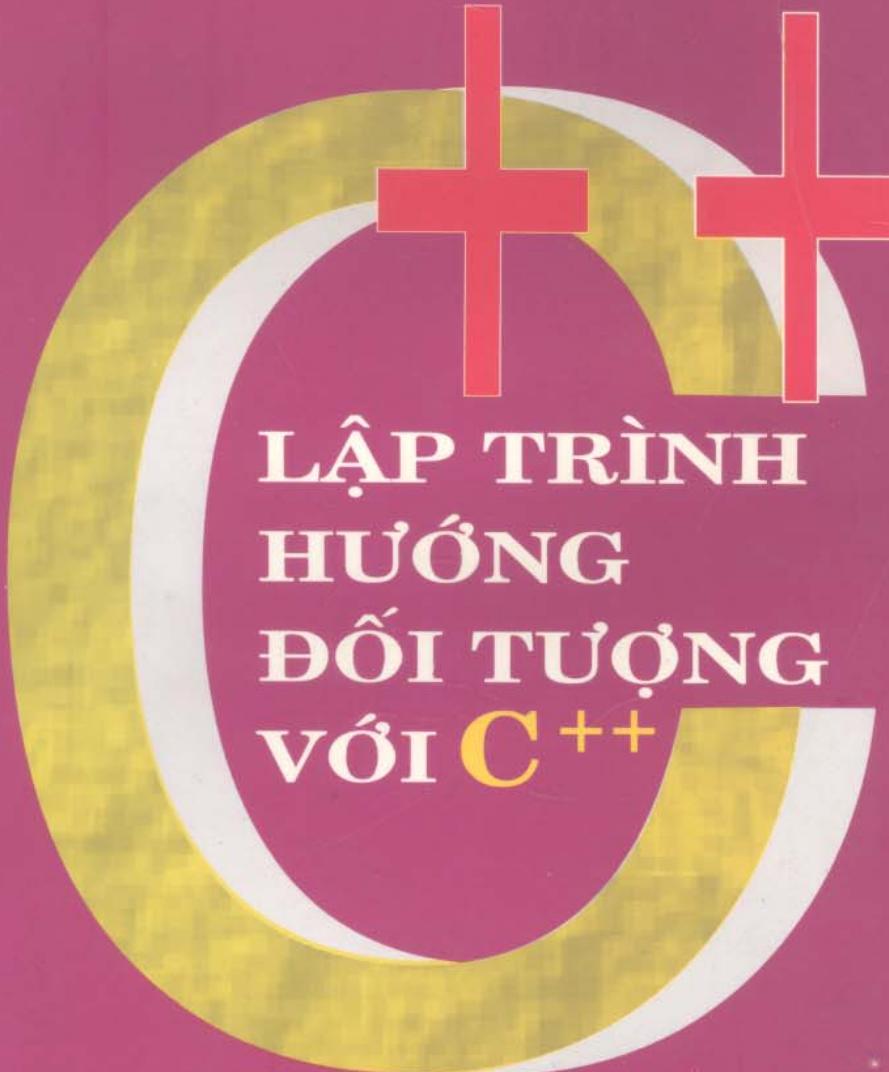


LÊ ĐĂNG HƯNG
TẠ TUẤN ANH
NGUYỄN HỮU ĐỨC
NGUYỄN THANH THỦY



NHÀ XUẤT BẢN
KHOA HỌC VÀ KỸ THUẬT

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN
LÊ ĐĂNG HƯNG, TẠ TUẤN ANH, NGUYỄN HỮU ĐỨC,
NGUYỄN THANH THỦY (Chủ biên)

**LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG
VỚI C++**



NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT - 1999

Chịu trách nhiệm xuất bản : Pgs. Pts. TÔ ĐĂNG HÀI
Biên tập : NGUYỄN NGỌC, MANH HÙNG
Chép bản : ĐĂNG HƯNG, VÂN CẨM
Vẽ bìa : HƯƠNG LAN

In 1000 cuốn khổ 16 x 24 cm tại Công ty in Hàng không. Giấy phép xuất
bản số 41-70-22/7/99. In xong và nộp lưu chiểu tháng 10/99.

LỜI GIỚI THIỆU

Vào thuở khai đầu sử dụng máy tính, cách lập trình là lập trình tuần tự. Khoảng những năm 70-80 xu hướng lập trình chủ yếu là lập trình có cấu trúc. Bước sang những năm 90, phương pháp lập trình hướng đối tượng trở nên phổ biến, được những người làm tin học quan tâm nghiên cứu nhiều.

Lập trình hướng đối tượng trở thành phương pháp lập trình hiện đại vì nó có những ưu điểm như:

- **Sự trừu tượng hoá:** Mỗi đối tượng trong chương trình là một trừu xuất của một đối tượng (vật lý hay phi vật lý) trong thực tế của bài toán. Điều này làm cho chương trình gần gũi hơn với người dùng và dễ thiết kế hơn đối với người lập trình.
- **Sự đóng gói:** Mỗi đối tượng là một sự đóng gói cả ba mặt: dữ liệu, trạng thái và thao tác, làm cho đối tượng là một đơn nguyên bền vững cho quá trình phân tích, thiết kế và lập trình. Xây dựng trên nguyên tắc giấu kín tối đa, công khai tối thiểu, đối tượng thích ứng dễ dàng với việc sửa lỗi, bảo trì hay phát triển.
- **Việc sử dụng lại:** Đối tượng là một đơn vị "lắp lẵn", có thể sử dụng lại cho bài toán khác.
- **Sự kế thừa được vận dụng cho các đối tượng làm tiết kiệm được mã nguồn, đồng thời tạo khả năng cho sự tiếp nối và mở rộng chương trình.**

Ngày nay đã có nhiều ngôn ngữ lập trình hỗ trợ cho đối tượng:

- Có những ngôn ngữ chỉ mới dựa vào đối tượng (có đóng gói mà không có kế thừa), như ADA (83).
- Có những ngôn ngữ hướng đối tượng thuần khiết, chỉ được phép sử dụng đối tượng trong lập trình, như SMALLTALK, JAVA...
- Có những ngôn ngữ lai, có thể lập trình hướng đối tượng một cách đầy đủ, mà cũng có thể lập trình dùng đối tượng. Turbo PASCAL, C++ thuộc loại này.

Ngôn ngữ C từ khi ra đời đã sớm khẳng định được vị thế là công cụ chủ yếu trong công nghệ phần mềm. C++ phát triển C thành ngôn ngữ hướng đối tượng đã thừa kế được các điểm mạnh của C và thích ứng với xu hướng lập trình hiện đại. Chọn C++ để giảng dạy cũng như làm công cụ phát triển phần mềm là hợp lý, vì không những nó mạnh, mà nó dễ hiểu cho những người vốn quen với lập trình truyền thống, nay đi vào lập trình hướng đối tượng. Như vậy chủ đề của cuốn sách là hoàn toàn thích hợp với nhu cầu giảng dạy lập trình ở các trường đại học, cũng như với nhu cầu của những lập trình viên đang hành nghề.

Cuốn sách gồm 6 chương và 4 phụ lục

- Chương 1: Lập trình hướng đối tượng - phương pháp giải quyết bài toán mới

- Chương 2: Những mở rộng của C++
- Chương 3: Đối tượng và lớp
- Chương 4: Định nghĩa toán tử trên lớp
- Chương 5: Kỹ thuật thừa kế
- Chương 6: Khuôn hình
- Phụ lục 1: Các kênh xuất nhập
- Phụ lục 2: Xử lý lỗi
- Phụ lục 3: Bài toán quan hệ gia đình
- Phụ lục 4: Mã chương trình bài toán quan hệ gia đình.

Như vậy cuốn sách bao trùm đủ các vấn đề cần đề cập và đã được cấu trúc khá hợp lý. Mỗi chương đều kết thúc bởi phần tóm tắt và các bài tập, thuận tiện cho việc học và thực hành.

Nội dung các phần đã được trình bày dễ hiểu, chuẩn xác, có nhiều thí dụ minh họa.

Cuốn sách có thể dùng làm:

- Giáo trình giảng dạy về lập trình hướng đối tượng ở các trường Đại học.
- Sách tham khảo về C++ cho những người lập trình chuyên nghiệp.

Tôi cho rằng quyển sách là rất có ích và hy vọng sẽ sớm được xuất bản để phục vụ bạn đọc. Xin trân trọng giới thiệu với bạn đọc cuốn sách “*Lập trình hướng đối tượng với C++*”, một sản phẩm lao động khoa học nghiêm túc của tập thể cán bộ giảng dạy khoa Công nghệ Thông tin, Đại học Bách khoa Hà nội.

Hà nội, ngày 4 tháng 5 năm 1999

GS. Nguyễn Văn Ba
Khoa CNTT, ĐHBK Hà nội

LỜI NÓI ĐẦU

Ngôn ngữ C được xem là một ngôn ngữ lập trình vận nòng và được sử dụng rộng rãi để giải quyết các bài toán khoa học kỹ thuật: xử lý ảnh, đồ họa, ghép nối máy tính. Ngôn ngữ C cũng là một công cụ mạnh cho phép xây dựng các chương trình hệ thống như hệ điều hành UNIX, các chương trình dịch và các công cụ tiện ích khác.

Điểm mạnh đáng chú ý của ngôn ngữ C chính là sự mềm dẻo và khả năng trao chuyển cao giữa các hệ thống tính toán. Trên cơ sở ngôn ngữ C người ta đã tiến hành xây dựng một phiên bản hướng đối tượng gọi là C++ nhằm thừa kế các điểm mạnh vốn có của C. Việc tìm hiểu ngôn ngữ C++ đang là một xu thế được quan tâm bởi các nhà lập trình nhằm phát triển các phần mềm ứng dụng ở Việt nam.

Hiện nay, đã có một số tài liệu tham khảo giới thiệu về C++. Tuy nhiên, các tài liệu này chỉ mới dừng lại ở mức giới thiệu và mô tả các khía cạnh cũ pháp của ngôn ngữ. Trong cuốn sách này, những trình bày về các khía cạnh lập trình hướng đối tượng có tính sư phạm cao, đi từ mức độ dễ đến khó nhằm giúp người đọc có thể linh hội các kiến thức cơ bản không chỉ lệ thuộc vào một ngôn ngữ lập trình cụ thể như C++ đã được quan tâm thích đáng.

Trong chương trình giảng dạy cử nhân, kỹ sư chuyên ngành Công nghệ Thông tin, ngôn ngữ lập trình C++ được chọn để minh họa cho lập trình hướng đối tượng nhờ các ưu điểm trong khả năng biểu diễn dữ liệu và thể hiện các khía cạnh lập trình. Việc nắm bắt các khía cạnh độc đáo của ngôn ngữ và làm chủ các yếu tố cơ bản khi lập trình trong ngôn ngữ C++ sẽ là cơ sở để nâng cao hiểu biết và kỹ năng lập trình bằng ngôn ngữ JAVA, một công cụ không thể thiếu được trong việc phát triển các ứng dụng trên mạng.

