết ba hàm riêng biệt. Những gì chúng ta tiết kiệm được là không phải gõ ba hàm riêng biệt vào file nguồn. Điều này làm cho chương trình ngắn hơn và dễ hiểu hơn. Ngoài ra nếu muốn thay đổi cách làm việc của hàm thì chúng ta chỉ cần thay đổi ở một nơi trong chương trình thay vì phải thay đổi ở ba nơi khi không sử dụng mẫu.

Trình biên dịch quyết định cách biên dịch hàm dựa trên toàn bộ kiểu dữ liệu được dùng trong đối số (hoặc các đối số) của lời gọi hàm. Kiểu trả về không tham gia vào quyết định này. Điều này tương tự như cách mà trình biên dịch quyết định gọi hàm nào trong số các hàm được chồng.

Một mẫu hàm không thực sự là một hàm, bởi vì nó không thực sự sinh ra mã chương trình để đặt vào trong bộ nhớ. Thay vào đó nó là một mẫu (**pattern**) hoặc một bản thiết kế, để tạo nhiều hàm. Điều này hoàn toàn phù hợp với triết lý của C++. Nó tương tự như cách mà một lớp không làm bất cứ cái gì cụ thể (chẳng hạn như tạo mã chương trình trong bộ nhớ), nhưng nó là một bản thiết kế để tạo nhiều đối tượng tương tự nhau.

6.1.4. Mẫu hàm có nhiều đối số

Chúng ta cùng xem một ví dụ khác về mẫu hàm. Mẫu hàm này có ba đối số: hai đối số mẫu và một đối số kiểu cơ bản. Mục đích của hàm này là tìm kiếm một giá trị cụ thể trong một mảng. Hàm trả về chỉ số mảng của giá trị mà nó tìm thấy, hoặc -1 nếu nó không tìm thấy. Các đối số là một con trỏ tới mảng, giá trị cần tìm và kích thước mảng. Trong hàm `main()`, chương trình định nghĩa bốn mảng thuộc các kiểu khác nhau và bốn giá trị để tìm kiếm (đối xử với kiểu `char` như một số). Sau đó nó gọi hàm mẫu một lần cho từng mảng. Bản 6-2 trình bày chương trình TEMPFIND.

Listing 6-2 TEMPFIND

```
//tempfind.cpp
//ham tra ve so chi so cua muc neu tim thay, neu khong tra ve -1
#include<iostream.h>
template <class atype>                                //ham mau
int find(atype* array, atype value, int size)
{
    for(int j=0; j<size; j++)
        if(array[j]==value) return j;
    return -1;
}
```

```

char chrArr[]={1,3,5,9,11,13};
char ch=5;
int intArr[]={1,3,5,9,11,13};
int in=6;
int lonArr[]={1L,3L,5L,9L,11L,13L};
int lo=11L;
int dubArr[]={1.0,3.0,5.0,9.0,11.0,13.0};
int db=4.0;

void main()
{
    cout<<"\n 5 trong chrArr: index = "<<find(chrArr,ch,6);
    cout<<"\n 6 trong intArr: index = "<<find(intArr,in,6);
    cout<<"\n 11 trong lonArr: index = "<<find(lonArr,lo,6);
    cout<<"\n 4 trong dubArr: index = "<<find(dubArr,db,6);
}

```

Ở đây chúng ta đặt tên đối số mẫu là **atype**. Nó xuất hiện trong hai đối số hàm: kiểu của một con trỏ trỏ tới mảng và kiểu của mục cần tìm kiếm. Đối số thứ ba là kích thước mảng, luôn luôn là kiểu **int**; nó không phải là đối số mẫu. Đây là kết quả đưa ra từ chương trình:

```

5 trong chrArr:   index = 2
6 trong intArr:   index = -1
11 trong lonArr:  index = 4
4 trong dubArr:   index = -1

```

Trình biên dịch tạo ra bốn phiên bản hàm, mỗi phiên bản cho một kiểu dùng để gọi nó. Nó tìm ra số 5 ở tại chỉ số 2 trong mảng ký tự, nó không tìm ra 6 trong mảng số nguyên v.v...

Có nhiều người lập trình đặt từ khóa **template** và phân khai báo hàm trên cùng một dòng:

```

template<class atype> int find(atype* array,
                               atype value,int size )
{
    //phân thân hàm
}

```

Tất nhiên trình biên dịch sẽ không phân nản gì với dạng này nhưng nó không rõ ràng bằng cách viết trên nhiều dòng.

1. Các đối số mẫu phải tương thích

Khi một hàm mẫu được gọi, tất cả cụ thể của cùng một đối số phải thuộc cùng một kiểu. Ví dụ, trong hàm **find()** ở chương trình trên nếu tên mảng là kiểu **int** thì giá trị cần tìm cũng phải thuộc kiểu **int**. Chúng ta không thể nói:

```

int intarray[]={1,3,5,7};    //mảng số nguyên
float f1=5.0;                //giá trị float
int value=find(intarray,f1,4); //không được

```

Bởi vì trình biên dịch mong đợi tất cả các cụ thể của **atype** phải cùng kiểu. Nó có thể tạo ra một hàm

```
find(int*,int,int);
```

nhưng không thể tạo ra:

```
find(int*,float,int);
```

bởi vì đối số thứ nhất và thứ hai phải cùng kiểu.

2. Hơn một đối số mẫu

Chúng ta có thể sử dụng hơn một đối số mẫu trong một mẫu hàm. Ví dụ, cho rằng chúng ta thích ý tưởng của mẫu hàm `find()` ở trên nhưng không chắc chắn một mảng được cung cấp lớn như thế nào. Nếu mảng quá lớn thì kiểu `long` sẽ cần cho kích thước mảng hơn là kiểu `int`. Mặt khác, chúng ta không muốn sử dụng kiểu `long` nếu không cần đến nó. Chúng ta muốn chọn kiểu của kích thước mảng, cũng như kiểu của đối tượng được lưu trữ, khi chúng ta gọi hàm. Để thực hiện được điều này, chúng ta có thể chuyển kích thước mảng thành một đối số mẫu, giả sử gọi nó là `btype`.

```
template<class atype,class btype>
btype find(atype* array,atype value,btype size)
{
    for(btype j=0;j<size;j++)
        if(array[j]==value) return j;
    return (btype) - 1;
}
```

Bây giờ chúng ta có thể sử dụng kiểu `int` hoặc kiểu `long` (hay thậm chí một kiểu được định nghĩa bởi người sử dụng) cho kích thước mảng. Trình biên dịch sẽ tạo ra các hàm khác nhau dựa trên không chỉ kiểu của mảng và giá trị cần tìm kiếm, mà còn dựa trên cả kiểu của kích thước mảng.

Chú ý rằng nhiều đối số mẫu có thể dẫn đến nhiều hàm được cụ thể từ một mẫu hàm. Với hai đối số như trên nếu có sáu kiểu dữ liệu cơ bản có thể dùng cho từng đối số thì cho phép tạo tới 36 hàm. Đây có thể mất nhiều bộ nhớ nếu các hàm là lớn. Mặt khác, chúng ta không thể cụ thể một phiên bản của hàm mẫu trừ khi chúng ta thực sự gọi nó.

6.1.5. Tại sao không dùng Macro?

Những người lập trình C trước kia có thể sẽ băn khoăn là tại sao không sử dụng `macro` để tạo các phiên bản khác nhau của một hàm cho các kiểu dữ liệu khác nhau. Ví dụ, hàm `abs()` có thể được định nghĩa như sau:

```
#define abs(n) ((n < 0)? (-n) : (n))
```

Định nghĩa này có tác dụng giống như mẫu hàm trong chương trình `TEMPABS` bởi vì nó thực hiện một sự thay thế văn bản đơn giản và do đó có thể làm việc với bất kỳ kiểu dữ liệu nào. Tuy nhiên, các `macro` ít được sử dụng trong C++. Có một vài vấn đề với chúng. Chúng không thực hiện bất kỳ một hành động kiểm tra nào, có thể có vài đối số cho `macro` mà đáng lẽ phải cùng kiểu, nhưng trình biên dịch sẽ không kiểm tra xem liệu chúng có cùng kiểu hay không. Ngoài ra, kiểu của giá trị trả về không xác định được, bởi vậy trình biên dịch không thể cảnh báo nếu chúng ta gán nó cho một biến không thích hợp. Trong bất kỳ trường hợp nào, các `macro` hạn chế các hàm mà có thể biểu diễn trong một lệnh. Cũng có những vấn đề khác (hơi tinh tế) với các `macro`. Nói chung, tốt nhất là nên tránh dùng chúng.

6.1.6. Điều cần biết khi mới làm việc

Làm sao biết được liệu có thể có cụ thể một hàm mẫu cho một kiểu dữ liệu cụ thể nào đó. Ví dụ, có thể sử dụng hàm `find()` trong chương trình `TEMPABS` để tìm kiếm một chuỗi `char*` trong một mảng chuỗi không? Để xem liệu có thể dùng được không, kiểm tra các toán tử được sử dụng trong hàm. Nếu tất cả chúng đều làm việc với kiểu dữ liệu đó thì chúng ta có thể sử dụng kiểu dữ liệu đó. Trong hàm `find()`, so sánh hai biến dùng toán tử bằng (`==`). Toán tử này không dùng được

với chuỗi **char***; phải sử dụng hàm thư viện **strcmp()**. Do đó, **find()** sẽ không làm việc với các chuỗi **char*** (tuy nhiên, nó sẽ làm việc với một lớp chuỗi được định nghĩa bởi người sử dụng, trong đó toán tử **==** được chồng).

6.1.7. Bắt đầu với một hàm thông thường

Khi viết một mẫu hàm, tốt hơn hết là bắt đầu với một hàm thông thường làm việc trên một kiểu dữ liệu cố định: **int** hoặc bất kỳ kiểu nào. Chúng ta có thể thiết kế và gỡ rối nó mà không cần phải lo lắng về cú pháp mẫu và nhiều kiểu. Sau khi mọi thứ đã làm việc tốt, chúng ta chuyển định nghĩa hàm sang một mẫu và kiểm tra nó làm việc với các kiểu dữ liệu thêm vào.

6.2. MẪU LỚP

Khái niệm mẫu có thể được áp dụng cho các lớp cũng như các hàm. Các mẫu lớp (class templates) thường được dùng cho các lớp lưu trữ dữ liệu (các **container**). Ngăn xếp mà chúng ta đã nói ở các chương trước là một ví dụ về lớp lưu trữ (chúng ta sẽ nói về các lớp lưu trữ dữ liệu trong phần II: Cấu trúc dữ liệu). Tuy nhiên, lớp **Stack** mà chúng ta đã nói đến chỉ có thể lưu trữ được duy nhất một kiểu dữ liệu cơ bản. Đây là một phiên bản rút gọn của lớp **Stack**:

```
class Stack
{
private:
    int st[MAX];    //mảng số nguyên
    int top;        //chỉ số đỉnh của ngăn xếp
public:
    Stack();        //hàm tạo
    void push(int var); //có đối số kiểu int
    int pop();      //trả về kiểu int
};
```

Lớp **Stack** này chỉ có thể lưu trữ dữ liệu kiểu **int**. Nếu bây giờ chúng ta muốn lưu trữ dữ liệu kiểu **float** thì chúng ta cần định nghĩa lại lớp hoàn toàn mới:

```
class LongStack
{
private:
    long st[MAX];    //mảng số nguyên
    int top;        //chỉ số đỉnh của ngăn xếp
public:
    LongStack();    //hàm tạo
    void push(long var); //có đối số kiểu long
    long pop();     //trả về kiểu long
};
```

Bởi vậy, nếu có thể viết một mô tả lớp mà có thể làm việc được với tất cả các kiểu dữ liệu cơ bản thì thật là tốt. Các mẫu lớp cho phép chúng ta làm điều này. Bản 6-3 là một chương trình mà ở đó áp dụng dạng mẫu cho lớp **Stack**. Chương trình này là **TEMPSTAK**.

Listing 6-3 TEMPSTAK

```
//tempstak.cpp
//cai dat lop Stack nhu mot mau
#include<iostream.h>
const int MAX =100;
template <class Type>
```



```

class Stack
{
private:
    Type st[MAX];           //ngăn xếp: mảng của bất kỳ kiểu nào
    int top;                //chỉ số của đỉnh ngăn xếp
public:
    Stack()                 //hàm tạo
    {top=-1;}
    void push(Type var)     //đặt vào ngăn xếp
    { st[++top]=var;}
    Type pop()
    {return st[top--];}
};

void main()
{
    Stack<float> s1;         //s1 là một đối tượng của lớp Stack<float>
    s1.push(1111.1);
    s1.push(2222.2);
    s1.push(3333.3);
    cout<<"1: "<<s1.pop()<<endl;
    cout<<"2: "<<s1.pop()<<endl;
    cout<<"3: "<<s1.pop()<<endl;

    Stack<long> s2;         //s2 là một đối tượng của lớp Stack<long>
    s2.push(123123123L);
    s2.push(234234234L);
    s2.push(345345345L);
    cout<<"1: "<<s2.pop()<<endl;
    cout<<"2: "<<s2.pop()<<endl;
    cout<<"3: "<<s2.pop()<<endl;
}

```

Ở đây lớp **Stack** được viết như một mẫu lớp. Cách cài đặt tương tự như một mẫu hàm. Từ khóa **template** báo hiệu rằng toàn bộ lớp sẽ là một mẫu:

```

template<class type>
class Stack
{
    //dữ liệu và các hàm thành viên sử dụng đối số mẫu "type"
};

```

Một đối số mẫu, trong ví dụ này tên là **type**, được sử dụng (thay cho một kiểu dữ liệu cố định) ở mọi nơi trong mô tả lớp mà có tham chiếu tới mảng **st[]**. Có ba nơi cần tên đối số mẫu này: định nghĩa **st**, kiểu đối số của hàm **push()** và kiểu trả về của hàm **pop()**.

Các mẫu lớp khác mẫu hàm ở cách chúng được cụ thể. Để tạo một hàm thực sự từ một mẫu hàm chỉ cần gọi nó sử dụng đối số có kiểu cụ thể. Tuy nhiên, các lớp được cụ thể bằng cách định nghĩa một đối tượng sử dụng đối số mẫu:

```
Stack<float> s1;
```

Câu lệnh này tạo một đối tượng, **s1** là một ngăn xếp lưu trữ các số kiểu **float**. Trình biên dịch cung cấp khoảng trống trong bộ nhớ cho dữ liệu của đối tượng này, sử dụng kiểu **float** ở bất kỳ đâu đối số mẫu **type** xuất hiện trong mô tả lớp. Nó cũng cung cấp bộ nhớ cho các hàm thành viên (nếu các hàm này chưa được đặt vào trong bộ nhớ bởi một đối tượng khác thuộc kiểu **Stack<float>**). Các

hàm thành viên này chỉ thao tác trên kiểu **float**. Hình 6-2 cho thấy cách mà một mẫu lớp và các định nghĩa các đối tượng cụ thể làm cho các đối tượng này được đặt trong bộ nhớ như thế nào.

Tạo một đối tượng **Stack** lưu trữ các đối tượng của một kiểu khác, như trong:

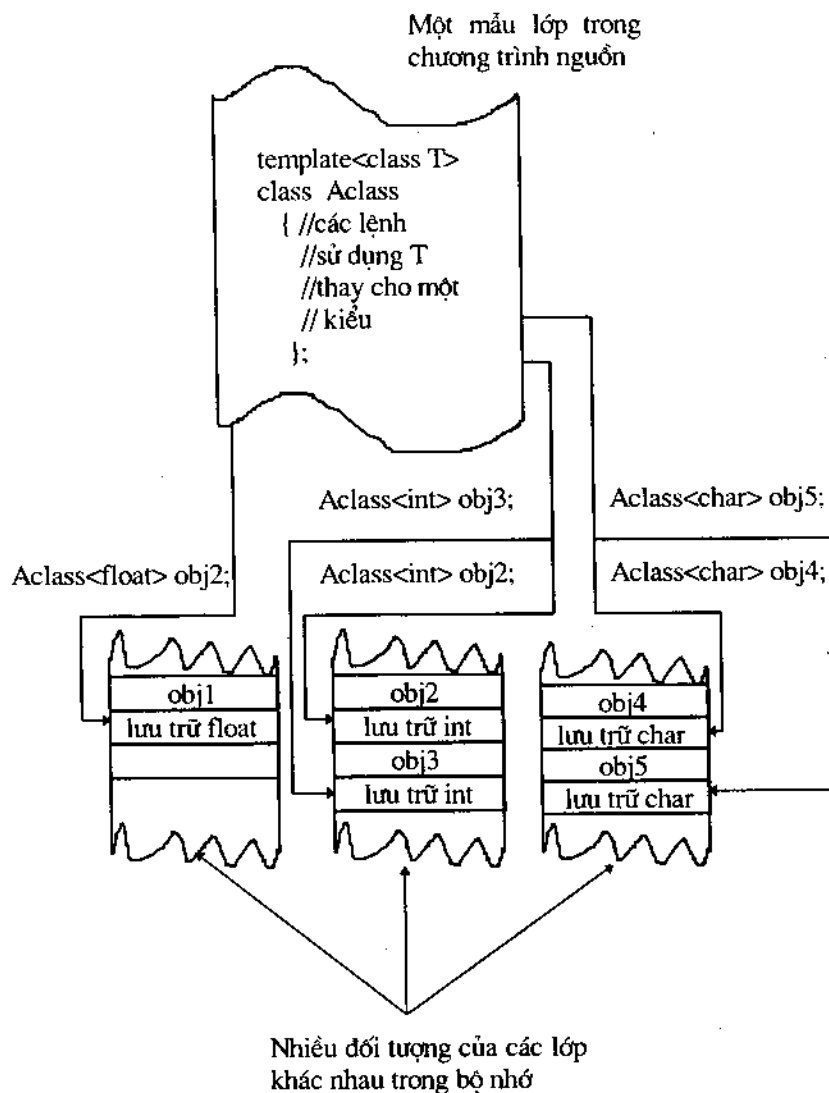
Stack<long> s2;

không chỉ tạo một vùng nhớ khác cho dữ liệu mà còn tạo một tập mới các hàm thành viên thao tác trên kiểu dữ liệu **long**.

Chú ý rằng tên kiểu của **s1** gồm có tên **Stack** cộng với đối số mẫu: **Stack<float>**. Điều này để phân biệt với các lớp khác có thể tạo cùng một mẫu lớp, chẳng hạn **Stack<int>** hay **Stack<long>**.

Chương trình của chúng ta sử dụng các ngăn xếp **s1** và **s2** bằng cách đặt vào nó và lấy ra ba giá trị trên từng ngăn xếp và hiển thị các giá trị lấy ra. Đây là kết quả đưa ra của chương trình:

1: 3333.3
2: 2222.2
3: 1111.1



Hình 6-2. Một mẫu lớp.

1: 345345345

2: 234234234

3: 123123123

Trong ví dụ này, mẫu lớp cho chúng ta hai lớp. Chúng ta có thể tạo các đối tượng lớp cho nhiều kiểu khác nhau chỉ với một dòng lệnh.

6.2.1. Tên lớp tùy thuộc vào ngữ cảnh

Trong ví dụ TEMPSTAK, các hàm thành viên đều được định nghĩa bên trong lớp. Nếu các hàm thành viên được định nghĩa bên ngoài lớp thì cần sử dụng cú pháp mới. Chương trình tiếp theo cho thấy cách làm này. Bản 6-4 là chương trình TEMPSTAK2.

Listing 6-4 TEMPSTAK2

```
//temstak2.cpp
//cai dat lop Stack nhu mot mau
//cac ham thanh vien dinh nghia ngoai lop
#include<iostream.h>
const int MAX =100;
template <class Type>
class Stack
{
private:
    Type st[MAX];           //ngan xep: mang cua bat ky kieu nao
    int top;                //chi so cua dinh ngan xep
public:
    Stack();                //ham tao
    void push(Type var);    //dat vao ngan xep
    Type pop();             //lay mot muc ra khoi ngan xep
};
template <class Type>       //ham tao
Stack<Type>::Stack()
{top=-1;}

template <class Type>       //ham tao
void Stack<Type>::push(Type var)
{st[++top]=var;}

template <class Type>       //ham tao
Type Stack<Type>::pop()
{return st[top--];}
void main()
{
    Stack<float> s1;         //s1 la mot doi cua lop Stack<float>
    s1.push(1111.1);
    s1.push(2222.2);
    s1.push(3333.3);
    cout<<"1: "<<s1.pop()<<endl;
    cout<<"2: "<<s1.pop()<<endl;
    cout<<"3: "<<s1.pop()<<endl;

    Stack<long> s2;          //s2 la mot doi cua lop Stack<long>
    s2.push(123123123L);
    s2.push(234234234L);
    s2.push(345345345L);
```

```

cout<<"1: "<<s2.pop()<<endl;
cout<<"2: "<<s2.pop()<<endl;
cout<<"3: "<<s2.pop()<<endl;
}

```

Biểu thức **template<class Type>** phải đứng trước không chỉ mô tả lớp mà còn đứng trước các hàm thành viên được định nghĩa bên ngoài. Đây là hàm thành viên **push()**:

```

template<class Type>
void Stack<Type>::push(Type var)
{ st[++top]=var; }

```

Tên **Stack<Type>** được sử dụng nhận ra lớp mà **push()** là thành viên. Trong các hàm thành viên không phải mẫu, chỉ cần một mình tên **Stack** là đủ:

```

void Stack::push(int var)
{ st[++top]=var; }

```

Tuy nhiên, đối với một mẫu hàm, chúng ta cần đối số mẫu: **Stack<Type>**.

Tên của lớp mẫu được biểu diễn khác nhau trong những ngữ cảnh khác nhau. Trong mô tả lớp, chỉ đơn giản là tên của nó **Stack**. Đối với các hàm thành viên được định nghĩa ngoài lớp phải có tên lớp cộng với tên đối số mẫu: **Stack<Type>**. Khi thực sự các đối tượng để lưu trữ một kiểu dữ liệu cụ thể lại là tên lớp cộng với kiểu cụ thể này: **Stack<float>** (hay bất cứ kiểu nào).

```

class Stack          //mô tả lớp
{
};
void Stack<Type>::push(Type var)    //định nghĩa push()
{
}
Stack<float> s1;      //định nghĩa đối tượng của Stack<float>

```

Chúng ta phải xem xét cẩn thận để sử dụng đúng tên đúng ngữ cảnh. Rất dễ quên thêm **<Type>** hoặc **<float>** vào tên lớp.

Mặc dù nó không được minh họa trong ví dụ này, cú pháp khi một hàm thành viên trả về một giá trị thuộc lớp của nó có thể tạo ra một vài vấn đề. Cho rằng chúng ta định nghĩa một lớp **Int** để tăng khả năng cho kiểu **int**, chẳng hạn như kiểm tra tràn. Thêm nữa, cho rằng chúng ta tạo một mẫu lớp để nó có thể dùng cho kiểu **int** hoặc **long**. Nếu định nghĩa hàm thành viên **xfunc()** của lớp này trả về kiểu **Int** thì chúng ta cần sử dụng **Int<Type>** cho kiểu trả về cũng như đứng trước toán tử quy định phạm vi.

```

Int<Type> Int<Type>::xfunc(Int arg)
{ }

```

Trái lại tên lớp được dùng như một kiểu của đối số hàm, không cần có **<Type>**.

6.2.2. Lưu trữ các kiểu dữ liệu được định nghĩa bởi người sử dụng

Trong các chương trình từ trước đến giờ chúng ta chỉ sử dụng các lớp mẫu để lưu trữ các kiểu dữ liệu cơ bản. Có thể lưu trữ các đối tượng thuộc các kiểu được định nghĩa bởi người sử dụng không? Câu trả lời là có. Ví dụ về cách này chúng ta sẽ nói ở phần II: Cấu trúc dữ liệu.

6.3. LỚP STRING CHUẨN

Bản thảo mới nhất của Ủy ban ANSI/ISO là lớp **string** chuẩn. Lớp này cho phép mảng ký tự được đối xử như một kiểu mới.

Lớp **string** chuẩn, được gọi là **basic_string**, được mẫu hóa và có thể được cụ thể không chỉ với kiểu **char**, mà còn với kiểu gọi là "ký tự mở rộng" được dùng trong bảng chữ cái của các nước khác nhau hoặc thậm chí với các kiểu được định nghĩa bởi người sử dụng. Chúng ta sẽ bỏ qua các khả năng này, chỉ làm việc với các chuỗi thuộc kiểu **char**. Trường hợp này lớp được định nghĩa kiểu có tên là **string**.

Các ví dụ trong phần này cho thấy một vài hoạt động với lớp **string**. Chúng ta không nói tất cả các hàm thành viên của lớp **string** mà chỉ nói vài hàm đại diện, các hàm còn lại có thể tham khảo trong các thư viện chương trình của trình biên dịch.

6.3.1. File tiêu đề

Các hãng sản xuất cài đặt lớp **string** chuẩn theo nhiều cách hơi khác nhau. Các ví dụ trong phần này làm việc với cài đặt của Borland (cụ thể là trình biên dịch Borland C++ 5.0). File tiêu đề cài đặt lớp **string** này là **CSTRING.H**. Với các trình biên dịch khác, tên file tiêu đề có thể khác. Các file này có thể là **BSTRING.H** hoặc **STRING** (không có **.H**) và một vài tên khác. Chúng ta nên thay thế các file tiêu đề cho thích hợp với từng trình biên dịch cụ thể.

6.3.2. Hàm tạo và các toán tử

Lớp **string** có nhiều hàm tạo cho phép chúng ta có thể tạo các đối tượng theo nhiều cách khác nhau. Chúng có thể được khởi tạo tới chiều dài bằng 0 (hàm tạo mặc định), tới các đối tượng **string** khác, tới các chuỗi **char*** (chuỗi C thông thường), tới một dãy liên tiếp các ký tự ở bất kỳ vị trí nào trong đối tượng **string** hoặc một chuỗi **char***, tới các ký tự riêng và tới một dãy liên tiếp các ký tự.

Ví dụ đầu tiên cho thấy vài khả năng này. Nó cũng minh họa hàm thành viên **length()** và các toán tử **<<**, **=**, **+**. Bản 6-5 trình bày chương trình **STRING1**.

Listing 6-5 STRING1

```
//string1.cpp
#include<iostream>
#include<cstring.h>
void main()
{
    string s1("IN HOC SIGNO VINCES");    //chuoi C
    string s2('-',19);                  //lap lai ky tu
    string s3(s1,7);                     //khởi tạo s3 từ s1 bắt đầu tại vị trí 7 của s1
    string s4(s1,13,16);                 //16 ký tự bắt đầu tại vị trí 13

    cout<<"s1= "<<s1<<endl;
    cout<<"s2= "<<s2<<endl;
    cout<<"s3= "<<s3<<endl;
    cout<<"s4= "<<s4<<endl<<endl;

    string s5;                          //ham tạo mặc định
    cout<<"Truoc lenh gan, chieu dai s5= "<<s5.length()<<endl;
    s5=s1;
    cout<<"Sau lenh gan, chieu dai s5= "<<s5.length()<<endl;

    string s6('-',47);                  //chuoi có kích thước cố định
    cout<<"chieu dai cua s6= "<<s6.length()<<endl;
    s6[0]='x';                          //su dụng ký hiệu mảng
    s6[46]='x';
    cout<<"s6= ";
```

```

for(int j=0;j<47;j++)
    cout<<s6[j];          //ky hieu mang
cout<<endl;
string s7=s1 + "(Motto of the Roman Empire)";
cout<<"s7= "<<s7<<endl<<endl;
cout<<"s1[13]= "<<s1[13]<<endl;
}

```

Các toán tử << và >> được chõng cho vào/ra. Hàm thành viên **length()** trả về số ký tự có trong đối tượng hiện tại ("hiện tại" có nghĩa là đối tượng gọi hàm thành viên đó). Toán tử chõng [] cho phép chúng ta truy nhập một ký tự riêng lẻ trong một đối tượng **string** nếu biết vị trí của nó. Toán tử = làm việc như mong đợi, gán một đối tượng **string** tới một đối tượng **string** khác và toán tử cộng nối hai đối tượng **string** lại. Đây là kết quả đưa ra từ chương trình **STRING1**:

```

s1= IN HOC SIGNO VINCES
s2= -----
s3= SIGNO VINCES
s4= VINCES

```

Truoc lenh gan, chieu dai s5= 0

Sau lenh gan, chieu dai s5= 19

chieu dai cua s6= 47

s6= x-----x

s7= IN HOC SIGNO VINCES(Motto of the Roman Empire)

s1[13]= V

Dòng lệnh cuối cùng của chương trình cho thấy toán tử [] được chõng cho phép truy nhập tới từng ký tự trong chuỗi sử dụng cú pháp y như với các chuỗi **char***. Toán tử [] cũng có thể được dùng ở bên trái dấu bằng để gán các giá trị tới các ký tự riêng biệt.

6.3.3. Các hàm thành viên

Các hàm thành viên khác nhau cho phép thao tác trên các đối tượng **string** thuận lợi hơn. Trong ví dụ tiếp theo, **STRING2** (bản 6-6), minh họa sáu hàm trong số các hàm này.

Listing 6-6 STRING2

```

//string1.cpp
#include<iostream.h>
#include<cstring.h>
void main()
{
    string s1("Don told Estelle he would get ring"); //chuoi C
    cout<<"s1= "<<s1<<endl;
    s1.insert(26,"not ");          //chen not truoc get
    cout<<"s1= "<<s1<<endl;

    s1.remove(26,4);              //xoa not
    cout<<"s1= "<<s1<<endl;

    s1.replace(9,7,"Pam",0,3);    //thay the "Estelle" bang "Pam"
    cout<<"s1= "<<s1<<endl;

    int loc1=s1.find("Pam");      //tim "Pam"
}

```

```

cout<<"Pam o vi tri: "<<loc1<<endl;
//tim ky tu trang dau tien sau loc1
int loc2=s1.find_first_of("\t\n",loc1);
//tao chuoai "Pam"
string s2=s1.substr(loc1,loc2-loc1);
cout<<"Hi, "<<s2<<endl;
}

```

Đây là kết quả của chương trình:

```

s1= Don told Estelle he would get ring
s1= Don told Estelle he would not get ring
s1= Don told Estelle he would get ring
s1= Don told Pam he would get ring
Pam o vi tri: 9
Hi, Pam he would get ring

```

1. Các hàm thành viên *insert()*, *remove()* và *replace()*

Hàm thành viên **insert(pos,ptr)** chèn chuỗi **char* ptr** vào đối tượng của nó, bắt đầu tại vị trí **pos**. Hàm **remove(pos,n)** xóa **n** ký tự khỏi đối tượng của nó, bắt đầu tại vị trí **pos**. Hàm **replace(pos,nptr)** xóa **n** ký tự khỏi đối tượng của nó, bắt đầu tại vị trí **pos** và thay thế chúng bằng chuỗi **char* ptr**.

2. Hàm thành viên *find()*

Hàm thành viên **find(ptr,pos)** tìm kiếm một mẫu, tạo ra từ chuỗi **char* ptr**, trong đối tượng của nó, bắt đầu từ vị trí **pos** và trả về vị trí của ký tự đầu tiên tìm được.

3. Hàm thành viên *find_first_of()*

Hàm **find_first_of(ptr,pos)** rất hữu ích khi muốn tìm kiếm một ký tự trong một đối tượng **string** nhưng không chắc chắn muốn tìm ký tự nào; nghĩa là tìm kiếm bất kỳ ký tự nào trong một số các ký tự. Đối số **ptr** là một chuỗi **char*** bao gồm tất cả các ký tự có thể và **pos** là vị trí bắt đầu tìm kiếm trong đối tượng **string**. Trong ví dụ chúng ta sử dụng hàm này để tìm kiếm ký tự trắng đầu tiên, dù là **space**, **tab** hay **newline**, theo sau vị trí đầu tiên của tên. Điều này cho phép ta tìm kiếm cuối của một tên dù nó ở cuối một dòng hay một cột.

4. Hàm thành viên *substr()*

Nếu biết vị trí bắt đầu và kết thúc của một dãy liên tiếp các ký tự trong một đối tượng **string** thì có thể đưa dãy này vào một đối tượng **string** bằng hàm **substr(pos,n)**, ở đây **pos** là vị trí bắt đầu của chuỗi con và **n** là chiều dài của nó. Trong chương trình chúng ta sử dụng hàm này để tạo một đối tượng chuỗi "Pam", nó được gán tới **s2**.

6.3.4. Truyền các đối tượng **string** như các đối số

Một trong những điều thú vị về lớp **string** là không cần truyền các con trỏ như các đối số tới hàm như làm với các chuỗi **char*** thông thường. Chúng ta chỉ đơn giản truyền đối tượng **string**. Ví dụ tiếp theo, **STRING3** (bản 6-7) cho thấy cách này như thế nào và minh họa một vài hàm quan trọng.

Listing 6-7 STRING3

```

//string3.cpp
#include<iostream.h>

```

```

#include<cstring.h>
void main()
{
    string func(string);           //khai bao ham
    string s1("IN HOC SIGNO VINCES"); //mot doi tuong string
    string s2=func(s1);           //goi ham
                                   //hien thi gia tri tra ve

    cout<<"main()hien thi: "<<s2<<endl;
    char char_arr[80];           //mang kieu char
    int len=s1.length();         //chieu dai cua doi tuong string
    int n=s1.copy(char_arr,len,0); //copy s1 toi char_arr

    char_arr[len]='\0';           //ket thuc chuoi char*
    cout<<"So ky tu da copy= "<<n<<endl;
    cout<<"char_arr= "<<char_arr<<endl;
    const char* ptr=s1.c_str();   //tro toi char*
    cout<<"ptr= "<<ptr<<endl;
}
string func(string s)           //dinh nghia ham
{
    cout<<"func() hien thi doi so: "<<s<<endl;
    return string("Tra ve tu func()\n");
}

```

Hàm **func()** được truyền tới một đối tượng **string**, nó hiển thị đối tượng này. Sau đó trả về một đối tượng **string** khác, hàm **main()** hiển thị đối tượng **string** trả về để thấy mọi thứ làm việc như mong đợi.

1. Hàm thành viên **copy()**

Hàm **copy()** cho phép copy một đối tượng **string** (hoặc một phần của nó) tới một chuỗi **char***. Đặc biệt hơn, **copy(ptr,n,pos)** copy n ký tự từ đối tượng của nó, bắt đầu từ vị trí **pos** tới chuỗi **char* ptr**.

2. Hàm thành viên **c_str()**

Hàm **c_str()** chuyển một đối tượng **string** sang một chuỗi **char***. Nghĩa là, nó trả về một con trỏ trỏ tới một mảng **char** chứa các ký tự từ đối tượng **string**, cộng với ký tự kết thúc '\0'. Con trỏ trả về từ **c_str()** là **const**, bởi vậy chúng ta không thể thay đổi bất kỳ cái gì trong mảng **char** đó, nó chỉ để đọc. Đây là kết quả của **STRING3**:

```

func() hien thi doi so: IN HOC SIGNO VINCES
main()hien thi: Tra ve tu func()

So ky tu da copy= 19
char_arr= IN HOC SIGNO VINCES
ptr= IN HOC SIGNO VINCES

```

6.3.5. Mảng các đối tượng **string**

1. Giới thiệu

Thật thú vị và dễ dàng lưu trữ các đối tượng **string** trong các mảng. Với các chuỗi **char*** thông thường, chúng ta phải cài đặt hoặc một mảng của các mảng kiểu **char**, để tất cả các chuỗi có cùng

độ dài hoặc một mảng con trở về tới các chuỗi. Ngược lại, các đối tượng **string** có thể được lưu trữ dễ dàng như các biến cơ bản chẳng hạn **int**, bởi vì tất cả chúng có kích thước giống nhau; chúng quan tâm đến yêu cầu bộ nhớ của riêng chúng ở bên trong.