Chúng tôi biên soạn tài liệu này với hy vọng rằng nó sẽ rất bổ ích và cần thiết đối với các sinh viên cao đẳng và đại học không chỉ trong chuyên ngành Công nghệ Thông tin mà cả các ngành kỹ thuật, công nghệ khác như Điện tử Viễn Thông, Tự động hoá điều khiển v.v...

Khi biên soạn, chúng tôi đã cố gắng đưa ra một bộ cục của cuốn sách sao cho bám sát được các nội dung cơ bản xung quanh các khía cạnh cũ pháp của ngôn ngữ và những kỹ năng lập trình trên đó. Các kiến thức được trình bày có dạng nhằm giúp người học và tự học dễ tiếp thu các kiến thức được truyền thụ.

Trong quá trình biên soạn cuốn sách, chúng tôi đã nhận được nhiều ý kiến đóng góp quý báu của GS. Nguyễn Văn Ba, GS. Vũ Lực, GS Đỗ Xuân Lôi, Ths. Đỗ Văn Uy và các thầy cô trong khoa Công nghệ Thông tin, trường Đại học Bách khoa Hà Nội. Chúng tôi xin chân thành cảm ơn sự giúp đỡ quý giá đó. Nhân dịp này chúng tôi cũng xin bày tỏ sự biết ơn tới GS Nguyễn Thúc Hải và Ban chủ nhiệm

khoa Công nghệ Thông tin và Nhà xuất bản Khoa học và Kỹ thuật đã tạo điều kiện vật chất và tinh thần để cuốn sách sớm ra mắt bạn đọc.

Tuy đã hết sức cố gắng nhưng chúng tôi nghĩ rằng sẽ không tránh khỏi những thiếu sót, rất mong nhận được các ý kiến đóng góp để nâng cao chất lượng trong các lần tái bản sau. Thư từ góp ý xin gửi về Nhà xuất bản Khoa học và Kỹ thuật, 70 Trần Hưng Đạo, Hà Nội

Hà Nội, ngày 1 tháng 5 năm 1999

Các tác giả

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

PHƯƠNG PHÁP GIẢI QUYẾT BÀI TOÁN MỚI

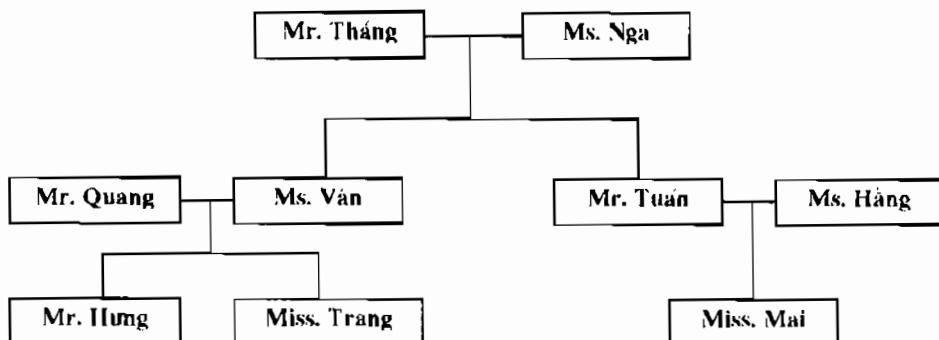
I. PHƯƠNG PHÁP LẬP TRÌNH

Từ nhiều năm nay chúng ta đã nghe nhiều đến thuật ngữ “Lập trình hướng đối tượng” (OOP - Object Oriented Programming). Vậy thực chất nó là gì? Để hiểu được vấn đề này chúng ta bắt đầu nhìn lại một chút lịch sử phát triển các phương pháp lập trình. Vào những ngày đầu phát triển của máy tính, khi các phần mềm còn rất đơn giản chỉ cất vài dòng lệnh, chương trình được viết tuân tự với các câu lệnh thực hiện từ đầu đến cuối. Cách viết chương trình như thế này gọi là phương pháp **lập trình tuyến tính**. Khoa học máy tính ngày càng phát triển, các phần mềm đòi hỏi ngày càng phức tạp và lớn hơn rất nhiều. Đến lúc này phương pháp lập trình tuyến tính tỏ ra kém hiệu quả và có những trường hợp người lập trình không thể kiểm soát được chương trình. Thế là phương pháp **lập trình cấu trúc** (LTCT) ra đời. Theo cách tiếp cận này, chương trình được tổ chức thành các chương trình con. Mỗi chương trình con đảm nhận xử lý một công việc nhỏ trong toàn bộ hệ thống. Mỗi chương trình con này lại có thể chia nhỏ thành các chương trình con nhỏ hơn. Quá trình phân chia như vậy tiếp tục diễn ra cho đến các chương trình con nhỏ nhất được đính kèm. Người ta gọi đó là quá trình làm mịn dần. Các chương trình con tương đối độc lập với nhau, do đó có thể phân công cho từng nhóm đảm nhận viết các chương trình con khác nhau. Ngôn ngữ lập trình thể hiện rõ nét nhất phương pháp lập trình cấu trúc chính là Pascal. Tuy nhiên, khi sử dụng phương pháp lập trình này vẫn còn gặp một khó khăn lớn là tổ chức dữ liệu của hệ thống như thế nào trong máy tính. Bởi vì theo quan điểm của LTCT thì *Chương trình = Cấu trúc dữ liệu + Giải thuật*. Để làm được việc này đòi hỏi người lập trình phải có kiến thức vững về cấu trúc dữ liệu. Một khó khăn nữa là giải thuật của chương trình phụ thuộc rất chặt chẽ vào cấu trúc dữ liệu, do vậy chỉ cần một sự thay đổi nhỏ ở cấu trúc dữ liệu cũng có thể làm thay đổi giải thuật và như vậy phải viết lại chương trình. Điều này rõ ràng không thể thích hợp khi phải xây dựng một dự án phần mềm rất lớn. Một phương pháp lập trình mới ra đời để khắc phục nhược điểm này và đó chính là phương pháp **lập trình hướng đối tượng** (LTHĐT). Điểm căn bản của phương pháp này là thiết kế chương trình xoay quanh dữ liệu của hệ thống. Nghĩa là lúc này các thao tác xử lý của hệ thống được gắn liền với dữ liệu và như vậy một sự thay đổi nhỏ của dữ liệu chỉ ảnh hưởng đến các một số nhỏ các hàm xử lý liên quan. Sự gắn kết giữa dữ liệu và các hàm xử lý trên chúng tạo ra đối tượng. Một ưu điểm nữa có ở phương pháp LTHĐT là cách tiếp cận bài toán trở nên gần gũi với thực tế hơn. Để hiểu rõ hơn về phương pháp lập trình này, không gì tốt hơn là chúng ta đi vào một bài toán cụ thể, chẳng hạn bài

toán quan hệ gia đình. Ở đây yêu cầu làm thế nào để thể hiện được các mối quan hệ giữa các thành viên trong một gia đình trên máy tính và có thể trả lời được câu hỏi dạng khá tổng quát: "A và B có quan hệ như thế nào trong gia đình ?" với A và B là hai cá thể bất kỳ. Chúng ta sẽ phân tích xem cách giải quyết bài toán này như thế nào.

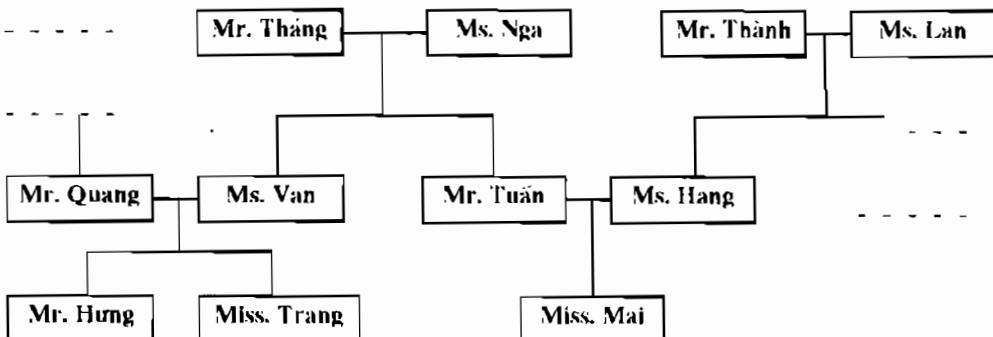
2. BÀI TOÁN QUAN HỆ GIA ĐÌNH

Trong xã hội, mỗi người đều có một gia đình, trong đó tồn tại nhiều mối quan hệ gia đình khá phức tạp như ông, bà, cha, mẹ, cô, chú, bác, v.v. Thông thường, để thể hiện các mối quan hệ này người ta biểu diễn bằng một sơ đồ cây quan hệ. Dưới đây là một ví dụ biểu diễn một gia đình ba thế hệ bằng hình 1.1.



Hình 1.1 Cây quan hệ trong một gia đình.

Để giải quyết bài toán này theo phương pháp LTCT, công việc đầu tiên là phải xây dựng một cấu trúc dữ liệu thể hiện được cây quan hệ trên. Trong qua có vẻ là đơn giản nhưng nếu thử làm xem sẽ thấy không đơn giản chút nào, thậm chí còn khó. Bởi vì nó đòi hỏi người lập trình phải rất thành thạo sử dụng con trỏ, phải xây dựng được giải thuật cập nhật thông tin trên cây quan hệ. Các giải thuật này tương đối phức tạp đối với một cấu trúc dữ liệu như trong bài toán. Yêu cầu của bài toán là trả lời được câu hỏi dạng như "Hưng và Mai có quan hệ như thế nào ?". Câu trả lời của chương trình phải là "Hưng là anh họ của Mai". Để có thể thực hiện được như vậy, rõ ràng chúng ta phải xây dựng được giải thuật tìm được mối quan hệ giữa hai nút trên cây quan hệ. Một vấn đề phức tạp và tinh vi hơn là tên gọi cho các mối quan hệ gia đình ở Việt nam rất phong phú! Một khó khăn là phải vét cạn hết các mối quan hệ có thể có trên một cây quan hệ. Một khó khăn nữa gặp phải là khi cần phát triển, chương trình phải quản lý được nhiều gia đình cùng một lúc và các gia đình này có mối quan hệ thông gia với nhau. Hình 1.2 là sơ đồ quan hệ được phát triển từ sơ đồ ví dụ trên minh họa cho vấn đề này.



Hình 1.2 Mở rộng quan hệ giữa các gia đình.

Một câu hỏi đặt ra: “Liệu với cấu trúc dữ liệu cũ có đám bảo giải quyết được vấn đề này không ?”. Rõ ràng câu trả lời là không. Số đó quan hệ trên hình vẽ sẽ phải mô tả quan hệ của một gia đình. Chỉ với chút ít sự thay đổi về cấu trúc dữ liệu cũng dẫn đến một loạt vấn đề đòi hỏi phải viết lại các giải thuật của chương trình. Phương pháp lập trình mới hướng đối tượng cho phép chúng ta khắc phục được các vấn đề đã nêu ra. Trong suốt các trình bày của cuốn sách này sẽ cố gắng nêu bật được cách giải quyết vấn đề nhờ LTHĐT.