Trong ví dụ tiếp theo, bản 6-8, chúng ta sẽ cài đặt một mảng các đối tượng **string** được sắp xếp theo thứ tự alphabe và sau đó sử dụng hàm thành viên **compare()** để chèn một chuỗi mới, nhập vào bởi người sử dụng, vào vị trí thích hợp trong mảng.

Listing 6-8 STRING4

```
//string4.cpp
#include<iostream.h>
#include<cstring.h>
void main()
{
    const int SZ=10;           //so ten ban dau
    string new_name;
    string arr[SZ+1]={"Adam","Bob","Clair","Doug","Emily","Frank",
                     "Gail","Harry","Ian","Joe"};

    cout<<"\nNhap vao mot ten: ";
    cin>>new_name;
    int j=0;
    while(j<SZ+1)
    {
        int len=arr[j].length();
        int lex_comp=new_name.compare(arr[j],0,len);
        if(lex_comp>0)          //neu ten moi lon hon
            j++;
        else                    //neu khong
        {
            for(int k=SZ-1;k>=j;k--) //di chuyen tat ca cac ten o tren
                arr[k+1]=arr[k];    //ten nay
            arr[j]=new_name;
            break;
        } //ket thuc else
    } //ket thuc while
    if(j==SZ+1)                 //neu ten moi dung sau tat ca cac ten khac
        arr[10]=new_name;
    cout<<"Cac ten co trong danh sach la:"<<endl;
    for(j=0;j<SZ+1;j++)        //hien thi tat ca cac ten
        cout<<arr[j]<<endl;
}
```

2. Hàm thành viên *compare()*

Hầu hết các lệnh trong chương trình đều liên quan đến việc tìm ra một vị trí trong mảng để chèn vào một tên mới. Hàm **compare(str,pos,n)** so sánh một dãy n ký tự trong đối tượng của nó, bắt đầu từ vị trí **pos**, với đối tượng **string** trong đối số của nó. Giá trị trả về cho biết hai đối tượng **string** được sắp xếp theo thứ tự alphabe như thế nào.

- Giá trị trả về < 0 Chuỗi hiện tại đứng sau đối số str theo thứ tự alphabe.
- Giá trị trả về = 0 Chuỗi hiện tại bằng str.
- Giá trị trả về > 0 Chuỗi hiện tại đứng trước str theo thứ tự alphabe.

Chương trình so sánh tên với từng phần tử của mảng. Nếu tên mới lớn hơn phần tử mảng, phần tử tiếp theo được kiểm tra. Nếu không, tên mới được chèn vào tại vị trí đó trong mảng và tất cả các đối tượng **string** từ điểm đó trở lên được di chuyển để có thể chèn vào tên mới.

Đây là vài mẫu tương tác với STRING4:

```
Nhap vao mot ten: Dilbert
Cac ten co trong danh sach la:
Adam
Bob
Clair
Dilbert
Doug
Emily
Frank
Gail
Harry
Ian
Joe
```

6.4. CHƯƠNG TRÌNH NHIỀU FILE

6.4.1. Tại sao cần các chương trình nhiều file

Có vài lý do để sử dụng các chương trình nhiều file. Những lý do này bao gồm có việc sử dụng các thư viện lớp, việc tổ chức những người lập trình làm việc trên cùng một đề án và việc thiết kế trên khái niệm của một chương trình. Chúng ta cùng xem xét một cách ngắn gọn về các vấn đề này.

1. Các thư viện lớp

Trong các ngôn ngữ thủ tục truyền thống, các nhà sản xuất phần mềm thường phải mất rất nhiều thời gian mới hoàn thành các thư viện hàm để quản lý các phép tính thống kê hoặc quản lý bộ nhớ. Ví dụ như thư viện hàm cung cấp các hàm cần thiết cho Graphics User Interface (GUI).

Bởi vì C++ tổ chức xung quanh các lớp hơn là các hàm nên không có gì ngạc nhiên khi thấy các thư viện cho các chương trình C++ chứa các lớp. Những gì gây ngạc nhiên là một thư viện lớp tốt hơn một thư viện hàm cổ điển bởi vì các lớp bao bọc cả dữ liệu và các hàm, đồng thời cũng bởi vì chúng rất gần gũi với các đối tượng mô hình trong thế giới thực, giao diện giữa một thư viện lớp và chương trình ứng dụng làm cho việc sử dụng nó rõ ràng hơn nhiều so với một thư viện hàm.

Vì những lý do này, các thư viện lớp đóng một vai trò quan trọng trong lập trình C++ hơn là các thư viện hàm trong lập trình truyền thống. Một thư viện lớp có thể chiếm một phần lập trình lớn hơn. Một người lập trình ứng dụng, nếu có sẵn một thư viện lớp, có thể thấy rằng chỉ cần một số lượng lập trình tối thiểu để tạo ra một sản phẩm cuối cùng. Ngoài ra, càng tạo ra nhiều thư viện lớp, càng có nhiều cơ hội tìm ra những giải pháp giải quyết các vấn đề lập trình ngày càng tăng.

Một thư viện lớp gồm có hai thành phần: **private** và **public**.

2. Thành phần **public**

Để sử dụng một thư viện lớp, người lập trình ứng dụng cần truy nhập các khai báo khác nhau, bao gồm các khai báo lớp. Các khai báo này có thể xem như là phần **public** của thư viện và thường được cung cấp dưới dạng mã nguồn như một file tiêu đề với phần mở rộng **.H**. File này thường được kết nối tới mã nguồn của khách hàng bằng một lệnh **include**.

Các khai báo trong một file tiêu đề như vậy cần là **public** vì mấy lý do. Thứ nhất, nó thuận tiện cho khách hàng để xem các định nghĩa thực sự hơn là đọc một mô tả về chúng. Quan trọng hơn, chương trình của khách hàng sẽ cần khai báo các đối tượng dựa trên các lớp này và gọi các hàm thành viên từ các đối tượng này. Chỉ bằng cách khai báo các lớp trong file nguồn là việc này có thể thực hiện được.

3. Thành phần *private*

Ngược lại, các công việc bên trong của các hàm thành viên của các lớp khác nhau không cần cho khách hàng biết. Những người phát triển thư viện lớp, cũng như bất kỳ người phát triển phần mềm nào khác, không muốn đưa ra mã nguồn bởi vì mã nguồn có thể bị thay đổi không đúng hoặc bị vi phạm quyền tác giả. Bởi vậy, các hàm thành viên, trừ các hàm ngắn, thường được phân bố trong một file dạng đối tượng .OBJ hay các thư viện LIB.

6.4.2. Tổ chức và khái niệm hóa

Các chương trình có thể được chia ra làm nhiều file vì các lý do khác với thư viện lớp. Như trong các ngôn ngữ khác, một tình huống thông dụng là một đề án đòi hỏi phải có vài người lập trình (hoặc một đội lập trình). Phân chia nhiệm vụ cho từng người lập trình vào một file riêng giúp cho tổ chức đề án và định nghĩa giao diện giữa các phần của chương trình rõ hơn.

Một chương trình thường được chia ra các file riêng biệt theo chức năng: ví dụ, một file có thể quản lý các mã liên quan đến màn hình đồ họa, trái lại một file khác quản lý việc phân tích toán học và file thứ ba quản lý vào/ra đĩa. Trong các chương trình lớn, một file có thể trở nên quá lớn dẫn đến khó quản lý, bởi vậy cần phải chia ra thành các file nhỏ.

Chúng ta sẽ nói các bước để tạo một chương trình nhiều file dùng trình biên dịch đại diện là Borland C++ (các trình biên dịch khác chỉ khác vài chi tiết).

6.4.3. Cách tạo một chương trình nhiều file

Cho rằng chúng ta đã mua một file lớp có tên là THEIR.OBJ. Nó có thể kèm theo một file tiêu đề, có tên là THEIR.H. Chúng ta cũng đã viết chương trình của riêng chúng ta để sử dụng các lớp đó, file nguồn của ta có tên là MINE.CPP. Bây giờ chúng ta muốn kết nối các file thành phần này: THEIR.OBJ, THEIR.H và MINE.CPP vào một chương trình thực hiện.

1. File tiêu đề

File tiêu đề THEIR.H dễ dàng kết hợp vào file nguồn của ta, MINE.CPP, chỉ bằng một lệnh **include**:

```
#include "THEIR.H"
```

Hai dấu nháy kép, không phải hai dấu ngoặc nhọn, bao quanh tên file bảo trình biên dịch đầu tiên tìm kiếm trong thư mục hiện tại hơn là thư mục mặc định INCLUDE.

2. Thư mục

Trong thực tế, khi tạo Project, chúng ta nên tạo một thư mục riêng để tránh nhầm lẫn và nên đảm bảo rằng tất cả các file thành phần, THEIR.OBJ, THEIR.H và MINE.CPP ở trong cùng một thư mục riêng này. Điều này không phải là cần thiết nhưng nó thuận lợi cho ta.

6.4.4. Vùng tên (namespaces)

Một chương trình càng lớn, càng nguy hiểm do sử dụng trùng tên, nghĩa là một tên được sử dụng, do không chú ý, cho nhiều thứ khác nhau. Đây là một vấn đề trong các chương trình nhiều file được tạo ra bởi các người lập trình khác nhau hoặc các công ty khác nhau.

Ví dụ, cho rằng lớp **alpha** được định nghĩa trong file A:

```
//file A
class alpha
{
};
```

Trong một file khác, sẽ liên kết với file A, cũng có một mô tả cho một lớp có cùng tên - có thể người tạo file này không biết người tạo file A:

```
//file B
class alpha
{
};
```

Sự trùng tên làm cho bộ liên kết lỗi. Những người lập trình thấy vậy có thể sửa vấn đề đó bằng cách đổi tên một lớp alpha. Tuy nhiên có một cách dễ hơn.

Phiên bản mới đây của tiêu chuẩn ANSI C++ giới thiệu một giải pháp mới cho vấn đề này: vùng tên (**namespace**). Một vùng tên là một phần của chương trình được xác định bởi từ khóa **namespace** và được bao trong các dấu ngoặc nhọn. Một tên được dùng trong vùng tên không phải là tên toàn cục; phạm vi của nó được giới hạn trong vùng tên.

1. Khai báo các vùng tên

Trong file A và B, người tạo hai file này là Thang và Binh, có thể giới hạn phạm vi nhìn thấy của các mô tả lớp **alpha** bằng cách đặt từng lớp vào vùng tên riêng của nó:

```
//file A
namespace NS_THANG
{
    class alpha
    {
    };
}
//file B
namespace NS_BINH
{
    class alpha
    {
    };
}
```

2. Truy nhập các phần tử của một vùng tên khác

Trong vùng tên, chúng ta có thể tham chiếu tới **alpha** theo cách thông thường. Tuy nhiên, ngoài vùng tên, chúng ta phải chỉ rõ là tham chiếu tới lớp nào. Có hai cách để làm việc này: sử dụng toán tử quy định phạm vi để xác định một vùng tên khác trong một câu lệnh hoặc sử dụng chỉ dẫn (**directive**) để cho phép suốt một vùng tên khác truy nhập tới một vài hoặc tất cả các phần tử trong vùng.

Để truy nhập tới một phần tử trong một vùng tên khác theo cách chỉ dùng một câu lệnh thì đứng trước tên của phần tử đó có tên của vùng tên và toán tử quy định phạm vi (::). Ví dụ, chúng ta có thể định nghĩa hai loại đối tượng **alpha** khác nhau như sau:

```
//file C
namespace NC_HUNG
{

```

```

....
NS_THANG::alpha anAlpha1;    //tao mot doi tuong alpha THANG
NS_BINH::alpha anAlpha2;    //tao mot doi tuong alpha BINH
....
}

```

Để cho phép truy nhập tới một phần tử cụ thể của một vùng tên trong suốt vùng tên thứ hai, chúng ta còn có thể sử dụng khai báo **using**. Đoạn chương trình dưới đây tạo một lớp **alpha** của **THANG** có thể được truy nhập trong suốt vùng tên **HUNG**.

```

//file C
namespace NC_HUNG
{
    using NS_THANG::alpha;
    ....
    alpha anAlpha1; //tao mot doi tuong alpha THANG
    alpha anAlpha2; //tao mot doi tuong alpha THANG
    ....
}

```

Để truy nhập tới tất cả các phần tử của một vùng tên trong suốt một vùng tên thứ hai, có thể sử dụng từ khóa **using** như một chỉ dẫn:

```

//file D
namespace NT_HOA
{
    using namespace NS_THANG;
    ....
    alpha anAlpha1; //tao mot doi tuong alpha THANG
    alpha anAlpha2; //tao mot doi tuong alpha THANG
    //truy nhap toi cac phan tu khac cua vùng ten THANG
    ....
}

```

Với sơ đồ này, **HOA** có thể truy nhập tới bất kỳ phần tử nào trong vùng tên của **THANG**.

PHẦN II

CẤU TRÚC DỮ LIỆU

Hầu hết các chương trình máy tính tồn tại là để xử lý dữ liệu. Dữ liệu có thể biểu diễn rất nhiều loại thông tin thế giới thực: các bản ghi nhân sự, các bản kiểm kê, văn bản, kết quả các cuộc thí nghiệm khoa học. Bất kỳ là biểu diễn cái gì, dữ liệu cũng được lưu trong bộ nhớ và được thao tác theo những cách giống nhau. Các chương trình máy tính ở cấp đại học thường có một nguồn gọi là cấu trúc dữ liệu và giải thuật. Cấu trúc dữ liệu là cách tổ chức lưu trữ dữ liệu trong bộ nhớ còn giải thuật là cách xử lý dữ liệu.

Như vậy, cùng với việc tổ chức dữ liệu thành cấu trúc, các giải thuật phải được thiết lập để xử lý dữ liệu và cho ra kết quả mong muốn. Việc chọn lựa cấu trúc dữ liệu, thiết lập các giải thuật đúng đắn, có cấu trúc tốt và có hiệu quả là những vấn đề mấu chốt của việc thiết lập phần mềm. Hai khía cạnh này của việc phát triển phần mềm có tầm quan trọng như nhau. Trong thực tế hai khía cạnh này không thể tách rời nhau. Thật vậy, Niklaus Wirth, người sáng lập ra ngôn ngữ Pascal, đã tổng kết:

Giải thuật + Cấu trúc dữ liệu = Chương trình

Từ *giải thuật* (algorithm) được lấy từ tên của nhà toán học Ả rập Abu Ja'far Mohammed ibn Musa al Khowarizmi (năm 825 sau công nguyên), người đã viết cuốn sách mô tả các thủ tục tính toán với các số Hindu. Theo cách nói hiện đại, từ này có nghĩa là "thủ tục từng bước một để giải quyết vấn đề hoặc để hoàn thành một đích cuối cùng nào đó". Trong tin học, thuật ngữ *giải thuật* dùng để chỉ một thủ tục có thể thực hiện được bằng máy tính. Do vậy điều này dẫn đến một số quy định đối với các lệnh tạo nên thủ tục, đó là:

1. Chúng phải được *xác định* và *không nhập nhằng* để có thể biết rõ lệnh nào thực hiện cái gì.
2. Chúng phải *đơn giản* để máy tính thực hiện được.
3. Chúng phải *thỏa mãn tính hữu hạn*, nghĩa là thuật toán phải kết thúc sau một số hữu hạn các phép toán.

Theo hai yêu cầu đầu tiên (xác định, không nhập nhằng và đơn giản) các giải thuật thường được mô tả dưới dạng có thể tạo thành chương trình máy tính với mục đích dễ dàng cài đặt từng bước của giải thuật bằng một lệnh hoặc một dãy lệnh của máy tính. Vì vậy, các giải thuật thường được mô tả dưới dạng giả mã (pseudocode), một ngôn ngữ lập trình giả chứa ngôn ngữ tự nhiên, các ký hiệu, thuật ngữ và các khía cạnh khác trong ngôn ngữ lập trình bậc cao. Bởi vì không có cú pháp tiêu chuẩn, các giải thuật thay đổi từ người lập trình này sang người lập trình khác, nhưng nó thường bao gồm các khía cạnh sau:

1. Các ký hiệu máy tính thông thường +, -, *, / dùng cho các phép toán số học cơ bản.
2. Các định danh (identifiers) dùng để thể hiện các đại lượng được xử lý bởi các giải thuật.
3. Một vài quy ước được tạo ra để chỉ các lời chú thích, ví dụ như chương trình viết bằng ngôn ngữ lập trình C++ chứa các lời chú thích sau hai dấu gạch chéo // hoặc /* và */.
4. Sử dụng các từ khóa thông thường trong các ngôn ngữ bậc cao, ví dụ *read* hoặc *input* cho các phép toán nhập; *display*, *print* hay *write* cho các phép toán xuất.
5. Dùng cách viết lùi vào để làm nổi bật các khối lệnh.

Yêu cầu thứ ba (tính hữu hạn) đòi hỏi giải thuật sớm hay muộn phải được kết thúc, nghĩa là sẽ kết thúc sau một số bước hữu hạn thực hiện các lệnh. Điều này có nghĩa là giải thuật không được

chứa các vòng lặp vô hạn. Ví dụ nếu một giải thuật có chứa các lệnh thực hiện việc lặp lại cho đến khi một biểu thức logic nào đó nhận giá trị đúng thì các lệnh đó phải làm thay đổi (vào một thời điểm nào đó) giá trị biểu thức logic này thành sai để chu trình được kết thúc. Đương nhiên là khi giải thuật được kết thúc thì chúng ta cũng chờ đợi là nó sẽ cho ta những kết quả mong muốn. Vì vậy, ngoài tính hữu hạn của giải thuật, chúng ta cũng cần phải kiểm tra tính đúng đắn nữa. Thực tế cho thấy rằng chỉ yêu cầu giải thuật sẽ kết thúc là chưa đủ. Ví dụ, một giải thuật tính nước cờ thắng cuộc trong mỗi giai đoạn đòi hỏi phải thử tất cả các trường hợp của các nước đi để xác định nước nào thắng, giải thuật này sẽ kết thúc nhưng nó cần quá nhiều thời gian và do đó không có giá trị thực tế. Những giải thuật có ích phải kết thúc sau một thời gian thích hợp.

Việc phân tích, kiểm nghiệm giải thuật và chương trình có thể tiến hành dễ dàng hơn nhiều nếu chúng ta có cấu trúc tốt. Chúng ta cũng cần nhớ lại rằng, các giải thuật và chương trình có cấu trúc được thiết kế bởi ba cấu trúc cơ bản sau đây:

1. Tuần tự (Sequential): các bước được thực hiện theo trình tự một cách chính xác, mỗi bước được thực hiện đúng một lần.
2. Chọn lọc (Selection): một trong nhiều thao tác sẽ được chọn và được thực hiện.
3. Lặp lại (Repetition): một hay nhiều bước được thực hiện lặp lại.

Ba cơ chế điều khiển này là đơn giản, nhưng trong thực tế chúng đủ mạnh để có thể xây dựng bất kỳ giải thuật nào.

Những giải thuật được thiết kế một cách cẩn thận bằng các cấu trúc điều khiển nói trên rất dễ đọc và dễ hiểu, vì vậy có thể phân tích và kiểm nghiệm chúng dễ dàng hơn nhiều so với các giải thuật không có cấu trúc.

Sau khi đã chọn lựa cấu trúc dữ liệu để lưu trữ các mục dữ liệu và thiết kế giải thuật để xử lý các mục dữ liệu này, chúng ta cần phải cài đặt chúng. Khi chọn cách cài đặt một cấu trúc dữ liệu, điều quan trọng là cần phải đánh giá hiệu suất của chúng. Hiệu suất (efficiency) thường được đo theo hai điều kiện. Điều kiện thứ nhất là **sự sử dụng không gian**, đó là lượng vùng nhớ cần thiết cho cấu trúc lưu trữ và điều kiện thứ hai là **hiệu suất thời gian**, đó là thời gian cần thiết để thực hiện các giải thuật. Thông thường không thể đồng thời làm cho cả thời gian lẫn không gian cần thiết cho việc cài đặt đều là cực tiểu. Các giải thuật cho các cấu trúc lưu trữ cần ít bộ nhớ nhất thường chậm hơn những giải thuật cho các cấu trúc lưu trữ cần nhiều bộ nhớ hơn. Vì vậy, các lập trình viên luôn gặp phải sự xung đột giữa hiệu suất không gian và thời gian.

Một xung đột khác thường gặp là giữa hiệu suất và sự rõ ràng. Một cách lý tưởng là các giải thuật cần phải có hiệu suất cao, dễ hiểu và dễ dịch thành chương trình. Những mục đích này đôi lúc mâu thuẫn nhau bởi vì giải thuật có hiệu suất cao nhất có thể không phải là dễ hiểu nhất hay không mã hóa dễ dàng nhất. Nếu một chương trình chỉ dùng cho vài lần hay với ít dữ liệu vào thì cách hợp lý là chọn giải thuật ít hiệu quả hơn và đơn giản hơn so với việc chọn giải thuật hiệu quả hơn nhưng phức tạp hơn. Thời gian tiết kiệm được trong khi viết, gỡ rối và bảo trì sẽ bù cho việc thực hiện chương trình ít hiệu quả hơn.

Bởi vì các cấu trúc dữ liệu và các giải thuật phải được cài đặt trong một ngôn ngữ nào đó, trong khi thiết kế cần phải sử dụng ưu điểm của ngôn ngữ ấy. Điều này có nghĩa là nếu có thể, chúng ta nên dùng các cấu trúc dữ liệu, các giải thuật có sẵn trong ngôn ngữ đó. Các lớp C++ cho một cơ chế rất tốt để tạo một thư viện cấu trúc dữ liệu. Từ khi phát triển C++ các nhà sản xuất trình biên dịch đã cung cấp các lớp chứa (**container classes**) để quản lý việc lưu trữ và xử lý dữ liệu. Mới đây, một cách tiếp cận mới đối với các thư viện lớp chứa đã được thêm vào bản thảo tiêu chuẩn ANSI/ISO C++. Nó được gọi là thư viện mẫu chuẩn (**Standard Template Library - STL**), do Alexander Stepanov và Meng Lee (người Mỹ) viết. Người ta hy vọng STL là một cách lưu trữ và xử lý dữ liệu chuẩn. Hiện nay các nhà sản xuất trình biên dịch đang bắt đầu đưa STL vào phần mềm của họ.

Trong phần này chúng ta sẽ nói về các cấu trúc dữ liệu và giải thuật của STL là chủ yếu và cách sử dụng chúng. Cấu trúc dữ liệu và giải thuật là một vấn đề lớn và phức tạp, nó đòi hỏi cả một cuốn sách lớn; do đó chúng ta không thể nói hết mọi thứ về chúng ở đây được. Trong phần này chúng ta quan tâm nhiều đến giới thiệu về STL và xem xét một vài côngtenơ và giải thuật thông dụng.

Bởi vì STL chưa chính thức đưa vào chuẩn C++ nên giữa các trình biên dịch có sự khác nhau, nhất là về các file tiêu đề. Để có những thông tin về cài đặt nó phải tra cứu tài liệu của hãng sản xuất trình biên dịch.

Những ví dụ trong phần này làm việc với STL được cài đặt trong Borland C++ 5.0. Các file tiêu đề sử dụng cho các giải thuật và các côngtenơ không có đuôi .H. Ví dụ, `#include<vector>`, `#include<algorithm>` v.v... Ngoài ra, trong chương trình chính phải xác định vùng tên **std** bằng câu lệnh sau:

```
using namespace std;
```

CHƯƠNG 7

THƯ VIỆN MẪU CHUẨN

STL chứa vài loại thực thể. Có ba loại quan trọng nhất là các côngtenơ (**container**), các giải thuật (**algorithm**), các **iterator** (con trỏ).

Một côngtenơ là một cách lưu trữ dữ liệu được tổ chức trong bộ nhớ. Ví dụ như ngăn xếp (**stack**) và danh sách liên kết (**linked list**). Một loại côngtenơ khác, mảng, cũng rất thông dụng và được đưa vào C++ (cũng như hầu hết các ngôn ngữ lập trình khác). STL còn có nhiều loại côngtenơ khác nữa mà rất có ích. Các côngtenơ STL được cài đặt bằng các lớp mẫu để chúng có thể lưu trữ các loại dữ liệu khác nhau.

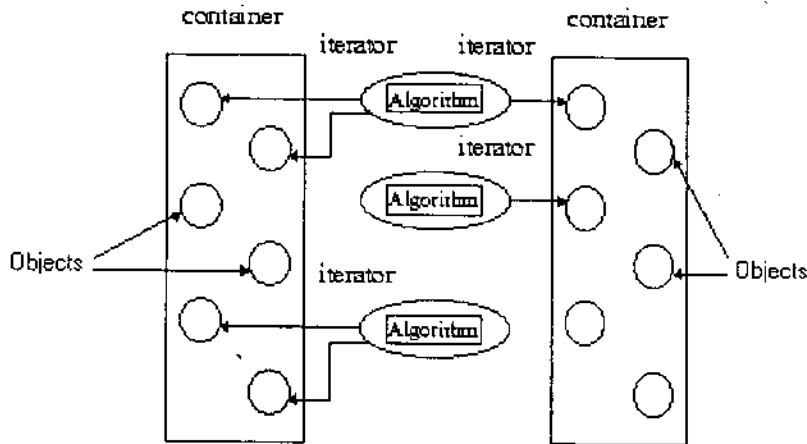
Các giải thuật (**algorithm**) là các thủ tục gắn với các côngtenơ để xử lý dữ liệu của chúng theo nhiều cách khác nhau. Ví dụ, có các giải thuật sắp xếp, sao chép, tìm kiếm và trộn dữ liệu. Trong STL, các giải thuật được biểu diễn bởi các hàm mẫu (**template function**). Các hàm này không phải là các hàm thành viên của các lớp côngtenơ (**container classes**). Chúng là các hàm đứng một mình. Trên thực tế, một đặc điểm nổi bật của STL là các giải thuật của nó rất chung. Chúng không chỉ được sử dụng với các côngtenơ mà còn có thể sử dụng với các mảng C++ thông thường và các côngtenơ do người sử dụng tạo ra.

Các **iterator** là một dạng khái quát hóa của khái niệm con trỏ. Chúng trỏ tới các phần tử trong một côngtenơ. Chúng ta có thể tăng một **iterator** như tăng một con trỏ để nó trỏ lần lượt tới từng phần tử trong một côngtenơ. Các **iterator** là phần chủ yếu của STL bởi vì chúng nối các giải thuật với các côngtenơ. Có thể coi chúng như một phiên bản phần mềm của các dây cáp, giống như dây cáp nối máy tính với các thiết bị ngoại vi. Các **iterator** cũng nối các thành phần khác với nhau.

Hình 7-1 cho thấy ba thành phần chính của STL. Trong chương này chúng ta sẽ giới thiệu khá chi tiết về các côngtenơ, các giải thuật và các **iterator**. Các chương tiếp theo sẽ khảo sát cụ thể với các ví dụ minh họa.

7.1. CÔNGTENƠ

Côngtenơ (**container**) là một cách lưu trữ dữ liệu, bao gồm cả các kiểu dữ liệu có sẵn như **int**, **float** và các đối tượng lớp. STL có sẵn bảy loại côngtenơ cơ bản, hơn ba loại côngtenơ được rút ra từ các loại cơ bản này. Ngoài ra chúng ta có thể tạo ra các côngtenơ của riêng chúng ta dựa trên các



Các giải thuật sử dụng các iterator để thao tác các đối tượng trong các côngtenơ

Hình 7-1. Các côngtenơ, giải thuật và iterator.

loại côngtenơ cơ bản. Có thể chúng ta sẽ băn khoăn là tại sao lại cần nhiều loại côngtenơ như vậy? Tại sao không sử dụng các mảng C++ trong tất cả các tình huống lưu trữ? Câu trả lời là hiệu suất. Một mảng sẽ gặp khó khăn hoặc chậm trong nhiều tình huống lưu trữ.

Các côngtenơ STL chia làm hai loại: côngtenơ tuần tự (**sequential container**) và côngtenơ liên kết (**associative container**). Các côngtenơ tuần tự là vector (**vector**), danh sách (**list**) và hàng đợi hai đầu (**deque**). Các côngtenơ liên kết là tập hợp (**set**), đa tập hợp (**multiset**), ánh xạ (**map**) và đa ánh xạ (**multimap**). Ngoài ra, có một vài côngtenơ gọi là các kiểu dữ liệu trừu tượng, chúng là các dạng đặc biệt của các côngtenơ khác. Các loại côngtenơ này là ngăn xếp (**stack**), hàng đợi (**queue**) và hàng đợi ưu tiên (**priority queue**). Chúng ta sẽ lần lượt xem xét các loại côngtenơ này.

7.1.1. Côngtenơ tuần tự

Một côngtenơ tuần tự (**sequential container**) lưu trữ một tập hợp các phần tử mà có thể hình dung như một đường thẳng, như các nhà trên một phố. Mỗi phần tử liên kết với một phần tử khác bằng vị trí của nó theo đường thẳng. Mỗi phần tử (trừ phần tử cuối cùng) đều có một phần tử xác định đứng trước và đứng sau nó. Một mảng C++ là một ví dụ về một côngtenơ tuần tự.

Một vấn đề với mảng C++ là phải xác định kích thước của nó tại thời điểm biên dịch, nghĩa là trong chương trình nguồn. Thật không may là khi viết chương trình chúng ta thường không biết có bao nhiêu dữ liệu sẽ được lưu trữ trong mảng. Bởi vậy chúng ta phải xác định một mảng đủ lớn để lưu trữ những gì mà chúng ta đoán là số lượng dữ liệu cực đại. Khi chương trình chạy, sẽ gây lãng phí bộ nhớ nếu không điền đầy mảng hoặc tạo ra một thông báo lỗi nếu vượt ngoài khả năng của mảng. STL cung cấp côngtenơ vector để khắc phục những nhược điểm này của mảng.

Đây là một vấn đề khác với mảng. Cho rằng chúng ta đang lưu trữ các bản ghi nhân viên và đã sắp xếp chúng theo thứ tự alphabe bởi tên của nhân viên. Nếu bây giờ chúng ta muốn chèn một tên mới, chẳng hạn tên bắt đầu với L, thì ta phải di chuyển tất cả các nhân viên từ M tới Z để tạo chỗ trống cho tên mới này. Công việc này có thể rất mất thời gian. STL cung cấp côngtenơ danh sách (**list**), nó dựa trên ý tưởng của một danh sách liên kết để giải quyết vấn đề này.

Côngtenơ tuần tự thứ ba là hàng đợi hai đầu (**deque**), nó có thể được xem như là sự kết hợp giữa ngăn xếp (**stack**) và hàng đợi (**queue**). Một ngăn xếp làm việc trên nguyên tắc vào sau ra trước (Last-In-First-Out, **LIFO**). Cả vào và ra được thực hiện trên một đầu của ngăn xếp. Trái lại, một hàng đợi lại sử dụng dạng sắp xếp vào trước ra trước (First In First Out- **FIFO**): dữ liệu đi vào ở

phía trước và đi ra ở phía sau, giống như một hàng hành khách trong một nhà băng. Hàng đợi hai đầu kết hợp cả hai cách này để có thể chèn và xóa dữ liệu ở cả hai đầu. Từ "deque" được rút ra từ Double-Ended-QUEUE (hàng đợi hai đầu). Nó là một cơ chế linh hoạt mà không chỉ có ích với riêng nó, mà nó còn được sử dụng làm cơ sở cho ngăn xếp và hàng đợi. Bảng 7-1 tóm tắt các đặc điểm của các côngtenơ tuần tự, bao gồm cả mảng C++ thông thường để tiện so sánh.

Bảng 7-1. Các côngtenơ tuần tự cơ bản

Côngtenơ	Đặc điểm	Những thuận lợi và không thuận lợi
Mảng C++ thông thường	Kích thước cố định.	+ Truy nhập ngẫu nhiên nhanh (qua chỉ số mảng). + Chèn và các phần tử ở giữa chậm. + Kích thước không thay đổi tại thời điểm chạy chương trình.
Vector	Tái định vị, mảng có thể mở rộng được.	+ Truy nhập ngẫu nhiên nhanh (qua chỉ số). + Chèn và xóa ở giữa chậm. + Chèn và xóa ở cuối nhanh.
List	Danh sách liên kết kép.	+ Chèn và xóa ở bất kỳ vị trí nào nhanh. + Truy nhập nhanh tới cả hai đầu. + Truy nhập ngẫu nhiên chậm.
Deque	Như vector nhưng có thể truy nhập ở cả hai đầu.	+ Truy nhập ngẫu nhiên nhanh (dùng chỉ số). + Chèn và xóa ở giữa chậm. + Chèn và xóa ở đầu hoặc ở cuối nhanh.

Việc tạo một đối tượng côngtenơ rất dễ. Đầu tiên phải có một file tiêu đề thích hợp. Sau đó sử dụng dạng mẫu với tham số là loại đối tượng cần lưu trữ. Ví dụ:

```
vector<int> avact;           //tao mot vecto cac so nguyen
```

hoặc

```
list<airtime> departure_list; //tao mot danh sach cac doi tuong airtime
```

Chú ý rằng không cần xác định kích thước cho côngtenơ. Các côngtenơ tự chúng quan tâm tới tất cả việc cấp phát bộ nhớ.

7.1.2. Côngtenơ liên kết

Một côngtenơ liên kết là một côngtenơ không tuần tự; thay vào đó nó sử dụng các khóa để truy nhập dữ liệu. Các khóa, điển hình là các số hoặc các chuỗi, được sử dụng tự động bởi côngtenơ để sắp xếp các phần tử lưu trữ theo một trật tự nhất định. Nó giống như một quyển từ điển tiếng Anh mà ở đó chúng ta có thể truy nhập dữ liệu bằng cách tra các từ đã được sắp xếp theo thứ tự alphabe. Nếu bắt đầu với một giá trị khóa, chẳng hạn như "aardvark", thì côngtenơ sẽ chuyển khóa này tới vị trí của phần tử trong bộ nhớ. Nếu biết khóa thì chúng ta có thể nhanh chóng truy nhập tới giá trị liên kết.

Có hai loại côngtenơ liên kết trong STL: ánh xạ (map) và tập hợp (set). Một ánh xạ liên kết một khóa (ví dụ từ mà chúng ta muốn tra) với một giá trị (ví dụ như định nghĩa của từ). Giá trị có thể là bất kỳ loại đối tượng nào. Một tập hợp (set) tương tự như một ánh xạ (map) nhưng nó chỉ lưu trữ các khóa; không có các giá trị liên kết. Nó giống như một danh sách các từ không có định nghĩa.

Các côngtenơ ánh xạ và tập hợp chỉ lưu trữ duy nhất một khóa cho một giá trị. Điều này giống như một danh bạ điện thoại, ở đó mỗi người chỉ có duy nhất một số điện thoại. Trái lại, các côngtenơ đa ánh xạ (multimap) và đa tập hợp (multiset) cho phép có nhiều khóa. Ví dụ, trong một quyển từ điển tiếng Anh có thể có vài mục cho từ "set".

Bảng 7-2 tóm tắt các côngtenơ liên kết có sẵn trong STL.

Bảng 7-2. Các côngtenơ liên kết cơ bản

Côngtenơ	Đặc điểm	Những thuận lợi và không thuận lợi
Map	+ Liên kết khóa với phần tử. + Chỉ cho phép duy nhất một khóa cho mỗi giá trị.	+ Truy nhập ngẫu nhiên nhanh (bằng khóa). + Không hiệu suất nếu các khóa không được phân bố đều.
Multimap	+ Liên kết khóa với phần tử. + Cho phép nhiều giá trị khóa.	+ Truy nhập ngẫu nhiên nhanh (bằng khóa). + Không hiệu suất nếu các khóa không được phân bố đều.
Set	+ Chỉ lưu trữ các khóa. + Chỉ cho phép một khóa duy nhất với mỗi giá trị.	+ Truy nhập ngẫu nhiên nhanh (bằng khóa). + Không hiệu suất nếu các khóa không được phân bố đều.
Multiset	+ Chỉ lưu trữ các khóa. + Cho phép nhiều giá trị khóa.	+ Truy nhập ngẫu nhiên nhanh (bằng khóa). + Không hiệu suất nếu các khóa không được phân bố đều.

Tạo các côngtenơ liên kết y như tạo các côngtenơ tuần tự:

```
map<int> IntMap;           //tao mot anh xa cac so nguyen
hoặc
multiset<employee> machinists; //tao mot da tap hop cac doi tuong employee
```

7.1.3. Hàm thành viên

Các giải thuật (algorithm) thực hiện các công việc phức tạp như sắp xếp và tìm kiếm. Tuy nhiên, các côngtenơ cũng cần các hàm thành viên để thực hiện những nhiệm vụ đơn giản hơn mà chỉ dành riêng cho một côngtenơ cụ thể. Bảng 7-3 trình bày một vài hàm thành viên mà tên và chức năng của chúng là chung cho tất cả các lớp côngtenơ.

Bảng 7-3. Một vài hàm thành viên chung cho tất cả các côngtenơ

Tên	Chức năng
size()	Trả về số mục trong côngtenơ.
empty()	Trả về true nếu côngtenơ rỗng.
max_size()	Trả về kích thước lớn nhất có thể có của côngtenơ.
begin()	Trả về một con trỏ (iterator) trỏ tới đầu côngtenơ để bắt đầu trỏ tiến về phía trước qua côngtenơ.
end()	Trả về một con trỏ trỏ tới vị trí quá vị trí cuối của một côngtenơ, được dùng để kết thúc một con trỏ tiến.
rbegin()	Trả về một con trỏ ngược trỏ tới cuối côngtenơ để bắt đầu trỏ lùi qua côngtenơ.
rend()	Trả về một con trỏ ngược trỏ tới đầu côngtenơ, được dùng để kết thúc một con trỏ lùi.

Nhiều hàm thành viên khác chỉ xuất hiện trong các côngtenơ nhất định hoặc các loại côngtenơ nhất định.

7.1.4. Kiểu dữ liệu trừu tượng

Có thể sử dụng các côngtenơ cơ bản để tạo một loại côngtenơ khác gọi là một kiểu dữ liệu trừu tượng hay ADT (**Abstract Data Type**). Một ADT là một loại côngtenơ đơn giản hóa tập trung vào các khía cạnh cụ thể của một côngtenơ cơ bản hơn; nó cung cấp một giao diện khác cho người lập trình. Các ADT được cài đặt trong STL là ngăn xếp (**stack**), hàng đợi (**queue**) và hàng đợi ưu tiên (**priority queue**). Một ngăn xếp hạn chế truy nhập bằng việc đặt vào và lấy ra một mục dữ liệu ở đỉnh của ngăn xếp. Còn trong một hàng đợi, đặt các mục dữ liệu ở một đầu và lấy chúng ra ở đầu khác. Trong một hàng đợi ưu tiên, dữ liệu được đặt vào ở một đầu theo một thứ tự ngẫu nhiên, nhưng khi lấy ra ở đầu khác luôn luôn lấy mục dữ liệu lớn nhất được lưu trữ: hàng đợi ưu tiên tự động sắp xếp dữ liệu giúp chúng ta.

Cơ chế STL dùng để tạo ADT từ các kiểu cơ bản là bộ thích ứng (**adaptor**). Các **adaptor** là các lớp mẫu chuyển các hàm được dùng trong ADT thành các hàm được dùng bởi các côngtenơ nằm trong ADT.

Ngăn xếp, hàng đợi, hàng đợi ưu tiên được tạo ra từ các côngtenơ tuần tự khác nhau nhưng hàng đợi hai đầu (**deque**) thường được sử dụng nhất. Bảng 7-4 trình bày các kiểu dữ liệu trừu tượng và các côngtenơ tuần tự có thể được sử dụng để cài đặt chúng.

Bảng 7-4. Các kiểu dữ liệu trừu tượng

Côngtenơ	Cài đặt	Đặc điểm
Stack	Có thể cài đặt như vector, list hoặc deque.	Chèn (push, insert) và xóa ở một đầu.
Queue	Có thể cài đặt như list, hoặc deque.	Chèn tại một đầu và xóa ở một đầu khác.
Priority queue	Có thể cài đặt như vector hoặc deque.	Chèn (push, insert) theo thứ tự ngẫu nhiên, xóa (remove, pop) theo thứ tự đã được sắp xếp ở đầu khác.

Chúng ta có thể sử dụng một mẫu (**template**) trong một mẫu để tạo một ADT. Ví dụ, đây là một ngăn xếp lưu trữ kiểu **int**, được tạo ra từ côngtenơ deque:

```
stack<deque<int>> > astak;
```

Chú ý, phải chèn một khoảng trắng vào giữa hai dấu ngoặc nhọn để trình biên dịch không tưởng là toán tử >>.

7.2. GIẢI THUẬT

7.2.1. Giải thuật tìm kiếm

Như trên ta đã biết, giải thuật là cách xử lý dữ liệu. Vấn đề tìm kiếm một mục dữ liệu cho trước trong một nhóm các mục dữ liệu là một trong những vấn đề quan trọng nhất về xử lý dữ liệu. Trong mục này chúng ta sẽ đi qua một số phương pháp tìm kiếm thông dụng. Đó là: tìm kiếm tuyến tính, tìm kiếm nhị phân, tìm kiếm sử dụng bảng băm. Trong nhiều ứng dụng, nhóm các mục dữ liệu cần tìm thường được tổ chức như một danh sách.

1. Tìm kiếm tuyến tính

Tìm kiếm tuyến tính là phương pháp tìm kiếm đơn giản nhất. Theo phương pháp này, ta bắt đầu so sánh từ phần tử đầu tiên rồi cứ như vậy tìm tuần tự theo danh sách cho đến khi tìm ra mục dữ liệu cho trước hoặc đạt đến cuối danh sách. Phương pháp tìm kiếm tuần tự thông thường mất nhiều thời gian.

2. Tìm kiếm nhị phân

Sau đây là giải thuật tìm kiếm nhị phân để tìm Item trong danh sách $X_1, X_2, \dots, X_n$ được sắp xếp theo thứ tự tăng dần. Nếu tìm thấy, biến Found sẽ nhận giá trị True và biến Position sẽ nhận giá trị là vị trí của phần tử có giá trị bằng Item. Nếu không tìm thấy, biến Found sẽ nhận giá trị False.

1. Khởi động Found tại giá trị False, First tại giá trị 1, Last tại giá trị n.
2. While $\text{First} \leq \text{Last}$ and not Found thực hiện các bước sau:
 - a) Tính $\text{Position} = (\text{First} + \text{Last})/2$
 - b) If $\text{Item} < X_{\text{Position}}$ then
Gán giá trị Mid - 1 vào Last
 - Else if $\text{Item} > X_{\text{Position}}$ then
Gán giá trị Mid + 1 vào First
 - Else
Gán giá trị True vào Found

Trong giải thuật trên, phần tử ở giữa được so sánh đầu tiên, nếu nó không phải là phần tử cần tìm thì ta tiếp tục tìm nó trong nửa trước (nếu nó nhỏ hơn phần tử ở giữa) hoặc nửa sau (nếu nó lớn hơn phần tử ở giữa) của dãy. Như vậy, sau mỗi lần đi qua vòng lặp while, kích thước của danh sách con bị giảm đi một nửa, chính vì vậy số lần so sánh để tìm ra Item cũng giảm đi so với phép tìm kiếm tuyến tính.

Mặc dù giải thuật trên là lặp ta cũng có thể mô tả nó một cách đệ quy. Trong mỗi lần lặp lại, ý tưởng cơ bản là tìm phần tử ở giữa danh sách (hoặc danh sách con) và nếu nó không phải là phần tử cần tìm, ta sẽ tiếp tục tìm ở một trong hai nửa của danh sách đó theo cách hoàn toàn giống trước. Sau đây là giải thuật tìm kiếm nhị phân đệ quy.

1. Khởi động First tại giá trị 1, và Last tại giá trị n.
2. If $\text{First} > \text{Last}$ then /* danh sách (hoặc danh sách con) là rỗng */
Đặt Found bằng giá trị False
Else thực hiện các bước sau:
 - a) Tính $\text{Position} = (\text{First} + \text{Last}) / 2$
 - b) If $\text{Item} < X_{\text{Position}}$ then
Áp dụng giải thuật này với $\text{Last} = \text{Position} - 1$
 - Else if $\text{Item} > X_{\text{Position}}$ then
Áp dụng giải thuật này với $\text{First} = \text{Position} + 1$
 - Else
Đặt Found bằng giá trị True

3. Tìm kiếm sử dụng bảng băm

Trong các giải thuật đã xét từ trước đến nay, việc định vị một mục được xác định bởi một dãy các phép so sánh. Trong mỗi trường hợp, mục mẫu cần tìm được so sánh nhiều lần với các mục ở những vị trí nào đó trong cấu trúc. Mặc dù giải thuật tìm kiếm nhị phân đòi hỏi ít lần so sánh hơn giải thuật tìm kiếm tuyến tính, nhưng trong một số trường hợp, những giải thuật này thực hiện còn quá chậm. Ví dụ như đối với bảng các ký hiệu được lập bởi bộ dịch để lưu trữ các định danh và những thông tin về chúng. Vận tốc xây dựng và tìm kiếm trong bảng này sẽ quyết định vận tốc dịch. Một cấu trúc dữ liệu cho phép thực hiện việc tìm kiếm nhanh hơn, được gọi là **bảng băm (hash**

table), trong đó vị trí của một mục được xác định trực tiếp bằng một hàm của chính nó chứ không phải bằng một dãy các so sánh thử-và-sai (trial-and-error). Thời gian cần thiết để định vị một mục trong bảng băm là một hằng số và không phụ thuộc vào số lượng các mục được lưu trữ.

Để minh họa, giả sử rằng cần lưu trữ 25 số nguyên trên đoạn 0...999 trong bảng băm. Có thể cài đặt bảng băm này bằng một mảng Table chứa các số nguyên được đánh chỉ số trên đoạn 0...999, trong đó mỗi phần tử của mảng được khởi động tại một giá trị cam nào đó, ví dụ tại -1. Nếu ta dùng mỗi số nguyên i trong tập hợp cần lưu trữ đó làm chỉ số, nghĩa là nếu ta lưu trữ i trong $table[i]$, thì ta có thể định vị một số nguyên Number cho trước chỉ bằng cách kiểm tra $table[Number] = Number$ hay không. Hàm h định nghĩa bởi $h(i) = i$ xác định vị trí của mục i trong bảng băm được gọi là **hàm băm (hash function)**.

Hàm băm trong ví dụ này thực hiện tìm kiếm một cách lý tưởng vì thời gian cần thiết để tìm kiếm một giá trị cho trước trong bảng là hằng số, chỉ cần thời gian để thử một vị trí. Như vậy sơ đồ này rất hiệu quả về thời gian, nhưng chắc chắn là không hiệu quả về không gian. Vì chỉ 25 trong 1000 vị trí có thể được dùng để lưu trữ các mục, còn 975 vị trí còn lại không được dùng; có nghĩa là chỉ 2,5% của không gian cho phép được dùng, còn 97,5% bị lãng phí!

Vì có thể lưu trữ 25 giá trị ở 25 vị trí, ta có thể sử dụng tốt hơn không gian bằng cách sử dụng mảng Table được đánh chỉ số trên đoạn 0... 24. Dĩ nhiên là không thể dùng hàm băm ban đầu $h(i) = i$ được nữa. Thay vào đó ta có thể dùng:

$$h(i) = i \bmod 25$$

vì hàm này luôn luôn có miền giá trị là những số nguyên trên đoạn 0...24. Như vậy, số nguyên 52 sẽ được lưu trữ ở Table[2] vì $h(52) = 52 \bmod 25 = 2$. Tương tự, 129, 500, 73 và 49 sẽ được lưu trữ tại các vị trí 4, 0, 23, và 24 theo thứ tự đó.

Bảng băm	
Table[0]	500
Table[1]	-1
Table[2]	52
Table[3]	-1
Table[4]	129
Table[5]	-1
.	.
Table[23]	273
Table[24]	49

Hình 7-2. Bảng băm sau khi lưu trữ các số 52, 129, 500, 73, 49.

Tuy nhiên có một trở ngại là có thể xảy ra sự **va chạm (collision)**. Ví dụ, sau khi 52 được lưu vào Table[2], nếu 77 cũng được lưu trữ, nó sẽ được đặt vào vị trí $h(77) = 77 \bmod 25 = 2$, nhưng như trên đã thấy, vị trí này đã bị chiếm bởi 52. Cũng tương tự như vậy, nhiều giá trị khác có thể trùng vào một vị trí cho trước, ví dụ, 2, 27, 102; và trong thực tế tất cả các số nguyên dạng $25k + 2$ đều được lưu trữ tại vị trí 2. Dĩ nhiên là cần phải có một cách nào đó để giải quyết những va chạm này. Một phương pháp đơn giản để xử lý các va chạm là **thăm dò tuyến tính (linear probing)**. Trong sơ đồ này, phép tìm kiếm tuyến tính trong bảng bắt đầu ở vị trí có va chạm và tiếp tục cho đến khi tìm

thấy một khe rỗng có thể lưu trữ được. Như vậy, trong ví dụ trên, khi 77 va chạm với giá trị 52 ở vị trí 2, một cách đơn giản là chúng ta đặt 77 ở vị trí 3 vì vị trí này đang rỗng (chứa số âm -1); để chèn 102, chúng ta đi theo **dãy thăm dò (probe sequence)** gồm các vị trí 2, 3, 4 và 5 để xác định vị trí đầu tiên có thể dùng được và thấy rằng 102 có thể lưu trữ ở Table[5]. Khi đạt đến cuối bảng, ta lại tiếp tục từ vị trí đầu tiên. Ví dụ, 123 được lưu trữ ở vị trí 1 vì nó va chạm với 273 ở vị trí 23, và dãy thăm dò 23, 24, 0, 1 xác định khe rỗng đầu tiên ở vị trí 1..

Bảng băm	
Table[0]	500
Table[1]	123
Table[2]	52
Table[3]	77
Table[4]	129
Table[5]	102
Table[23]	273
Table[24]	49

Hình 7-3. Bảng băm sau khi lưu trữ thêm các số có va chạm.

Để tìm kiếm một giá trị cho trước có nằm trong bảng băm này hay không, trước hết ta dùng hàm băm để tính vị trí có thể tìm thấy trong bảng.

Ta cần xét ba trường hợp. Trường hợp thứ nhất, nếu vị trí này rỗng (chứa -1), ta có thể kết luận ngay rằng giá trị này không nằm trong bảng. Trường hợp thứ hai, nếu vị trí này chứa giá trị cần tìm, ta biết ngay là đã tìm ra. Trường hợp thứ ba, vị trí này chứa một giá trị khác với giá trị đang tìm do sự va chạm gây ra trong khi lập bảng. Với trường hợp này, ta thực hiện tìm kiếm tuyến tính "vòng" tại vị trí này, và tiếp tục cho đến khi tìm thấy giá trị đó hoặc đạt đến vị trí rỗng hay vị trí khởi đầu, vì điều này chỉ ra rằng mục cần tìm không có trong bảng. Như vậy, thời gian tìm kiếm trong hai trường hợp đầu là hằng số (chỉ cho một phép so sánh), còn trong trường hợp thứ ba thì không phải là hằng. Nếu bảng gần đầy, ta phải thử hầu hết tất cả các vị trí trước khi tìm ra mục đó hay trước khi kết luận rằng mục đó không có trong bảng.

Để quá trình tìm kiếm có hiệu quả hơn, khi thiết kế bảng băm, ta có thể thực hiện ba việc sau:

1. Mở rộng kích thước của bảng.
2. Dùng phương pháp khác để giải quyết va chạm.
3. Dùng một hàm băm khác.

Việc làm cho kích thước của bảng bằng số lượng các mục được lưu trữ như trong ví dụ trên không thực tế lắm, nhưng nếu dùng bảng nhỏ hơn có khả năng sẽ dẫn đến va chạm. Trong thực tế, ngay cả khi có thể lưu trữ nhiều hơn hẳn số lượng mục cần thiết thì vẫn có thể xảy ra va chạm. Như vậy rõ ràng là việc mong đợi một bảng băm hoàn thiện không bị va chạm là không xác đáng. Thay vào đó, chúng ta đành phải thỏa mãn với các bảng băm với số va chạm đủ ít. Những nghiên cứu thực nghiệm cho thấy, chúng ta nên dùng những bảng có độ dài khoảng từ $1\frac{1}{2}$ đến 2 lần số lượng mục cần lưu trữ.

Một cách giải quyết va chạm khác là dùng phương pháp **dây chuyền (chaining)**. Trong phương pháp này, ta dùng các danh sách liên kết để lưu trữ các giá trị va chạm và bảng băm là một mảng các con trỏ chỉ đến các danh sách liên kết này. Ta cũng có thể dùng một số phương pháp khác để giải quyết va chạm. Ví dụ, có thể dùng một mảng các con trỏ chỉ đến nút gốc của các cây tìm kiếm nhị phân sau đó dùng các thủ tục chèn và tìm kiếm.

Công việc cuối cùng có thể thực hiện để tăng hiệu quả của quá trình tìm kiếm là chọn hàm băm. Dáng điệu của hàm này đương nhiên là sẽ ảnh hưởng đến tần số va chạm. Một hàm băm lý tưởng để tính và rải đều các mục trong bảng băm làm giảm xác suất va chạm đến giá trị nhỏ nhất. Mặc dù không có phương pháp băm nào hoàn thiện trong mọi trường hợp, một phương pháp thông dụng hiện thời là **băm ngẫu nhiên (random hashing)** với kỹ thuật tạo số ngẫu nhiên để rải đều các mục trong bảng băm một cách ngẫu nhiên. Trước hết mục đó được chuyển sang một số nguyên lớn ngẫu nhiên bằng cách dùng một lệnh, chẳng hạn như:

$$\text{RandomInt} = ((25173 * \text{Item}) + 13849) \bmod 65536$$

và giá trị này được chia theo môđun kích thước của bảng để xác định vị trí của mục:

$$\text{Location} = \text{RandomInt} \bmod \text{TableSize};$$

Có thể dùng hàm băm này với các mục không phải là số nguyên nếu trước đó ta mã hóa những mục này thành các số nguyên. Ví dụ, mỗi một tên có thể được mã hóa như là tổng của các mã số ASCII của một vài hoặc tất cả các ký tự của nó.

7.2.2. Giải thuật sắp xếp

Khi xử lý dữ liệu, hai vấn đề tìm kiếm và sắp xếp (sorting) đều quan trọng bởi chúng thường đi đôi với nhau. Thật vậy, như ta đã thấy ở trên, một số thuật toán tìm kiếm có hiệu quả chỉ dùng được khi danh sách các mục dữ liệu đã được sắp xếp theo một thứ tự nào đó. Trong phần này, chúng ta sẽ xem xét vấn đề sắp thứ tự một danh sách $X_1, X_2, \dots, X_n$. Điều này nghĩa là sắp lại các phần tử của nó sao cho (theo một trường khóa nào đó) chúng có thứ tự tăng dần $X_1 < X_2 < \dots < X_n$ hoặc thứ tự giảm dần $X_1 > X_2 > \dots > X_n$. Chúng ta sẽ đi qua các giải thuật sắp xếp: sắp xếp bằng chọn lựa đơn giản, sắp xếp kiểu sủi bọt, sắp xếp kiểu chèn, một biến thể của cách sắp xếp kiểu chọn lựa đơn giản được gọi là heapsort và cuối cùng là giải thuật sắp xếp quicksort.

1. Sắp xếp bằng chọn lựa đơn giản

Ý tưởng cơ bản của cách sắp xếp này là duyệt qua danh sách hay một phần của nó nhiều lần và cứ mỗi lần duyệt qua thì sẽ có một phần tử được sắp xếp theo đúng vị trí cần thiết. Ví dụ, mỗi lần duyệt qua danh sách con, phần tử nhỏ nhất trong danh sách này được tìm ra và được chuyển đến đúng vị trí cần thiết của nó.

Giải thuật sắp xếp bằng chọn lựa đơn giản cho các danh sách được cài đặt bằng mảng được mô tả như sau:

```
/* Giải thuật sắp xếp n phần tử trong mảng X [1], [2], ..., X [n] theo thứ tự tăng dần*/
```

```
For i = 1 to n - 1 thực hiện các bước sau:
```

```
/* chọn phần tử nhỏ nhất trong danh sách con X[i], ..., X[n] */
```

```
a) Gán giá trị i cho SmallPos
```

```
b) Gán giá trị X [SmallPos] cho Smallest.
```

```
c) For j = i + 1 to n thực hiện các bước sau:
```

```
if X [j] < Smallest /* tìm ra phần tử nhỏ hơn */ then
```

```
    i. Gán giá trị j vào SmallPos
```

```
    ii. Gán giá trị X[SmallPos] cho Smallest
```

```
/* chuyển phần tử nhỏ nhất này vào đầu danh sách con */
```


d) Gán giá trị $X[i]$ vào $X[\text{SmallPos}]$

e) Gán giá trị Smallest vào $X[i]$

Để sử dụng giải thuật trên cho các danh sách liên kết, ta chỉ cần thay thế chỉ số i và j bằng các con trỏ chạy trong danh sách và danh sách con.

2. Sắp xếp kiểu sủi bọt

Giải thuật sắp xếp này cũng duyệt qua danh sách hoặc các danh sách con nhiều lần, so sánh và hoán vị các cặp phần tử khác nhau. Tuy nhiên thay vì chỉ hoán vị một lần trong mỗi lần duyệt qua, mỗi cặp các phần tử sẽ được hoán vị ngay khi chúng không có thứ tự đúng.

/ Giải thuật này sắp xếp kiểu sủi bọt một danh sách các mục $X_1, X_2, \dots, X_n$ theo thứ tự tăng dần */*

1. Khởi động NumPairs tại $n - 1$ và Last tại 1

/ NumPairs là cặp số cần được so sánh trong lần đi qua hiện thời và Last đánh dấu vị trí của phần tử cuối cùng đã cần phải hoán vị */*

2. While NumPairs $< > 0$ lặp lại các bước sau:

a) For $i = 1$ to NumPairs:

 If $X_i > X_{i+1}$ then

 i. Hoán vị X_i và X_{i+1}

 ii. Đặt Last bằng i

b) Đặt NumPairs bằng Last - 1

3. Sắp xếp kiểu chèn

Cách sắp xếp kiểu chèn được xây dựng trên cơ sở: chèn phần tử mới vào danh sách đã được sắp theo thứ tự để cho danh sách mới nhận được cũng có thứ tự đúng.

Thuật giải sau đây mô tả cách sắp xếp kiểu chèn cho danh sách được cài đặt trên cơ sở mảng

/ Giải thuật này sắp xếp một danh sách chứa các phần tử trong mảng $X[1], X[2], \dots, X[n]$ theo thứ tự tăng dần bằng cách sắp xếp kiểu chèn. Ký hiệu $-\infty$ ngụ ý một giá trị nào đó nhỏ hơn tất cả những giá trị trong danh sách */*

1. Khởi động $X[0]$ tại giá trị $-\infty$

2. For $i = 2$ to n thực hiện các bước sau:

/ Chèn $X[i]$ vào đúng vị trí của nó trong danh sách con chứa các phần tử $X[1], \dots, X[i-1]$ */*

a) Đặt NextElement bằng $X[i]$

b) Đặt j bằng $i - 1$

c) While NextElement $< X[j]$ thực hiện các bước sau:

/ Chuyển một phần tử sang để tạo ô trống */*

 i. Đặt $X[j+1]$ bằng $X[j]$

 ii. Tăng j thêm 1

/ Chuyển NextElement vào ô trống */*

d) Đặt $X[j+1]$ bằng NextElement.

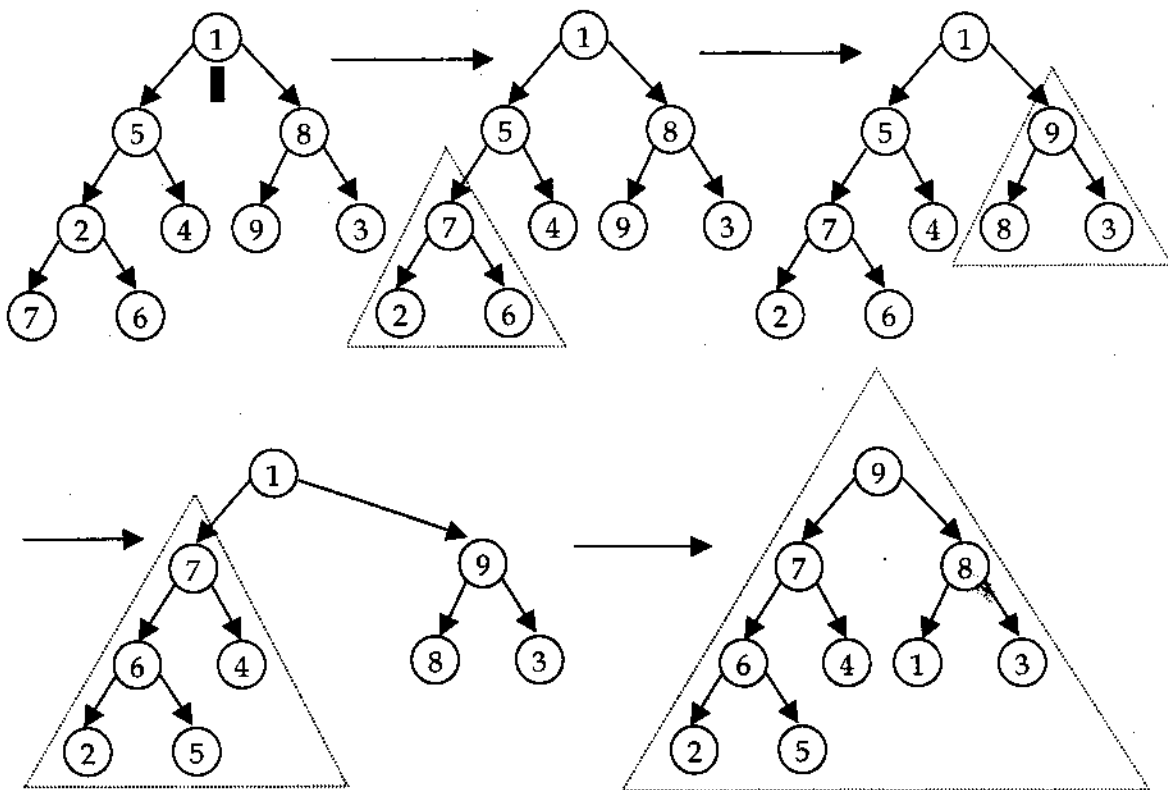
4. Heapsort

Heapsort là một biến thức của cách sắp xếp bằng chọn lựa đơn giản. Heapsort dùng một cấu trúc dữ liệu mới gọi là khối (heap) để tổ chức các phần tử của danh sách sao cho có thể thực hiện việc sắp xếp một cách có hiệu quả hơn.

Giống như các cây tìm kiếm nhị phân (BST) (xem giới thiệu về cây tìm kiếm nhị phân ở mục 12.4), khối là loại cây nhị phân đặc biệt. Nó khác với các BST ở hai điểm:

1. Nó là **đầy đủ**; nghĩa là các lá trên cây nằm nhiều nhất ở hai mức liên tiếp nhau và các lá ở mức dưới nằm ở những vị trí "bên trái nhất".
2. Giá trị của mục dữ liệu trong mỗi nút lớn hơn giá trị của những mục ở các con nó (dĩ nhiên là nếu các mục dữ liệu là bản ghi thì một trường nào đó trong bản ghi phải thỏa mãn điều kiện này).

Thuật toán chuyển từ cây nhị phân đầy đủ sang khối là cơ bản cho đa số các thao tác khác trên khối. Đối với bài toán tổng quát chuyển cây nhị phân đầy đủ sang khối, ta bắt đầu từ nút cuối cùng không phải là lá, áp dụng thủ tục swap-down để chuyển cây con có gốc tại nút này thành khối, chuyển lên nút trước và áp dụng thủ tục swap-down, v.v... cứ thế cho đến khi ta đạt đến nút gốc của cây. Dãy các biểu đồ sau đây minh họa quá trình "tạo khối" này, các cây con đang chuyển thành khối được đánh dấu.



Hình 7-4. Quá trình tạo khối.

Giải thuật cài đặt quá trình trên như sau:

SWAPDOWN.

/* Cho trước một cây nhị phân đầy đủ các vị trí $r...n$ của mảng Heap với các cây con bên trái và bên phải tạo thành khối, thủ tục này chuyển cây nói trên thành một khối*/.

1. Khởi động biến logic Done tại giá trị sai và chỉ số c (child) tại giá trị $2 * r$

2. While not Done and $c \leq n$ thực hiện các bước sau:

/* Tìm con lớn nhất */

a) If $c < n$ and $\text{Heap}[c] < \text{Heap}[c+1]$ then đặt c bằng $c+1$

/* Hoán vị một nút với con lớn nhất nếu cần thiết rồi chuyển xuống cây con tiếp theo*/

b) If $\text{Heap}[r] < \text{Heap}[c]$ then:

i. Hoán vị $\text{Heap}[r]$ với $\text{Heap}[c]$

ii. Đặt r bằng c

iii. Đặt c bằng $2 * c$

Else

Đặt Done bằng giá trị đúng.

Bây giờ ta có thể viết một cách dễ dàng giải thuật chuyển một cây nhị phân đầy đủ bất kỳ sang khối:

HEAPIFY

/* Giải thuật này chuyển một cây nhị phân đầy đủ lưu trữ ở các vị trí 1...n của mảng Heap thành một khối*/

Để cho $r = n \text{ div } 2$ đến 1 thực hiện:

/* bắt đầu từ nút cuối cùng không phải là lá */

Áp dụng thủ tục SwapDown cho cây được lưu trữ trong các vị trí từ r đến n của Heap

Giải thuật dưới đây tổng kết sơ đồ sắp xếp HeapSort đơn giản nhưng hiệu quả này:

HEAPSORT

/* Giải thuật này dùng giải thuật heapsort để sắp xếp n phần tử trong mảng $X[1], X[2], \dots, X[n]$ theo thứ tự tăng dần */

1. Xem X như là cây nhị phân đầy đủ và dùng giải thuật Heapify để chuyển cây này sang khối.

2. Để cho $i = n$ đến 2, thực hiện các bước sau:

a) Hoán vị $X[1]$ và $X[i]$, bằng cách này ta đặt phần tử lớn nhất trong danh sách con $X[1], \dots, X[i]$ vào đầu danh sách ấy.

b) Dùng giải thuật SwapDown để chuyển cây nhị phân tương ứng với danh sách con lưu trữ ở các vị trí từ 1 đến $i - 1$ của X .

5. Quicksort

Trong các sơ đồ sắp xếp trước đây, ý tưởng cơ bản là chọn phần tử nhỏ nhất hay lớn nhất trong một danh sách con nào đó của danh sách và đặt nó vào đúng vị trí trong danh sách con. Quicksort cũng là giải thuật chọn một mục và định vị đúng cho mục ấy. Tuy nhiên trong sơ đồ này, mục được chọn không nhất thiết phải là phần tử nhỏ nhất hay lớn nhất; trong thực tế nó hoạt động tốt nhất nếu phần tử này không phải là một trong các giá trị biên nói trên, mà là một giá trị nào đó ở giữa danh sách con.

Phần tử được chọn sẽ được định vị cho đúng bằng cách sắp xếp lại danh sách hay danh sách con sao cho tất cả những phần tử bên trái của phần tử này nhỏ hơn hay bằng nó và tất cả những phần tử bên phải lớn hơn nó. Như vậy danh sách (hoặc danh sách con) được chia thành những danh sách con nhỏ hơn và cũng bằng cách đó mỗi danh sách con được sắp xếp một cách độc lập. Một cách tự nhiên, phương pháp "chia để trị" này dẫn đến một thuật toán sắp xếp đệ quy.

7.2.3. Giải thuật trong STL

Như chúng ta đã thấy, để ứng dụng một giải thuật nào đó dù là đơn giản nhưng cũng phải ít nhất là lập chương trình con với các nam số khác nhau. Trong thực tế chúng ta còn phải sử dụng rất nhiều giải thuật khác nhau để giải quyết các bài toán khác nhau. Rất may, trong STL, một giải thuật là một hàm cho trước để làm việc gì đó cho các mục dữ liệu bên trong một côngtenơ (hay trong các côngtenơ).

Chú ý là các giải thuật trong STL không phải là các hàm thành viên hay thậm chí là các hàm bạn của các lớp côngtenơ. Chúng ta có thể sử dụng các giải thuật với các mảng có sẵn trong C++ hoặc với các lớp côngtenơ tự tạo (cho các lớp các hàm cơ bản xác định).

Bảng 7-5 trình bày một vài giải thuật đại diện.

Bảng 7-5. Một vài giải thuật STL tiêu biểu

Giải thuật	Chức năng
Find	Trả về giá trị đầu tiên tương ứng với một giá trị cụ thể.
Count	Đếm số phần tử có giá trị được chỉ rõ.
Equal	So sánh nội dung của hai côngtenơ và trả về true nếu tất cả các phần tử tương ứng bằng nhau.
Search	Tìm kiếm một dãy các phần tử trong một côngtenơ mà tương ứng với cùng dãy đó trong một côngtenơ khác.
Copy	Sao chép một dãy các giá trị từ một côngtenơ tới một côngtenơ khác hoặc tới một vị trí khác trong cùng một côngtenơ.
Swap	Đổi chỗ một giá trị ở một nơi với một giá trị ở nơi khác.
Iter_swap	Đổi chỗ một dãy giá trị ở một nơi với một dãy giá trị ở một nơi khác.
Fill	Sao chép một giá trị vào một dãy các vị trí.
Sort	Sắp xếp các giá trị trong một côngtenơ theo một thứ tự được chỉ rõ.
Merge	Kết nối hai khoảng phần tử được sắp xếp thành một khoảng phần tử được sắp xếp lớn hơn.
Accumulate	Trả về tổng số phần tử trong một khoảng cho trước.
For_each	Thực hiện một hàm được chỉ rõ cho từng phần tử trong một côngtenơ.

Giả sử chúng ta tạo một mảng kiểu **int** có dữ liệu trong đó:

```
int arr[]={42,31,7,80,2,26,19,75};
```

Khi đó chúng ta có thể dùng giải thuật `sort()` để sắp xếp mảng này:

```
sort(arr,arr+8);
```

Ở đây `arr` là địa chỉ đầu của mảng và `arr+8` là địa chỉ quá cuối mảng (một mục quá cuối mảng).

7.3. CON TRỎ

Các **con trỏ** (iterators) là các thực thể giống như các con trỏ được sử dụng để truy nhập các mục dữ liệu (nó thường được gọi là các phần tử) trong một côngtenơ. Thông thường, chúng được sử dụng

để di chuyển liên tiếp từ phần tử này sang phần tử kia. Một quá trình được gọi là **trở** (iterating) qua côngtenơ. Có thể tăng một **iterator** với các toán tử ++ để nó trở tới phần tử tiếp theo và có thể tham chiếu ngược (dereference) nó với toán tử * để lấy giá trị của phần tử nó trở tới.