Theo cách tiếp cận LTHĐT, bài toán quan hệ gia đình được xem xét dưới góc độ quản lý tập các đối tượng **Con người**. Để biết mỗi quan hệ gia đình của mỗi cá thể, cần thể hiện một số quan hệ cơ bản như cha, mẹ, anh em, con cái, vợ chồng của cá thể đó. Như vậy, mỗi đối tượng con người của bài toán có các thuộc tính riêng, nói lên rằng cha mẹ, anh em, v.v.. của họ là ai. Ngoài ra cũng cần có một thuộc tính nữa cho biết tên cá thể là gì. Có thể mô tả một lớp các đối tượng con người như hình 1.3.

Con người
Tên ?
Cha ?
Mẹ ?
Anh em ?
Con cái ?
Vợ / Chồng ?

Hình 1.3 Mô tả một lớp các đối tượng con người.

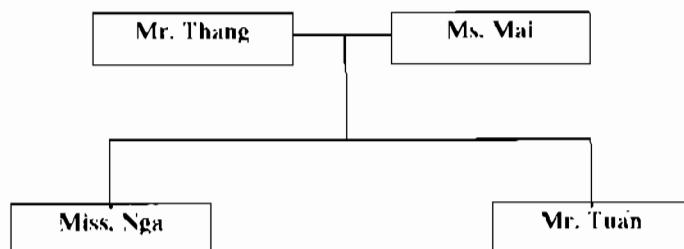
Nếu chỉ có như vậy thì chẳng khác gì một cấu trúc hay bảng ghi trong cấu trúc dữ liệu được sử dụng ở phương pháp LTCT. Vấn đề ở đây là phương pháp LTHĐT xem các mối quan hệ trong gia đình được hình thành một cách tự nhiên do các sự

kiện cụ thể trong cuộc sống tạo nên. Ví dụ, khi người phụ nữ sinh con, đứa con cô ta sinh ra sẽ có mẹ là cô ta và cha là chồng cô ta, đồng thời anh chồng phải được cập nhật để có thêm đứa con này. Những đứa con trước của cô ta sẽ có thêm đứa em này và đứa bé có thêm những người anh hoặc người chị đó. Để dễ dàng thấy rằng có hai sự kiện chính tác động đến mối quan hệ gia đình là sự sinh con của người phụ nữ và hôn nhân giữa hai cá thể khác giới trong xã hội. Các sự kiện này gắn liền với từng con người trong bài toán. Điều này có nghĩa là khi nói đến một sự kiện nào thì phải chỉ ra nó được phát sinh bởi người nào. Ví dụ, khi nói sự kiện sinh con thì phải biết người nào sinh. Khi một sự kiện của một con người nào đó xảy ra (ví dụ như sinh con) thì các thuộc tính của chính anh ta sẽ bị thay đổi, đồng thời thuộc tính của một số đối tượng liên quan cũng có thể thay đổi theo. Quá trình đóng gói giữa các sự kiện và thuộc tính sẽ tạo ra **Đối tượng**, khái niệm cơ bản của phương pháp LTHĐT. Một mô tả chung cho các đối tượng con người của bài toán được gọi là một **Lớp**. Hình 1.4 minh họa một lớp Con người có thêm các sự kiện của bài toán.

Con người
Tên ?
Cha ?
Mẹ ?
Anh em ?
Con cái ?
Vợ / Chồng ?
Sinh con
Cưới

Hình 1.4 Các sự kiện bổ sung gắn với con người.

Sau khi đã gắn kết các sự kiện vào đối tượng như trên, vẫn để là tạo một sơ đồ quan hệ gia đình như thế nào. Dưới đây là một ví dụ minh họa việc tạo ra một quan hệ gia đình dựa trên các sự kiện cuộc sống. Giả thiết là đã có hai đối tượng là ông Thắng và bà Mai.



Các sự kiện để tạo ra cây quan hệ trên có thể viết theo trật tự như sau:

Tháng.Cưới (Mai)

Mai.Sinh con (gái, Nga)

Mai.Sinh con (trai,Tuấn)

Các sự kiện viết theo cú pháp:

Đối tượng tạo sự kiện . Sự kiện (thông số kèm theo sự kiện)

Như vậy các bạn đã thấy rằng chúng ta không cần phải quan tâm đến cách tạo một cấu trúc cây quan hệ như thế nào bên trong dữ liệu của chương trình mà vẫn có thể cung cấp dữ liệu bài toán cho chương trình thông qua các sự kiện như trên. Chúng ta quay lại vấn đề chính của bài toán là trả lời các câu hỏi về mối quan hệ gia đình như thế nào khi tiếp cận bài toán theo phương pháp này. Để trả lời được câu hỏi tổng quát “X và Y có quan hệ gia đình như thế nào ?” ta cần phải trả lời các câu hỏi nhỏ như “X có phải là anh của Y không ?”, “X có phải là ông nội của Y không ?”, v.v.. Câu hỏi có thể nhìn từ góc độ đối tượng X như : “Đối tượng có phải là anh của Y không ?”, “có phải là ông nội của Y không ?”, v.v.. Như vậy câu hỏi lúc này đã giao về cho đối tượng để trả lời. Các đối tượng lúc này cần phải có các phương thức để trả lời các câu hỏi như vậy. Và bây giờ một lớp đối tượng Con người được minh họa như hình 1.5.

Con người
Tên ?
Cha ?
Mẹ ?
Anh em ?
Con cái ?
Vợ / Chồng ?
Sinh con
Cưới
Là anh
Là ông nội
.....

Hình 1.5 Thêm các phương thức trả lời câu hỏi.

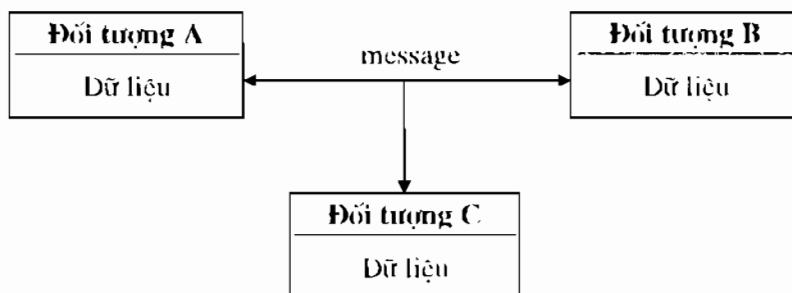
Ta xem xét các đối tượng trả lời các câu hỏi như thế nào? Chẳng hạn X trả lời câu hỏi “Đối tượng có phải là anh của Y không ?” hoàn toàn đơn giản. Nó chỉ cần kiểm tra xem Y có phải là anh em mà trong thuộc tính của nó lui giữ không. Hoàn toàn tương tự đối với các câu hỏi quan hệ gần như là em, là chị, là bố, là mẹ,... Còn

câu hỏi như “Đối tượng có phải là ông nội của Y không ?” phức tạp hơn chút ít. Để trả lời được các câu hỏi có quan hệ xa như thế ta phải dựa vào kết quả trả lời của các câu hỏi về các quan hệ gần gũi hơn. Để biết được X đúng là ông nội của Y thì phải chỉ ra một người Z nào đó mà X là bố của Z và Z là bố của Y. Nếu không chỉ ra được Z thì X không phải là ông nội của Y. Việc tìm kiếm Z hoàn toàn đơn giản bởi vì chương trình quản lý tập các đối tượng con người. Hãy tìm Z trong tập đối tượng Con người. Có thể thấy câu hỏi ban đầu đã được phân chia thành hai câu hỏi đơn giản với chúng mà đã có cách trả lời. Tóm lại, các vấn đề của bài toán đã được giải quyết khi tiếp cận theo phương pháp LTHĐT. Một lợi điểm có thể thấy ngay là bài toán được phân tích rất gần với thực tế và tự nhiên.

Trên đây mới chỉ là sự phân tích sơ khai bài toán dựa theo phương pháp LTHĐT. Để làm hoàn chỉnh được bài toán còn cần một số kỹ thuật của LTHĐT như tính kế thừa, tính đa hình, ... Chúng tôi hy vọng rằng qua sự phân tích một bài toán nhỏ trên đã chứng tỏ được lợi ích của phương pháp LTHĐT. Trong mục tiếp theo chúng tôi sẽ tóm tắt và đưa ra tổng quan sơ bộ về LTHĐT.

3. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Lập trình hướng đối tượng đặt trọng tâm vào đối tượng, yếu tố quan trọng trong quá trình phát triển chương trình và không cho phép dữ liệu biến động tự do trong hệ thống. Dữ liệu được gắn chặt với các hàm thành các vùng riêng mà chỉ có các hàm đó tác động lên và cấm các hàm bên ngoài truy nhập tới một cách tuy tiện. LTHĐT cho phép chúng ta phân tích bài toán thành các thực thể được gọi là các đối tượng và sau đó xây dựng các dữ liệu cùng các hàm xung quanh các đối tượng đó. Các đối tượng có thể tác động, trao đổi thông tin với nhau thông qua cơ chế thông báo (message). Tổ chức một chương trình hướng đối tượng có thể mô tả như trong hình 1.6.



Hình 1.6 Các đối tượng trao đổi qua thông báo.

LTHĐT có các đặc tính chủ yếu sau:

1. Tập trung vào dữ liệu thay cho các hàm
2. Chương trình được chia thành các đối tượng.

3. Các cấu trúc dữ liệu được thiết kế sao cho đặc tả được đối tượng.
4. Các hàm thao tác trên các vùng dữ liệu của đối tượng được gắn với cấu trúc dữ liệu đó.
5. Dữ liệu được đóng gói lại, được che giấu và không cho phép các hàm ngoại lai truy nhập tự do.
6. Các đối tượng tác động và trao đổi thông tin với nhau qua các hàm.
7. Có thể dễ dàng bổ sung dữ liệu và các hàm mới vào đối tượng nào đó khi cần thiết.
8. Chương trình được thiết kế theo cách tiếp cận từ dưới lên (bottom-up).

Sau đây là một số khái niệm được sử dụng trong LTHĐT.

3.1 Một số khái niệm

Đối tượng (object)

Đối tượng là sự kết hợp giữa dữ liệu và thủ tục (hay còn gọi là các phương thức - method) thao tác trên dữ liệu đó. Có thể đưa ra công thức phản ánh bản chất kỹ thuật của LTHĐT như sau:

$$\text{Đối tượng} = \text{Dữ liệu} + \text{Phương thức}$$

Lớp (class)

Lớp là một khái niệm mới trong LTHĐT so với các kỹ thuật lập trình khác. Đó là một tập các đối tượng có cấu trúc dữ liệu và các phương thức giống nhau (hay nói cách khác là một tập các đối tượng cùng loại). Như vậy khi có một lớp thì chúng ta sẽ biết được một mô tả cấu trúc dữ liệu và phương thức của các đối tượng thuộc lớp đó. Mỗi đối tượng sẽ là một thể hiện cụ thể (instance) của lớp đó. Trong lập trình, chúng ta có thể coi một lớp như là một kiểu, còn các đối tượng sẽ là các biến có kiểu của lớp.

Nguyên tắc đóng gói dữ liệu

Trong LTCT ta đã thấy là các hàm hay thủ tục được sử dụng mà không cần biết đến nội dung cụ thể của nó. Người sử dụng chỉ cần biết chức năng của hàm cũng như các tham số cần truyền vào để gọi hàm chạy mà không cần quan tâm đến những lệnh cụ thể bên trong nó. Người ta gọi đó là sự đóng gói về chức năng.

Trong LTHĐT, không những các chức năng được đóng gói mà cả dữ liệu cũng như vậy. Với mỗi đối tượng người ta không thể truy nhập trực tiếp vào các thành phần dữ liệu của nó mà phải thông qua các thành phần chức năng (các phương thức) để làm việc đó.

Chúng ta sẽ thấy sự đóng gói thực sự về dữ liệu chỉ có trong một ngôn ngữ LTHĐT “thuần khiết” (pure) theo nghĩa các ngôn ngữ được thiết kế ngay từ đầu chỉ cho LTHĐT. Còn đối với các ngôn ngữ “lai” (hybrid) được xây dựng trên các ngôn

ngữ khác ban đầu chưa phải là HĐT như C++ được nói đến trong cuốn sách này, vẫn có những ngoại lệ nhất định vì phạm nguyên tắc đóng gói dữ liệu.

Tính kế thừa (inheritance)

Một khái niệm quan trọng của LTHĐT là sự kế thừa. Sự kế thừa cho phép chúng ta định nghĩa một lớp mới trên cơ sở các lớp đã tồn tại, tất nhiên có bổ sung những phương thức hay các thành phần dữ liệu mới. Khả năng kế thừa cho phép chúng ta sử dụng lại một cách dễ dàng các module chương trình mà không cần một thay đổi các module đó. Rõ ràng đây là một điểm mạnh của LTHĐT so với LTCT.

Tính đa hình (polymorphism)

Tính đa hình xuất hiện khi có khái niệm kế thừa. Giả sử chúng ta có một kế thừa lớp hình tứ giác và lớp hình tam giác kế thừa từ lớp hình đa giác (hình tam giác và tứ giác sẽ có đầy đủ các thuộc tính và tính chất của một hình đa giác). Lúc này một đối tượng thuộc lớp hình tam giác hay tứ giác đều có thể hiểu rằng nó là một hình đa giác. Mặt khác với mỗi đa giác ta có thể tính diện tích của nó. Như vậy làm thế nào mà một đa giác có thể sử dụng đúng công thức để tính diện tích phù hợp với nó là hình tam giác hay tứ giác. Ta gọi đó là tính đa hình.

3.2 Các ưu điểm của LTHĐT

LTHĐT đem lại một số lợi thế cho người thiết kế lẫn người lập trình. Cách tiếp cận hướng đối tượng giải quyết được nhiều vấn đề tồn tại trong quá trình phát triển phần mềm và tạo ra được những phần mềm có độ phức tạp và chất lượng cao. Phương pháp này mở ra một triển vọng to lớn cho người lập trình. Những ưu điểm chính của LTHĐT là:

1. Thông qua nguyên lý kế thừa, chúng ta có thể loại bỏ được những đoạn chương trình lặp lại trong quá trình mô tả các lớp và có thể mở rộng khả năng sử dụng của các lớp đã xây dựng mà không cần phải viết lại.
2. Chương trình được xây dựng từ những đơn vị (đối tượng) trao đổi với nhau nên việc thiết kế và lập trình sẽ được thực hiện theo quy trình nhất định chứ không phải dựa vào kinh nghiệm và kỹ thuật như trước nữa. Điều này đảm bảo rút ngắn được thời gian xây dựng hệ thống và tăng năng suất lao động.
3. Nguyên lý đóng gói hay che giấu thông tin giúp người lập trình tạo ra được những chương trình an toàn không bị thay đổi bởi những đoạn chương trình khác.
4. Có thể xây dựng được ánh xạ các đối tượng của bài toán vào đối tượng chương trình.
5. Cách tiếp cận thiết kế đặt trọng tâm vào dữ liệu, giúp chúng ta xây dựng được mô hình chi tiết và dễ dàng cài đặt hơn.
6. Các hệ thống hướng đối tượng dễ mở rộng, nâng cấp thành những hệ lớn hơn.

7. Kỹ thuật truyền thông báo trong việc trao đổi thông tin giữa các đối tượng làm cho việc mô tả giao diện với các hệ thống bên ngoài trở nên đơn giản hơn.
8. Có thể quản lý được độ phức tạp của những sản phẩm phần mềm.

3.3 Những ứng dụng của LTHĐT

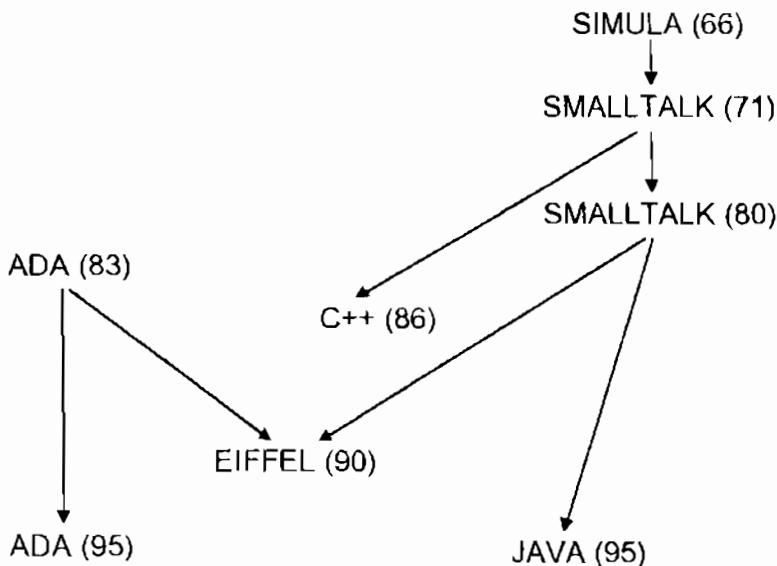
LTHĐT là một trong những thuật ngữ được nhắc đến nhiều nhất hiện nay trong công nghệ phần mềm và nó được ứng dụng để phát triển phần mềm trong nhiều lĩnh vực khác nhau. Trong số đó, ứng dụng quan trọng và nổi tiếng nhất hiện nay là thiết kế giao diện với người sử dụng, kiểu như Windows. Các hệ thống tin quản lý trong thực tế thường rất phức tạp, chứa nhiều đối tượng với các thuộc tính và hàm phức tạp. Để giải quyết những hệ thống tin phức tạp như thế, LTHĐT tỏ ra rất hiệu quả. Các lĩnh vực ứng dụng phù hợp với kỹ thuật LTHĐT có thể liệt kê như dưới đây:

- Các hệ thống làm việc theo thời gian thực.
- Các hệ mô hình hóa hoặc mô phỏng các quá trình.
- Các hệ cơ sở dữ liệu hướng đối tượng.
- Các hệ siêu văn bản (hypertext), đa phương tiện (multimedia).
- Các hệ thống trí tuệ nhân tạo và các hệ chuyên gia.
- Các hệ thống song song và mạng net-ron.
- Các hệ tự động hóa văn phòng hoặc trợ giúp quyết định.
- Các hệ CAD/CAM.

Với nhiều đặc tính phong phú của LTHĐT nói riêng, của phương pháp phân tích thiết kế và phát triển hướng đối tượng nói chung chúng ta hy vọng công nghiệp phần mềm sẽ có những cải tiến nhảy vọt không những về chất lượng, mà còn tăng nhanh về số lượng trong tương lai.

4. CÁC NGÔN NGỮ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

LTHĐT không phải là đặc quyền của một ngôn ngữ đặc biệt nào. Cũng giống như kỹ thuật lập trình có cấu trúc, các khái niệm trong LTHĐT được thể hiện trong nhiều ngôn ngữ lập trình khác nhau. Những ngôn ngữ cung cấp được những khả năng LTHĐT được gọi là ngôn ngữ lập trình hướng đối tượng. Tuy vẫn có những ngôn ngữ chỉ cung cấp khả năng tạo lớp và đối tượng mà không cho phép kế thừa, do đó hạn chế khả năng LTHĐT. Hình 1.7 cho chúng ta một cái nhìn tổng quan về sự phát triển các ngôn ngữ LTHĐT.



Hình 1.7 Sự phát triển của các ngôn ngữ LTHĐT.

Các ngôn ngữ SIMULA, SMALLTALK, JAVA thuộc họ ngôn ngữ LTHĐT thuần khiết, nghĩa là nó không cho phép phát triển các chương trình cấu trúc trên các ngôn ngữ loại này. Còn ngôn ngữ C++ thuộc loại ngôn ngữ “lai” bởi vì nó được phát triển từ ngôn ngữ C. Do đó trên C++ vẫn có thể sử dụng tính cấu trúc và đối tượng của chương trình. Điều này tỏ ra rất phù hợp khi chúng ta mới bắt đầu học một ngôn ngữ lập trình. Đó chính là lý do mà chúng tôi sử dụng ngôn ngữ C++ để giới thiệu phương pháp LTHĐT trong cuốn sách này. Một lý do khác nữa là C++ sử dụng cú pháp của ngôn ngữ C là ngôn ngữ rất thông dụng trong lập trình chuyên nghiệp.

5. NGÔN NGỮ LẬP TRÌNH C++

Vào năm 1983, giáo sư Bjarne Stroustrup bắt đầu nghiên cứu và phát triển việc cải đặt khả năng LTHĐT vào ngôn ngữ C tạo ra một ngôn ngữ mới gọi là C++. Tên gọi này có thể phân tích ý nghĩa rằng nó là ngôn ngữ C mà có hai đặc điểm mới tương ứng với hai dấu cộng. Đặc điểm thứ nhất là một số khả năng mở rộng so với C như tham chiếu, chồng hàm, tham số mặc định... Đặc điểm thứ hai chính là khả năng LTHĐT. Hiện nay C++ chưa phải là một ngôn ngữ hoàn toàn ổn định. Kể từ khi phiên bản đầu tiên ra đời vào năm 1986 đã có rất nhiều thay đổi trong các phiên bản C++ khác nhau: bản 1.1 ra đời vào năm 1986, 2.0 vào năm 1989 và 3.0 vào năm 1991. Phiên bản 3.0 này được sử dụng để làm cơ sở cho việc định nghĩa một ngôn ngữ C++ chuẩn (kiểu như Ansi C).

Trên thực tế hiện nay tất cả các chương trình dịch C++ đều tương thích với phiên bản 3.0. Vì vậy C++ hầu như không gây bất kỳ một khó khăn nào khi

chuyển đổi từ một môi trường này sang môi trường khác, như chúng ta đã biết C++ như là một sự bổ sung khả năng LTHĐT vào ngôn ngữ C. Sẽ có nhiều người nghĩ rằng ngôn ngữ C nói ở đây là C theo chuẩn ANSI. Thực ra không phải hoàn toàn như vậy. Tên thực tế vẫn tồn tại một vài điểm không tương thích giữa ANSI C và C++.

Mặt khác cũng cần thấy rằng những mở rộng có trong C++ so với Ansi C không chỉ là để phục vụ cho mục đích tạo cho ngôn ngữ khả năng LTHĐT. Có những thay đổi chỉ với mục đích đơn thuần là tăng sức mạnh cho ngôn ngữ C hiện thời.