Giống như một con trỏ trở tới một phần tử mảng, một vài kiểu **iterator** có thể lưu trữ (hoặc nhớ) vị trí của một phần tử côngtenơ cụ thể. Trong STL, một **iterator** biểu diễn bằng một đối tượng của một lớp **iterator**.

Các **iterator** khác nhau phải được dùng với các kiểu côngtenơ khác nhau. Có ba lớp **iterator** chính là: **iterator** tiến (**forward iterator**), **iterator** hai chiều (**bidirectional iterator**) và **iterator** truy nhập ngẫu nhiên (**random-access iterator**). Một **iterator** tiến chỉ có thể di chuyển tiến về phía trước qua côngtenơ mỗi lần một mục dữ liệu. Toán tử ++ của nó thực hiện việc này. Nó không thể di chuyển lùi lại và không thể tới một vị trí tùy ý ở giữa côngtenơ. Một **iterator** hai chiều có thể di chuyển cả tiến và lùi, bởi vậy nó định nghĩa cả hai toán tử ++ và --. Một **iterator** truy nhập ngẫu nhiên, ngoài khả năng di chuyển tiến và lùi, nó có thể nhảy tới một vị trí tùy ý trong côngtenơ. Ví dụ, chúng ta có thể bảo nó truy nhập tới vị trí 27.

Cũng có hai loại **iterator** đặc biệt **iterator** vào (**input iterator**) và **iterator** ra (**output iterator**). Một **iterator** vào có thể "trở tới" một thiết bị vào (**cin** hoặc **file**) để đọc các mục dữ liệu liên tiếp vào một côngtenơ. Một **iterator** ra có thể "trở tới" một thiết bị ra (như **cout** hoặc **file**) và ghi các phần tử từ một côngtenơ tới thiết bị đó.

Mặc dù các giá trị của các **iterator** tiến, **iterator** hai chiều, **iterator** truy nhập ngẫu nhiên có thể được lưu trữ (để chúng có thể được sử dụng sau đó), nhưng các giá trị của **iterator** vào và ra không thể lưu trữ. Điều này có lý, bởi vì ba **iterator** đầu trở tới các vùng nhớ, trái lại **iterator** vào và ra trở các thiết bị I/O nên việc lưu trữ các giá trị "con trỏ" là không có ý nghĩa. Bảng 7-6 trình bày các đặc điểm của ba loại **iterator** khác nhau này.

Bảng 7-6. Các đặc điểm của iterator

Iterator	Đọc/ghi	Iterator có thể lưu trữ ?	Hướng	Truy nhập
Random-access	Đọc và ghi	Có	Tiến và lùi	Ngẫu nhiên
Bidirectional	Đọc và ghi	Có	Tiến và lùi	Tuyến tính
Forward	Đọc và ghi	Có	Chỉ tiến	Tuyến tính
Output	Chỉ ghi	Không	Chỉ tiến	Tuyến tính
Input	Chỉ đọc	Không	Chỉ tiến	Tuyến tính

7.4. VẤN ĐỀ TIỀM ẨN VỚI STL

STL vẫn đang được tinh chế và các nhà sản xuất vẫn đang đưa ra các sản phẩm để làm việc với các trình biên dịch cụ thể, bởi vậy các vấn đề mà chúng ta đề cập ở đây có thể đã được giải quyết xong rồi. Tuy nhiên, cũng cần đưa ra một vài khía cạnh của STL yêu cầu cần thận. Các vấn đề này được phát sinh từ sự khác nhau về tư duy lớp mẫu trên nhiều trình biên dịch.

Thứ nhất, đôi khi khó tìm lỗi bởi vì trình biên dịch mô tả chúng ở sâu trong các file tiêu đề khi chúng thực sự ở trong chương trình của người sử dụng lớp.

Sự biên dịch trước của các file tiêu đề, mà làm tăng tốc mạnh mẽ sự biên dịch trên các trình biên dịch có nó, có thể gây ra các vấn đề với STL. Nếu mọi cái dường như không làm việc thì bỏ việc biên dịch trước các file tiêu đề.

.STL sinh ra rất nhiều các cảnh báo biên dịch từ sâu trong các file tiêu đề STL. Thường hay gặp cảnh báo **conversion may lose significant digits**. Những cảnh báo này không có gì hại và có thể được bỏ qua.

Không để ý tới những phiền phức nhỏ thì STL là một hệ thống mạnh và linh hoạt đến ngạc nhiên. Các lỗi có xu hướng được phát hiện tại thời điểm biên dịch hơn là thời điểm chạy chương trình. Các giải thuật và các côngtenơ khác nhau biểu hiện một giao diện rất nhất quán. Những gì làm việc với một côngtenơ hay một giải thuật này thường sẽ làm việc với một côngtenơ hay một giải thuật khác.

CÁC GIẢI THUẬT

Các giải thuật STL thực hiện các thao tác trên các bộ dữ liệu. Các giải thuật (ALGORITHMS) này được thiết kế để làm việc với các côngtenơ STL, nhưng một trong những điều thú vị về chúng là chúng ta có thể áp dụng chúng cho các mảng C++ thông thường. Điều này có thể tiết kiệm được một lượng công việc khá lớn khi lập trình với các mảng. Nó cũng cung cấp một cách học dễ dàng về các giải thuật, không vướng víu gì tới các côngtenơ. Trong chương này chúng ta sẽ nói tới cách sử dụng một vài giải thuật đại diện hay dùng.

8.1. GIẢI THUẬT FIND()

Giải thuật tìm kiếm phần tử đầu tiên trong một côngtenơ mà có giá trị được chỉ rõ. Chương trình ví dụ FIND sẽ cho thấy giải thuật làm việc như thế nào khi tìm kiếm một giá trị trong một mảng.

Listing 8-1 FIND

```
//find.cpp
//tìm phần tử đầu tiên có giá trị được chỉ rõ
#include<iostream.h>
#include<algorithm>
void main()
{
    int arr[]={11,22,33,44,55,66,77,88};
    int* ptr;

    ptr=find(arr,arr+8,33);    //tìm phần tử đầu tiên có giá trị 33
    cout<<"Phần tử đầu tiên có giá trị 33 được tìm thấy ở vị trí: "
        <<(ptr-arr)<<endl;
}
```

Kết quả đưa ra là:

Như thường lệ phần tử đầu tiên trong mảng là số 0, bởi vậy 3 được tìm thấy ở vị trí 2.

8.1.1. File tiêu đề

Trong chương trình FIND chúng ta sử dụng file tiêu đề ALGORITHM. Trong phiên bản STL mà chúng ta sử dụng ở đây, file tiêu đề này chứa các khai báo của tất cả các giải thuật. Các file tiêu đề khác được dùng cho các côngtenơ. Với phiên bản STL của hãng cung cấp khác, có thể phải sử dụng một file tiêu đề khác cho các giải thuật. Một vài trình biên dịch chỉ cài đặt một file tiêu đề cho tất cả các hoạt động STL, trái lại một vài trình biên dịch khác lại cài đặt nhiều file và gọi chúng bằng các tên khác nhau. Ở đây chúng ta sử dụng trình biên dịch **BorlandC++ 5.0**, file tiêu đề của nó cho các giải thuật là ALGORITHM (không có đuôi .H). Các trình biên dịch khác có thể là ALGO.H hoặc ALGORITHM.H hoặc là ALGORITH (không có .H).

8.1.2. Khoảng giới hạn

Hai tham số đầu tiên cho giải thuật `find()` xác định khoảng phần tử được kiểm tra. Các giá trị này được xác định bởi các **iterator**. Trong ví dụ FIND chúng ta sử dụng **iterator** là các giá trị con trỏ C++ thông thường, chúng là trường hợp đặc biệt của các **iterator**.

Tham số là **iterator** của (trong trường hợp này là con trỏ trỏ tới) giá trị đầu tiên được kiểm tra. Tham số thứ hai là **iterator** của vị trí đứng sau phần tử cuối cùng được kiểm tra. Bởi vì có 8 phần tử, giá trị này là giá trị đầu tiên cộng thêm 8. Nó được gọi là giá trị *past-the-end*. Nó trỏ tới phần tử ngay sau cuối khoảng được kiểm tra.

Cú pháp này gọi lại thuật ngữ C++ thông thường trong một vòng lặp **for**:

```
for(int j=0;j<8;j++)
{
    if (arr[j]==33)
    {
        cout<<"Phan tu dau tien co gia tri 33 tim thay o vi tri: "
        <<j<<endl;
        break;
    }
}
```

Trong ví dụ **FIND**, giải thuật **find()** tiết kiệm được việc viết vòng lặp **for**. Trong các tình huống phức tạp hơn, các giải thuật có thể tiết kiệm được việc viết các lệnh phức tạp.

8.2. GIẢI THUẬT COUNT()

Chúng ta cùng xét một giải thuật khác là **count()**, nó đếm xem có bao nhiêu phần tử trong một côngtenơ có giá trị được chỉ rõ. Giải thuật này, thay vì trả về một giá trị **iterator** (con trỏ) như giải thuật **find()**, nó cộng số đếm vào một biến nguyên được cung cấp như một đối số cho **count()**. Ví dụ **COUNT** sau đây cho thấy cách làm này.

Listing 8-2 COUNT

```
//count.cpp
//dem so phan tu co gia tri duoc chi ro
#include<iostream.h>
#include<algorithm>
void main()
{
    int arr[]={33,22,33,44,33,55,66,77};
    int n=0;

    count(arr,arr+8,33,n);    //dem so 33, cong ket qua vao n
    cout<<"Co "<<n<<" so 33 trong mang "<<endl;
}
```

Đây là kết quả đưa ra:

Co ba so 33 trong mang.

8.3. GIẢI THUẬT SORT()

Giải thuật **sort()** sắp xếp các phần tử của một côngtenơ theo một thứ tự nào đó. Bản 8-3 là một ví dụ về giải thuật này được áp dụng cho mảng. Nó sắp xếp các số trong mảng theo chiều tăng.

Listing 8-3 SORT

```
//sort.cpp
//sap xep mot mang cac so nguyen
#include<iostream.h>
```



```

#include<algorithm>
void main()
{
    int arr[]={45,2,22,-17,0,-30,25,77};
    int n=0;

    sort(arr,arr+8);    //sap xep cac so
    for(int j=0;j<8;j++) //hien thi
        cout<<arr[j]<<' ';
}

```

Đây là kết quả đưa ra của chương trình:

-30 -17 0 2 25 45 77

8.4. GIẢI THUẬT SEARCH()

Có một vài giải thuật thao tác trên hai côngtenơ cùng một lúc. Ví dụ, trái với giải thuật **find()** tìm kiếm một giá trị cho trước trong một côngtenơ, giải thuật **search()** tìm kiếm một dãy các giá trị cho bởi một côngtenơ trong một côngtenơ khác. Ví dụ SEARCH (bản 8-4) cho thấy **search()** hoạt động như thế nào.

Listing 8-4 SEARCH

```

//search.cpp
//tim kiem mot day gia tri cho boi mot container trong mot container khac
#include<iostream.h>
#include<algorithm>
void main()
{
    int source[]={11,44,33,11,22,33,11,22,44};
    int pattern[]={11,22,33};
    int* ptr;
    ptr=search(source,source+9,pattern,pattern+3);
    if(ptr==source+9)
        cout<<"\nKhong tim thay su phu hop";
    else
        cout<<"\nPhu hop tai vi tri "<<(ptr-source);
}

```

Giải thuật **search()** tìm kiếm dãy số 11,22,33, cho bởi mảng **pattern**, trong mảng **source**. Ta thấy dãy này được tìm thấy trong **source** tại vị trí thứ tư (phần tử 3). Đây là kết quả đưa ra từ chương trình:

Phu hop tai vi tri 3

Nếu con trỏ **ptr** kết thúc quá cuối của **source** thì sẽ không tìm thấy sự phù hợp. Các đối số cho giải thuật như **search()** không cần cùng kiểu côngtenơ. **Source** có thể tìm thấy trong một hằng sản xuất STL còn **pattern** có thể thấy trong một mảng. Dạng tổng quát này là một đặc điểm rất quan trọng của STL.

8.5. GIẢI THUẬT MERGE()

Giải thuật **merger()** dùng để trộn các côngtenơ vào một côngtenơ. Bản 8-5 trình bày một chương trình trộn các phần tử của hai côngtenơ nguồn vào một côngtenơ đích.

Listing 8-5 MERGE

```
//merge.cpp
//trộn hai container vào một container thu ba
#include<iostream.h>
#include<algorithm>
void main()
{
    int src1[]={2,3,4,6,8};
    int src2[]={1,3,5};
    int dest[8];
    merge(src1,src1+5,src2,src2+3,dest);    //trộn các số nguyên
    cout<<"\nCac số nguyên trong đây đã trộn là:\n";
    for(int j=0;j<8;j++)
        cout<<dest[j]<<' ';
}
```

Kết quả đưa ra là:

1 2 3 3 4 5 6 8

Chúng ta có thể thấy việc trộn dữ liệu giữ nguyên thứ tự, hợp hai dãy phần tử nguồn vào một container đích. Chú ý là trước khi trộn, hai container nguồn phải được sắp xếp theo cùng một thứ tự.

8.6. ĐỐI TƯỢNG HÀM

Một vài giải thuật có thể có một cái gì đó gọi là một đối tượng hàm làm đối số. Một đối tượng hàm, đối với người sử dụng, trông rất giống một hàm mẫu. Tuy nhiên nó thực sự là một đối tượng của một lớp mẫu có một hàm thành viên đơn là **toán tử chống** (). Điều này dường như kỳ lạ nhưng lại dễ sử dụng.

Cho rằng chúng ta muốn sắp xếp một mảng số theo thứ tự giảm dần thay vì tăng dần. Chương trình SORTEMP (bản 8-6) cho thấy cách làm này.

Listing 8-6 SORTEMP

```
//sortemp.cpp
//sắp xếp mảng số float theo thứ tự ngược lại (giảm dần)
#include<iostream.h>
#include<algorithm>
void main()
{
    float fdata[]={19.2,87.4,33.6,55.0,11.5,42.2};
    //sắp xếp các số float theo thứ tự tăng dần
    sort(fdata,fdata+6,greater<float>());
    cout<<"\nSau khi sắp xếp:\n";
    for(int j=0;j<6;j++)
        cout<<fdata[j]<<endl;
}
```

Mảng các giá trị float được sắp xếp dùng đối tượng hàm `greater<float>()`. Đây là kết quả đưa ra:

87.4
55
42.2
33.6
19.1
11.5

Ngoài các đối tượng so sánh, còn có các đối tượng hàm cho các phép toán số học và logic. Chúng ta sẽ xem xét kỹ các đối tượng hàm trong chương 14.

8.7. HÀM VIẾT BỞI NGƯỜI SỬ DỤNG ĐẶT Ở VỊ TRÍ ĐỐI TƯỢNG HÀM

Các đối tượng hàm chỉ tính toán trên các kiểu dữ liệu cơ bản C++ và trên các lớp có định nghĩa các toán tử thích hợp (+,<,&&,v.v...). Nếu làm việc với các giá trị không phải ở trong các trường hợp này thì chúng ta phải thay thế một đối tượng hàm bằng một hàm được định nghĩa bởi người sử dụng. Ví dụ, toán tử < không được định nghĩa cho các chuỗi **char*** thông thường nhưng chúng ta có thể viết một hàm thực hiện việc so sánh và sử dụng địa chỉ của hàm này (tên của nó) thay cho đối tượng hàm. Ví dụ SORTCOM (bản 8-7) cho thấy cách sắp xếp một mảng chuỗi **char***.

Listing 8-7 SORTCOM

```
//sortcom.cpp
//sắp xếp mảng chuỗi với hàm so sánh viết bởi người sử dụng
#include<iostream.h>
#include<string.h>
#include<algorithm>

char* names[]={"George","Penny","Estelle","Don","Mike","Bob"};
bool alpha_comp(char*,char*);
void main()
{
    sort(names,names+6,alpha_comp); //sắp xếp các chuỗi
    cout<<"\nSau khi sắp xếp:\n";
    for(int j=0;j<6;j++)
        cout<<names[j]<<endl;
}
bool alpha_comp(char* s1,char* s2) //trả về true nếu s1<s2
{
    return (strcmp(s1,s2)<0) ? true:false;
}
```

Đối số thứ ba cho giải thuật **sort()** là địa chỉ của hàm **alpha_comp()**, nó so sánh hai chuỗi **char*** và trả về **true** hoặc **false** tùy thuộc vào liệu đối số thứ nhất có nhỏ hơn đối số thứ hai không (theo thứ tự alphabe). Kết quả đưa ra như chúng ta mong đợi:

```
Bob
Don
Estelle
George
Mike
Penny
```

Nếu sử dụng lớp **string** từ thư viện chuẩn thì chúng ta không cần viết hàm mà có thể sử dụng các hàm có sẵn **less<>()** thay cho **greater<>()**.

8.8. THÊM _IF VÀO CÁC GIẢI THUẬT

Một vài giải thuật có các phiên bản kết thúc bằng **_if**. Các giải thuật này có một tham số phụ gọi là **predicate** (dự đoán), nó là một đối tượng hàm hoặc một hàm. Ví dụ, giải thuật **find()** tìm tất cả các phần tử bằng một giá trị cho trước. Tuy nhiên, nó chỉ làm điều này nếu toán tử bằng (==) được định nghĩa cho kiểu dữ liệu của đối tượng được tìm kiếm. Nếu toán tử == không được định

nghĩa, chúng ta có thể tạo một hàm dấu bằng == làm việc với giải thuật **find_if()** để tìm phần tử có bất kỳ đặc điểm tùy ý nào.

Bản 8-8 sử dụng các chuỗi **char***. Giải thuật **find_if()** được cung cấp một hàm viết bởi người sử dụng là **isDon()** để tìm chuỗi đầu tiên trong mảng có giá trị "Don".

Listing 8-8 FIND_IF

```
//find_if.cpp
//tìm kiếm trong mảng tên đầu tiên phù hợp với "Don"
#include<iostream.h>
#include<string.h>
#include<algorithm>

char* names[]={"George","Estelle","Don","Mike","Bob"};
bool isDon(char*);
void main()
{
    char** ptr;

    ptr=find_if(names,names+5,isDon);           //tìm chuỗi đầu tiên là "Don"
    if(ptr==names+5)
        cout<<"\nDon không có tên trong danh sách";
    else
        cout<<"\nDon phần tử "<<(ptr-names)<<" trên danh sách";
}
bool isDon(char* name)
{
    return (strcmp(name,"Don")) ? false:true;
}
```

Bởi vì "Don" thực sự là một tên trong mảng, kết quả đưa ra sẽ là:

Don là phần tử 2 trên danh sách

Địa chỉ của hàm **isDon()** là đối số thứ ba của **find_if()**, trong khi đó đối số thứ nhất và thứ hai, như thường lệ, là địa chỉ đầu tiên và địa chỉ quá cuối của mảng.

Giải thuật **find_if()** áp dụng hàm **isDon()** cho mọi phần tử trong mảng. Nếu **isDon()** trả về **true** cho phần tử nào đó **find_if()** trả về giá trị con rỏ của phần tử đó. Nếu không, **find_if()** trả về con trỏ tới địa chỉ quá cuối của mảng (the past-the-end).

Nhiều giải thuật khác như **count()**, **replace()** và **remove()** cũng có phiên bản **_if**.

8.9. GIẢI THUẬT FOR_EACH()

Giải thuật **for_each()** cho phép ta làm cái gì đó với mọi mục trong một côngtenơ. Chúng ta có thể viết một hàm để xác định "cái gì đó" là gì. Hàm mà chúng ta viết không thể thay đổi các phần tử trong côngtenơ, nhưng nó có thể sử dụng hoặc hiển thị các giá trị của chúng.

Đây là một ví dụ trong đó **for_each()** được sử dụng để chuyển tất cả các giá trị của một mảng từ inch sang centimeter và hiển thị chúng. Hàm mà chúng ta viết có tên là **in_to_cm()**, nó nhận một giá trị với 2.54. Địa chỉ của hàm được dùng làm đối số thứ ba cho **for_each()**. Bản 8-9 trình bày chương trình **FOR_EACH**.

Listing 8-9 FOR_EACH

```
//for_each.cpp
//su dung for_each() de dua ra cac gia tri mang in la centimeters
#include<iostream.h>
#include<algorithm>
void main()
{
    float inches[]={3.5,6.2,1.0,12.75,4.33};
    void in_to_cm(float);           //khai bao ham
    for_each(inches,inches+5,in_to_cm);
}
void in_to_cm(float in)
{ cout<<(in*2.54)<<' '; }
```

Kết quả đưa ra như sau:

8.89 15.748 2.54 32.385 10.9982

8.10. GIẢI THUẬT TRANSFORM()

Giải thuật **transform()** làm một việc gì đó với tất cả các mục dữ liệu trong một côngtenơ và đặt các giá trị kết quả trong một côngtenơ khác (hay một côngtenơ giống như vậy). Ngoài ra, một hàm viết bởi người sử dụng sẽ xác định là làm gì với các mục dữ liệu của côngtenơ. Kiểu trả về của hàm này phải giống kiểu của côngtenơ đích. Ví dụ sau tương tự như ví dụ FOREACH nhưng thay vì hiển thị giá trị được chuyển đổi, hàm **in_to_cm()** đưa các giá trị centimeter vào một mảng khác, **centi[]**. Sau đó chương trình chính hiển thị nội dung của mảng **centi[]**. Bản 8-10 trình bày chương trình TRANSFO.

Listing 8-10 TRANSFO

```
//transfo.cpp
//su dung transform() de chuyen doi mang cac gia tri inch sang centimeter
#include<iostream.h>
#include<algorithm>

void main()
{
    float inches[]={3.5,6.2,1.0,12.75,4.33};
    float centi[5];

    void in_to_cm(float); //khai bao ham
                          //chuyen doi va dua vao mang centi[]
    transform(inches,inches+5,centi,in_to_cm);
}
void in_to_cm(float in)
{
    return (in*2.54);
}
```

Kết quả đưa ra giống như chương trình FOR_EACH.

Chúng ta vừa trình bày một vài giải thuật trong STL. Còn có nhiều giải thuật khác mà chúng ta không thể nói hết ở đây được.

ITERATORS

Các **iterator** (con trỏ) dường như hơi kỳ lạ nhưng chúng lại là trung tâm đối với các hoạt động của STL. Trong chương này chúng ta sẽ tìm hiểu hai vai trò của **iterator** là: vai trò con trỏ thông minh (**smart iterator**) và vai trò kết nối giữa các giải thuật và các côngtenơ. Một số ví dụ trong chương này sử dụng các côngtenơ tuần tự, nó được trình bày ở chương 10.

9.1. CON TRỎ THÔNG MINH

Thực hiện một thao tác trên tất cả các phần tử trong một côngtenơ (hoặc có thể là một khoảng phần tử trong một côngtenơ) thường là cần thiết. Ví dụ, hiển thị giá trị của tất cả các phần tử trong côngtenơ hoặc cộng giá trị của nó tới một tổng. Trong một mảng C++ thông thường, các thao tác như vậy được thực hiện nhờ con trỏ (hoặc toán tử []). Ví dụ, đoạn mã sau đây trỏ qua một mảng số **float** để hiển thị giá trị của từng phần tử:

```
float* ptr=start_address;
for(int j=0;j<size;j++)
    cout<<*ptr++;
```

Với các côngtenơ phức tạp, các con trỏ C++ thông thường có những bất tiện. Vì một điều là việc quản lý con trỏ trở nên quá phức tạp; chúng ta không thể tăng nó bằng toán tử ++ để nó trỏ tới phần tử tiếp theo. Ví dụ, trong việc di chuyển tới một mục dữ liệu tiếp theo trong một danh sách liên kết, chúng ta không thể coi mục dữ liệu đó ở cạnh mục dữ liệu trước nó; chúng ta phải đi theo một chuỗi các con trỏ thì mới tới được mục dữ liệu đó.

Có thể chúng ta muốn lưu giữ địa chỉ của một phần tử côngtenơ nào đó trong một biến con trỏ để có thể di chuyển tới phần tử đó tại một thời điểm nào đó trong tương lai. Chuyện gì sẽ xảy ra nếu thực hiện việc chèn và xóa ở giữa côngtenơ đó? Giá trị của con trỏ có thể sẽ không còn hợp lệ nếu nội dung của côngtenơ được sắp xếp lại. Thật thuận lợi nếu không phải lo lắng về việc sửa đổi tất cả các giá trị mà các con trỏ lưu trữ khi xảy ra việc chèn hoặc xóa.

Một giải pháp cho những vấn đề này là tạo một lớp các "con trỏ thông minh" (**smart iterator**). Đối tượng của một lớp như vậy chủ yếu bao các hàm thành viên của nó quanh một con trỏ thông thường. Các toán tử ++ và * được chồng để chúng biết cách trỏ tới phần tử trong côngtenơ của chúng cho dù các phần tử đó không nằm cạnh nhau trong bộ nhớ hoặc vị trí của chúng bị thay đổi. Đây là dạng khung của cách làm này:

```
class SmartPointer
{
private:
    float* p;           //mọt con tro thong thuong
public:
    float operator*()
        {}
    float operator++()
        {}
};
void main()
{
    ....
```

```

SmartPointer sptr=start_address;
for(int j=0;j<size;j++)
    cout<<*sptr++;
}

```

Cho rằng chúng ta cần tạo một lớp các con trỏ thông minh, vậy trách nhiệm tạo lớp này thuộc về ai? Với tư cách là người sử dụng lớp, chắc chắn chúng ta không muốn viết thêm đoạn chương trình phức tạp này.

Mặt khác, có một vấn đề với việc tạo các thành viên con trỏ thông minh của lớp côngtenơ. Chúng ta có thể cần nhiều con trỏ trong một côngtenơ và sẽ là phức tạp cho một côngtenơ để lưu trữ các giá trị này. Nên có bao nhiêu con trỏ? Mỗi côngtenơ nên có ba con trỏ cho các phần tử của riêng nó? Hay nên lưu trữ một bảng các con trỏ như vậy? Chương trình ứng dụng có thể tạo một con trỏ như vậy bất kỳ lúc nào nó cần, không có hạn chế hay phức tạp gì.

Cách lựa chọn của STL là tạo các con trỏ thông minh (**smart iterator**) gọi là **iterator**, đưa vào một lớp hoàn toàn riêng biệt (thường là một họ các lớp mẫu). Người sử dụng tạo các **iterator** bằng cách định nghĩa chúng.

9.2. ITERATOR NHƯ MỘT GIAO DIỆN

Ngoài việc đóng vai trò như các con trỏ thông minh trỏ tới các mục dữ liệu trong các côngtenơ, các **iterator** còn có các mục đích khác trong STL. Chúng quyết định giải thuật nào sẽ được sử dụng với côngtenơ nào. Tại sao điều này lại là cần thiết?

Trong một vài ngữ cảnh lý thuyết, chúng ta có thể áp dụng mọi giải thuật cho mọi côngtenơ. Trong thực tế nhiều giải thuật làm việc với tất cả các côngtenơ STL. Tuy nhiên, có một vài giải thuật sẽ không hiệu quả (chậm) khi được dùng với một vài côngtenơ. Ví dụ, giải thuật **sort()** cần sự truy nhập ngẫu nhiên tới các côngtenơ mà nó sắp xếp; nếu không nó phải trỏ qua côngtenơ để tìm từng phần tử trước khi di chuyển nó, cách này rất tốn thời gian.

Các **iterator** cho ta một cách thức giản đơn đáng ngạc nhiên để làm phù hợp các giải thuật với các côngtenơ. Chúng ta có thể coi một **iterator** như một dây cáp cắm vào một côngtenơ và đầu kia cắm vào một giải thuật. Tuy nhiên, không phải tất cả các dây cáp đều cắm được vào tất cả các côngtenơ và không phải tất cả các dây cáp đều cắm được vào tất cả các giải thuật. Nếu cố dùng một giải thuật quá mạnh cho một côngtenơ cho trước thì có thể sẽ không tìm thấy một dây cáp (một **iterator**) để nối chúng với nhau. Nếu cố sử dụng nó chúng ta sẽ nhận được lỗi biên dịch báo cho biết có vấn đề.

Chúng ta cần bao nhiêu **iterator** để cho hệ thống này làm việc? Như đã nói trước đây, chỉ cần năm kiểu là đủ. Bảng 9-1 trình bày năm loại **iterator** này, chúng được sắp xếp từ dưới lên trên theo thứ tự mạnh dần, trừ **input** và **output** là ngang nhau (chú ý, đây không phải là một sơ đồ kế thừa).

Nếu một giải thuật chỉ cần đi qua một côngtenơ theo một chiều tiến, đọc (nhưng không ghi) từng mục dữ liệu, thì nó có thể sử dụng **iterator input** để nối nó với côngtenơ. Nếu giải thuật đi qua theo chiều tiến nhưng lại ghi tới côngtenơ thay vì đọc thì nó có thể dùng **iterator output**. Nếu nó đi qua côngtenơ theo chiều tiến và cả đọc lẫn ghi thì nó phải sử dụng một **iterator forward** (**iterator** tiến). Và nếu nó phải truy nhập bất kỳ mục dữ liệu nào trong một côngtenơ ngay lập tức, không trỏ qua côngtenơ, thì nó phải sử dụng một **iterator random-access** (**iterator** truy nhập ngẫu nhiên). Bảng 9-2 trình bày các hoạt động mà từng côngtenơ trợ giúp.

Như chúng ta có thể thấy, tất cả các **iterator** đều trợ giúp toán tử ++ để bước tiến qua côngtenơ. **Iterator input** cũng sử dụng toán tử * bên phải dấu bằng (nhưng không phải bên trái): `v=*i;`

Bảng 9-1. Các khả năng của các loại iterator

Iterator	Tiến ++	Đọc v=*j	Ghi *j=v	Lùi --	Truy nhập ngẫu nhiên [n]
Random-access	x	x	x	x	x
Bidirectional	x	x	x	x	
Forward	x	x	x		
Output	x		x		
Input	x	x			

Iterator output có thể sử dụng toán tử * nhưng chỉ dùng ở bên trái dấu bằng: *i=v;

Iterator tiến (Forward) quản lý cả đọc và ghi, **iterator hai chiều** (Bidirectional) có thể tăng cũng như giảm. **Iterator truy nhập ngẫu nhiên** (Random-access) có thể sử dụng toán tử [] để truy nhập tới bất kỳ phần tử nào một cách nhanh chóng.

Một giải thuật có thể luôn luôn sử dụng một **iterator** có khả năng nhiều hơn nó cần đến. Ví dụ, nếu nó cần một **iterator tiến** thì có thể sử dụng **iterator hai chiều** hoặc **iterator truy nhập ngẫu nhiên** cho nó.

9.2.1. Phù hợp các giải thuật với các côngtenơ

Chúng ta đã sử dụng một dây cáp để so sánh với một **iterator** bởi vì một **iterator** nối một giải thuật và một côngtenơ. Còn bây giờ chúng ta cùng tập trung vào hai đầu dây cáp tương tự như này: đầu côngtenơ và đầu giải thuật.

1. Cắm cáp vào một côngtenơ

Nếu chúng ta chỉ sử dụng các côngtenơ cơ bản thì chỉ cần hai loại **iterator** là đủ. Như chỉ ra trong bảng 9-2, **vector** và **deque** yêu cầu một **iterator truy nhập ngẫu nhiên**, trái lại **list**, **set**, **multiset**, **map** và **multimap** chỉ yêu cầu các **iterator hai chiều**.

Bảng 9-2. Các kiểu iterator được các côngtenơ dùng

Iterator	Vector	List	Deque	Set	Multiset	Map	Multimap
Random-access	x		x				
Bidirectional	x	x	x	x	x	x	x
Forward	x	x	x	x	x	x	x
Input	x	x	x	x	x	x	x
Output	x	x	x	x	x	x	x

STL thực hiện việc sử dụng **iterator** đúng với một côngtenơ như thế nào? Đó là khi định nghĩa một **iterator**, chúng ta phải xác định loại côngtenơ sử dụng nó. Ví dụ, nếu chúng ta đã định nghĩa một danh sách giữ các phần tử kiểu **int**:

```
list<int> iList; //danh sách các số nguyên
```


Sau đó chúng ta định nghĩa một **iterator** cho danh sách này:

```
list<int>::iterator it;//iterator tro toi danh sach cac so nguyen.
```

Khi chúng ta làm điều này, STL tự động tạo **iterator** là một **iterator hai chiều** (bidirectional iterator) bởi vì đó là những gì mà một danh sách yêu cầu. Trái lại, một **iterator** cho một vectơ hoặc một hàng đợi hai đầu được tự động tạo là một **iterator truy nhập ngẫu nhiên** vì đây là những gì mà chính hai côngtenơ này yêu cầu.

Quá trình lựa chọn tự động được cài đặt, trong file tiêu đề **ITERATOR.H** và các file tiêu đề khác, bằng một lớp **iterator** cho một côngtenơ cụ thể được rút ra từ một lớp **iterator** tổng quát hơn thích hợp với một côngtenơ cụ thể. Do đó, các **iterator** cho vectơ và hàng đợi hai chiều được rút ra từ lớp **random_access_iterator**, trái lại các **iterator** cho danh sách lại được rút ra từ lớp **bidirectional_iterator**.

Bây giờ chúng ta đã thấy các côngtenơ được làm phù hợp với các dây cáp **iterator** như thế nào. Một dây cáp thực ra không cắm vào một côngtenơ, nó được nối cứng với côngtenơ. Các vectơ và hàng đợi hai chiều luôn luôn được nối với cáp **iterator hai chiều**.

2. Cắm cáp vào giải thuật

Chúng ta đã thấy một đầu của một cáp **iterator** được nối với một côngtenơ như thế nào, thế còn đầu kia thì như thế nào? Mỗi giải thuật, tùy thuộc vào những gì nó sẽ làm với các phần tử trong một côngtenơ, yêu cầu một loại **iterator** cụ thể. Nếu giải thuật phải truy nhập tới các phần tử ở các vị trí tùy ý trong côngtenơ thì nó yêu cầu một **iterator truy nhập ngẫu nhiên**, nó có thể sử dụng **iterator** tiến nhưng kém hơn. Bảng 9-3 trình bày một số giải thuật đại diện với các **iterator** mà chúng yêu cầu.

Bảng 9-3. Các kiểu iterator được yêu cầu bởi các giải thuật

Algorithm	Input	Output	Forward	Bidirectional	Random-access
For_each	x				
Find	x				
Count	x				
Copy	x	x			
Replace			x		
Unique			x		
Reverse				x	
Sort					x
nth_element					x
Merge	x	x			
Accumulate	x				

Ngoài ra, mặc dù mỗi giải thuật yêu cầu một **iterator** với một mức độ khả năng nhất định, nhưng một **iterator** mạnh hơn cũng sẽ làm việc được với nó. Giải thuật **replace()** yêu cầu **iterator** tiến nhưng nó cũng sẽ làm việc với một **iterator hai chiều** hoặc một **iterator truy nhập ngẫu nhiên**.

Bây giờ hãy tưởng tượng rằng các giải thuật có các bộ nối với các chân nhô ra. Giải thuật yêu cầu **iterator truy nhập ngẫu nhiên** có năm chân, giải thuật yêu cầu **iterator hai chiều** có bốn chân, giải thuật yêu cầu **iterator tiến** có ba chân v.v...

Phía đầu nối với giải thuật của một **iterator** (một dây cáp) có một bộ nối với một số lỗ xác định. Chúng ta có thể cắm một **iterator** có năm lỗ vào một giải thuật có năm chân và cũng có thể cắm nó vào giải thuật có bốn hoặc vài chân. Tuy nhiên, chúng ta không thể cắm một **iterator** có bốn lỗ (**iterator hai chiều**) vào một giải thuật có năm chân (yêu cầu truy nhập ngẫu nhiên). Bởi vậy, vectơ và hàng đợi hai đầu, có **iterator truy nhập ngẫu nhiên**, có thể cắm vào bất cứ giải thuật nào, trái lại danh sách và các côngtenơ liên kết, chỉ có các **iterator hai chiều** bốn lỗ chỉ có thể được cắm vào các giải thuật kém mạnh hơn.

9.2.2. Các bảng cho biết điều gì?

Từ bảng 9-2 và bảng 9-3 chúng ta có thể tìm ra một giải thuật làm việc cho một côngtenơ cho trước. Ví dụ, bảng 9-3 chỉ ra rằng giải thuật **sort()** yêu cầu một **iterator truy nhập ngẫu nhiên**, bảng 9-2 cho thấy côngtenơ duy nhất quản lý các **iterator truy nhập ngẫu nhiên** là vectơ và hàng đợi hai đầu. Bởi vậy, thật vô ích nếu cố áp dụng giải thuật **sort()** cho danh sách, tập hợp, ánh xạ...

Bất kỳ giải thuật nào không yêu cầu một **iterator truy nhập ngẫu nhiên** sẽ làm việc với tất cả các loại côngtenơ bởi vì tất cả các côngtenơ chỉ sử dụng **iterator hai chiều**, nó chỉ ở dưới **iterator truy nhập ngẫu nhiên** một bậc.

Một vài giải thuật so sánh yêu cầu **iterator truy nhập ngẫu nhiên** còn hầu hết các giải thuật làm việc với hầu hết các côngtenơ.

9.2.3. Sự trùng lặp các hàm thành viên và các giải thuật

Đôi khi chúng ta phải quyết định giữa việc sử dụng một hàm thành viên hay một giải thuật có cùng tên. Ví dụ, giải thuật **find()** chỉ yêu cầu một **iterator đọc (input iterator)**, bởi vậy nó có thể được dùng cho bất kỳ côngtenơ nào. Tuy nhiên, tập hợp và ánh xạ có hàm thành viên **find()** của riêng nó (không giống như các côngtenơ tuần tự). Vậy nên sử dụng phiên bản hàm **find()** nào? Nói chung, nếu có một phiên bản hàm thành viên thì phiên bản giải thuật sẽ không hiệu suất bằng. Trong các trường hợp này nên sử dụng các hàm thành viên.

9.3. ITERATOR LÀM VIỆC NHƯ THẾ NÀO?

9.3.1. Truy nhập dữ liệu trong côngtenơ

Trong các côngtenơ có các **iterator truy nhập ngẫu nhiên** (vectơ và hàng đợi hai đầu) có thể dễ dàng trở qua côngtenơ bằng toán tử **[]**. Các côngtenơ như danh sách không trợ giúp truy nhập ngẫu nhiên cần có một cách khác. Một cách hay sử dụng là định nghĩa một **iterator** cho côngtenơ. Chương trình LISTOUT cho cách này.

Listing 9-1 LISTOUT

```
//listout.cpp
//dùng iterator và vòng lặp for để đưa ra
#include<iostream.h>
#include<list>

void main()
```

```

{
    int arr[]={2,4,6,8};           //mang cac so nguyen
    list<int> iList(arr,arr+4);    //khoi tao danh sach toi mang
    list<int>::iterator it;        //iterator tro toi danh sach cac so nguyen

    for(it=iList.begin();it!=iList.end();it++)
        cout<<*it<<' ';
}

```

Chương trình chỉ đơn giản hiển thị nội dung của côngtenơ `iList`. Kết quả đưa ra là:

2 4 6 8

Trong chương trình chúng ta đã định nghĩa một **iterator** kiểu `list<int>` phù hợp với kiểu côngtenơ. Như với một biến con trỏ, chúng ta phải cho **iterator** một giá trị trước khi sử dụng nó. Trong vòng lặp `for`, `it` được khởi tạo bằng `iList.begin()`, điểm bắt đầu của côngtenơ. Có thể tăng `it` với toán tử `++` để nó trở qua các phần tử trong một côngtenơ và có thể tham chiếu ngược nó với toán tử `*` để lấy giá trị của từng phần tử mà nó trở tới. Cũng có thể so sánh bằng toán tử `!=` để có thể thoát khỏi vòng lặp khi nó tới côngtenơ ở `iList.end()`.

Một cách tương tự, sử dụng vòng lặp `while` thay cho `for`, có thể là:

```

it=iList.begin();
while(it!=iList.end())
    cout<<*it++<<' ';

```

9.3.2. Chèn dữ liệu vào côngtenơ

Chúng ta có thể sử dụng chương trình tương tự như trên để đặt dữ liệu vào các phần tử đã tồn tại trong một côngtenơ, như chỉ ra trong chương trình `LISTFILL` dưới đây.

Listing 9-2 LISTFILL

```

//listfill.cpp
//dung iterator de dien du lieu vao danh sach
#include<iostream.h>
#include<list>
void main()
{
    list<int> iList(5);           //danh sach rong giu 5 so nguyen
    list<int>::iterator it;       //iterator tro toi danh sach cac so nguyen
    int data=0;

                                //dat du lieu vao danh sach
    for(it=iList.begin();it!=iList.end();it++)
        *it=data+=2;
    it=iList.begin();             //hien thi danh sach
    while(it!=iList.end())
        cout<<*it++ <<' ';
}

```

Vòng lặp thứ nhất để điền các giá trị 2,4,6,8 vào côngtenơ, cho thấy toán tử chồng `*` làm việc bên trái dấu bằng cũng như bên phải ở ví dụ trước. Vòng lặp thứ hai hiển thị các giá trị này.

9.3.3. Các giải thuật và các iterator

Các giải thuật sử dụng các **iterator** như các đối số (đôi khi như các giá trị trả về). Ví dụ `ITERFIND` (bản 9-3) cho thấy giải thuật `find()` được dùng với danh sách. Chúng ta có thể dùng `find()` với danh sách bởi vì nó yêu cầu một **iterator đọc** (input iterator).

Listing 9-3 ITERFIND

```
//iterfind.cpp
//giải thuật find() tra ve mot iterator danh sach
#include<iostream.h>
#include<list>
#include<algorithm>
void main()
{
    list<int> iList(5);           //danh sach rong giu 10 so nguyen
    list<int>::iterator it;       //iterator tro toi danh sach cac so nguyen
    int data=0;

                                //dat du lieu vao danh sach
    for(it=iList.begin();it!=iList.end();it++)
        *it=data+=2;

    it=find(iList.begin(),iList.end(),8);
    if(it!=iList.end())
        cout<<"\nDa tim thay 8 trong danh sach.";
    else
        cout<<"\nKhong tim thay 8 trong danh sach.";
}
```

Đưa vào danh sách với các giá trị 2,4,6,8,10 như trong ví dụ trước. Tiếp theo chúng ta sử dụng giải thuật **find()** để tìm số 8 trong danh sách. Nếu **find()** trả về giá trị **iList.end()** thì chúng ta biết đã tới cuối côngtenơ mà không tìm thấy phần tử thích hợp. Nếu không, nó định vị tại mục dữ liệu có giá trị là 8. Đây là kết quả đưa ra:

Da tim thay 8 trong danh sach

Chúng ta có thể sử dụng giá trị của **iterator** trả về để biết được nơi đặt số 8 trong danh sách. Có thể chúng ta nghĩ vị trí đó là độ lệch của mục dữ liệu tìm thấy tính từ đầu côngtenơ được tính như sau:

(it-iList.begin())

Tuy nhiên đây không phải là phép toán hợp lệ trên các **iterator** dùng cho danh sách. Một **iterator** danh sách là một **iterator hai chiều**, bởi vậy không thể thực hiện phép toán trên nó. Chúng ta chỉ có thể làm phép toán trên các **iterator truy nhập ngẫu nhiên**, chẳng hạn như các phép toán với các **iterator** cho vectơ và hàng đợi hai đầu. Do đó, nếu tìm kiếm trong một vectơ hơn là một danh sách thì chúng ta có thể viết lại đoạn cuối của chương trình ITERFIND như sau:

```
it=find(iList.begin(),iList.end(),8);
if(it!=v.end())
    cout<<"\nDa tim thay 8 trong vector tai vi tri: "<<(it-v.begin());
else
    cout<<"\nKhong tim thay 8 trong vector.";
```

Và kết quả đưa ra sẽ là:

Da tim thay 8 trong vector tai vi tri: 3

Bản 9-4 trình bày một ví dụ khác trong đó một giải thuật sử dụng các **iterator** làm các đối số. Ví dụ này sử dụng giải thuật **copy()** với một vectơ. Người sử dụng xác định đoạn phần tử cần được copy từ một vectơ tới một vectơ khác và chương trình copy chúng. Các **iterator** xác định đoạn phần tử này.

Listing 9-4 ITERCOPY

```
//itercopy.cpp
//su cac iterator cho giai thuat copy
#include<iostream.h>
#include<vector>
#include<algorithm>
void main()
{
    int beginRange,endRange;
    int arr[]={11,13,15,17,19,21,23,25,27,29};
    vector<int> v1(arr,arr+10);           //khoi tao mot vector
    vector<int> v2(10);                   //khong khoi tao vector

    cout<<"\nNhap vao doan copy (vi du 2 5): ";
    cin>>beginRange>>endRange;
                                   //tao cac iterator va khoi tao chung
    vector<int>::iterator it1=v1.begin()+beginRange;
    vector<int>::iterator it2=v1.begin()+endRange;
    vector<int>::iterator it3;

                                   //copy doan do tu v1 toi v2
    it3=copy(it1,it2,v2.begin());//it3 tro toi phan tu cuoi cung duoc copy
    it1=v2.begin();                 //tro qua doan
    while(it1!=it3)
        cout<<*it1++<<' ';
}
```

Đây là vài mẫu tương tác với chương trình:

Nhap vao doan copy (vi du 2 5): 3 6
17 19 21

Chúng ta không muốn hiển thị toàn bộ nội dung của v2, chỉ hiển thị đoạn được copy. Thật là may mắn, `copy()` trả về một **iterator** trở tới phần tử cuối cùng (thường sau phần tử cuối cùng một phần tử) được copy tới côngtenơ đích, trong trường hợp này là v2. Chương trình hiển thị giá trị này trong vòng lặp `while` để hiển thị các mục dữ liệu được copy.

9.4. ITERATOR ĐẶC BIỆT

Trong phần này chúng ta sẽ xem xét hai loại **iterator** đặc biệt: các bộ thích ứng **iterator** (**iterator adaptors**), nó có thể thay đổi cách đối xử của các **iterator** theo nhiều cách thú vị và các **iterator** dòng (**stream iterator**), nó cho phép các dòng vào/ra được đối xử như các **iterator**.

9.4.1. Bộ thích ứng iterator

STL cung cấp ba dạng biến đổi của **iterator** thông thường. Ba dạng này là **iterator ngược** (**reverse iterator**), **iterator chèn** (**insert iterator**) và **iterator lưu trữ thô** (**raw storage iterator**). **Iterator đảo** cho phép trở lùi qua một côngtenơ. **Iterator chèn** thay đổi cách cư xử của nhiều giải thuật khác nhau, chẳng hạn `copy()` và `merge()`, để chúng chèn dữ liệu vào một côngtenơ hơn là ghi đè lên dữ liệu đã có. **Iterator lưu trữ thô** cho phép đưa ra các **iterator** để lưu trữ dữ liệu trong bộ nhớ không được khởi tạo, nhưng lại được sử dụng trong nhiều tình huống đặc biệt mà chúng ta sẽ khảo sát kỹ ở đây.

1. Iterator ngược

Cho rằng chúng ta muốn trở qua côngtenơ từ cuối tới đầu, có người sẽ nghĩ là làm như thế này:

```
list<int>::iterator it;      //iterator thông thường
it=iList.end();             //bat dau tu cuoi
while(it!=iList.begin())    //di toi dau
    cout<<*it--<<' ';    //giam iterator
```

Nhưng thật không may cách này không làm việc. Vì một điều là khoảng sẽ sai (từ n tới 1 chứ không phải từ n-1 tới 0).

Để trở theo hướng lùi, chúng ta phải sử dụng **iterator ngược**. Chương trình ITEREV (bản 9-5) trình bày một ví dụ mà ở đó sử dụng một **iterator ngược** để hiển thị một danh sách theo thứ tự ngược lại.

Listing 9-5 ITEREV

```
//iterev.cpp
//minh hoa iterator ngược
#include<iostream.h>
#include<list>

void main()
{
    int arr[]={2,4,6,8,10};           //mang so nguyen
    list<int> iList(arr,arr+5);        //danh sach duoc khoi tao toi mang
    list<int>::reverse_iterator revit; //iterator ngược
    revit=iList.rbegin();
    while(revit!=iList.rend())         //tro lui qua danh sach
        cout<<*revit++<<' ';        //hien thi
}
```

Kết quả đưa ra như sau:

10 8 6 4 2

Chúng ta phải sử dụng hàm thành viên **rbegin()** và **rend()** khi dùng **iterator ngược** (đừng thử với **iterator tiến - forward iterator**). Sự nhầm lẫn hay gặp là mặc dù chúng ta bắt đầu ở cuối côngtenơ nhưng lại gọi hàm thành viên **rbegin()**. Ngoài ra, không được giảm **iterator ngược**, **revit--** sẽ không làm những gì ta mong muốn. Với **reverse_iterator**, luôn đi từ **rbegin()** tới **rend()** dùng toán tử tăng ++.

2. Iterator chèn

Một vài giải thuật, chẳng hạn **copy()**, ghi đè lên nội dung đã tồn tại của côngtenơ đích. Chương trình COPYDEQ (bản 9-6) copy từ một hàng đợi hai đầu tới một hàng đợi hai đầu khác sẽ cho một ví dụ về **iterator chèn**.

Listing 9-6 COPYDEQ

```
//copydeq
//minh hoa giai thuat copy() thông thường voi cac hang doi
#include<iostream.h>
#include<deque>
#include<algorithm>

void main()
```

```

{
    int arr1[]={1,3,5,7,9};           //mang so nguyen
    deque<int> d1(arr1,arr1+5);        //khởi tạo d1
    int arr2[]={2,4,6,8,10};          //mang so nguyen
    deque<int> d2(arr2,arr2+5);        //khởi tạo d2

                                     //copy d1 vào d2
    copy(d1.begin(),d1.end(),d2.begin());
    for(int j=0;j<d2.size();j++)      //hiển thị d2
        cout<<d2[j]<<' ';
}

```

Kết quả đưa ra là:

1 3 5 7 9

Nội dung của d1 đã bị ghi đè bởi nội dung của d2, không còn dấu vết gì về nội dung của nó trước đây (các số nguyên chẵn). Thông thường thì chúng ta muốn điều này. Tuy nhiên, trong một số trường hợp chúng ta muốn nhờ `copy()` chèn các phần tử mới vào một côngtenơ cùng với các phần tử cũ hơn là ghi đè lên chúng. Chúng ta có thể thực hiện việc này bằng cách sử dụng **iterator chèn**. Có ba dạng của **iterator** này:

- + **back_inserter** chèn các mục mới ở cuối.
- + **front_inserter** chèn các mục mới ở đầu.
- + **inserter** chèn các mục mới ở vị trí xác định.

Chương trình DINSITER (bản 9-7) cho thấy cách sử dụng **back_inserter**.

Listing 9- 7 DINSITER

```

//dinsiter.cpp
//minh họa iterator chèn vào hàng đôi
#include<iostream.h>
#include<deque>
#include<algorithm>

void main()
{
    int arr1[]={1,3,5,7,9};           //mang so nguyen
    deque<int> d1(arr1,arr1+5);        //khởi tạo d1
    int arr2[]={2,4,6};               //mang so nguyen
    deque<int> d2(arr2,arr2+3);        //khởi tạo d2

                                     //copy d1 vào d2
    copy(d1.begin(),d1.end(),back_inserter(d2));
    cout<<"\nd2= ";
    for(int j=0;j<d2.size();j++)      //hiển thị d2
        cout<<d2[j]<<' ';
}

```

back_inserter sử dụng hàm thành viên **push_back()** của côngtenơ để chèn các mục mới vào cuối côngtenơ đích d2, theo sau các mục dữ liệu đã tồn tại. Côngtenơ nguồn d1 không thay đổi. Kết quả đưa ra của chương trình là:

2 4 6 1 3 5 7 9

Nếu sử dụng **front_inserter** thay thế **back_inserter**

```
copy(d1.begin(),d1.end(),front_inserter);
```

thì các mục mới sẽ được chèn vào trước côngtenơ. Cơ chế bên trong là sử dụng hàm thành viên **push_front()**, nó đẩy từng mục một vào trước côngtenơ, làm đảo ngược thứ tự của côngtenơ. Kết quả đưa ra sẽ là:

```
9 7 5 1 2 4 6
```

Chúng ta cũng có thể chèn các mục mới bắt đầu tại phần tử tùy ý trong côngtenơ dùng phiên bản **inserter** của **iterator** chèn. Ví dụ, để chèn các mục mới tại đầu của d2, chúng ta dùng:

```
copy(d1.begin(),d1.end(),inserter(d2,d2.begin()));
```

Đối số thứ nhất của **inserter** là côngtenơ được copy vào và đối số thứ hai là một **iterator** trở tới nơi bắt đầu copy tới. Bởi vì **inserter** sử dụng hàm thành viên **insert()** của côngtenơ nên thứ tự của các phần tử không bị đảo ngược. Kết quả đưa ra từ câu lệnh này sẽ là:

```
1 3 5 7 9 2 4 6
```

Bằng cách thay đổi đối số thứ hai của **inserter**, chúng ta có thể chèn dữ liệu mới vào bất cứ đâu trong côngtenơ.

Chú ý rằng **front_inserter** không dùng được với vectơ bởi vì các vectơ không có hàm thành viên **push_front()**; chúng chỉ có thể được truy nhập tại một đầu.

9.4.2. Iterator dòng

Các **iterator** dòng (stream iterators) cho phép đối xử với các file và các thiết bị vào/ra (chẳng hạn như cin và cout) y như thể chúng là các **iterator**. Điều này làm cho dễ dàng sử dụng các file và các thiết bị vào/ra làm đối số cho các giải thuật. (Đây là một minh họa khác về sự phổ dụng của việc sử dụng **iterator** để liên kết các giải thuật và các côngtenơ).

Mục đích chính của các loại **iterator** đọc và ghi là trợ giúp các lớp **iterator** dòng. Các **iterator** đọc và ghi (**input/output iterator**) làm cho nó có thể phù hợp với các giải thuật được sử dụng trực tiếp trên các dòng vào/ra.

Các **iterator** dòng thực sự là các đối tượng của các lớp được mẫu hóa cho các kiểu vào/ra khác nhau. Có hai loại **iterator** dòng: **ostream_iterator** và **istream_iterator**. Chúng ta sẽ lần lượt xem xét chúng.

1. Lớp ostream_iterator

Một đối tượng **ostream_iterator** có thể được dùng làm đối số cho bất kỳ giải thuật nào yêu cầu một **iterator** ghi (**output iterator**). Chương trình OUTITER (bản 9-8) sử dụng nó làm đối số cho giải thuật **copy()**.

Listing 9-8 OUTITER

```
//outiter.cpp
//minh hoa iterator dòng
#include<iostream.h>
#include<list>
#include<algorithm>

void main()
{
    int arr1[]={10,20,30,40,50};           //mang so nguyen
    list<int> iList(arr,arr+5);             //khởi tạo danh sách
```



```

ostream_iterator<int> ositer(cout,"--");    //iterator dòng
cout<<"\nNoi dung cua danh sach:";
copy(iList.begin(),iList.end(),ositer);    //hien thi danh sach
}

```

Chúng ta định nghĩa một **iterator dòng** để đọc các giá trị kiểu **int**. Có hai đối số cho hàm tạo này là **stream** mà các giá trị sẽ được ghi ra và một giá trị chuỗi sẽ được hiển thị theo sau mỗi giá trị. Giá trị **stream** điển hình là một tên file hoặc **cout**; ở đây là **cout**. Khi ghi ra **cout**, chúng ta có thể tạo chuỗi bao gồm bất kỳ ký tự nào; ở đây dùng hai dấu gạch ngang.

Giải thuật **copy()** sao chép nội dung của danh sách tới **cout**. **Iterator dòng** được dùng như đối số thứ ba cho **copy()**; nó là nơi dữ liệu đến.

Kết quả của chương trình là:

```
10--20--30--40--50--
```

Bản 9-9 FOUTITER, cho thấy cách sử dụng **ostream_iterator** để ghi ra file.

Listing 9-9 FOUTITER

```

//foutiter.cpp
//minh hoa iterator dòng voi file
#include<fstream.h>
#include<list>
#include<algorithm>
void main()
{
    int arr1[]={11,21,31,41,51};    //mang so nguyen
    list<int> iList(arr,arr+5);    //khởi tạo danh sách
    ofstream outfile("ITER.DAT");    //tạo đối tượng file
    ostream_iterator<int> ositer(outfile," ");    //iterator dòng
    copy(iList.begin(),iList.end(),ositer);    //hien thi danh sach
}

```

Chúng ta phải định nghĩa một đối tượng file **ofstream** và gắn nó tới một file, ở đây là **ITER.DAT**. Đối tượng này là đối số thứ nhất cho **ostream_iterator**. Khi ghi tới một file, cần sử dụng một ký tự trắng trong đối số chuỗi, chứ không phải là các ký tự như **--**. Điều này làm cho dễ đọc dữ liệu trở lại từ file.

Không có kết quả được hiển thị từ chương trình **FOUTITER**, nhưng xem file **ITER.DAT** bằng một hệ soạn thảo bất kỳ chúng ta sẽ thấy:

```
11 21 31 41 51
```

2. Lớp **istream_iterator**

Một đối tượng **istream_iterator** có thể được dùng làm đối số cho bất kỳ giải thuật nào xác định một **iterator đọc** (input iterator).

Bản 9-10 cho thấy các đối tượng như vậy được dùng làm hai đối số đầu cho **copy()**. Chương trình này đọc các số dấu phẩy động nhập từ bàn phím (**cin**) bởi người sử dụng và lưu chúng vào trong một danh sách.

Listing 9-10 INITER

```

//initer.cpp
//minh hoa istream_iterator

```

```

#include<fstream.h>
#include<list>
#include<algorithm>

void main()
{
    list<int> fList(5);           //tao danh sach rong

    cout<<"\nNhap vao 5 so dau phay dong (go Ctrl+Z de ket thuc):";
                                //tao cac istream_iterator
    istream_iterator<float,ptrdiff_t> cin_iter(cin); //cin
    istream_iterator<float,ptrdiff_t> end_of_stream; //eof

    copy(cin_iter,end_of_stream,fList.begin());
    cout<<endl;                 //hien thi danh sach
    ostream_iterator<float> ositer(cout,"--");
    copy(fList.begin(),fList.end(),ositer);
}

```

Vài mẫu tương tác với chương trình như sau:

```

Nhap vao 5 so dau phay dong (go Ctrl+Z de ket thuc):
1.1 2.2 3.3 4.4 5.5
1.1--2.2--3.3--4.4--5.5--

```

Chú ý rằng đối với **copy()**, bởi vì dữ liệu đến từ **cin** là nguồn chứ không phải là đích nên chúng ta phải xác định cả đầu và cuối của đoạn được copy. Điểm đầu là **istream_iterator** nối với **cin**, chúng ta định nghĩa nó là **cin_iter** dùng hàm tạo một đối số. Nhưng điểm cuối của đoạn được copy là gì? Hàm tạo không đối số (mặc định) đóng vai trò đặc biệt này. Nó luôn luôn tạo một đối tượng **istream_iterator** biểu diễn cuối **stream**.

Người sử dụng tạo ra giá trị kết thúc **stream** này như thế nào khi họ nhập vào dữ liệu? Bằng cách gõ CTRL+Z, nó đưa vào ký tự **end_of_file** thường được sử dụng cho các **stream** (ấn ENTER sẽ không kết thúc file mặc dù nó sẽ giới hạn các số).

Chúng ta dùng một **ostream_iterator** để hiển thị nội dung của danh sách do người sử dụng nhập vào mặc dù có nhiều cách làm việc này.

Các dòng thông báo đưa ra màn hình, chẳng hạn Nhap vao 5 so dau phay dong (go Ctrl+Z de ket thuc):, phải được thực hiện trước khi định nghĩa **istream_iterator**, bởi vì ngay sau khi định nghĩa **iterator** này màn hình sẽ bị khoá để đợi nhập vào.

Bản 9-11, FINITER, sử dụng một file thay cho **cin** làm đầu vào cho giải thuật **copy()**.

Listing 9-11 FINITER

```

//finiter.cpp
//minh hoa istream_iterator voi cac file
#include<fstream.h>
#include<list>
#include<algorithm>

void main()
{
    list<int> iList;           //tao danh sach rong
    ifstream infile("ITER.DAT"); //tao doi tuong file de doc vao
                                //file ITER.DAT phai ton tai
}

```

```

//tao cac istream_iterator
istream_iterator<int,ptrdiff_t> file_iter(infile); //file
istream_iterator<int,ptrdiff_t> end_of_stream; //eof
//copy tu file vao danh sach iList
copy(file_iter,end_of_stream,back_inserter(iList));
cout<<endl; //hien thi danh sach
ostream_iterator<int> ositer(cout,"--");
copy(fList.begin(),fList.end(),ositer);
}

```

Chúng ta định nghĩa một **ifstream** để biểu diễn file ITE.DAT, file phải tồn tại và chứa dữ liệu (nếu chạy chương trình FOUTIER sẽ tạo ra file này).

Thay vì sử dụng **cin** như trong ví dụ INTER, chúng ta sử dụng đối tượng **ifstream** là **infile**. Đối tượng kết thúc **stream** giống như trong ví dụ trước.

Ở trong chương trình này chúng ta có một sự thay đổi: sử dụng một **back_inserter** để chèn dữ liệu vào **iList**. Vì sử dụng **back_inserter** nên có thể định nghĩa **iList** là một côngtenơ rỗng thay cho một côngtenơ có kích thước xác định. Điều này thường có ý nghĩa khi đọc vào, bởi vì chúng ta có thể không biết có bao nhiêu dữ liệu sẽ được nhập vào.

CHƯƠNG 10

CÔNGTENƠ TUẦN TỰ

Trong chương này chúng ta sẽ giới thiệu ba côngtenơ tuần tự: vectơ (**vector**), danh sách (**list**) và hàng đợi hai đầu (**deque**); tập trung vào cách làm việc của các côngtenơ này và các hàm thành viên của chúng.

Các chương trình ví dụ sẽ giới thiệu một vài hàm thành viên cho côngtenơ được mô tả. Tuy nhiên, nên nhớ rằng các loại côngtenơ khác nhau vẫn sử dụng các hàm thành viên có tên và chức năng giống nhau.

10.1. VÉCTƠ

Chúng ta có thể nghĩ về các vectơ như các mảng thông minh. Chúng quản lý việc cấp phát bộ nhớ giúp chúng ta, co giãn kích thước vectơ khi chèn hoặc xóa dữ liệu. Chúng ta có thể sử dụng vectơ y như các mảng, truy nhập các phần tử của nó với toán tử []. Việc truy nhập ngẫu nhiên như vậy rất nhanh với các vectơ. Nó cũng nhanh khi thêm (**add**, **push**) một mục dữ liệu mới vào cuối (**back**) của vectơ. Khi những việc này xảy ra, kích thước của vectơ được tự động tăng lên để giữ mục dữ liệu thêm vào hoặc tự động giảm đi khi xóa một mục khỏi vectơ.

Các hàm thành viên của vectơ sẽ thực hiện những thao tác cơ bản trên vectơ. Có những hàm thành viên cơ bản sau:

- **push_back()** Đặt một mục dữ liệu vào cuối vectơ.
- **back()** Đọc một mục dữ liệu ở cuối vectơ.
- **pop_back()** Lấy một mục dữ liệu ở cuối vectơ ra khỏi vectơ.
- **insert()** Chèn một mục dữ liệu vào một vị trí trong vectơ.
- **erase()** Xóa một mục dữ liệu ở một vị trí nào đó trong vectơ.
- **swap()** Hoán đổi dữ liệu giữa hai vectơ.
- **empty()** Nếu vectơ rỗng trả về true.
- **size()** Trả về số phần tử hiện có của vectơ.
- toán tử [] Truy nhập tới một phần tử bất kỳ trong vectơ.

10.1.1. Hàm thành viên **push_back()**, **size()** và toán tử []

Bản 10-1, VECTOR, cho thấy những phép toán trên vectơ thông dụng nhất.

Listing 10-1 VECTOR

```
//vector.cpp
//minh hoa push_back(), operator[],size()
#include<iostream.h>
#include<vector>

void main()
{
    vector<int> v;           //tao mot vector so nguyen
```

```

v.push_back(10);           //dat cac gia tri vao cuoi vector
v.push_back(11);
v.push_back(12);
v.push_back(13);
v[0]=20;
v[3]=23;
for(int j=0;j<v.size();j++) //hien thi noi dung cua vector
    cout<<v[j]<<' ';
}

```

Chúng ta sử dụng hàm tạo mặc định (không đối số) của vectơ để tạo một vectơ *v*. Như với tất cả các côngtenơ STL, dạng mẫu được sử dụng để xác định kiểu của biến mà côngtenơ sẽ giữ; trong ví dụ này là kiểu **int**. Chúng ta không xác định kích thước của côngtenơ, bởi vậy nó có kích thước ban đầu bằng 0.

Hàm thành viên `push_back()` chèn giá trị của đối số của nó vào sau vectơ (sau vectơ là nơi có chỉ số cao nhất). Trước một vectơ (nơi phần tử có chỉ số bằng 0), không giống như một danh sách (list) hay một hàng đợi (queue), không thể truy nhập. Ở đây chúng ta đưa vào các giá trị 10, 11, 12 và 13 để `v[0]` chứa 10, `v[1]` chứa 11, `v[2]` chứa 12 và `v[3]` chứa 13.

Khi một vectơ đã có dữ liệu trong nó, dữ liệu này có thể được truy nhập - cả đọc và ghi - bằng toán tử `[]`, y như thể nó ở trong một mảng. Trong chương trình chúng ta sử dụng toán tử này để thay đổi phần tử đầu tiên từ 10 thành 20 và phần tử cuối cùng từ 13 thành 23. Đây là kết quả đưa ra từ chương trình:

```
20 11 12 23
```

Chú ý rằng, mặc dù có thể truy nhập dữ liệu đã có nhưng chúng ta không thể sử dụng toán tử `[]` để làm thay đổi kích thước của vectơ như với `push_back()`. Ví dụ, cố đọc hoặc ghi tới `v[27]` sẽ không làm việc bởi vì vectơ chỉ có bốn phần tử.

Hàm thành viên `size()` trả về số phần tử hiện có trong côngtenơ, trong chương trình VECTOR nó là 4. Chúng ta sử dụng giá trị này trong vòng lặp `for` để cho các giá trị của các phần tử trong côngtenơ.

Một hàm thành viên khác, `max_size()`, trả về kích thước cực đại mà một côngtenơ có thể mở rộng được tới. Số này tùy thuộc vào kiểu dữ liệu được lưu trữ trong côngtenơ (kiểu phần tử càng lớn càng lưu trữ được ít), tùy thuộc vào kiểu của côngtenơ và hệ điều hành. Ví dụ, `max_size()` trả về 32767 cho kiểu **int**.

10.1.2. Hàm thành viên `swap()`, `empty()`, `back()` và `pop_back()`

Bản 10-2, VECTCON, cho thấy một vài hàm tạo vectơ khác và các hàm thành viên.

Listing 10-2 VECTCON

```

//vectcon.cpp
//minh hoa cac ham tao,swap(), empty(),back(), pop_back()
#include<iostream.h>
#include<vector>

void main()
{
    float arr[]={1.1,2.2,3.3,4.4}; //mot mang float
    vector<float> v1(arr,arr+4); //khoi tao mot vector toi mang
    vector<float> v2(4);          //vector rong co kích thước bằng 4
    v1.swap(v2);                  //hoan doi v1 cho v2
}

```

```

while(!v2.empty())
{
    cout<<v2.back()<<' ';    //hien thi phan tu cuoi
    v2.pop_back();            //xoa phan tu cuoi
}

```

Chúng ta đã sử dụng hai hàm tạo vectơ mới trong chương trình này. Hàm tạo thứ nhất khởi tạo vectơ `v1` với các giá trị của một mảng C++ thông thường được truyền tới nó như một đối số. Các đối số cho hàm tạo này là các con trỏ tới đầu của mảng và phần tử sau phần tử cuối mảng. Hàm tạo thứ hai thiết lập `v2` có kích thước ban đầu là 4 nhưng không cung cấp giá trị ban đầu cho nó. Cả hai vectơ đều giữ giá trị kiểu `float`.

Hàm thành viên `swap()` hoán đổi tất cả dữ liệu trong một vectơ với một vectơ khác, giữ phần tử theo đúng thứ tự. Trong chương trình này, `v2` chứa giá trị vô nghĩa, nó được hoán đổi với `v1`. Chương trình hiển thị `v2` để thấy bây giờ nó chứa dữ liệu của `v1`. Kết quả đưa ra là:

4.4 3.3 2.2 1.1

Hàm thành viên `back()` trả về giá trị của phần tử cuối cùng trong vectơ và được hiển thị với `cout`. Hàm thành viên `pop_back()` xóa phần tử cuối cùng trong vectơ. Do đó, mỗi lần qua vòng lặp có một phần tử cuối cùng khác (hơi ngạc nhiên là `pop_back()` không đồng thời trả về giá trị của phần tử cuối cùng và xóa nó khỏi vectơ, vì vậy phải dùng `back()`).

Một vài hàm thành viên, chẳng hạn như `swap()`, cũng tồn tại một giải thuật có cùng tên. Khi gặp trường hợp hàm thành viên và giải thuật giống nhau cả về tên và chức năng thì nên sử dụng hàm thành viên vì nó năng suất hơn giải thuật. Đôi khi chúng ta cũng sử dụng phiên bản giải thuật, ví dụ, để hoán đổi các phần tử trong hai loại côngtenơ khác nhau.

10.1.3. Hàm thành viên `insert()` và `erase()`

Hàm thành viên `insert()` và `erase()` chèn hoặc xóa một phần tử từ một vị trí bất kỳ trong côngtenơ. Các hàm này rất không hiệu suất với các vectơ bởi vì tất cả các phần tử ở trên vị trí chèn hoặc xóa phải được di chuyển để tạo khoảng trống cho phần tử mới hoặc đóng khoảng trống mà ở đó mục dữ liệu được xóa. Tuy nhiên việc chèn và xóa có thể hữu ích nếu không sử dụng quá nhiều.

Bản 10-3, VECTINS, cho thấy cách sử dụng các hàm này.