Ngoài ra có một vài thay đổi nhỏ ở C++ so với ANSI C như sau:

- Định nghĩa các hàm: khai báo, truyền tham số và giá trị trả lại.
- Sự tương thích giữa các con trỏ.
- Tính linh hoạt của các hằng (const).

Các đặc điểm mở rộng trong C++

Như đã đề cập ở trên C++ chứa cả những mở rộng so với C mà không liên quan đến kỹ thuật hướng đối tượng. Những mở rộng này sẽ được mô tả cụ thể trong chương sau, ở đây chúng ta chỉ tóm tắt lại một vài điểm chính.

- Khả năng viết các dòng chú thích mới.
- Khả năng khai báo linh hoạt hơn.
- Khả năng định nghĩa lại các hàm: các hàm cùng tên có thể thực hiện theo những thao tác khác nhau. Các lời gọi hàm sẽ dùng kiểu và số tham số để xác định đúng hàm nào cần thực hiện.
- Có thêm các toán tử định nghĩa bộ nhớ động mới: new và delete.
- Khả năng định nghĩa các hàm inline để tăng tốc độ thực hiện chương trình.
- Tạo các biến tham chiếu đến các biến khác.

LTHĐT trong C++

C++ chứa đựng khái niệm lớp. Một lớp bao gồm các thành phần dữ liệu hay là thuộc tính và các phương thức hay là hàm thành phần. Từ một lớp ta có thể tạo ra các đối tượng hoặc bằng cách khai báo thông thường một biến có kiểu là lớp đó hoặc bằng cách cấp phát bộ nhớ động nhờ sử dụng toán tử new. C++ cho phép chúng ta đóng gói dữ liệu nhưng nó không bắt buộc chúng ta thực hiện điều đó. Đây là một nhược điểm của C++. Tuy nhiên cũng cần thấy rằng bản thân C++ chỉ là sự mở rộng của C nên nó không thể là một ngôn ngữ LTHĐT thuần khiết được.

C++ cho phép ta định nghĩa các hàm thiết lập (constructor) cho một lớp. Hàm thiết lập là một phương thức đặc biệt được gọi đến tại thời điểm một đối tượng của lớp được tạo ra. hàm thiết lập có nhiệm vụ khởi tạo một đối tượng: cấp phát bộ

nhớ, gán các giá trị cho các thành phần dữ liệu cũng như việc chuẩn bị chỗ cho các đối tượng mới. Một lớp có thể có một hay nhiều hàm thiết lập. Để xác định hàm thiết lập nào cần gọi đến, chương trình biên dịch sẽ so sánh các đối số với các tham số truyền vào. Tương tự như hàm thiết lập, một lớp có thể có một hàm hủy bỏ (destructor), một phương thức đặc biệt được gọi đến khi đối tượng được giải phóng khỏi bộ nhớ.

Lớp trong C++ thực chất là một kiểu dữ liệu do người sử dụng định nghĩa. Khái niệm định nghĩa không toán tử cho phép định nghĩa các phép toán trên một lớp giống như các kiểu dữ liệu chuẩn của C. Ví dụ ta có thể định nghĩa một lớp số phức với các phép toán cộng, trừ, nhân, chia.

Cũng giống như C, C++ có khả năng chuyển đổi kiểu. Không những thế, C++ còn cho phép mở rộng sự chuyển đổi này sang các kiểu do người sử dụng tự định nghĩa (tác lớp). Ví dụ, ta có thể chuyển đổi từ kiểu chuẩn int của C sang kiểu số phức mà ta định nghĩa chẳng hạn.

C++ cho phép thực hiện kế thừa các lớp đã xây dựng. Từ phiên bản 2.0 trở đi, C++ còn cho phép một lớp kế thừa cùng một lúc từ nhiều lớp khác nhau (gọi là sự đa kế thừa).

Cuối cùng C++ cung cấp những thao tác vào ra mới dựa trên cơ sở khái niệm luồng dữ liệu (flow). Sự ưu việt của các thao tác này ở chỗ:

- Sử dụng đơn giản.
- Kích thước bộ nhớ được rút gọn.
- Khả năng áp dụng trên các kiểu do người sử dụng định nghĩa bằng cách sử dụng cơ chế định nghĩa không toán tử.

NHỮNG MỞ RỘNG CỦA C++

Mục đích chương này:

1. Giới thiệu những điểm khác biệt chủ yếu giữa C và C++
2. Các điểm mới của C++ so với C (những vấn đề cơ bản nhất).

I. CÁC ĐIỂM KHÔNG TƯƠNG THÍCH GIỮA C++ VÀ ANSI C

1.1 Định nghĩa hàm

Trong định nghĩa hàm, ANSI C cho phép hai kiểu khai báo dòng tiêu đề của hàm, trong khi đó C++ chỉ chấp nhận một cách:

```
/*C++ không có khai báo kiểu này*/
double example(int u, double v)
int u;
double v;
```

```
/*cả C và C++ cho phép*/
double example(int u, double v)
int u;
double v;
```

1.2 Khai báo hàm nguyên mẫu

Trong ANSI C, khi sử dụng một hàm chưa được định nghĩa trước đó trong cùng một tệp, ta có thể:

1. không cần khai báo (khi đó ngầm định giá trị trả về của hàm là int)
2. chỉ cần khai báo tên hàm và giá trị trả về, không cần danh sách kiểu của các tham số
3. khai báo hàm nguyên mẫu.

Với C++, chỉ có phương pháp thứ 3 là chấp nhận được. Nói cách khác, một lời gọi hàm chỉ được chấp nhận khi trình biên dịch biết được kiểu của các tham số, kiểu của giá trị trả về. Mỗi khi trình biên dịch gặp một lời gọi hàm, nó sẽ so sánh các kiểu của các đối số được truyền với các tham số hình thức tương ứng. Trong trường hợp có sự khác nhau, có thể thực hiện một số chuyển kiểu tự động để cho hàm nhận được có danh sách các tham số đúng với kiểu đã được khai báo của hàm. Tuy nhiên phải tuân theo nguyên tắc chuyển kiểu tự động sau đây:

char-->int-->longint-->float-->double

Ví dụ 2.1

double example (int , double) /*khai báo hàm example*/
--

```

    ...
main(){
int n;
char c;
double z,res1,res2,res3;
...
res1 = fexple(n,z); /* không có chuyển đổi kiểu*/
res2 = fexple(c,z); /* có chuyển đổi kiểu, từ char (c) thành int*/
res3 = fexple(z,n); /* có chuyển đổi kiểu, từ double(z) thành int và từ int(n)
thành double*/
...
}

```

Trong C++ bắt buộc phải có từ khoá **void** trước tên của hàm trong phân khai báo để chỉ rằng hàm không trả về giá trị. Trường hợp không có, trình biên dịch ngầm hiểu kiểu của giá trị trả về là **int** và như thế trong thân hàm bắt buộc phải có câu lệnh **return**. Điều này hoàn toàn không cần thiết đối với mô tả trong ngôn ngữ C.

Thực ra, các khả năng vừa mô tả không hoàn toàn là điểm không tương thích giữa C và C++ mà đó chỉ là sự “gạn lọc” các điểm yếu và hoàn thiện các mặt còn chưa hoàn chỉnh của C.

1.3 Sự tương thích giữa con trỏ **void** và các con trỏ khác

Trong ANSI C, kiểu **void *** tương thích với các kiểu trỏ khác cả hai chiều. Chẳng hạn với các khai báo sau :

```

void *gen;
int   *adj;

```

hai phép gán sau đây là hợp lệ trong ANSI C:

```

gen = adj;
adj = gen;

```

Thực ra, hai câu lệnh trên đã kèm theo các phép “chuyển kiểu ngầm định”:

int* --->**void*** đổi với câu lệnh thứ nhất, và

void* --->**int*** đổi với câu lệnh thứ hai.

Trong C++, chỉ có chuyển đổi kiểu ngầm định từ một kiểu trả tự ý thành **void*** là chấp nhận được, còn muốn chuyển đổi ngược lại, ta phải thực hiện chuyển kiểu tường minh như cách viết sau đây:

```
gen = adj;
adj = (int *)gen;
```

2. CÁC KHẢ NĂNG VÀO/RA MỚI CỦA C++

Các tiện ích vào/ra (hàm hoặc macro) của thư viện C chuẩn đều có thể sử dụng trong C++. Để sử dụng các hàm này chúng ta chỉ cần khai báo tệp tiêu đề trong đó có chứa khai báo hàm nguyên mẫu của các tiện ích này.

Bên cạnh đó, C++ còn cài đặt thêm các khả năng vào/ra mới dựa trên hai toán tử “<<”(xuất) và “>>”(nhập) với các đặc tính sau đây:

1. đơn giản trong sử dụng
2. có khả năng mở rộng đối với các kiểu mới theo nhu cầu của người lập trình.

Trong tệp tiêu đề **iostream.h** người ta định nghĩa hai đối tượng **cout** và **cin** tương ứng với hai thiết bị chuẩn ra/vào được sử dụng cùng với “<<” và “>>”. Thông thường ta hiểu **cout** là màn hình còn **cin** là bàn phím.

2.1 Ghi dữ liệu lên thiết bị ra chuẩn (màn hình) **cout**

Trong phần này ta xem xét một số ví dụ minh họa cách sử dụng **cout** và “<<” để đưa thông tin ra màn hình.

Ví dụ 2.2

Chương trình sau minh họa cách sử dụng **cout** để đưa ra màn hình một xâu ký tự.

```
#include <iostream.h> /*phải khai báo khi muốn sử dụng cout*/
main()
{
    cout << "Welcome C++";
}
```

Welcome C++

“<<” là một toán tử hai ngôi, toán hạng ở bên trái mô tả nơi kết xuất thông tin (có thể là một thiết bị ngoại vi chuẩn hay là một tập tin), toán hạng bên phải của “<<” là một biểu thức nào đó. Trong chương trình trên, câu lệnh **cout << "Welcome C++"** đưa ra màn hình xâu ký tự “Welcome C++”.

Ví dụ 2.3

Sử dụng **cout** và “**<<**” đưa ra các giá trị khác nhau:

```
#include <iostream.h> /*phải khai báo khi muốn sử dụng cout*/
void main() {
    int n = 25;
    cout << "Value : ";
    cout << n;
```

Value : 25

Trong ví dụ này chúng ta đã sử dụng toán tử “**<<**” để in ra màn hình đầu tiên là một xâu ký tự, sau đó là một số nguyên. Chức năng của toán tử “**<<**” rõ ràng là khác nhau trong hai lần kết xuất dữ liệu: với câu lệnh thứ nhất, chỉ đưa ra màn hình một dãy các ký tự, ở câu lệnh sau, đã sử dụng một khuôn mẫu để chuyển đổi một giá trị nhị phân thành một chuỗi các ký tự chữ số. Việc một toán tử có nhiều vai trò khác nhau liên quan đến một khái niệm mới trong C++, đó là “**Định nghĩa chồng toán tử**”. Điều này sẽ được đề cập đến trong chương 4.

Ví dụ 2.4

Trong ví dụ này ta gộp cả hai câu lệnh kết xuất trong ví dụ 2.3 thành một câu lệnh phức tạp hơn, tuy kết quả không khác trước:

```
#include <iostream.h> /*phải khai báo khi muốn sử dụng cout*/
void main() {
    int n = 25;
    cout << "Value : " << n;
}
```

Value : 25

2.2 Các khả năng viết ra trên **cout**

Chúng ta vừa xem xét một vài ví dụ viết một xâu ký tự, một số nguyên. Một cách tổng quát, chúng ta có thể sử dụng toán tử “**<<**” cùng với **cout** để đưa ra màn hình giá trị của một biểu thức có các kiểu sau:

1. kiểu dữ liệu cơ sở (**char**, **int**, **float**, **double**),
2. xâu ký tự: (**char** *),
3. con trỏ (trừ con trỏ **char** *)

Trong trường hợp muốn đưa ra địa chỉ biến xâu ký tự phải thực hiện phép chuyển kiểu tường minh, chẳng hạn (**char ***) \rightarrow (**void ***).

Xét ví dụ sau đây:

Ví dụ 2.4

```
#include <iostream.h>
void void main(){
int n = 25;
long p = 250000;
unsigned q = 63000;
char c = 'a';
float x = 12.3456789;
double y = 12.3456789e16;
char * ch = "Welcome C++";
int *ad = &n;
cout <<"Value of n : " << n <<"\n";
cout <<"Value of p : " << p <<"\n";
cout <<"Value of c : " << c <<"\n";
cout <<"Value of q : " << q <<"\n";
cout <<"Value of x : " << x <<"\n";
cout <<"Value of y : " << y <<"\n";
cout <<"Value of ch : " << ch <<"\n";
cout <<"Addrese of n : " << ad <<"\n";
cout <<"Addrese de ch : " << (void *)ch <<"\n";
}
```

```
Value of n: 25
Value of p: 250000
Value of c: a
Value of q: 63000
Value of x: 12.345679
Value of y: 1.234567e+17
Value of ch: Welcome C++
Addrese of n: 0x1f12
Addrese de ch: 0x00b2
```

2.3 Đọc dữ liệu từ thiết bị vào chuẩn (bàn phím) **cin**

Nếu như **cout** dùng để chỉ thiết bị ra chuẩn, thì **cin** được dùng để chỉ một thiết bị vào chuẩn. Một cách tương tự, toán tử “>>” được dùng kèm với **cin** để nhập vào các giá trị; hai câu lệnh

```
int n;
cin >> n;
```

yêu cầu đọc các ký tự trên bàn phím và chuyển chúng thành một số nguyên và gán cho biến **n**.

Giống như **cout** và “<<”, có thể nhập nhiều giá trị cùng kiểu hay khác kiểu bằng cách viết liên tiếp tên các biến cần nhập giá trị cùng với “>>” ngay sau **cin**. Chẳng hạn:

```
int n;
float p;
char c;
cin >> c >> n >> p;
```

Có thể sử dụng toán tử “>>” để nhập dữ liệu cho các biến có kiểu **char**, **int**, **float**, **double** và **char ***.

Giống với hàm **scanf()**, **cin** tuân theo một số qui ước dùng trong việc phân tích các ký tự:

- (i) Các giá trị số được phân cách bởi: SPACE, TAB, CR, LF. Khi gặp một ký tự “không hợp lệ” (dấu “.” đối với số nguyên, chữ cái đối với số, ...) sẽ kết thúc việc đọc từ **cin**; ký tự không hợp lệ này sẽ được xem xét trong lần đọc sau.
- (ii) Đối với giá trị xâu ký tự, dấu phân cách cũng là SPACE, TAB, CR, còn đối với giá trị ký tự, dấu phân cách là ký tự CR. Trong hai trường hợp này không có khái niệm “ký tự không hợp lệ”. Mô sinh ra do bấm phím Enter của lần nhập trước vẫn được xét trong lần nhập xâu/ký tự tiếp theo và do vậy sẽ có “nguy cơ” không nhập được đúng giá trị mong muốn khi đưa ra lệnh nhập xâu ký tự hoặc ký tự ngay sau các lệnh nhập các giá trị khác. Giải pháp khắc phục vấn đề này để đảm bảo công việc diễn ra đúng theo ý là trước mỗi lần gọi lệnh nhập dữ liệu cho xâu/ký tự ta sử dụng một trong hai chỉ thị sau đây:

```
fflush(stdin); //khai báo trong stdio.h
cin.clear(); //hàm thành phần của lớp định nghĩa đối tượng cin
```

Ta tham khảo chương trình sau:

Ví dụ 2.6

```
#include <iostream.h>
#include <conio.h>

void main() {
    int n;
    float x;
    char t[81];
    clrscr();
    do {
        cout << "Nhập vào một số nguyên, một xâu, một số thực : ";
        cin >> n >> t >> x;
        cout << "Đã nhập " << n << "," << t << " và " << x << "\n";
    } while (n);
}
```

```
Nhập vào một số nguyên, một xâu, một số thực : 3 long 3.4
Đã nhập 3, long và 3.4
Nhập vào một số nguyên, một xâu, một số thực : 5 hung 5.6
Đã nhập 5, hung and 5.6
Nhập vào một số nguyên, một xâu, một số thực : 0 4
3
Đã nhập 0,4 và 3
```

3. NHỮNG TIỆN ÍCH CHO NGƯỜI LẬP TRÌNH

3.1 Chú thích cuối dòng

Mỗi ký hiệu đi sau “//” cho đến hết dòng được coi là chú thích, được chương trình dịch bỏ qua khi biên dịch chương trình.

Xét ví dụ sau:

```
cout << "Xin chào\n"; //lời chào hỏi
```

Thường ta sử dụng chú thích cuối dòng khi muốn giải thích ý nghĩa của một câu lệnh gì đó. Đối với một đoạn chương trình kiểu chú thích giới hạn bởi “/*” và “*/” cho phép mô tả được nhiều thông tin hơn.

3.2 Khai báo mảng nội

Trong C++ không nhất thiết phải nhóm lên đầu các khai báo đặt bên trong một hàm hay một khối lệnh, mà có thể đặt xen kẽ với các lệnh xử lý. Ví dụ:

```
{
int n;
n=23;
cout <<n<<"\n";
...
int *p=&n;
cout <<p<<"\n";
...
}
```

Giá trị khởi đầu cho các biến có thể thay đổi tại các thời điểm chạy chương trình khác nhau.

Một ví dụ minh họa cho khả năng định nghĩa khắp mọi nơi là có thể khai báo các biến điều khiển ngay bên trong các vòng lặp nếu biến đó chưa được khai báo trước đó trong cùng khối lệnh cũng như không được khai báo lại ở phần sau. Xem đoạn chương trình sau:

```
{
    ... //chưa có i
for(int i=0;...;...)
    ... //không được khai báo i
}
```

3.3 Toán tử phím vi "::"

Bình thường, biến cục bộ che lấp biến toàn cục cùng tên. Chẳng hạn:

```
#include <iostream.h>
int x;
main() {
    int x = 10; //x cục bộ
    cout<<x<<"\n"; //x cục bộ
}
```

Trong những trường hợp cần thiết, khi muốn truy xuất tới biến toàn cục phải sử dụng toán tử “::” trước tên biến:

```
#include <iostream.h>
int x;
main() {
    int x = 10; //x cục bộ
    ::x = 10; //toàn cục
    cout<<x<<"\n";//x cục bộ
    cout<<::x<<"\n";//toàn cục
}
```

4. HÀM inline

Trong C++ có thể định nghĩa các hàm được thay thế trực tiếp thành mã lệnh máy tại chỗ gọi (inline) mỗi lần được tham chiếu. Điểm này rất giống với cách hoạt động của các macro có tham số trong C. Ưu điểm của các hàm **inline** là chúng không đòi hỏi các thủ tục bổ sung khi gọi hàm và trả giá trị về. Do vậy hàm **inline** được thực hiện nhanh hơn so với các hàm thông thường.

Một hàm **inline** được định nghĩa và được sử dụng giống như bình thường. Điểm khác nhau duy nhất là phải đặt mô tả **inline** trước khai báo hàm.

Xét ví dụ sau đây:

Ví dụ 2.7

```
#include <iostream.h>
#include <math.h>
#include <conio.h>
inline double norme(double vet[3]); //khai báo hàm inline
void main() {
clrscr();
double v1[3], v2[3];
int i;
for(i=0;i<3;i++) {
    v1[i]=i; v2[i]=2*i+1;
}
cout << "norme của v1 : "<<norme(v1) << " norme của v2 : "<<norme(v2);
```

```

getch();
}

/* Định nghĩa hàm inline */
inline double norme(double vec[3]) {
    int i; double s=0;
    for(i=0;i<3;i++)
        s+=vec[i]*vec[i];
    return(sqrt(s));
}

```

norme của v1 : 2.236068 - norme của v2 : 3.316625

Hàm norme() nhằm mục đích tính chuẩn của vector với ba thành phần.

Từ khoá **inline** yêu cầu chương trình biên dịch xử lý hàm norme khác với các hàm thông thường. Cụ thể là, mỗi lần gọi norme(), trình biên dịch ghép trực tiếp các chỉ thị tương ứng của hàm vào trong chương trình (ở dạng ngôn ngữ máy). Do đó cơ chế quản lý lời gọi và trả về không cần nữa (không cần lưu trữ cảnh, sao chép các thông số...), nhờ vậy tiết kiệm thời gian thực hiện. Bên cạnh đó các chỉ thị tương ứng sẽ được sinh ra mỗi khi gọi hàm do đó chi phí lưu trữ tăng lên khi hàm được gọi nhiều lần.

Điểm bất lợi khi sử dụng các hàm **inline** là nếu chúng quá lớn và được gọi thường xuyên thì kích thước chương trình sẽ tăng lên rất nhanh. Vì lý do này, chỉ những hàm đơn giản, không chứa các cấu trúc lặp mới được khai báo là hàm **inline**.

Việc sử dụng hàm **inline** so với các macro có tham số có hai điểm lợi. Trước hết hàm **inline** cung cấp một cách có cấu trúc hơn khi mở rộng các hàm đơn giản thành hàm **inline**. Thực tế cho thấy khi tạo một macro có tham số thường hay quên các dấu đóng ngoặc, rất cần đến để đảm bảo sự mở rộng nội tuyến riêng trong mỗi trường hợp. Với macro có thể gây ra hiệu ứng phụ hoặc bị hạn chế khả năng sử dụng. Chẳng hạn với macro:

```
#define square(x) {x++*x++}
```

Với lời gọi

square(a)

với a là biến sẽ sản sinh ra biểu thức a++*a++ và kết quả là làm thay đổi giá trị của biến a tới hai lần

Còn lời gọi

square (3)

sẽ gây lỗi biên dịch vì ta không thể thực hiện các phép toán tăng giảm trên các toán hạng là hằng số.

Việc sử dụng hàm **inline** như một giải pháp thay thế sẽ tránh được các tình huống như thế.

Ngoài ra, các hàm **inline** có thể được tối ưu bởi chương trình biên dịch.