Listing 10-3 VECTINS

```

//vectins.cpp
//minh hoa ham insert() va erase()
#include<iostream.h>
#include<vector>

void main()
{
    int arr[]={100,110,120,130};    //mot mang so nguyen
    vector<int> v(arr,arr+4);        //khoi tao mot vector toi mang
    int j;

    cout<<"\nTruoc khi chen: ";
    for(j=0;j<v.size();j++)          //hien thi tat ca cac phan tu
        cout<<v[j]<<' ';

    v.insert(v.begin()+2,115);        //chen 115 tai phan tu 2
    cout<<"\nSau khi chen: ";

```

```

for(j=0;j<v.size();j++)           //hien thi tat ca cac phan tu
    cout<<v[j]<<' ';
v.erase(v.begin()+2);             //xoa phan tu 2
cout<<"\nSau khi xoa: ";
for(j=0;j<v.size();j++)           //hien thi tat ca cac phan tu
    cout<<v[j]<<' ';
}

```

Hàm thành viên **insert()** (ít nhất là phiên bản này của nó) có hai đối số: nơi mà một phần tử sẽ được chèn vào một côngtenơ và giá trị của phần tử được chèn vào. Chúng ta cộng hai vào hàm thành viên **begin()** để xác định phần tử 2 (phần tử thứ ba) trong vectơ. Các phần tử từ điểm chèn tới cuối côngtenơ được di chuyển lên để tạo khoảng trống và kích thước của côngtenơ được tăng lên 1.

Hàm thành viên **erase()** xóa một phần tử tại một vị trí xác định. Các phần tử phía trên điểm xóa được di chuyển xuống (chỉ số giảm đi một) và kích thước của côngtenơ được giảm đi 1. Đây là kết quả đưa ra từ chương trình VECTINS.

10.1.4. Sắp xếp vectơ với giải thuật sort()

Chương trình ví dụ tiếp theo cho thấy giải thuật **sort()** làm việc với vectơ như thế nào. Chương trình để người sử dụng nhập vào một dãy số nguyên, sau đó nó sắp xếp dãy số nguyên đó theo chiều tăng. Bản 10-4 trình bày chương trình này, SORTINTS.

Listing 10-4 SORTINTS

```

//sortints.cpp
//minh hoa giai thuat sort() voi vector
#include<iostream.h>
#include<vector>
#include<algorithm>
void main()
{
    using namespace std;
    vector<int> v;           //tao mot vector rong
    int in,n;                //luu tru so nguyen nhap vao
    char ch;
    cout<<"\nNhap vao mot day so nguyen ";
    cout<<"\nSo luong so nguyen muon nhap: ";
    cin>>n;
    for(int j=0;j<n;j++)
    {
        cout<<"So thu "<<j+1<<" : ";
        cin>>in;
        v.push_back(in);
    }
    sort(v.begin(),v.end()); //sap xep theo chieu tang
    cout<<"\n\nDay so sap xep theo chieu tang: ";
    for(int j=0;j<v.size();j++)
        cout<<v[j]<<' ';
}

```

Chúng ta sử dụng giá trị trả về của hai hàm thành viên **begin()** và **end()** làm đối số cho **sort()**. Hàm thành viên **begin()** trả về một **iterator** trở tới điểm bắt đầu của vectơ, còn **end()** trả về một **iterator** trở tới vị trí quá cuối của vectơ một phần tử. Đây là vài mẫu tương tác với chương trình:

Nhap vao mot day so nguyen

So luong so nguyen muon nhap: 6

So thu 1 : 9

So thu 2 : 34

So thu 3 : 2

So thu 4 : 4

So thu 5 : 9

So thu 6 : -5

Day so sap xep theo chieu tang: -5 2 4 9 9 34

Phiên bản `sort()` có hai tham số sẽ sắp xếp khoảng phần tử được chỉ ra theo chiều tăng. Để sắp xếp theo chiều giảm chúng ta phải dùng thêm tham số thứ ba là đối tượng hàm `greater<>()`:

```
sort(v.begin(),v.end(),greater<int>());
```

Kết quả của phiên bản này sẽ là:

34 9 9 4 2 -5

10.1.5. Tìm một phần tử trong vectơ bằng giải thuật `find()`

Vì các côngtenơ tuần tự không có hàm thành viên `find()` nên muốn tìm kiếm một phần tử trong vectơ chúng ta phải sử dụng giải thuật `find()`. Chương trình ví dụ tiếp theo sẽ minh họa cách sử dụng giải thuật này. Đầu tiên chương trình để cho người sử dụng nhập vào một dãy số nguyên và lưu chúng trong một vectơ. Sau đó yêu cầu người sử dụng nhập vào một giá trị để tìm. Chương trình sẽ tìm số đó trong mảng và thông báo cho người sử dụng biết kết quả tìm kiếm. Bản 10-5 là chương trình `FINDINT`.

Listing 10-5 **FINDINT**

```
//findint.cpp
//minh hoa giai thuat find() voi vector
#include<iostream.h>
#include<vector>
#include<algorithm>
void main()
{
    using namespace std;
    vector<int> v;           //tao mot vector rong
    int in,n;               //luu tru so nguyen nhap vao

    cout<<"\nNhap vao mot day so nguyen ";
    cout<<"\nSo luong so nguyen muon nhap: ";
    cin>>n;
    for(int j=0;j<n;j++)
    {
        cout<<"So thu "<<(j+1)<<" : ";
        cin>>in;
        v.push_back(in);
    }

    int search_int;
    cout<<"\nNhap vao so nguyen can tim: ";
    cin>>search_int;
    vector<int>::iterator it; //tao mot iterator vector
    it=find(v.begin(),v.end(),search_int);
```



```

if(it!=v.end())
    cout<<"\nDa tim thay so nay o phan tu: "<<it-v.begin();
else
    cout<<"\nKhong tim thay so nay trong vector.";
cout<<endl<<endl;
for(int j=0;j<v.size();j++)
    cout<<v[j]<<' ';
}

```

Trong chương trình chúng ta định nghĩa một **iterator**, sử dụng nó để lưu giá trị trả về của giải thuật **find()**. Nếu **find()** tìm thấy thì nó trả về một **iterator** trở tới vị trí của phần tử tìm thấy trong vectơ. Nếu không tìm thấy nó trả về một **iterator** trở tới vị trí quá cuối của vectơ. Sau đây là vài mẫu tương tác với chương trình:

```

Nhap vao mot day so nguyen
So luong so nguyen muon nhap: 6
So thu 1 : 23
So thu 2 : 4
So thu 3 : 45
So thu 4 : 34
So thu 5 : 56
So thu 6 : 7

```

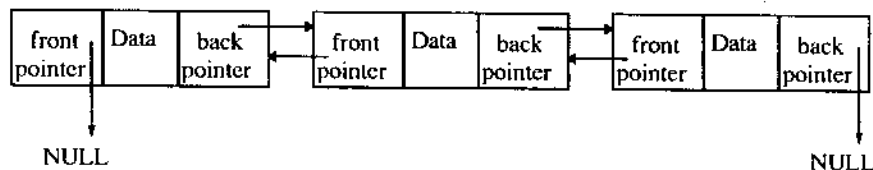
Nhap vao so nguyen can tim: 45

Da tim thay so nay o phan tu: 2

23 4 45 34 56 7

10.2. DANH SÁCH (LIST)

Một côngtenơ danh sách là một danh sách liên kết kép (**double linked list**). Danh sách liên kết kép là một tập hợp các phần tử mà trong đó mỗi phần tử liên kết với cả phần tử đứng trước và đứng sau nó, giống như một dây xích. Để tạo liên kết, mỗi phần tử chứa hai con trỏ tới phần tử đứng trước và đứng sau. Hình 10-1 cho thấy cấu trúc của một danh sách.



Hình 10-1. Cấu trúc danh sách.

Con trỏ trước (**front pointer**) của phần tử đầu tiên và con trỏ sau (**back pointer**) của phần tử cuối cùng có giá trị **NULL** để báo hiệu là hai đầu của danh sách.

Vì mỗi phần tử trong danh sách lưu trữ địa chỉ của cả phần tử đứng trước và đứng sau nó nên việc truy nhập tới cả hai đầu của danh sách là rất nhanh.

Các hàm thành viên chính của danh sách là:

- **push_front()** Đặt một phần tử vào đầu danh sách.

- `front()` Đọc một phần tử ở đầu danh sách.
- `pop_front()` Lấy phần tử đầu tiên ra khỏi danh sách.
- `push_back()` Đặt một phần tử vào cuối danh sách.
- `back()` Đọc một phần tử ở cuối danh sách.
- `pop_back()` Lấy phần tử cuối cùng ra khỏi danh sách.
- `reverse()` Đảo ngược danh sách: đầu thành cuối, cuối thành đầu.
- `merge()` Trộn một danh sách vào một danh sách khác.
- `unique()` Làm cho mỗi phần tử trong danh sách là duy nhất.
- `size()` Trả về số phần tử hiện có của danh sách.
- `insert()` Chèn một phần tử vào danh sách.
- `erase()` Xóa một phần tử khỏi danh sách.
- `sort()` Sắp xếp danh sách.
- `empty()` Trả về true nếu danh sách rỗng.

10.2.1. Hàm thành viên `push_front()`, `front()` và `pop_front()`

Bản 10-6, LIST, cho thấy dữ liệu được đặt vào, đọc ra và lấy khỏi danh sách như thế nào.

Listing 10-6 LIST

```
//list.cpp
//minh hoa cac ham thanh vien push_front(),front(),pop_front()
#include<iostream.h>
#include<list>
#include<algorithm>
void main()
{
    using namespace std;
    list<int> iList;           //tao mot danh sach rong
    iList.push_back(30);       //dat cac muc vao tu phia sau
    iList.push_back(40);
    iList.push_front(20);      //dat cac muc vao tu phia truoc
    iList.push_front(10);

    int size=iList.size();     //so cac muc
    for(int j=0;j<size;j++)
    {
        cout<<iList.front()<<" "; //doc muc du lieu tu phia truoc
        iList.pop_front();         //lay muc du lieu o dau ra khoi danh sach
    }
}
```

Các hàm thành viên `push_front()`, `pop_front()` và `front()` tương tự như `push_back()`, `pop_back()` và `back()` mà chúng ta đã thấy làm việc với vectơ.

Chú ý rằng chúng ta không thể sử dụng truy nhập ngẫu nhiên đối với các phần tử danh sách bởi vì toán tử `[]` không được định nghĩa cho danh sách. Nếu cần truy nhập ngẫu nhiên chúng ta nên sử dụng một vectơ hoặc một hàng đợi hai đầu.

Danh sách thích hợp khi thường xuyên phải chèn và xóa ở giữa danh sách. Việc chèn và xóa sẽ không hiệu suất đối với các vectơ (**vector**) và hàng đợi hai đầu (**deque**) bởi vì tất cả các phần tử ở trên điểm chèn hoặc xóa phải được di chuyển. Còn đối với danh sách thì công việc này được thực hiện nhanh chóng bởi vì chỉ cần thay đổi một vài con trỏ để chèn một mục mới hoặc xóa một mục.

10.2.2. Hàm thành viên `reverse()`, `merge()` và `unique()`

Có một số hàm thành viên chỉ dành cho danh sách; các hàm này không được định nghĩa cho các côngtenơ khác. Bản 10-7, **LISTPLUS**, cho thấy một vài hàm này. Nó bắt đầu bằng việc khởi tạo hai đối tượng danh sách kiểu `int` sử dụng các mảng.

Listing 10-7 **LISTPLUS**

```
//listplus.cpp
//minh hoa cac ham thanh vien reverse(),merge(),unique();
#include<iostream.h>
#include<list>
#include<algorithm>
void main()
{
    using namespace std;
    int arr1[]={40,30,20,10};
    int arr2[]={15,20,25,30,35};
    list<int> list1(arr1,arr1+4);           //khởi tạo danh sách từ các mảng
    list<int> list2(arr2,arr2+5);
    list1.reverse();                       //đảo ngược list1: 10 20 30 40
    list1.merge(list2);                    //trộn list2 vào list1
    list1.unique();                        //xóa các số 20 và 30 bởi
    int size=list1.size();
    for(int j=0;j<size;j++)
    {
        cout<<list1.front()<<' ';       //đọc dữ liệu ở trước
        list1.pop_front();                //lấy mục đầu ra khỏi danh sách
    }
}
```

Danh sách thứ nhất có thứ tự giảm dần, chúng ta đưa nó về thứ tự sắp xếp thông thường (tăng dần) bằng hàm thành viên `reverse()` (việc đảo ngược một danh sách rất nhanh bởi vì cả hai đầu của danh sách đều có thể truy nhập được). Điều này là cần thiết bởi vì hàm thành viên thứ hai, `merge()`, tác động trên hai danh sách và yêu cầu cả hai danh sách phải ở thứ tự đã sắp xếp. Sau khi đảo `list1`, hai danh sách là:

```
10 20 30 40
15 20 25 30 35
```

Bây giờ hàm thành viên `merge()` trộn `list1` vào `list2`, giữ thứ tự đã được sắp xếp và mở rộng `list1` để giữ các mục mới. Kết quả của `list1` sau khi đã trộn là:

```
10 15 20 20 25 30 30 35 40
```

Cuối cùng chúng ta dùng hàm thành viên `unique()` cho `list1`. Hàm này tìm những cặp gần nhau có cùng giá trị, với mỗi cặp tìm thấy nó xóa đi một phần tử. Tiếp theo nội dung của `list1` được hiển thị. Đây là kết quả đưa ra từ chương trình **LISTPLUS**:

```
10 15 20 25 30 35 40
```

Để hiển thị nội dung của danh sách chúng ta sử dụng các hàm thành viên **front()** và **pop_front()** trong một vòng lặp **for()**. Mỗi phần tử từ đầu tới cuối danh sách được hiển thị và sau đó được lấy ra khỏi danh sách. Quá trình hiển thị danh sách phá hủy nó. Có thể đây không phải luôn luôn là những gì chúng ta mong muốn. Dưới đây chúng ta cùng xem xét một cách hiển thị danh sách mà không phá hủy nó.

10.2.3. Hiển thị danh sách dùng iterator

Ví dụ tiếp theo, bản 10-8 DISPLIST, cho phép người sử dụng nhập vào một dãy số nguyên với số lượng tùy ý. Với mỗi số nguyên mà người sử dụng nhập vào, chương trình đặt nó vào cuối một danh sách với hàm thành viên **push_back()**. Sau khi người sử dụng kết thúc việc nhập, nó hiển thị toàn bộ nội dung của danh sách. Dưới đây là toàn bộ chương trình.

Listing 10-8 DISPLIST

```
//displist.cpp
//hien thi danh sach dung iterator
#include<iostream.h>
#include<list>
#include<algorithm>
void main()
{
    using namespace std;
    list<int> iList;           //tao mot danh sach rong
    list<int>::iterator it;    //tao mot iterator danh sach
    int in,n;                 //luu tru so nguyen nhap vao
    cout<<"\nNhap vao mot day so nguyen ";
    cout<<"\nSo luong so nguyen muon nhap: ";
    cin>>n;
    for(int j=0;j<n;j++)
    {
        cout<<"So thu"<<(j+1)<<" : ";
        cin>>in;
        iList.push_back(in);
    }
    cout<<"\n\nDay so nguyen vua nhap la:"<<endl;
    for(it=iList.begin();it!=iList.end();it++)
        cout<<*it<<" ";
}
```

Hàm thành viên **begin()** trả về một **iterator** trỏ tới đầu danh sách, giá trị này được sử dụng để khởi đầu cho **it**; hàm thành viên **end()** trả về một **iterator** trỏ tới phần tử quá cuối của danh sách, nó được dùng làm dấu hiệu kết thúc vòng lặp. Toán tử **++** làm cho **it** trỏ tới phần tử tiếp theo. Đây là vài mẫu tương tác với chương trình:

```
Nhap vao mot day so nguyen
So luong so nguyen muon nhap: 6
So thu 1 : 34
So thu 2 : 56
So thu 3 : 12
So thu 4 : 28
So thu 5 : 45
So thu 6 : 23
```

```
Day so nguyen vua nhap la:
34 56 12 28 45 23
```

```

        return (p1.FirstName<p2.FirstName) ? true : false;
        return (p1.LastName<p2.LastName) ? true : false;
    }
//operator == for person class
bool operator==(const person& p1,const person& p2)
{
    return (p1.LastName==p2.LastName &&
            p1.FirstName==p2.FirstName) ? true : false;
}
//-----
void main()
{
    person pers1("Deauville","William",8435150);
    person pers2("McDonald","Stacey",3327563);
    person pers3("Bartoski","Peter",6946473);
    person pers4("KuangThu","Bruce",4157300);
    person pers5("Wellington","John",9207404);
    person pers6("McDonald","Amanda",8435150);
    person pers7("Fredericks","Roger",7049982);
    person pers8("McDonald","Stacey",7764987);
    //multiset of persons
    multiset<person,less<person> > persSet;
    //iterator to a multiset of persons
    multiset<person,less<person> >::iterator iter;

    //put persons in multiset
    persSet.insert(pers1);
    persSet.insert(pers2);
    persSet.insert(pers3);
    persSet.insert(pers4);
    persSet.insert(pers5);
    persSet.insert(pers6);
    persSet.insert(pers7);
    persSet.insert(pers8);

    cout<<"\nNumber of entries = "<<persSet.size();
    //display contents of multiset
    while(iter!=persSet.end())
        (*iter++).display();
    //get last and first name
    string searchLastName,searchFirstName;
    cout<<"\n\nEnter last name of person to search for: ";
    cin>>searchLastName;
    cout<<"Enter first name: ";
    cin>>searchFirstName;
    //creat person with this name
    person searchPerson(searchLastName,searchFirstName,0);
    //get count of such persons
    int cntPersons = persSet.count(searchPerson);
    cout<<"Number of persons with this name = "<<cntPersons;
    //display all matches
    iter = persSet.lower_bound(searchPerson);
    while(iter != persSet.upper_bound(searchPerson))
        (*iter++).display();
} //end main()

```

13.1.1. Hàm thành viên cần thiết

Lớp **person** hầu như được tạo theo cách thông thường, nhưng để được với các côngtenơ STL nó phải được cung cấp một vài hàm thành viên chung. Các hàm này là một hàm tạo mặc định (không có đối số) (thực tế trong ví dụ này nó không cần thiết nhưng thường rất quan trọng), toán tử chồng < và toán tử chồng ==. Các hàm thành viên này được các hàm thành viên lớp danh sách (list) và nhiều giải thuật sử dụng. Chúng ta có thể cần đến các hàm thành viên khác trong những tình huống cụ thể khác (như trong hầu hết các lớp, chúng ta cũng nên cung cấp toán tử gán chồng, các hàm tạo copy và một hàm hủy, ở đây chúng ta bỏ qua các hàm này).

Toán tử chồng < và == nên sử dụng các đối số hằng **const**. Nói chung để chúng là các hàm bạn thì tốt, nhưng cũng có thể để chúng là các hàm thành viên.

13.1.2. Sắp xếp

Toán tử chồng < xác định cách mà các phần tử trong tập hợp sẽ được sắp xếp. Trong chương trình SETPERS, chúng ta định nghĩa toán tử này để sắp xếp tên người và nếu tên trùng nhau thì sắp xếp theo họ đệm (có thể sử dụng số điện thoại để sắp xếp hoặc bất kỳ một mục dữ liệu nào khác trong lớp).

Dưới đây là vài mẫu tương tác với chương trình. Đầu tiên chương trình hiển thị toàn bộ danh sách (điều này sẽ không thiết thực với các cơ sở dữ liệu có số phần tử lớn). Bởi vì chúng được lưu trữ trong một đa tập hợp nên các phần tử được sắp xếp tự động. Khi đó tại dòng thông báo, người sử dụng gõ vào tên "McDonald, Stacey". Có hai người trên danh sách có tên này, bởi vậy chúng được hiển thị cả:

Number of entries = 8

Bactoski,	Peter	phone:	6946473
Deauville,	William	phone:	8435150
Fredecicks,	Roger	phone:	7049982
KuangThu,	Bruce	phone:	4157300
McDonald,	Amanda	phone:	8435150
McDonald,	Stacey	phone:	3327563
McDonald,	Stacey	phone:	7764987
Wellington,	John	phone:	9207404

Enter last name of person to search for: McDonald

Enter first name: Stacey

Number of persons of this name = 2

McDonald,	Stacey	phone:	3327563
McDonald,	Stacey	phone:	7764987

13.1.3. Lưu ý

Chúng ta có thể thấy, khi một lớp được định nghĩa, các đối tượng của lớp đó được các côngtenơ quản lý giống như các biến thuộc các kiểu dữ liệu cơ bản.

Đầu tiên chúng ta dùng hàm thành viên **size()** để hiển thị tổng số mục. Sau đó trở qua đa tập hợp để hiển thị tất cả các mục.

Bởi vì chúng ta sử dụng một đa tập hợp nên có sẵn các hàm **lower_bound()** và **upper_bound()** để hiển thị tất cả các mục trong một khoảng nào đó. Ở cả hai giới hạn trên và dưới giống nhau, bởi vậy tất cả những người có cùng tên được hiển thị. Chú ý là chúng ta phải tạo trước một vài người có

10.2.4. Hàm thành viên insert(), erase() và sort()

Các hàm thành viên **insert()** và **erase()** dùng để chèn và xóa một phần tử tại một vị trí nào đó trong danh sách. Hàm thành viên **sort()** dùng để sắp xếp danh sách theo thứ tự tăng dần. Bản 10-9, LIST2, sẽ cho thấy các công việc này được thực hiện như thế nào.

Listing 10-9 LIST2

```
//list2.cpp
//hien minh hoa insert(),erase(),va sort()
#include<iostream.h>
#include<list>
#include<algorithm>
void main()
{
    using namespace std;
    int arr[]={10,50,90,20,40,30,80,60};
    list<int> iList(arr,arr+8);           //khởi tạo danh sách từ một mảng
    list<int>::iterator it;               //tạo một iterator danh sách

                                     //hien thi danh sach
    cout<<"\nDanh sach so nguyen ban dau:\n";
    for(it=iList.begin();it!=iList.end();it++)
        cout<<*it<<' ';

    int n,loc;
    cout<<"\nNhap vao so nguyen can chen vao danh sach: ";
    cin>>n;
    cout<<"Vi tri chen: ";
    cin>>loc;
    while(loc>iList.size()-1)           //kiem tra vi tri nhap vao
    {
        cout<<"\nKhong co vi tri nay, vi tri cuc dai la "
            <<(iList.size()-1);
        cout<<"\nNhap lai vi chen: ";
        cin>>loc;
    }
    it=iList.begin();                   //di chuyen iterator toi vi tri chen
    for(int j=0;j<loc;j++)
        it++;
    iList.insert(it,n);                 //chen vao danh sach

    cout<<"\nDanh sach so nguyen sau khi chen:\n";
    for(it=iList.begin();it!=iList.end();it++)
        cout<<*it<<' ';
    cout<<"\n\nXoa mot phan tu khoi danh sach tai vi tri: ";
    cin>>loc;
    while(loc>iList.size()-1)           //kiem tra vi tri nhap vao
    {
        cout<<"\nKhong co vi tri nay, vi tri cuc dai la "
            <<(iList.size()-1);
        cout<<"\nNhap lai vi chen: ";
        cin>>loc;
    }
    it=iList.begin();                   //di chuyen iterator toi vi tri chen
```

```

for(int j=0;j<loc;j++)
    it++;
iList.erase(it);    //xoa mot phan tu khoi danh sach

cout<<"\nDanh sach so nguyen sau khi xoa:\n";
for(it=iList.begin();it!=iList.end();it++)
    cout<<*it<<' ';

iList.sort();    //sap xep danh sach theo chieu tang
cout<<"\nDanh sach so nguyen sau khi sap xep:\n";
for(it=iList.begin();it!=iList.end();it++)
    cout<<*it<<' ';
}

```

Đầu tiên chương trình khởi tạo danh sách tới một mảng số nguyên. Sau đó nó định nghĩa một **iterator** danh sách, **it**, để dùng cho việc hiển thị danh sách và trở tới phần tử cần chèn hoặc xóa. Số nguyên chèn vào danh sách và vị trí chèn được nhập vào từ người sử dụng. Vị trí chèn được chương trình kiểm tra, nếu lớn hơn hoặc bằng số phần tử hiện có của danh sách thì bắt nhập lại. Vị trí xóa nhập vào từ người sử dụng cũng được kiểm tra như vậy. Sau mỗi lần chèn hoặc xóa chương trình đều hiển thị toàn bộ danh sách để xem có thực hiện được không.

Cuối chương trình chúng ta sử dụng hàm thành viên **sort()** để sắp xếp danh sách theo chiều tăng dần (mặc định). Cũng có một giải thuật sắp xếp **sort()** nhưng chúng ta không sử dụng nó bởi vì phiên bản hàm thành viên hiệu suất hơn.

Sau đây là vài mẫu tương tác với chương trình:

Danh sach so nguyen ban dau:

10 50 90 20 40 30 80 60

Nhap vao so nguyen can chen vao danh sach: 70

Vị trí chèn: 3

Danh sach so nguyen sau khi chen:

10 50 90 70 20 40 30 80 60

Xoa mot phan tu khoi danh sach tai vi tri: 3

Danh sach so nguyen sau khi xoa:

10 50 90 20 40 30 80 60

Danh sach so nguyen sau khi sap xep:

10 20 30 40 50.60 80 90

10.2.5. Tìm một phần tử trong danh sách dùng giải thuật **find()**

Côngtenơ danh sách không có hàm thành viên tìm kiếm, bởi vậy để tìm kiếm một phần tử trong danh sách chúng ta phải sử dụng giải thuật **find()**. Bản 10-10, **FINDITEM**, cho thấy công việc này.

Listing 10-10 FINDITEM

```

//finditem.cpp
//tim kiem mot phan tu trong danh sach dung giai thuat find()
#include<iostream.h>
#include<list>
#include<algorithm>
void main()
{
    using namespace std;
    int arr[]={10,50,90,20,40,30,80,60};
}

```



```

list<int> iList(arr,arr+8);           //khởi tạo danh sách từ một mảng
list<int>::iterator it;               //tạo một iterator danh sách
int n;
char ch;
do
{
    cout<<"\nNhập vào số nguyên cần tìm: ";
    cin>>n;
    it=find(iList.begin(),iList.end(),n);
    if(it!=iList.end())
        cout<<"\nĐã tìm thấy số này trong danh sách.";
    else
        cout<<"\nKhông tìm thấy số này trong danh sách.";
        cout<<"\n\tCó tìm nữa không(c/k)?";
        cin>>ch;
}
while(ch!='k');
}

```

Chúng ta khởi tạo danh sách từ một mảng số nguyên cố định. Sau đó để người sử dụng nhập vào một số nguyên cần tìm trong danh sách. Giải thuật **find()** được dùng để tìm số nguyên đó trong danh sách. Nếu tìm thấy **find()** trả về một **iterator** trỏ tới vị trí của số tìm được. Nếu không tìm thấy **find()** trả về một **iterator** trỏ tới phần tử quá cuối danh sách.

Để người sử dụng có thể tìm nhiều lần chúng ta sử dụng vòng lặp **do while**. Chương trình chỉ kết thúc khi người sử dụng không muốn tìm kiếm nữa.

10.3. HÀNG ĐỢI HAI ĐẦU (DEQUE)

Hàng đợi hai đầu là một biến dạng của một vectơ. Giống như một vectơ, nó trợ giúp truy nhập ngẫu nhiên dùng toán tử []. Tuy nhiên, không giống như một vectơ (mà giống một danh sách), một hàng đợi hai đầu có thể truy nhập cả trước và sau. Nó là một vectơ hai đầu, có trợ giúp hàm thành viên **push_front()**, **pop_front()** và **front()**.

Bộ nhớ được cấp phát cho vectơ và hàng đợi là khác nhau. Một vectơ luôn luôn chiếm một vùng nhớ liên tiếp nhau. Nếu một vectơ phát triển quá lớn, nó có thể cần được di chuyển tới một vị trí mới sao cho vừa với nó. Trái lại, một hàng đợi hai đầu có thể lưu trữ trong vài vùng nhớ không liên tiếp nhau; nó được phân đoạn. Một hàm thành viên, **capacity()**, trả về số phần tử lớn nhất mà một vectơ có thể lưu trữ mà không cần di chuyển, nhưng **capacity()** không được định nghĩa cho hàng đợi hai đầu bởi vì hàng đợi hai đầu không cần di chuyển. Bản 10-11 trình bày chương trình ví dụ DEQUE.

Listing 10-11 DEQUE

```

//deque.cpp
//minh họa push_back(),push_front()
#include<iostream.h>
#include<deque>
#include<algorithm>
void main()
{
    using namespace std;
    deque<int> deq;           //tạo một hàng đợi hai đầu rỗng
    deq.push_back(30);        //đặt dữ liệu vào từ phía sau
    deq.push_back(40);
    deq.push_back(50);
}

```

```

deq.push_front(20);           //dat du lieu vao tu phia truoac
deq.push_front(10);

deq[2]=33;                    //thay doi muc du lieu o giua
for(int j=0;j<deq.size();j++)
    cout<<deq[j]<<' ';      //hien thi cac muc du lieu
}

```

Chúng ta đã thấy các ví dụ về **push_back()**, **push_front()** và toán tử []. Chúng làm việc với hàng đợi giống như với các côngtenơ khác. Kết quả của chương trình là:

```
10 20 33 40 50
```

Hình 10-2 cho thấy một vài hàm thành viên quan trọng cho ba côngtenơ tuần tự.

10.3.1. Hàm thành viên insert(), erase()

Để chèn một phần tử vào hàng đợi hai đầu tại một vị trí nào đó, chúng ta sử dụng hàm thành viên **insert()** và để xóa một phần tử tại một vị trí nào đó chúng ta sử dụng hàm thành viên **erase()**. Bản 10-12, DEQUE2, cho thấy hai hàm này làm việc như thế nào.

Listing 10-12 DEQUE2

```

//deque2.cpp
//minh hoa insert() va erase()
#include<iostream.h>
#include<deque>
#include<algorithm>
void main()
{
    using namespace std;
    int arr[]={10,20,30,40,50,60};           //mang so nguyen
    deque<int> deq(arr,arr+6);              //khoi tao hang doi toi mang
    int j;
    cout<<"\nHang doi ban dau:\n";
    for(j=0;j<deq.size();j++)
        cout<<deq[j]<<' ';
    int n,loc;
    cout<<"\nNhap vao mot so nguyen de chen vao hang doi: ";
    cin>>n;
    cout<<"Vi tri chen: ";
    cin>>loc;
    while(loc>deq.size()-1)                  //kiem tra vi tri nhap vao
    {
        cout<<"\nKhong co vi tri nay, vi tri cuc dai la "
            <<(deq.size()-1);
        cout<<"\nNhap lai vi chen: ";
        cin>>loc;
    }
    deq.insert(deq.begin()+loc,n);           //chen vao hang doi
    cout<<"\nHang doi sau khi chen:\n";
    for(j=0;j<deq.size();j++)
        cout<<deq[j]<<' ';

    cout<<"\nXoa mot phan tu khoi hang doi tai vi tri: ";
    cin>>loc;
    while(loc>deq.size()-1)                  //kiem tra vi tri nhap vao

```

```

    {
        cout<<"\Khong co vi tri nay, vi tri cuc dai la "
            <<(deq.size()-1);
        cout<<"\nNhap lai vi chen: ";
        cin>>loc;
    }
    deq.erase(deq.begin()+loc);
    cout<<"\nHang doi sau khi xoa:\n";
    for(j=0;j<deq.size();j++)
        cout<<deq[j]<<' ';
}

```

Chương trình yêu cầu người sử dụng nhập vào một số nguyên muốn chèn vào hàng đợi và vị trí chèn. Sau đó chương trình chèn số nguyên đó vào hàng đợi sử dụng hàm thành viên **insert()**. Tiếp theo, nó yêu cầu người sử dụng nhập vào vị trí cần xóa một phần tử và xóa phần tử tại vị trí đó bằng hàm thành viên **erase()**. Các vị trí chèn và xóa mà người sử dụng nhập vào đều được kiểm tra. Sau mỗi lần chèn hoặc xóa chương trình đều hiển thị nội dung của hàng đợi để thấy việc chèn và xóa được thực hiện đúng.

Đây là kết quả của chương trình:

```

Hang doi ban dau:
10 20 30 40 50 60
Nhap vao mot so nguyen de chen vao hang doi: 45
Vi tri chen: 4

Hang doi sau khi chen:
10 20 30 40 45 50 60
Xoa mot phan tu khoi hang doi tai vi tri: 4

Hang doi sau khi xoa:
10 20 30 40 50 60

```

10.3.2. Tìm kiếm một phần tử trong hàng đợi dùng giải thuật **find()**

Ví dụ tiếp theo sử dụng giải thuật **find()** để tìm kiếm một phần tử trong hàng đợi. Bản 10-13 là chương trình **FINDEQUE**.

Listing 10-13 FINDEQUE

```

//findeq.cpp
//tim kiem mot phan tu trong hang doi
#include<iostream.h>
#include<deque>
#include<algorithm>
void main()
{
    using namespace std;
    int arr[]={10,7,1,5,2,9,8,15};           //mang so nguyen
    deque<int> deq(arr,arr+8);               //khoi tao hang doi toi mang
    deque<int>::iterator it;                 //iterator cua hang doi
    int n;
    char ch;
    do
    {
        cout<<"\nNhap vao so nguyen can tim: ";
        cin>>n;
        it=find(deq.begin(),deq.end(),n);
    }
}

```

```

if(it!=deq.end())
{
    cout<<"\nĐã tìm thấy số này trong hàng doi:\n";
    for(int j=0;j<deq.size();j++)
        cout<<deq[j]<<' ';           //hiển thị hàng doi
}
else
    cout<<"\nKhông tìm thấy số này trong hàng doi.";
    cout<<"\n\tCó tìm nữa không(c/k)?";
cin>>ch;
}
while(ch!='k');
}

```

Chúng ta khởi tạo hàng đợi tới một mảng số nguyên cố định. Sau đó định nghĩa một **iterator** của hàng đợi để lưu trữ giá trị do giải thuật **find()** trả về. Đây là vài mẫu tương tác với chương trình:

Nhập vào số nguyên cần tìm: 5

Đã tìm thấy số này trong hàng đợi:
10 7 1 5 2 9 8 15

Có tìm nữa không(c/k)?c

Nhập vào số nguyên cần tìm: 10

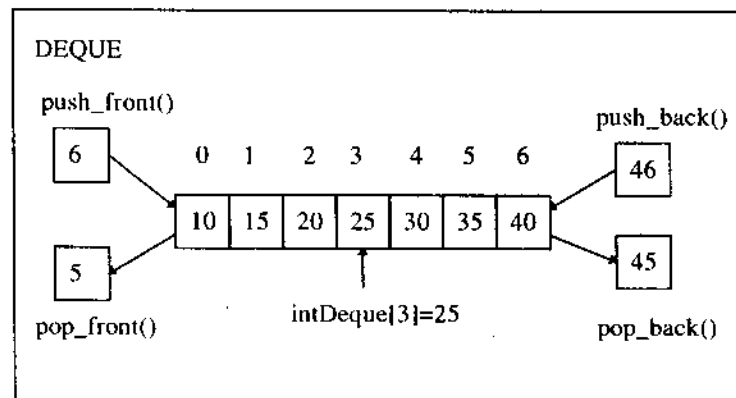
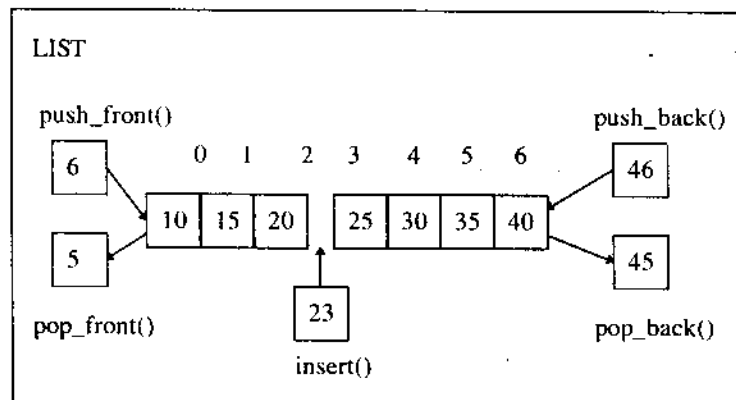
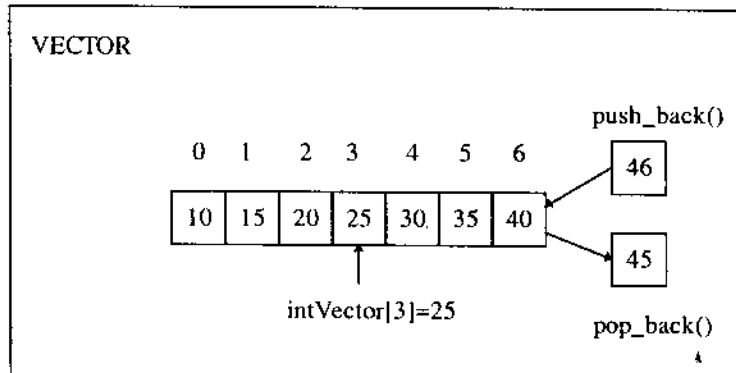
Đã tìm thấy số này trong hàng đợi:
10 7 1 5 2 9 8 15

Có tìm nữa không(c/k)?c

Nhập vào số nguyên cần tìm: 35

Không tìm thấy số này trong hàng đợi

Có tìm nữa không(c/k)?k



Hình 10-2. Các công thức tuần tự.