Điều quan trọng là đặc tả **inline** chỉ là một yêu cầu, chứ không phải là một chỉ thị đối với trình biên dịch. Nếu vì một lý do nào đó trình biên dịch không thể đáp ứng được yêu cầu (chẳng hạn khi bên trong định nghĩa hàm **inline** có các cấu trúc lặp) thì hàm sẽ được biên dịch như một hàm bình thường và yêu cầu **inline** sẽ bị bỏ qua.

Hàm **inline** phải được khai báo bên trong tệp tin nguồn chứa các hàm sử dụng nó. Không thể dịch tách biệt các hàm **inline**.¹

5. THAM CHIẾU

Ngôn ngữ C++ giới thiệu một khái niệm mới “reference” tạm dịch là “tham chiếu”. Về bản chất, tham chiếu là “bí danh” của một vùng nhớ được cấp phát cho một biến nào đó.

Một tham chiếu có thể là một biến, tham số hình thức của hàm hay dùng làm giá trị trả về của một hàm. Các phần tiếp sau lần lượt giới thiệu các khả năng của tham chiếu được sử dụng trong chương trình viết bằng ngôn ngữ C++.

5.1 Tham chiếu tới một biến

Xét hai chỉ thị:

```
int n;
int &p = n;
```

Trong chỉ thị thứ hai, dấu “&” để xác định p là một biến tham chiếu còn dấu “=” và tên biến n để xác định vùng nhớ mà p tham chiếu tới. Lúc này cả hai định danh p và n cùng xác định vùng nhớ được cấp phát cho biến n. Như vậy các chỉ thị sau:

```
n = 3;
cout << p;
```

¹ “Biên dịch tách biệt” cho phép khai báo hàm trong một tệp tiêu đề, còn định nghĩa hàm đó lại ở trong tệp tin chương trình sử dụng hàm.

cho kết quả 3 trên thiết bị hiển thị

Xét về bản chất tham chiếu và tham trỏ giống nhau vì cùng chỉ đến các đối tượng có địa chỉ, cùng được cấp phát địa chỉ khi khai báo. Nhưng cách sử dụng chúng thì khác nhau. Khi nói đến tham chiếu “`&p`” ta phải gắn nó với một biến nào đó đã khai báo qua “`&p`” , trong khi đó khai báo con trỏ “`*p`” không nhất thiết phải khởi tạo giá trị cho nó. Trong chương trình biên trỏ có thể tham chiếu đến nhiều biến khác nhau còn biến tham chiếu thì “vẫn đã đóng thuyền” từ khi khai báo, có nghĩa là sau khi khởi tạo cho tham chiếu gắn với một biến nào đó rồi thì ta không thể thay đổi để gắn tam chiếu với một biến khác.

Xét các chi tiết sau:

Ví dụ 2.8

```
int n=2,m=4;
int *p;
p=&n; //p chỉ đến n
*p=7; //gán giá trị của m cho n thông qua con trỏ p
...
p=&m; //cho p chỉ đến m
int &q=n; //khai báo tham chiếu q chỉ đến n
q=8; //gán cho biến n giá trị 8
...
p=q; //gán giá trị của biến m cho biến n
```

Nói một cách đơn giản, tham chiếu của một biến giống như bí danh của một con người nào đó. Có nghĩa là để chỉ đến một con người cụ thể nào đó, ta có thể đồng thời sử dụng tên của anh ta hoặc bí danh. Do vậy, để truy nhập đến vùng nhớ tương ứng với một biến, chúng ta có thể sử dụng hoặc là tên biến hoặc là tên tham chiếu tương ứng. Đối với con người, bí danh bao giờ cũng nhằm nói đến một người đã tồn tại, và như vậy tham chiếu cũng phải được khai báo và khởi tạo sau khi biến được khai báo. Chương trình sau đây sẽ gây lỗi biên dịch do tham chiếu `y` chưa được khởi tạo.

Ví dụ 2.9

Chương trình sai

```
#include <iostream.h>
int x,y,z;
y=x*x*x;
cout<<y;
int a=b; //lỗi vì y phải được khởi tạo
```

```

cout << " x = "<<x<<"\n"
      << " y = "<<y<<"\n";
y = 7;
cout << " x = "<<x<<"\n"
      << " y = "<<y<<"\n";
}

```

Chương trình đúng như sau:

```

#include <iostream.h>
void main() {
    int x=3, &y=x;//y bây giờ là một "bí danh" của x
    cout << " x = "<<x<<"\n"
        << " y = "<<y<<"\n";
    y = 7;
    cout << " x = "<<x<<"\n"
        << " y = "<<y<<"\n";
}

```

```

x = 3
y = 3
x = 7
y = 7

```

Lưu ý cuối cùng là không thể gắn một tham chiếu với một hàng số trừ trường hợp có từ khoá const đứng trước khai báo tham chiếu. Để dễ dàng kiểm tra các nhận xét sau:

```

int &p =3; //không hợp lệ
const int & p =3; //hợp lệ

```

5.2 Truyền tham số cho hàm bằng tham chiếu

Trong C, các tham số và giá trị trả về của một hàm được truyền bằng giá trị. Để giả lập cơ chế truyền tham biến ta phải sử dụng con trỏ.

Trong C++, việc dùng khái niệm tham chiếu trong khai báo tham số hình thức của hàm sẽ yêu cầu chương trình biên dịch truyền địa chỉ của biến cho hàm và hàm sẽ thao tác trực tiếp trên các biến đó. Chương trình sau đưa ra ba cách viết khác nhau của hàm thực hiện việc hoán đổi nội dung của hai biến.

Ví dụ 2.10

```
/*swap.cpp*/
#include <conio.h>
#include <iostream.h>
/*Hàm swap1 được gọi với các tham số được truyền theo tham trị*/
void swap1(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

/*Hàm swap2 thực hiện việc truyền tham số bằng tham trỏ*/
void swap2(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

/*Hàm swap3 thực hiện việc truyền tham số bằng tham chiếu*/
void swap3(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}

void main() {
    int a=3, b=4;
    clrscr();
    cout << "Trước khi gọi swap1:\n";
    cout << "a = " << a << " b = " << b << "\n";
    swap1(a,b);
    cout << "Sau khi gọi swap:\n";
    cout << "a = " << a << " b = " << b << "\n";
}
```

```

a=3; b =4;
cout <<"Truoc khi goi swap2:\n";
cout<<"a = "<<a<<" b = "<<b<<"\n";
swap2 (&a, &b);
cout <<"Sau khi goi swap2:\n";
cout<<"a = "<<a<<" b = "<<b<<"\n";
a=3;b=4;
cout <<"Truoc khi goi swap3:\n";
cout<<"a = "<<a<<" b = "<<b<<"\n";
swap3 (a,b);
cout <<"Sau khi goi swap3:\n";
cout<<"a = "<<a<<" b = "<<b<<"\n";
getch();
}

```

```

Truoc khi goi swap1:
a = 3 b = 4
Sau khi goi swap1:
a = 3 b = 4
Truoc khi goi swap2
a = 3 b = 4
Sau khi goi swap2:
a = 4 b = 3
Truoc khi goi swap3:
a = 3 b = 4
Sau khi goi swap3:
a = 4 b = 3

```

Trong chương trình trên, ta truyền tham số a, b cho hàm swap1() theo tham trị cho nên giá trị của chúng không thay đổi trước và sau khi gọi hàm.

Giải pháp đưa ra trong hàm swap2() là thay vì truyền trực tiếp giá trị hai biến a và b người ta truyền địa chỉ của chúng rồi thông qua các địa chỉ này để xác định giá trị biến. Bằng cách đó giá trị của hai biến a và b sẽ hoán đổi cho nhau sau lời gọi hàm. Khác với swap2(), hàm swap3() đưa ra giải pháp sử dụng tham chiếu. Các tham số hình thức của hàm swap3() bây giờ là các tham chiếu đến các tham số thực được truyền cho hàm. Nhờ vậy mà giá trị của hai tham số thực a và b có thể hoán đổi được cho nhau.

Có một vấn đề mà chắc rằng bạn đọc sẽ phản vấn: “làm thế nào để khởi tạo các tham số hình thức là tham chiếu”. Xin thưa rằng điều đó được thực hiện tự động trong mỗi lời gọi hàm chứ không phải trong định nghĩa. Từ đó này sinh thêm một nhận xét quan trọng: “tham số ứng với tham số hình thức là tham chiếu phải là biến trừ trường hợp có từ khoá const đứng trước khai báo tham số hình thức”. Chẳng hạn không thể thực hiện lời gọi hàm sau:

```
swap3(2, 3);
```

Bạn đọc có thể xem thêm phần “Hàm thiết lập sao chép lại” để thấy được ích lợi to lớn của việc truyền tham số cho hàm bằng tham chiếu.

Chú ý: khi muốn truyền bằng tham biến một biến trả thì viết như sau:

```
int*& adr; //adr là một tham chiếu tới một biến trả chỉ đến một biến nguyên.
```

5.3 Giá trị trả về của hàm là tham chiếu

Định nghĩa của hàm có dạng như sau:

```
<type> & fct( ... ) {  
    ...  
    return <bien co pham vi toan cuc>;  
}
```

Trong trường hợp này biểu thức được trả lại trong câu lệnh **return** phải là tên của một biến xác định từ bên ngoài hàm, bởi vì chỉ khi đó mới có thể sử dụng được giá trị của hàm. Khi ta trả về một tham chiếu đến một biến cục bộ khai báo bên trong hàm, biến cục bộ này sẽ bị mất đi khi kết thúc thực hiện hàm và do vậy, tham chiếu của hàm cũng không còn có ý nghĩa nữa.

Khi giá trị trả về của hàm là tham chiếu, ta có thể gặp các câu lệnh gán “kỳ dị” trong đó vẽ trái là một lời gọi hàm chứ không phải là tên của một biến. Điều này hoàn toàn hợp lý, bởi lẽ bản thân hàm đó có giá trị trả về là một tham chiếu. Nói cách khác, vẽ trái của lệnh gán (biểu thức gán) có thể là lời gọi đến một hàm có giá trị trả về là một tham chiếu.

Xét ví dụ sau đây:

Ví dụ 2.11

```
/*lr.cpp*/  
#include <iostream.h>  
#include <conio.h>  
int a[5];
```

```

int &fr(int *d,int i);
void main() {
    clrscr();
    cout<<"Nhập giá trị cho mảng a:\n";
    for(int i=0;i<5;i++) {
        cout<<"a["<<i<<"] = ";
        cin>>fr(a,i);
    }
    cout<<"Mảng a sau khi nhập\n";
    for(i=0;i<5;i++)
        cout<<a[i]<<" ";
    cout<<"\n";
    getch();
}
int &fr(int *d,int i) {
    return d[i];
}

```

```

Nhập giá trị cho mảng a:
a[0]= 6
a[1]= 4
a[2]= 3
a[3]= 5
a[4]= 6
Mảng a sau khi nhập
6 4 3 5 6

```

Bạn đọc có thể xem thêm phần “Định nghĩa chồng toán tử” để thấy được lợi ích của vấn đề trả về tham chiếu cho hàm.