CÔNGTENƠ LIÊN KẾT

Hai loại côngtenơ liên kết chính trong STL là ánh xạ (**map**) và tập hợp (**set**). Một ánh xạ (đôi khi gọi là từ điển hay bảng ký hiệu) lưu trữ các cặp khóa và giá trị. Các khóa được sắp đặt theo thứ tự sắp xếp. Chúng ta có thể tìm một phần tử sử dụng khóa và điều này cho phép chúng ta truy nhập tới giá trị. Một côngtenơ ánh xạ (**map**) giống như một cuốn từ điển thông thường. Các từ được sắp xếp theo thứ tự alphabe tương đương với các khóa và các định nghĩa của các từ là các giá trị. Bởi vì các từ trong một cuốn từ điển được sắp xếp theo thứ tự alphabe nên chúng ta có thể nhanh chóng tra được một từ và định nghĩa của nó. Tương tự như vậy, trong chương trình, với một khóa cho trước chúng ta có thể nhanh chóng tìm được cặp khóa - giá trị.

Một tập hợp (**set**) tương tự như một cuốn từ điển nhưng nó chỉ lưu trữ các khóa; không có các giá trị. Chúng ta có thể nghĩ về một tập hợp như một danh sách các từ không có định nghĩa.

Trong cả tập hợp và ánh xạ, chỉ có một mục cụ thể duy nhất cho từng khóa được lưu trữ. Nó giống như một cuốn từ điển không cho phép nhiều hơn một mục cho mỗi từ, tức là mỗi từ chỉ có duy nhất một mục để tra. Tuy nhiên, STL có các phiên bản khác của tập hợp và ánh xạ. Trên thực tế có bốn loại côngtenơ liên kết: tập hợp (**set**), đa tập hợp (**multiset**), ánh xạ (**map**) và đa ánh xạ (**multimap**). Một đa tập hợp tương tự như một tập hợp và một ánh xạ, nhưng có thể gồm nhiều mục cụ thể của cùng một khóa.

Thuận lợi của côngtenơ liên kết là, với một khóa cho trước, có thể nhanh chóng truy nhập thông tin liên kết với các khóa này; nó nhanh hơn rất nhiều so với việc tìm kiếm từng mục qua một côngtenơ tuần tự. Trên các côngtenơ liên kết thông thường, chúng ta có thể nhanh chóng trở qua một côngtenơ theo thứ tự đã được sắp xếp.

Các côngtenơ liên kết sử dụng chung nhiều hàm thành viên với các côngtenơ khác. Tuy nhiên, một vài hàm thành viên, chẳng hạn **lower_bound()** và **upper_bound()**, chỉ tồn tại với các côngtenơ liên kết. Ngược lại, một vài hàm thành viên tồn tại với các côngtenơ khác, như họ hàm **push** và **pop**, không có trong các côngtenơ liên kết (chẳng có ý nghĩa gì để sử dụng họ hàm **push** và **pop** với các côngtenơ liên kết bởi vì các phần tử luôn luôn phải được chèn vào các vị trí theo thứ tự của chúng, không phải ở đầu hay ở cuối côngtenơ).

Chúng ta sẽ bắt đầu tìm hiểu về các côngtenơ liên kết với các tập hợp bởi vì nó đơn giản hơn.

11.1. TẬP HỢP VÀ ĐA TẬP HỢP (SET AND MULTiset)

Các tập hợp thường được dùng để lưu trữ các đối tượng của các lớp được định nghĩa bởi người sử dụng, chẳng hạn như các nhân viên trong một cơ sở dữ liệu (chúng ta sẽ có ví dụ về các đối tượng này ở chương 12). Tuy nhiên, cũng có thể sử dụng chúng để lưu trữ các phần tử đơn giản hơn như các chuỗi ký tự. Hình 11-1 cho thấy điều này trông như thế nào. Các đối tượng được sắp xếp theo một dạng thứ tự và được truy nhập bằng khóa.

Bản 11-1, SET, cho thấy một tập hợp các đối tượng lớp **string**. Ở đây chúng ta sử dụng trình biên dịch Borland C++ 5.0 nên file tiêu đề **CSTRING.H** chứa mô tả lớp **string**. Với các trình biên dịch khác có thể phải thay đổi file tiêu đề chẳng hạn như **BCSTRING.H**, **BSTRING.H** ...

Listing 11-1 SET

```
//set.cpp
//tap hop lưu trữ các đối tượng string
#include<iostream.h>
#include<set>
#include<cstring.h>
```

```

void main()
{
    using namespace std;
    string names[]={"Juanita","Robert","Marry","Amanda","Marie"};
                                //khởi tạo tập hợp tôi mang
    set<string,less<string> > nameSet(names,names+5);
    set<string,less<string> >::iterator iter; //iterator tro toi tap hop
    nameSet.insert("Yvette"); //chen mot vai ten vao tap hop
    nameSet.insert("Larry");
    nameSet.insert("Robert"); //khong co tac dung, vi da co ten nay
    nameSet.insert("Barry");
    nameSet.erase("Marry"); //xoa mot ten khoi danh sach
                                //hien thi kích thước của tập hợp
    cout<<"\n Size= "<<nameSet.size()<<endl;
    iter=nameSet.begin(); //hien thi danh sach ten
    while(iter!=nameSet.end())
        cout<<*iter++<<"\n";
    string searchName; //lay ten tu nguoi su dung
    cout<<"\nNhap vao mot ten de tim kiem: ";
    cin>>searchName;
    iter=nameSet.find(searchName);
    if(iter==nameSet.end())
        cout<<"Ten "<<searchName<<" khong co trong danh sach.";
    else
        cout<<"Ten "<<*iter<<" co trong danh sach.";
}

```

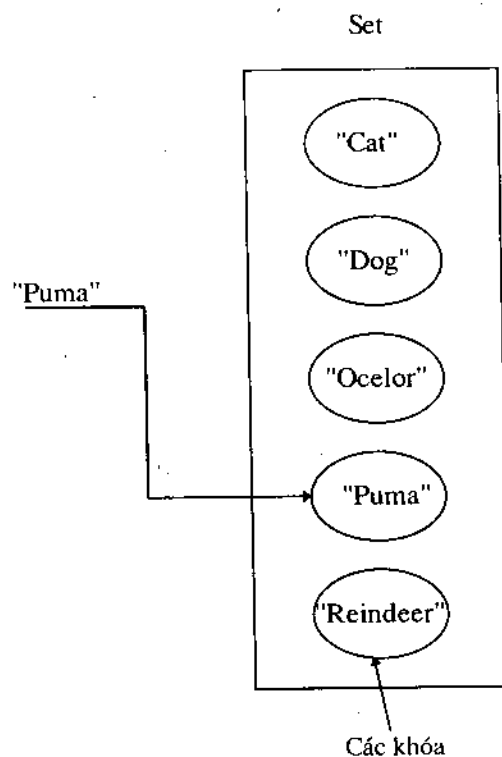
Để định nghĩa một tập hợp, chúng ta xác định kiểu đối tượng được lưu trữ (trong trường hợp này là lớp **string**) và đối tượng hàm sẽ dùng để sắp xếp các phần tử của tập hợp. Ở đây chúng ta sử dụng đối tượng hàm **less<>()** áp dụng cho các đối tượng **string**.

Như chúng ta có thể thấy, một tập hợp có một giao diện tương tự như các côngtenơ khác. Có thể khởi tạo một tập hợp tới một mảng và chèn các phần tử mới vào một tập hợp bằng hàm thành viên **insert()**. Để hiển thị nội dung một tập hợp, chúng ta sử dụng một **iterator** để trở qua cả tập hợp.

Để tìm một mục cụ thể trong tập hợp, chúng ta sử dụng hàm thành viên **find()** (các côngtenơ tuần tự không có hàm thành viên **find()**, chúng phải sử dụng giải thuật **find()** để tìm kiếm).

Còn để xóa một mục cụ thể, chúng ta sử dụng hàm thành viên **erase()** với đối số là mục cần xóa, không cần chỉ ra vị trí của mục cần xóa. Dưới đây là vài mẫu tương tác với chương trình SET, ở đó người sử dụng nhập vào tên "George" để tìm kiếm:

Kích thước = 7



Hình 11-1. Tập hợp các đối tượng chuỗi.

```

Amanda
Barry
Juanita
Larry
Marie
Robert
Yvette
Nhập vào tên cần tìm: George
Tên George không có trong tập hợp

```

Tất cả các phép toán cơ bản với tập hợp: chèn, xóa, hiển thị, tìm kiếm đều được minh họa trong chương trình SET. Bây giờ chúng ta cùng xem xét một ví dụ khác, ở đó cho thấy một cặp hàm thành viên quan trọng chỉ tồn tại với các côngtenơ liên kết: **lower_bound()** và **upper_bound()**. Bản 11-2, SETRANGE, trình bày ví dụ này.

Listing 11-2 SETRANGE

```

//setrange.cpp
//kiểm tra các khoảng với một tập hợp
#include<iostream.h>
#include<set>
#include<string>
void main()
{
    set<string,less<string>> organic;    //tập hợp các đối tượng chuỗi
    set<string,less<string>>::iterator iter;//iterator trở tới tập hợp
    organic.insert("Metan");             //chèn các hợp chất hữu cơ
    organic.insert("Axetilen");
    organic.insert("Etilen");
    organic.insert("Benzen");
    organic.insert("glucozo");
    organic.insert("Lipit");
    organic.insert("Protein");
    organic.insert("Saccarozo");
    organic.insert("Phenol");
    organic.insert("Ruou Etylic");
    organic.insert("Tinh Bột");
    iter=organic.begin();
    while(iter!=organic.end())
        cout<<*iter++<<"\n";
    string lower,upper;
    cout<<"\nNhập vào khoảng (ví dụ c czz): ";
    cin>>lower>>upper;
    iter=organic.lower_bound(lower);
    while(iter!=organic.upper_bound())
        cout<<*iter++<<"\n";
}

```

Đầu tiên chương trình hiển thị toàn bộ tập hợp các hợp chất hữu cơ. Sau đó thông báo cho người sử dụng gõ vào hai giá trị khóa và chương trình hiển thị các khóa nằm trong khoảng hai khóa này.

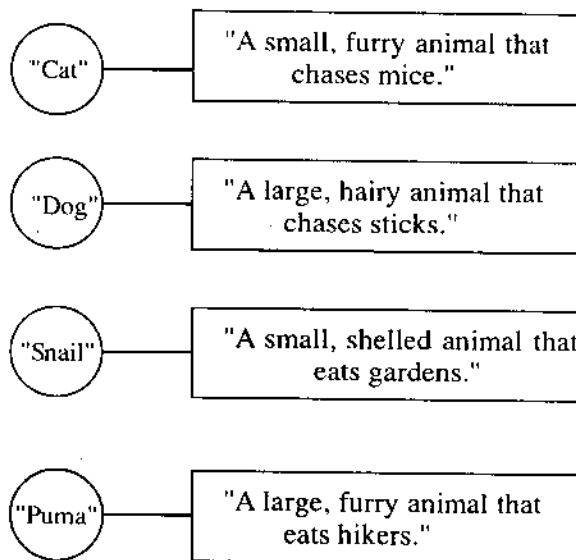
Hàm thành viên **lower_bound()** có một đối số có giá trị cùng kiểu với khóa. Nó trả về một **iterator** trở tới mục đầu tiên không nhỏ hơn đối số này (ở đây ý nghĩa của từ "nhỏ hơn" được xác định bởi đối tượng hàm dùng trong định nghĩa tập hợp). Hàm thành viên **upper_bound()** trả về một **iterator** trở tới mục đầu tiên lớn hơn đối số của nó. Ngoài ra, các hàm thành viên này cho phép ta truy nhập tới một khoảng giá trị xác định.

Một đa tập hợp (**multiset**) y như một tập hợp, nó cũng có tất cả các hàm thành viên của một tập hợp. Tuy nhiên, nó khác tập hợp ở chỗ nó cho phép lưu trữ các khóa giống nhau, trong khi đó tập hợp chỉ có các khóa duy nhất. Bởi vì tập hợp chỉ cho phép duy nhất một khóa nên trong chương trình SET, khi chúng ta chèn vào một tên đã có trong tập hợp thì tên đó không được chèn vào tập hợp. Đối với đa tập hợp thì tên này vẫn được đưa vào.

11.2. ÁNH XẠ VÀ ĐA ÁNH XẠ (MAP AND MULTIMAP)

Một ánh xạ (**map**) lưu trữ các khóa và các giá trị. Các khóa có thể là các chuỗi hoặc các số. Các giá trị thường là các đối tượng phức tạp, chẳng hạn như các đối tượng **employee** hoặc **student_record**, mặc dù chúng cũng có các số và các chuỗi. Ví dụ, khóa có thể là một từ và giá trị có thể là một số biểu diễn số lần từ xuất hiện trong văn bản. Một ánh xạ như vậy có thể được sử dụng để tạo một bảng tần suất. Hoặc khóa có thể là một mã nhân viên và giá trị liên kết có thể là file nhân viên.

Hình 11-2 trình bày một ánh xạ mà ở đó các khóa là các từ và các giá trị là các cụm từ. Nó tương tự như một cuốn từ điển thông thường.



Hình 11-2. Một ánh xạ các cặp từ - ngữ.

11.2.1. Trỏ qua các ánh xạ

Bản 11-3, MAP, minh họa sự sắp xếp: một từ điển tạo ra các cặp từ - ngữ được lưu trữ trong một ánh xạ.

Listing 11-3 MAP

```

//map.cpp
//môt ánh xạ được sử dụng như một cuốn từ điển tiếng Anh
#include<iostream.h>
#include<map>
void main()
{
    enum{SIZE=80};
    char def[SIZE]; //định nghĩa chuỗi kí tự C thông thường
    string word=""; //khóa là đối tượng chuỗi
    typedef map<string,string,less<string>> map_type; //ten kiểu ngôn
    map_type diction; //định nghĩa một từ điển
    //chèn vào từ điển một vài mục mẫu
    diction.insert(map_type::value_type("cat",
                                        "A small furry animal that chases mice.");
    diction.insert(map_type::value_type("dog",
                                        "A large hairy animal that chases sticks.");

    while(true)
    {
        cout<<"\nNhập vào từ (hoặc\"Thoi\" để kết thúc:";
        cin>>word;
        if(word=="Thoi")
  
```



```

        break;
    cout<<"Nhập vào định nghĩa:";
    cin.get(def,SIZE);          //đọc cả ký tự trống
    diction.insert(map_type::value_type(word,def));
}
map_type::iterator iter;       //tạo một iterator
iter=diction.begin();          //đặt tại đầu của từ điển
cout<<endl<<endl;
while(iter!=diction.end())     //hiển thị từ -- định nghĩa
{
    cout<<(*iter).first<<"--"<<(*iter).second<<endl;
    ++iter;
}
}

```

Các giá trị khóa "cat" và "dog" và các định nghĩa của chúng được cài đặt trong ánh xạ bởi chương trình, nhưng người sử dụng được khuyến khích nhập vào các cặp từ - định nghĩa khác.

Trong chương trình MAP, người sử dụng không thể chèn các định nghĩa cho "cat" hoặc "dog" bởi vì các khóa này đã có trong côngtenơ và một ánh xạ chỉ cho phép duy nhất một khóa với một giá trị cho trước. Một đa ánh xạ sẽ cho phép giá trị khóa giống nhau xuất hiện hơn một lần.

Biểu thức cho kiểu dữ liệu của côngtenơ ánh xạ là:

```
map<string,string,less<string>>
```

hơi khó viết. Bởi vì biểu thức này xuất hiện nhiều lần trong chương trình nên chúng ta sử dụng định danh **typedef** để đưa nó vào một biểu thức ngắn hơn là **map_type**:

```
typedef map<string,string,less<string>> map_type;
```

Các định danh **typedef** được sử dụng chủ yếu trong cách này khi làm việc với các côngtenơ STL mẫu hóa, đặc biệt là các ánh xạ và tập hợp.

Định danh **typedef** được sử dụng để cho một kiểu dữ liệu một tên khác. Đối với các mẫu có vài đối số thì tên kiểu trở nên lớn và khó viết. Trong các trường hợp này, sử dụng **typedef** sẽ làm cho các tên dài gọn hơn và rõ ràng hơn.

11.2.2. Các cặp

Đối số cho hàm **insert()** là **map_type::value(word,def)** có vẻ hơi nặng nề. Nó xuất hiện bởi vì các ánh xạ và đa ánh xạ thực sự giữ các đối tượng thuộc kiểu **pair** (cặp). Lớp **pair** tự mẫu hóa để các đối tượng của nó có thể chứa hai đối tượng của hai lớp khác. Chúng ta cũng có thể tạo các đối tượng **pair** của riêng mình:

```
pair<string,string> dictpair=make_pair("cat","A cat is a ...");
```

Trong file tiêu đề MAP, **value_type** là một cặp giá trị:

```
typedef pair<const key,T> value_type;
```

Điều này làm cho việc cung cấp các giá trị cho **insert()** thuận tiện hơn, như được làm trong ví dụ này. Chú ý rằng các chuỗi **char***, chẳng hạn như "cat" được tự động chuyển đổi thành các đối tượng **string** để phù hợp với kiểu mà **value_type** mong đợi.

Tên của hai hàm thành viên của **pair** là **first** và **second**. Các hàm thành viên này được dùng trong lặp **while** để hiển thị nội dung của ánh xạ:

```
cout<<(*iter).first<<" -- "<<(*iter).second<<endl;
```

Nói tóm lại, khi chúng ta kết nối các giá trị để đặt chúng vào một ánh xạ, chúng ta kết nối chúng với **value_type**, nhưng để tách chúng ra, chúng ta sử dụng **first** và **second**.

11.2.3. Toán tử []

Chúng ta có thể sử dụng các hàm thành viên `find()` hoặc `lower_bound()` / `upper_bound()` để tìm kiếm một mục cụ thể trong một ánh xạ, y như trong tập hợp. Tuy nhiên, chúng ta cũng có thể sử dụng toán tử []. Chúng ta sẽ minh họa cách làm này với một ánh xạ lưu giữ các cặp gồm các số và các cách diễn đạt hàng hải. Ở thế kỷ 18, các con tàu luôn mang theo các quyển sách liệt kê hàng trăm các thông báo cờ. Để gửi một thông báo, một con tàu sẽ kéo cờ chỉ thị một số xác định; một con tàu khác, có thể cách đó vài dặm, sẽ đọc các cờ bằng kính viễn vọng và sau đó sử dụng số đọc được để tra thông báo trong sách cờ của riêng nó. Chương trình ví dụ lưu trữ các cặp bao gồm các số cờ và các thông báo tương ứng. Nếu các con tàu ở thế kỷ 18 được trang bị máy tính thì chương trình sẽ giúp cho việc tra thông báo nhanh hơn rất nhiều. Bản 11-4 trình bày ví dụ MAPBRACK.

Listing 11-4 MAPBRACK

```
//mapbrack.cpp
//minh hoa toan tu [] dung cho cac anh xa
#include<iostream.h>
#include<map>
void main()
{
    typedef map<long,char*,less<int>>> map_type;
    map_type flaglist;           //anh xa giu cac thong bao co
    long code_number;
    flaglist.insert(map_type::value_type(14072,
        "I am listing sharly and soon founder."));
    flaglist.insert(map_type::value_type(12023,
        "The enemy is within sight and approaching rapidly."));
    flaglist.insert(map_type::value_type(16067,
        "I have dispatches. Prepare to receive out long boat."));
    flaglist.insert(map_type::value_type(13045,
        "Fall in line astern of me."));
    flaglist.insert(map_type::value_type(19092,
        "Stand off. This coast is rocky anduncharted."));
    while(true)                 //lay so ma tu nguoi su dung
    {
        cout<<"\n\nNhap vao so ma co (0 de ket thuc):";
        cin>>code_number;
        if(!code_number)
            break;
        cout<<"Thong bao la:"<<endl;
        cout<<flaglist[code_number];    //truy nhap gia tri voi khoa
    }
}
```

Biểu thức `flaglist[code_number]` sử dụng khóa (số mã) làm chỉ số và trả về giá trị ánh xạ được liên kết, trong trường hợp này là một chuỗi `char*` và sau đó nó được hiển thị. Toán tử chồng [] cung cấp một cách rõ ràng trực quan để truy nhập toàn bộ một ánh xạ.

11.3. CÁC PHIÊN BẢN BẢNG BĂM (HASH TABLE VERSIONS)

Có các phiên bản khác cho tất cả các côngtenơ liên kết mà có thể tăng tốc độ truy nhập tới các mục dữ liệu trong côngtenơ. Các phiên bản này sử dụng một bảng băm (hash table) làm cơ chế cài đặt bên trong. Bất lợi của chúng là không cho phép trở nhanh qua một côngtenơ đã được sắp xếp. Các côngtenơ này gọi là `hash_set`, `hash_multiset`, `hash_map` và `hash_multimap`. Chúng được sử dụng như các phiên bản thông thường, nhưng thích hợp khi tốc độ tìm kiếm quan trọng hơn khả năng trở qua côngtenơ.

CÁC KIỂU DỮ LIỆU TRỪ TƯỢNG

Các kiểu dữ liệu trừu tượng là các côngtenơ được tạo ra từ các côngtenơ cơ bản. Nó là một loại côngtenơ được đơn giản hóa để nhấn mạnh vào các khía cạnh xác định của một côngtenơ cơ bản hơn. Các côngtenơ được cài đặt trong STL là ngăn xếp (*stack*), hàng đợi (*queue*), hàng đợi ưu tiên (*priority queue*) và cây nhị phân (*tree*).

12.1. NGĂN XẾP

Như trên ta đã biết, ngăn xếp được hoạt động theo cơ chế vào trước - ra sau (Last In - First Out hay viết tắt là LIFO). Điều này có nghĩa là, mục dữ liệu nào được đặt vào ngăn xếp sau cùng thì sẽ được lấy ra đầu tiên. Kiểu xử lý "vào trước - ra sau" này xuất hiện trong nhiều ứng dụng, vì vậy một cấu trúc dữ liệu trừu tượng thể hiện ý tưởng này là rất có ích. Khi xem xét điều gì xảy ra trong hệ thống máy tính lúc các chương trình con được gọi, chúng ta sẽ thấy một trong những ứng dụng quan trọng của ngăn xếp. Hệ thống hay chương trình chính cần phải ghi nhớ nơi xuất hiện lời gọi chương trình con để có thể quay lại nơi đó khi chương trình con được thực hiện xong. Ngoài ra nó còn cần ghi nhớ tất cả các biến địa phương, các thanh ghi của CPU cùng các thông tin tương tự sao cho thông tin không bị mất đi trong quá trình chương trình con đang làm việc. Chúng ta có thể coi tất cả các thông tin trên được lưu trong một vùng nhớ lớn, mỗi chương trình con có một vùng riêng tạm thời. Giả thiết là chúng ta có ba chương trình con được gọi là A, B, C và đồng thời cũng giả thiết là chương trình A gọi chương trình B và chương trình B gọi chương trình C. Như vậy, B sẽ chưa thể kết thúc công việc của mình cho đến khi C kết thúc và quay lại B. Tương tự như vậy, tuy A được bắt đầu trước tiên nhưng nó lại phải kết thúc sau cùng vì phải đợi cho đến khi B kết thúc và quay lại A. Trình tự hoạt động của các chương trình con này có thể tổng kết lại là mang tính chất "vào trước - ra sau". Nếu chúng ta coi nhiệm vụ của máy tính là gán các vùng nhớ tạm thời cho các chương trình con để chúng sử dụng thì các vùng nhớ này cần phải được định vị trong một danh sách với cùng tính chất "vào trước - ra sau" như trên, đó chính là *stack*.

Như vậy, ngăn xếp sẽ gồm một danh sách hay một dãy các mục dữ liệu trong đó các phép toán thêm vào hay bớt đi đều được thực hiện ở một đầu được gọi là đỉnh của ngăn xếp. Hay nói một cách khác, các phép toán cơ bản cho một ngăn xếp bao gồm:

1. Stack dùng để tạo một ngăn xếp rỗng
2. Empty xác định ngăn xếp có rỗng hay không
3. Pop tìm lại và lấy ra một mục ở đỉnh của ngăn xếp
4. Push thêm một phần tử mới vào đỉnh của ngăn xếp.

Bước đầu tiên để cài đặt một cấu trúc dữ liệu kiểu ngăn xếp là chọn một cấu trúc lưu trữ để lưu trữ các phần tử của ngăn xếp. Vì ngăn xếp là một dãy các mục dữ liệu, chúng ta có thể dùng mảng hoặc danh sách liên kết để lưu trữ các mục này.

Để hiểu rõ hoạt động của ngăn xếp, chúng ta sẽ xét bài toán chuyển đổi một số nguyên dương thành một số nhị phân. Thuật toán chuyển đổi là dùng phép chia liên tiếp cho 2 với các số dư là các chữ số nhị phân trong biểu diễn cơ số hai. Số dư tạo ra cuối cùng là số nhị phân đầu tiên. Bởi vậy, để hiển thị số nhị phân theo đúng thứ tự thì các bit (các số dư) phải được hiển thị theo cách "được tạo ra cuối cùng thì được hiển thị đầu tiên". Ngăn xếp thích hợp cho việc hiển thị các số dư này. Sau đây là thuật toán chuyển đổi cơ số như vậy.

1. While $N \neq 0$ thực hiện các bước sau:
 - a) Tính số dư Remainder khi chia N cho 2
 - b) Đặt Remainder vào đỉnh của ngăn xếp chứa các số dư
 - c) Thay N bằng phần nguyên của kết quả phép chia N cho 2.
2. While ngăn xếp chứa số dư không rỗng, thực hiện các bước sau:
 - a) Lấy Remainder ra từ đỉnh của ngăn xếp chứa các số dư
 - b) Hiển thị Remainder

Với côngtenơ ngăn xếp (stack) được cài đặt sẵn trong STL, chương trình chuyển đổi cơ số TenToTwo được viết như sau:

Dưới đây là chương trình TenToTwo.

```
//tentotwo.cpp
#include<iostream.h>
#include<stack>
void main()
{
    stack<int> s;           //tao mot ngan xep rong
    int n,remainder;
    char ch;
    do
    {
        cout<<"\nNhap vao mot so nguyen duong: ";
        cin>>n;
        while (n)
        {
            remainder=n%2;
            s.push(remainder);
            n/=2;
        }
        cout<<"\nBieu dien co so 2 la: ";
        while(!s.empty())
            cout<<s.pop();
        cout<<"\n\nCo tiep tục nữa không (c/k)?";
        cin>>ch;
    }
    while (ch!='k');
}
```

12.2. HÀNG ĐỢI

Trong khi ngăn xếp là một cấu trúc vào "vào sau ra trước" (LIFO) thì hàng đợi lại là cấu trúc "vào trước ra trước" (First In - First Out - FIFO).

Giống như ngăn xếp, hàng đợi là một kiểu danh sách đặc biệt, trong đó các phép toán chen và xóa được giới hạn ở một đầu của danh sách. Nó khác ngăn xếp ở chỗ, các phần tử được đẩy vào ở một đầu nhưng lấy ra từ một đầu khác của hàng đợi.

Các hàm thành viên cơ bản của hàng đợi là pop(), push(), empty. Đây là các phép toán cơ bản với hàng đợi.

12.3. HÀNG ĐỢI ƯU TIÊN

Trong hàng đợi ưu tiên (**priority queue**), một sự ưu tiên nào đó được gán cho mỗi mục và mỗi mục này được lưu trữ sao cho những mục được ưu tiên nhất ở đầu hàng đợi và được lấy ra trước những mục có sự ưu tiên thấp hơn. Không giống như hàng đợi, ở đó các mục được đặt vào ở một đầu và lấy ra ở đầu kia theo đúng thứ tự đặt vào, trong hàng đợi ưu tiên, các mục được đặt vào một đầu nhưng lấy ra ở đầu kia mục cao nhất (mục có sự ưu tiên cao nhất).

12.4. CÂY NHỊ PHÂN

Chúng ta đã biết, danh sách liên kết (**Linked List**) là cấu trúc rất có ích để xử lý những danh sách động có độ dài cực đại không được biết trước và độ dài của chúng có thể thay đổi được một cách đáng kể bởi các thao tác chèn và xóa. Nhưng một điều không may là giải thuật tìm kiếm nhị phân không thể dùng cho các danh sách liên kết mặc dù nó hiệu quả hơn giải thuật tìm kiếm tuyến tính. Cây tìm kiếm nhị phân cho phép sử dụng giải thuật tìm kiếm nhị phân với cấu trúc liên kết. Chúng là trường hợp đặc biệt của một cấu trúc tổng quát hơn với nhiều ứng dụng phong phú được gọi là cây.

Một cây bao gồm một tập hợp hữu hạn các phần tử được gọi là **nút** (hay **đỉnh**) và một tập hợp hữu hạn những **cạnh có hướng** để nối các cặp nút với nhau. Nếu cây không rỗng thì sẽ có một nút được gọi là **nút gốc** (**root**), không có cạnh nào hướng đến nó, nhưng từ nút này, bằng cách đi theo một dãy duy nhất các cạnh liên tiếp, có thể đến được một nút bất kỳ trong tất cả các nút còn lại. Các nút có thể truy nhập trực tiếp từ một nút cho trước (bằng cách chỉ đi qua một cạnh) được gọi là **con** (**child**) của nút ấy và nút ấy được gọi là **bố** (**parent**) của những nút con đó.

Một cây trong đó mỗi nút có nhiều nhất hai con được gọi là **cây nhị phân** (**Binary Tree**). Chúng có thể cài đặt bằng các cấu trúc liên kết trong đó mỗi nút có hai liên kết, một liên kết chỉ đến con bên trái (hoặc là rỗng nếu không có con bên trái) và một liên kết chỉ đến con bên phải (nếu có). Có thể truy nhập đến một nút bất kỳ nếu ta bảo trì được một con trỏ chỉ đến nút gốc của cây.

Cây nhị phân trên được gọi là **cây tìm kiếm nhị phân** (**Binary Search Tree - BST**) nếu nó đảm bảo tính chất sau: giá trị ở mỗi nút lớn hơn giá trị của con bên trái của nó (nếu có) và nhỏ hơn giá trị của con bên phải của nó (nếu có). Cây nhị phân thỏa mãn tính chất trên được gọi là cây tìm kiếm nhị phân vì có thể dùng giải thuật rất giống với giải thuật tìm kiếm nhị phân cho các danh sách để thực hiện việc tìm kiếm trên nó.

Cây nhị phân trong C++ là một côngtenơ liên kết, ở đó mỗi nút có hai con trỏ: một trỏ đến nút trái, một trỏ đến nút phải.

Các hàm thành viên của cây nhị phân là: `insert()`, `erase()`...

LƯU TRỮ CÁC ĐỐI TƯỢNG ĐƯỢC ĐỊNH NGHĨA BỞI NGƯỜI SỬ DỤNG

Đỉnh cao của STL là có thể sử dụng nó để lưu trữ và thao tác trên các đối tượng của các lớp mà chúng ta tạo ra (hoặc do một ai đó viết). Trong chương này chúng ta sẽ có một số ví dụ về vấn đề này.

13.1. MỘT TẬP HỢP CÁC ĐỐI TƯỢNG PERSON

Chúng ta sẽ bắt đầu với lớp **person** bao gồm có tên, họ đệm và số điện thoại của một người. Chúng ta sẽ tạo vài đối tượng của lớp này và chèn chúng vào trong một tập hợp, làm như vậy để tạo một cơ sở dữ liệu danh sách điện thoại. Người sử dụng tương tác với chương trình bằng cách gõ vào tên của một người. Sau đó chương trình tìm kiếm trong danh sách và hiển thị dữ liệu cho người đó nếu nó tìm thấy. Chúng ta sẽ sử dụng một đa tập hợp để cho hai hoặc nhiều đối tượng **person** có thể có cùng tên. Bản 13-1 trình bày chương trình SETPERS.

Listing 13-1 SETPERS

```
//setpers.cpp
//sử dụng multiset để lưu trữ các đối tượng person
#include<iostream.h>
#include<multiset.h>
#include<cstring.h>
class person
{
private:
    string LastName;
    string FirstName;
    long PhoneNumber;
public:
    //default constructor
    person():LastName("blank"),FirstName("blank"),phoneNumber(0L)
    {}
    //3-arg constructor
    person(string lana,string fina,long pho):
        LastName(lana),FirstName(fina),PhoneNumber(pho)
    {}
    friend bool operator<(const person&,const person&);
    friend bool operator==(const person&,const person&);
    void display() const //display person's data
    {
        cout<<endl<<LastName<<","<<"\t"<<FirstName
            <<"\t\tPhone: "<<PhoneNumber;
    }
}; //end of class
//toán tử < dùng cho lớp person
bool operator<(const person& p1,const person& p2)
{
    if(p1.LastName==p2.LastName)
```

tên và họ đem trùng nhau và phải giống với người tìm kiếm. Lúc đó các hàm `lower_bound()` và `upper_bound()` sẽ tìm ra người mà người sử dụng gõ vào giống với người có trên danh sách.

13.2. MỘT DANH SÁCH CÁC ĐỐI TƯỢNG PERSON

Chúng ta có thể nhanh chóng tìm kiếm một người có tên biết trước trong một tập hợp hoặc một đa tập hợp, như trong ví dụ SETPERS. Tuy nhiên, nếu muốn chèn hoặc xóa một đối tượng `person` nhanh thì nên sử dụng một danh sách. Ví dụ, LISTPERS (bản 13-2) cho thấy cách làm này.

Listing 13-2 LISTPERS

```
//setpers.cpp
//sử dụng một đa tập hợp để giữ các đối tượng person
#include<iostream.h>
#include<set>
#include<string>
class person
{
private:
    string firstName;           //ten va ho dem
    string lastName;           //ten
    long   phoneNumber;         //so dien thoai
public:
    person():firstName("blank"),lastName("blank"),phoneNumber(0L)
    {}
    person(string fname,string lname,long phone):
    firstName(fname),lastName(lname),phoneNumber(phone)
    {}
    friend bool operator<(const person&,const person&);
    friend bool operator==(const person&,const person&);
    void display() const
    {
        cout<<endl<<firstName<<"\t"<<lastName<<"\t\t"<<phoneNumber;
    }
};
//toán tử <
bool operator<(const person& p1,const person& p2)
{
    if(p1.lastName==p2.lastName)
        return (p1.firstName<p2.firstName)? true:false;
    return (p1.lastName<p2.lastName)? true:false;
}
bool operator==(const person& p1,const person& p2)
{
    return
    (p1.lastName==p2.lastName && p1.firstName==p2.firstName)? true:false;
}

using namespace std;
void main()
{
    //tao cac doi tuong
    person pers1("Hoang Van"," Cong",8693959);
    person pers2("Nguyễn Thanh","Bình",8750370);
    person pers3("Nguyễn Thu","Hoa",8335787);
    person pers4("Kieu Duc","Hung",8826344);
```

```

person pers5("Pham Anh","Nam",8289106);
person pers6("Pham Thi ","Hang",8548666);
person pers7("Bui Trung","Kien",8548162);
person pers8("Tham Hai","Hoa",8854290);
person pers9("Nguyen Cong","Thanh",8681616);
multiset<person,less<person> > persSet;
multiset<person,less<person> >::iterator iter;

persSet.insert(pers1);
persSet.insert(pers2);
persSet.insert(pers3);
persSet.insert(pers4);
persSet.insert(pers5);
persSet.insert(pers6);
persSet.insert(pers7);
persSet.insert(pers8);
persSet.insert(pers9);
cout<<"\nSo nguoi = "<<persSet.size();
iter=persSet.begin();
while(iter!=persSet.end())
    (*iter++).display();
}

```

13.2.1. Tìm tất cả những người có tên biết trước

Chúng ta không thể sử dụng các hàm thành viên **lower_bound()** và **upper_bound()** bởi vì chúng ta đang làm việc với một danh sách chứ không phải một tập hợp hay ánh xạ. Thay vào đó chúng ta sử dụng giải thuật **find()** tìm tất cả những người có tên cho trước. Nếu hàm này báo cáo là tìm thấy, chúng ta phải sử dụng lại nó, bắt đầu từ người đứng sau người tìm thấy, để xem liệu còn người nào khác có tên như vậy không. Điều này làm phức tạp việc lập trình; chúng ta phải sử dụng một vòng lặp và hai **iterator**.

13.2.2. Tìm tất cả những người có số điện thoại biết trước

Nhớ rằng trong lớp **person**, toán tử chồng == so sánh hai phần tử dựa trên tên của chúng, cả tên và họ đệm. Điều này dẫn đến khó tìm kiếm một người có số điện thoại biết trước bởi vì giải thuật **find()** so sánh tên của người cần tìm kiếm với tên các người trên danh sách. Trong ví dụ này giải pháp tốn sức để tìm kiếm số điện thoại, trở qua danh sách và làm phép so sánh "bằng tay" số điện thoại mà chúng ta cần tìm với số trong danh sách.

```

if(snumber==( *iter).get_phone())
    .....

```

Việc này sẽ không nhanh bằng các hàm thành viên **find()** hoặc **lower_bound()/upper_bound()** được thiết kế để quản lý việc tìm kiếm (trong chương sau chúng ta sẽ trình bày một cách khác, sử dụng các đối tượng hàm).

Đầu tiên chương trình hiển thị tất cả các phần tử trong danh sách, sau đó yêu cầu người sử dụng nhập vào một tên và tìm người có tên như vậy. Sau đó nó lại yêu cầu nhập vào một số điện thoại và lại tìm người có số điện thoại đó.

13.3. MỘT DANH SÁCH CÁC ĐỐI TƯỢNG AIRTIME

Như trong ví dụ trước, chúng ta sẽ trình bày một danh sách các đối tượng **airtime** (nhớ rằng dữ liệu cho đối tượng **airtime** chỉ có giờ và phút, không có giây). Bản 13-3, LISTAIR, cho thấy một số

đối tượng **airtime**, lưu trữ trong danh sách, có thể được cộng với nhau như thế nào. Các công ty hàng không có thể sử dụng cách này để cộng lượng thời gian mà một phi công đã sử dụng trong không gian trong một tuần. Phép cộng được thực hiện bởi một toán tử chồng + trong lớp **airtime**.

Listing 13-3 LISTAIR

```
//listair.cpp
//adding airtime objects stored in a list
#include<iostream.h>
#include<list.h>
class airtime
{
private:
    int hours;    //0 to 23
    int minutes; //0 to 59
public:
    //default constructor
    airtime():hours(0),minutes(0)
    {}

    //2-arg constructor
    airtime(int h,int m):hours(h),minutes(m)
    {}

    void display() const    //output to screen
    {
        cout<<hours<<":"<<minutes;
    }

    void get()    //input from user
    {
        char dummy;
        cout<<"\nEnter airtime (format 12:59): ";
        cin>>hours>>dummy>>minutes;
    }

    airtime operator+(airtime right) const
    {
        //add numbers
        int temp_h = hours + right.hours;
        int temp_m = minutes + right.minutes;
        if(temp_m >= 60)
        {
            temp_h++;
            temp_m-=60;
        }
        return airtime(temp_h,temp_m);    //return sum
    }

    bool operator<(const airtime& at2) const
    {
        if(hours<at2.hours)
            return true;
        if(hours==at2.hours && minutes < at2.minutes)
            return true;
        return false;
    }

    bool operator==(const airtime& at2) const
    {
        if(hours==at2.hours && minutes == at2.minutes)
            return true;
        return false;
    }
}
```

```

    }
}; //end of class airtime
//-----
void main()
{
    char answer;
    airtime temp,sum;
    list<airtime> airlist; //list and iterator
    list<airtime>::iterator iter;
    do
    {
        temp.get(); //get airtime from user
        airlist.push_back(temp);
        cout<<"Enter another (y/n)? ";
        cin>>answer;
    }
    while(answer!='n');
    iter=airtime.begin(); //iterate through list
    while(iter != iterlist.end())
        sum=sum + *iter++; //add this airtime to sum
    cout<<"\nSum = ";
    sum.display(); //display sum
} //end main()
/*

```

Các giá trị **airtime** được người sử dụng nhập vào. Một khi đã có tất cả các giá trị này và lưu chúng trên danh sách, chương trình trở qua danh sách, cộng từng giá trị vào một đối tượng **airtime** gọi là **sum**. Cuối cùng hiển thị **sum** (chúng ta sẽ thấy một cách hay hơn nhiều để làm việc này ở chương sau). Đây là vài mẫu tương tác với chương trình LISTAIR:

```

Enter airtime (format 12:59): 0:19
Enter another (y/n)? y

```

```

Enter airtime (format 12:59): 2:13
Enter another (y/n)? y

```

```

Enter airtime (format 12:59): 1:45
Enter another (y/n)? n

```

```

Sum = 5:39

```

```

*/

```

Trong chương trình này, chúng ta tập trung vào sự tìm kiếm các phần tử cụ thể trong một côngtenơ sử dụng giải thuật **find()**. Tuy nhiên, cũng có thể áp dụng bất kỳ giải thuật STL nào cho các đối tượng của các lớp được định nghĩa bởi người sử dụng đặt trong các côngtenơ STL. Như chúng ta thấy, việc lưu trữ các đối tượng của hầu hết các lớp trong một côngtenơ STL và thao tác trên các đối tượng này với các giải thuật STL khá đơn giản.

CHƯƠNG 14

ĐỐI TƯỢNG HÀM

Các đối tượng hàm (function objects) được sử dụng rất nhiều trong STL như các đối số cho các giải thuật cụ thể. Chúng cho phép ta điều chỉnh hoạt động của các giải thuật này theo ý muốn. Trong chương 8, chúng ta đã đề cập đến các đối tượng hàm, ở đó có một ví dụ về đối tượng hàm định nghĩa trước **greater** được dùng để sắp xếp dữ liệu theo thứ tự ngược (giảm dần). Trong chương này chúng ta sẽ nói về các đối tượng hàm định nghĩa trước và trình bày cách tự viết các đối tượng hàm để có thể điều khiển những gì mà các giải thuật làm.

14.1. CÁC ĐỐI TƯỢNG HÀM ĐỊNH NGHĨA TRƯỚC

Các đối tượng hàm STL định nghĩa trước, đặt trong file tiêu đề **FUNCTION.H**, được chỉ ra trong bảng 14-1. Chữ **T** chỉ bất kỳ lớp nào, được viết bởi người sử dụng hay một kiểu dữ liệu cơ bản. Các biến **x** và **y** biểu diễn các đối tượng của lớp **T** được truyền tới các đối tượng hàm như các đối số.

Bảng 14-1. Các đối tượng hàm định nghĩa trước

Các đối tượng hàm định nghĩa trước	Các kiểu	Giá trị trả về
plus	T=plus(T,T)	x+y
minus	T=minus(T,T)	x-y
times	T=times(T,T)	x*y
divide	T=divide(T,T)	x/y
modulus	T=modulus(T,T)	x%y
negate	T=negate(T)	-x
equal_to	Bool=equal_to(T,T)	x==y
not_equal_to	Bool=not_equal_to(T,T)	x!=y
greater	Bool=greater(T,T)	x>y
less	Bool=less(T,T)	x<y
greater_equal	Bool=greater_equal(T,T)	x>=y
less_equal	Bool=less_equal(T,T)	x<=y
logical_and	Bool=logical_and(T,T)	x&& y
logical_or	Bool=logical_or(T,T)	x y
logical_not	Bool=logical_not(T)	! x

Có các đối tượng hàm cho các phép toán số học, so sánh và các phép toán logic. Ví dụ **SORTTEMP** trong chương 8 đã chỉ ra cách sử dụng một đối tượng hàm so sánh, **greater<>()**. Xét một ví dụ mà ở đó một đối tượng hàm số học có thể một lúc nào đó sẽ có ích. Ví dụ (một dạng khác của ví dụ **LISTAIR** ở chương 13) cho thấy cách sử dụng đối tượng hàm **plus<>()** để cộng tất cả các giá trị **airtime** trong một côngtenơ. Bản 14-1 là chương trình **PLUSAIR**.

Listing 14-1 PLUSAIR

```
//plusair.cpp
//uses accumulate() algorithm and plus function object
#include<iostream.h>
#include<list.h>
#include<algo.h>
class airtime
```

```

{
private:
    int hours;        //0 to 23
    int minutes;      //0 to 59
public:               //default constructor
    airtime():hours(0),minutes(0)
    {
        //2-arg constructor
    }
    airtime(int h,int m):hours(h),minutes(m)
    {
    }
    void display() const    //output to screen
    {
        cout<<hours<<":"<<minutes;
    }
    void get()            //input from user
    {
        char dummy;
        cout<<"\nEnter airtime (format 12:59): ";
        cin>>hours>>dummy>>minutes;
    }
    airtime operator+(airtime right) const
    {
        //add numbers
        int temp_h = hours + right.hours;
        int temp_m = minutes + right.minutes;
        if(temp_m >= 60)
        {
            temp_h++;
            temp_m-=60;
        }
        return airtime(temp_h,temp_m);    //return sum
    }
    bool operator<(const airtime& at2) const
    {
        if(hours<at2.hours)
            return true;
        if(hours==at2.hours && minutes < at2.minutes)
            return true;
        return false;
    }
    bool operator==(const airtime& at2) const
    {
        if(hours==at2.hours && minutes == at2.minutes)
            return true;
        return false;
    }
};    //end of class airtime
//-----
void main()
{
    char answer;
    airtime temp,sum;
    list<airtime> airtlist;    //list and iterator
    list<airtime>::iterator iter;
    do
    {

```

```

        temp.get();           //get airtime from user
        airlist.push_back(temp);
        cout<<"Enter another (y/n)? ";
        cin>>answer;
    }
    while(answer != 'n');
    //sum all the airtime
    sum=accumulate(airlist.begin(),airlist.end(),
                  airtime(0,0),plus<airtime>());
    cout<<"\nSum = ";
    sum.display();           //display sum
} //end main()

```

Chương trình này chủ yếu là giải thuật **accumulate()**, nó trả về tổng các phần tử có trong một khoảng phần tử. Bốn đối số cho **accumulate()** là các **iterator** của phần tử thứ nhất và cuối cùng trong khoảng, giá trị khởi đầu của tổng **sum** (thường là 0) và phép toán được áp dụng cho các phần tử. Trong ví dụ này chúng ta đã cộng các phần tử, cũng có thể trừ chúng, nhân chúng hoặc thực hiện các phép toán khác dùng các đối tượng hàm khác nhau.

Giải thuật **accumulate()** không chỉ dễ và rõ ràng hơn việc trở qua côngtenơ để cộng mà còn hiệu suất hơn (một phiên bản khác của **accumulate()** có thể thực hiện bất cứ phép toán nào chúng ta định nghĩa, hơn là chỉ cộng dữ liệu).

Đối tượng hàm **plus<>()** yêu cầu toán tử + phải được chồng cho lớp **airtime** (toán tử này có thể nên là một hàm hằng **const** bởi vì đó là những gì mà đối tượng hàm **plus<>()** mong đợi).

Các đối tượng hàm số học khác làm việc theo cách tương tự. Các đối tượng hàm logic chẳng hạn như **logical_and<>()** có thể được sử dụng trên các đối tượng của các lớp mà ở đó các phép toán này là có nghĩa.

14.2. TỰ VIẾT CÁC ĐỐI TƯỢNG HÀM

Nếu một trong các đối tượng hàm không làm được những gì chúng ta mong muốn, chúng ta có thể viết các đối tượng hàm của riêng chúng ta. Ví dụ tiếp theo cho thấy một lý do cần phải viết một đối tượng hàm riêng.

Sắp xếp một nhóm các phần tử dựa trên mối quan hệ được chỉ rõ trong toán tử < là dễ. Chỉ đơn giản chèn các phần tử vào một tập hợp hay một đa tập hợp sẽ có được loại sắp xếp này hoặc nếu các phần tử ở trong một côngtenơ tuần tự, có thể sắp xếp chúng bằng phiên bản **sort()** thông thường với hai tham số.

14.2.1. Các tiêu chuẩn khác nhau để sắp xếp

Cho rằng chúng ta muốn sắp xếp theo một thứ tự khác thứ tự được xác định trong toán tử <. Một giải pháp là xác định tiêu chuẩn sắp xếp sử dụng một đối tượng hàm. Ví dụ sau cho thấy một danh sách (list) các đối tượng **person** có thể được sắp xếp theo số điện thoại dùng một đối tượng hàm tự viết, có tên là **Phoneless()**. Bản 14-2 là chương trình SORTPERS.

Listing 14-2 SORTPERS

```

//sortpers.cpp
//sorts person objects by name and phone number
#include<iostream.h>
#include<vector.h>
#include<multiset.h>
#include<algo.h>
#include<cstring.h>

```

```

class person
{
private:
    string LastName;
    string FirstName;
    long PhoneNumber;
public:
    //default constructor
    person():LastName("blank"),FirstName("blank"),phoneNumber(0L)
    {
        //3-arg constructor
    }
    person(string lana,string fina,long pho):
        LastName(lana),FirstName(fina),PhoneNumber(pho)
    {
    }
    friend bool operator<(const person&,const person&);
    friend bool operator==(const person&,const person&);
    void display() const //display person's data
    {
        cout<<endl<<LastName<<","<<"\t"<<FirstName
            <<"\t"<<PhoneNumber;
    }

    long get_phone() const //return phone number
    {return PhoneNumber;}
}; //end of class

//overloaded < for person class
bool operator<(const person& p1,const person& p2)
{
    if(p1.LastName==p2.LastName)
        return (p1.FirstName<p2.FirstName) ? true : false;
    return (p1.LastName<p2.LastName) ? true : false;
}

//overloaded == for person class
bool operator==(const person& p1,const person& p2)
{
    return (p1.LastName==p2.LastName &&
        p1.FirstName==p2.FirstName) ? true : false;
}

//function object to compare person's phone number
class phoneLess:binary_function<person,person,bool>
{
public:
    bool operator() (const person& p1,const person& p2) const
    {
        return p1.get_phone() < p2.get_phone();
    }
};

//-----
void main()
{
    //a multiset of persons
    multiset<person,less<person> > persSet;
    //put persons in set
    persSet.insert(person("Deauville","William",8435150));
    persSet.insert(person("McDonald","Stacey",3327563));
    persSet.insert(person("Bartoski","Peter",6946473));
}

```

```

persSet.insert(person("KuangThu","Bruce",4157300));
persSet.insert(person("Wellington","John",9207404));
persSet.insert(person("McDonald","Amanda",8435150));
persSet.insert(person("Fredericks","Roger",7049982));
persSet.insert(person("McDonald","Stacey",7764987));
//iterator to multiset
multiset<person,less<person> >::iterator iterset;
cout<<"\nPersons sorted by name:";
iterSet=persSet.begin();           //display contents of set
while(iterSet != persSet.end())
    (*iterSet++).display();
//an empty vector of persons
vector<person> persVect(persSet.size());
//copy from set to vector
copy(persSet.begin(),persSet.end(),persVect.begin());
//sort vector by phone number
sort(persVect.begin(),persVect.end(),phoneLess());
//iterator to a vector of persons
vector<person>::iterator itervect;
cout<<"\n\nPersons sorted by phone number:";
itervect=persVect.begin();         //display contents of vector
while(itervect != persVect.end())
    (*itervect++).display();
} //end main()

```

Chương trình bắt đầu bằng việc đặt các đối tượng **person** trong một tập hợp và hiển thị nội dung của nó để thấy thứ tự sắp xếp theo tên. Sau đó tập hợp được copy vào một vectơ và vectơ được sắp xếp dùng đối tượng hàm **phoneLess()** trong dòng:

```
sort(persVect.begin(),persVect.end(),phoneLess());
```

Sau đây là kết quả đưa ra của chương trình SORTPERS:

Persons sorted by name:

Bartoski,	Peter	phone: 6946473
Deauville,	William	phone: 8435150
Fredericks,	Roger	phone: 7049982
KuangThu,	Bruce	phone: 4157300
McDonald,	Amanda	phone: 8435150
McDonald,	Stacey	phone: 3327563
McDonald,	Stacey	phone: 7764987
Wellington,	John	phone: 9207404

Persons sorted by phone number:

McDonald,	Stacey	phone: 3327563
KuangThu,	Bruce	phone: 4157300
Bartoski,	Peter	phone: 6946473
Fredericks,	Roger	phone: 7049982
McDonald,	Stacey	phone: 7764987
Deauville,	William	phone: 8435150
McDonald,	Amanda	phone: 8435150
Wellington,	John	phone: 9207404

Lớp **phoneLess** được định nghĩa như sau:

```

class phoneLess : binary_function<person, person, bool>
{
    public:
    bool operator() (const person& p1, const person& p2) const
    { return p1.get_phone() < p2.get_phone(); }
};

```

Chúng ta có thể viết lớp này như một cấu trúc **struct**, nó sẽ không cần dùng từ **public** (bởi vì mọi thứ trong một **struct** mặc định là **public**). Tuy nhiên, ở đây chúng ta sử dụng **class** để nhấn mạnh rằng chúng ta đang làm việc với lớp.

Thay vì so sánh tên, như đối tượng hàm có sẵn đã làm, toán tử chông () trong lớp **phoneLess** so sánh các số điện thoại của hai người được truyền tới như các đối số. Nó trả về **true** nếu số điện thoại thứ nhất nhỏ hơn số điện thoại thứ hai. Giải thuật **sort()** sử dụng mối quan hệ này để sắp xếp vectơ.

Lớp **phoneLess** được rút ra từ lớp **binary_function** (thường là một cấu trúc **struct**), nó được định nghĩa trong **FUNCTION.H**. Lớp **binary_function** xác định hai đối số phải được cung cấp cho đối tượng hàm. Việc này đơn giản hóa cú pháp được dùng để viết hàm chông **operator()** (cũng có một lớp **unary_function**, có thể rút ra các đối tượng hàm một đối số từ nó).

14.2.2. Sự kết nối: cung cấp các giá trị tới các đối tượng hàm

Giải thuật **find()** tìm kiếm trong một côngtenơ các đối tượng được định nghĩa bởi người sử dụng. Có thể tìm kiếm các phần tử dùng các tiêu chuẩn tìm kiếm khác nhau được không? Có một cách là sử dụng giải thuật **find_if** và một đối tượng hàm thích hợp. Tuy nhiên để làm điều này, chúng ta cần biết về một khái niệm mới: sự kết nối (**binding**), trong đó một giá trị được nối tới một đối tượng hàm để tạo một đối tượng hàm khác.

Ví dụ **FUNCPERS** (bản 14-3) cho thấy cách mà một đối tượng hàm có tên là **phoneEqual()** có thể tìm tất cả các mục với một số điện thoại cho trước.

Listing 14-3 FUNCPERS

```

//funcpers.cpp
//demonstrates function objects with person class
#include<iostream.h>
#include<list.h>
#include<algo.h>
#include<cstring.h>
class person
{
    private:
        string LastName;
        string FirstName;
        long PhoneNumber;
    public:
        //default constructor
        person():LastName("blank"),FirstName("blank"),phoneNumber(0L)
        {}
        //3-arg constructor
        person(string lana,string fina,long pho):
            LastName(lana),FirstName(fina),PhoneNumber(pho)
        {}
        friend bool operator<(const person&,const person&);
        friend bool operator==(const person&,const person&);
        void display() const //display person's data

```



```

        {
            cout<<endl<<LastName<<"\t"<<FirstName
                <<"\t\tPhone: "<<PhoneNumber;
        }
        long get_phone() const    //return phone number
        {return PhoneNumber;}
    };    //end of class
//overloaded < for person class
bool operator<(const person& p1,const person& p2)
{
    if(p1.LastName==p2.LastName)
        return (p1.FirstName<p2.FirstName) ? true : false;
    return (p1.LastName<p2.LastName) ? true : false;
}
//overloaded == for person class
bool operator==(const person& p1,const person& p2)
{
    return (p1.LastName==p2.LastName &&
        p1.FirstName==p2.FirstName) ? true : false;
}
//function object to compare person's phone number
class phoneEqual:binary_function<person,long,bool>
{
public:
    bool operator() (const person& p,const long& n) const
    {
        return (p.get_phone()==n) ? true : false;
    }
};
//-----
void main()
{
    list<person> persList;    //list of persons
    list<person>::iterator iter1;//iterators to a list of persons
    list<person>::iterator iter2;
    //put persons in list
    persList.push_back(person("Deauville","William",8435150));
    persList.push_back(person("McDonald","Stacey",3327563));
    persList.push_back(person("Bartoski","Peter",6946473));
    persList.push_back(person("KuangThu","Bruce",4157300));
    persList.push_back(person("Wellington","John",9207404));
    persList.push_back(person("McDonald","Amanda",8435150));
    persList.push_back(person("Fredericks","Roger",7049982));
    persList.push_back(person("McDonald","Stacey",7764987));
    //find person or persons with specified phone number
    cout<<"\n\nEnter phone number (format 1234567): ";
    long sNumber;    //get search number
    cin>>sNumber;
    //search for first match
    iter1=find_if(persList.begin(),persList.end(),
        bind2nd(phoneEqual(),sNumber)); //func object
    if(iter1 != persList.end())
    {
        cout<<"\nPerson(s) with that phone is(are)";
        do

```

```

    {
        (*iter1).display();    //display match
        iter2=++iter1;        //search for another match
        iter1=find_if(iter2,persList.end(),bind2nd(phoneEqual(),
            sNumber));
    }
    while(iter1 != persList.end());
}
else
    cout<<"There is no person with that phone number.";
} //end main()

```

Đối tượng hàm **phoneEqual()** tương tự như đối tượng hàm **phoneLess()** trong ví dụ SORTPERS. Tuy nhiên, nó được gọi hơi khác bởi giải thuật **find_if()**.

Đối số thứ ba cho **find_if()** yêu cầu một đối tượng hàm. Đối tượng hàm này, **phoneEqual()**, mong đợi **find_if()** cung cấp hai đối số cho nó: một đối tượng **person** và một giá trị **long** biểu diễn số điện thoại. Khi **find_if()** trở qua côngtenơ **persList** để kiểm tra lần lượt từng đối tượng, nó lặp lại lời gọi hàm **phoneEqual()**. Nó biết sử dụng đối tượng **person** nào để truyền tới **phoneEqual()** bởi vì nó biết nơi đối tượng ở trong quá trình trở qua côngtenơ. Nhưng **find_if()** lấy số điện thoại để cung cấp cho **phoneEqual()** ở đâu? Chúng ta không thể cung cấp số này trực tiếp cho đối tượng hàm như là một đối số như sau:

```

iter1 = find_if(persList.begin(),persList.end(),
               phoneEqual(snumber)); //không thể làm thế này được

```

bởi vì **find_if()** phải tự nó cung cấp tất cả các giá trị cho **phoneEqual()**. Thay vào đó, chúng ta phải nối số điện thoại và đối tượng hàm với nhau để tạo một đối tượng hàm khác. Bộ nối **bind2nd** được sử dụng cho mục đích này:

```

iter1 = find_if(persList.begin(),persList.end(),
               bind2nd(phoneEqual(),snumber) );

```

Chúng ta có thể sử dụng một trong hai bộ nối: **bind1st** hoặc **bind2nd**, tùy thuộc vào giá trị mà chúng ta cung cấp cho đối tượng hàm là đối số thứ nhất hay thứ hai. Trong **phoneEqual()**, nó là đối số thứ hai, trong đối tượng **person** là đối số thứ nhất.

Đây là vài mẫu tương tác với chương trình FUNCPERS:

Enter phone number (format 1234567): 9207474

Person(s) with that phone number is(are)
 Wellington, John Phone: 9207474

Nếu có nhiều số điện thoại giống nhau thì chương trình sẽ tìm và hiển thị tất cả chúng.

TÀI LIỆU THAM KHẢO

- [1] Al Stevens, "Teach Yourself ... C++", *MIS : Press*, 1995
- [2] Bruce Eckel, " C++ Inside & Out", *Oshorne McGraw Hill*, 1994
- [3] Herbert Schildt, " C++ Nuts and Bolts : For Experienced Programmers",
Berkeley : McGraw Hill, 1995
- [4] H.M. Deitel, P.J. Deitel, " C++ How to program", *New York : Prentice Hall*, 1998
- [5] Jan Skansholm, " C++ from the beginning", *Masachusetts : Addison Wesley*, 1997
- [6] Kenneth A.Barclay, Brian J.Gordon, " C++ problem solving and Programming",
New York : Prentice - Hall, 1994
- [7] Namir C. Shammas, "C/ C++ Mathematical Algorithms for Scientis and Engineers",
New York : Prentice Hall, 1995
- [8] Margaret Ellis & Gjarne Stroustrup, "The Annotated ANSI C++ Reference Manual",
Addison : Wesley, 1990
- [9] Marshall P. Cline, Greg A.Lomow, " C++ FAQS : Frequently Asked Questions",
Masachusetts: Addison Wesley, 1995
- [10] Saba Zamir, " C++ Primer for non C programmers", *New York : Mc Graw Hill*, 1995
- [11] Pobert Lafore, "C++ Interactive Course", *Corte Madera : Waite Group Press*, 1996
- [12] Iarry Nyhoff - Sanford Leedstma, người dịch : TS. Lê Minh Trung ; "Lập trình
nâng cao bằng Pascal với các cấu trúc dữ liệu". *NXB Đà Nẵng*, 1977

MỤC LỤC

Trang

Lời nói đầu

3

PHẦN I

NGÔN NGỮ LẬP TRÌNH C++

Chương 1. CÁI NHÌN ĐẦU TIÊN VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VÀ C++

- | | |
|---|---|
| 1.1. Tại sao chúng ta cần lập trình hướng đối tượng ? | 5 |
| 1.2. Đặc điểm của ngôn ngữ hướng đối tượng | 7 |

Chương 2. SỰ KẾ THỪA

- | | |
|---|----|
| 2.1. Giới thiệu về sự kế thừa | 10 |
| 2.2. Thiết kế chương trình : lớp employee | 16 |
| 2.3. Sự sử dụng lại : lớp stack cải tiến | 19 |
| 2.4. Hàm tạo và sự kế thừa | 21 |
| 2.5. Điều khiển truy nhập | 26 |
| 2.6. Sự kế thừa nhiều mức | 32 |
| 2.7. Sự hợp thành | 37 |
| 2.8. Sự kế thừa bội | 39 |

Chương 3. CON TRỎ

- | | |
|---------------------------------------|----|
| 3.1. Địa chỉ và con trỏ | 40 |
| 3.2. Con trỏ, mảng và hàm | 47 |
| 3.3. Quản lý bộ nhớ với new và delete | 56 |
| 3.4. This và const | 63 |

Chương 4. HÀM ẢO VÀ HÀM BẠN

- | | |
|---|-----|
| 4.1. Giới thiệu về hàm ảo | 69 |
| 4.2. Ví dụ về hàm ảo | 76 |
| 4.3. Gỡ kết nối với sự đa hình thái | 83 |
| 4.4. Lớp trừu tượng và hàm hủy ảo | 89 |
| 4.5. Định danh kiểu tại thời điểm chạy chương trình | 98 |
| 4.6. Hàm bạn | 101 |
| 4.7. Lớp bạn | 108 |

Chương 5. STREAM VÀ FILE

- | | |
|--------------------------------------|-----|
| 5.1. Lớp stream | 115 |
| 5.2. Các lỗi stream | 122 |
| 5.3. Vào/ra file đĩa với các stream | 128 |
| 5.4. Các lỗi file và con trỏ file | 138 |
| 5.5. Vào/ra file dùng hàm thành viên | 142 |
| 5.6. Các toán tử chống <<và>> | 150 |
| 5.7. Bộ nhớ như một đối tượng stream | 155 |

Chương 6. CÁC BẢN MẪU

- | | |
|------------------------------|-----|
| 6.1. Mẫu hàm | 160 |
| 6.2. Mẫu lớp | 166 |
| 6.3. Lớp string chuẩn | 170 |
| 6.4. Chương trình nhiều file | 176 |

PHẦN II

CẤU TRÚC DỮ LIỆU

Chương 7. THƯ VIỆN MẪU CHUẨN

7.1. Côngtenơ	182
7.2. Giải thuật	186
7.3. Con trỏ	194
7.4. Vấn đề tiềm ẩn với STL	195

Chương 8. CÁC GIẢI THUẬT

8.1. Giải thuật find ()	197
8.2. Giải thuật count ()	198
8.3. Giải thuật sort ()	198
8.4. Giải thuật search ()	199
8.5. Giải thuật merge ()	199
8.6. Đối tượng hàm	200
8.7. Hàm viết bởi người sử dụng đặt ở vị trí đối tượng hàm	201
8.8. Thêm -If vào các giải thuật	201
8.9. Giải thuật for - each ()	202
8.10. Giải thuật transform ()	203

Chương 9. ITERATORS

9.1. Con trỏ thông minh	204
9.2. Iterator như một giao diện	205
9.3. Iterator làm việc như thế nào ?	208
9.4. Iterator đặc biệt	211

Chương 10. CÔNGTENƠ TUẦN TỰ

10.1. Véc tơ	218
10.2. Danh sách (list)	223
10.3. Hàng đợi hai đầu (deque)	229

Chương 11. CÔNGTENƠ LIÊN KẾT

11.1. Tập hợp và đa tập hợp (set and multiset)	233
11.2. Ánh xạ và đa ánh xạ (map and multimap)	236
11.3. Các phiên bản bảng băm (hash table versions)	238

Chương 12. CÁC KIỂU DỮ LIỆU TRỪU TƯỢNG

12.1. Ngăn xếp	239
12.2. Hàng đợi	240
12.3. Hàng đợi ưu tiên (priority queue)	241
12.4. Cây nhị phân (binary tree)	241

Chương 13. LƯU TRỮ CÁC ĐỐI TƯỢNG ĐƯỢC ĐỊNH NGHĨA BỞI NGƯỜI SỬ DỤNG

13.1. Một tập hợp các đối tượng person	242
13.2. Một danh sách các đối tượng person	245
13.3. Một danh sách các đối tượng airtime	246

Chương 14. ĐỐI TƯỢNG HÀM

14.1. Các đối tượng hàm định nghĩa trước	249
14.2. Tự viết các đối tượng hàm	251

TÀI LIỆU THAM KHẢO

257

MỤC LỤC

258

Chịu trách nhiệm xuất bản :

Chủ tịch HĐQT kiêm Tổng Giám đốc NGÔ TRẦN ÁI
Phó Tổng Giám đốc kiêm Tổng biên tập NGUYỄN QUÝ THAO

Biên tập lần đầu và tái bản :

DƯƠNG VĂN BẰNG

Biên tập kỹ thuật :

LƯU CHÍ ĐỒNG

Sửa bản in :

DƯƠNG VĂN BẰNG

Trình bày bìa :

LƯU CHÍ ĐỒNG

Chế bản :

HẢI NAM

NGÔN NGỮ LẬP TRÌNH C++ VÀ CẤU TRÚC DỮ LIỆU

Mã số: 7B558y8 – DAI

In 1.500 bản (QĐ : 54), khổ 19 x 27 cm. In tại Công ty Cổ phần In Phúc Yên.

Địa chỉ : Đường Trần Phú, thị xã Phúc Yên.

Số ĐKKH xuất bản : 04 – 2008/CXB/139 – 1999/GD.

In xong và nộp lưu chiểu tháng 10 năm 2008.



**CÔNG TY CỔ PHẦN SÁCH ĐẠI HỌC - DẠY NGHỀ
HEVOBCO**

25 HÀN THUYỀN – HÀ NỘI

Website : www.hevobco.com.vn

TÌM ĐỌC SÁCH THAM KHẢO KỸ THUẬT CỦA NHÀ XUẤT BẢN GIÁO DỤC

-  **LẬP TRÌNH BẰNG NGÔN NGỮ ASSEMBLY CHO MÁY TÍNH PC - IBM**
NGUYỄN MẠNH GIANG
-  **NGÔN NGỮ LẬP TRÌNH C++ VÀ CẤU TRÚC DỮ LIỆU**
NGUYỄN VIỆT HƯƠNG
-  **CƠ SỞ KỸ THUẬT MẠNG INTERNET**
PHẠM MINH VIỆT – TRẦN CÔNG NHƯỢNG
-  **KỸ THUẬT GHÉP NỐI MÁY VI TÍNH**
NGUYỄN MẠNH GIANG
-  **CẤU TRÚC MÁY VI TÍNH**
TRẦN QUANG VINH
-  **CƠ SỞ ĐỒ HOẠ MÁY VI TÍNH**
PHAN HỮU PHÚC
-  **AUTOCAD CHO TỰ ĐỘNG HOÁ THIẾT KẾ**
NGUYỄN VĂN HIẾN

Bạn đọc có thể mua tại các Công ti Sách - Thiết bị trường học ở các địa phương hoặc các Cửa hàng của Nhà xuất bản Giáo dục :

Tại Hà Nội : 25 Hàn Thuyên ; 187B Giảng Võ ; 232 Tây Sơn ; 23 Tràng Tiền ;

Tại Đà Nẵng : Số 15 Nguyễn Chí Thanh ; Số 62 Nguyễn Chí Thanh ;

Tại Thành phố Hồ Chí Minh : 104 Mai Thị Lựu, Quận 1 ; Cửa hàng 451B - 453,

Hai Bà Trưng, Quận 3 ; 240 Trần Bình Trọng – Quận 5.

Tại Thành phố Cần Thơ : Số 5/5, đường 30/4 ;

Website : www.nxbgd.com.vn



8 934980 858295



Giá: 39.000đ