6. ĐỊNH NGHĨA CHỒNG HÀM (Overloading functions)

C++ cho phép sử dụng một tên cho nhiều hàm khác nhau ta gọi đó là sự “chồng hàm”. Trong trường hợp đó, các hàm sẽ khác nhau ở giá trị trả về và danh sách kiểu các tham số. Chẳng hạn chúng ta muốn định nghĩa các hàm trả về số nhỏ nhất trong:

1. hai số nguyên
2. hai số thực
3. hai ký tự
4. ba số nguyên
5. một dãy các số nguyên ...

Đĩ nhiên có thể tìm cho mỗi hàm như vậy một tên phân. Lợi dụng khả năng “định nghĩa chồng hàm” của C++, chúng ta có thể viết các hàm như sau:

Ví dụ 2.12

```
#include <iostream.h>

//Hàm nguyên mẫu
int min(int, int); //Hàm 1

double min(double, double); //Hàm 2

char min(char, char); //Hàm 3

int min(int, int, int); //Hàm 4

int min(int, int *); //Hàm 5

main() {
    int n=10, p =12, q = -12;
    double x = 2.3, y = -1.2;
    char c = 'A', d= 'Q';
    int td[7] = {1,3,4,-2,0,23,9};

    cout<<"min (n,p) : "<<min(n,p)<<"\n"; //Hàm 1
    cout<<"min (n,p,q) : "<<min(n,p,q)<<"\n"; //Hàm 4
    cout<<"min (c,d) : "<<min(c,d)<<"\n"; //Hàm 3
    cout<<"min (x,y) : "<<min(x,y)<<"\n"; //Hàm 2
    cout<<"min (td) : "<<min(td)<<"\n"; //Hàm 5
    cout<<"min (n,x) : "<<min(n,x)<<"\n"; //Hàm 2
    //cout<<"min (n,p,x) : "<<min(n,p,x)<<"\n"; //Lỗi
}

int min(int a, int b) {
    return (a> b? a: b);
}

int min(int a, int b, int c) {
```

```
    return (min(min(a,b),c));
}

double min(double a, double b) {
    return (a> b? a: b);
}

char min(char a, char b) {
    return (a> b? a: b);
}

int min(int n, int *t) {
int res = t[0];
for (int i=1; i<n; i++)
    res = min(res,t[i]);
return res;
}
```

Nhận xét

1. Một hàm có thể gọi đến hàm cùng tên với nó (ví dụ như hàm 4.5 gọi hàm 1).
2. Trong trường hợp có các hàm trùng tên trong chương trình, việc xác định hàm nào được gọi do chương trình dịch đảm nhiệm và tuân theo các nguyên tắc sau:

Trường hợp các hàm có một tham số

Chương trình dịch tìm kiếm “sự tương ứng nhiều nhất” có thể được; có các mức độ tương ứng như sau (theo độ ưu tiên giảm dần):

- a) Tương ứng thật sự: ta phân biệt các kiểu dữ liệu cơ sở khác nhau đồng thời lưu ý đến cả dấu.
- b) Tương ứng dữ liệu số nhưng có sự chuyển đổi kiểu dữ liệu tự động (“numeric promotion”): **char** và **short** -->**int**; **float** -->**int**.
- c) Các chuyển đổi kiểu chuẩn được C và C++ chấp nhận.
- d) Các chuyển đổi kiểu do người sử dụng định nghĩa.

Quá trình tìm kiếm bắt đầu từ mức cao nhất và dừng lại ở mức đầu tiên cho phép tìm thấy sự phù hợp. Nếu có nhiều hàm phù hợp ở cùng một mức, chương trình dịch đưa ra thông báo lỗi do không biết chọn hàm nào giữa các hàm phù hợp.

Trường hợp các hàm có nhiều tham số

Ý tưởng chung là phải tìm một hàm phù hợp nhất so với tất cả những hàm còn lại. Để đạt mục đích này, chương trình dịch chọn cho mỗi tham số các hàm phù hợp (ở tất cả các mức độ). Trong số các hàm được lựa chọn, chương trình dịch chọn ra (nếu tồn tại và tồn tại duy nhất) hàm sao cho đối với mỗi đối số nó đạt được sự phù hợp hơn cả so với các hàm khác.

Trong trường hợp vẫn có nhiều hàm thỏa mãn, lỗi biên dịch xảy ra do chương trình dịch không biết chọn hàm nào trong số các hàm thỏa mãn. Đặc biệt lưu ý khi sử dụng định nghĩa chống hàm cùng với việc khai báo các hàm với tham số có giá trị ngầm định sẽ được trình bày trong mục tiếp theo.

7. THAM SỐ NGẦM ĐỊNH TRONG LỜI GỌI HÀM

Ta xét ví dụ sau:

Ví dụ 2.13

```
#include <iostream.h>
void main() {
    int n=10,p=20;
    void fct(int, int = 12); //khai báo hàm với một giá trị ngầm định
    fct(n,p); //lời gọi thông thường, có hai tham số
    fct(n); //lời gọi chỉ với một tham số
    //fct() sẽ không được chấp nhận
}
//khai báo bình thường
void fct(int a, int b) {
    cout << "tham so thu nhat : " <<a << "\n";
    cout << "tham so thu hai : " <<b << "\n";
}
```

```
tham so thu nhat : 10
tham so thu hai : 20
tham so thu nhat : 10
tham so thu hai : 12
```

Trong khai báo của fct() bên trong hàm main():

```
void fct(int,int =12);
```

khai báo

```
int = 12
```

chỉ ra rằng trong trường hợp vắng mặt tham số thứ hai ở lời gọi hàm fct() thì tham số hình thức tương ứng sẽ được gán giá trị ngầm định 12.

Lời gọi

```
fct();
```

không được chấp nhận bởi vì không có giá trị ngầm định cho tham số thứ nhất.

Ví dụ 2.14

```
#include <iostream.h>
void main(){
    int n=10,p=20;
    void fct(int = 0, int = 12); //khai báo hàm với hai tham số có giá trị ngầm định
    fct(n,p); //lời gọi thông thường, có hai tham số
    fct(n); //lời gọi chỉ với một tham số
    fct(); //fct() đã được chấp nhận
}
void fct(int a, int b) //khai báo bình thường
{
    cout<<"tham so thu nhat : "<<a<<"\n";
    cout<<"tham so thu hai : "<<b<<"\n";
}
```

```
tham so thu nhat : 10
tham so thu hai : 20
tham so thu nhat : 10
tham so thu hai : 12
tham so thu nhat : 0
tham so thu hai : 12
```

Chú ý

- Các tham số với giá trị ngầm định phải được đặt ở cuối trong danh sách các tham số của hàm để tránh nhầm lẫn các giá trị.
- Các giá trị ngầm định của tham số được khai báo khi sử dụng chử không phải trong phần định nghĩa hàm. Ví dụ sau đây gây ra lỗi biên dịch:

Ví dụ 2.15

```
#include <conio.h>
#include <iostream.h>
void f();
void main() {
    clrscr();
    int n=10,p=20;
    void fct(int =0,int =12);
    cout<<"Gọi fct trong main\n";
    fct(n,p);
    fct(n);
    fct();
    getch();
}
void fct(int a=10,int b=100) {
    cout<<"Tham số thứ nhất : "<<a<<"\n";
    cout<<"Tham số thứ hai : "<<b<<"\n";
}
```

3. Nếu muốn khai báo giá trị ngầm định cho một tham số biến trả, thì phải chú ý viết * và = cách xa nhau ít nhất một dấu cách.
4. Các giá trị ngầm định có thể là một biểu thức bất kỳ (không nhất thiết chỉ là biểu thức hằng), có giá trị được tính tại thời điểm khai báo:

```
float x;
int n;
void fct(float = n*2+1.5);
```

5. Chỗng hàm và gọi hàm với tham số có giá trị ngầm định có thể sẽ dẫn đến lỗi biên dịch khi chương trình dịch không xác định được hàm phù hợp.

Xét ví dụ sau:

Ví dụ 2.16

```
#include <iostream.h>
void fct(int, int=10);
void fct(int);
```

```
void main() {  
    int n=10, p=20;  
    fct(n,p); //OK  
    fct(n); //ERROR  
}
```

8. BỔ SUNG THÊM CÁC TOÁN TỬ QUẢN LÝ BỘ NHỚ ĐỘNG: **new** và **delete**

8.1 Toán tử cấp phát bộ nhớ động **new**

Với khai báo

```
int * adr;
```

chỉ thị

```
ad = new int;
```

cho phép cấp phát một vùng nhớ cần thiết cho một phân tử có kiểu int và gán cho adr địa chỉ tương ứng. Lệnh tương đương trong C:

```
ad =(int *) malloc(sizeof(int));
```

Với khai báo

```
char *adc;
```

chỉ thị

```
adc =new char [100];
```

cho phép cấp phát vùng nhớ đủ cho một bảng chứa 100 ký tự và đặt địa chỉ đầu của vùng nhớ cho biến adc. Lệnh tương ứng trong C như sau:

```
adc =(char *)malloc(100);
```

Hai cách sử dụng **new** như sau:

Dạng I

 new type;

 giá trị trả về là

(i) một con trỏ đến vị trí tương ứng khi cấp phát thành công

(ii) NULL trong trường hợp trái lại.

Dạng 2

```
new type[n];
```

trong đó n là một biểu thức nguyên không âm nào đó, khi đó toán tử **new** xin cấp phát vùng nhớ đủ để chứa n thành phần kiểu type và trả lại con trỏ đến đầu vùng nhớ đó nếu như cấp phát thành công.

8.2 Toán tử giải phóng vùng nhớ động **delete**

Một vùng nhớ động được cấp phát bởi **new** phải được giải phóng bằng **delete** mà không thể dùng **free** được, chẳng hạn:

```
delete adr;
delete adc;
```

Ví dụ 2.17

Cấp phát bộ nhớ động cho mảng hai chiều

```
#include<iostream.h>
void Nhap(int **mat); //nhập ma trận hai chiều
void In(int *mat); //In ma trận hai chiều
void main() {
    int **mat;
    int i;
    /*cấp phát mảng 10 con trỏ nguyên*/
    mat = new int *[10];
    for(i=0; i<10; i++)
        /*mỗi con trỏ nguyên xác định vùng nhớ 10 số nguyên*/
        mat[i] = new int [10];
    /*Nhập số liệu cho mảng vừa được cấp phát*/
    cout<<"Nhập số liệu cho matran 10*10\n";
    Nhap(mat);
    /*In ma trận*/
    cout<<"Ma tran vua nhap \n";
    In(mat);
    /*Giải phóng bộ nhớ*/
}
```

```

for(i=0;i<10;i++)
    delete mat[i];
delete mat;
}

void Nhap(int ** mat) {
    int i,j;
    for(i=0; i<10;i++)
        for(j=0; j<10;j++) {
            cout<<"Thanh phan thu ["<<i<<"] ["<<j<<")= ";
            cin>>mat[i][j];
        }
}

void In(int ** mat) {
    int i,j;
    for(i=0; i<10;i++) {
        for(j=0; j<10;j++)
            cout<<mat[i][j]<< " ";
        cout<<"\n";
    }
}

```

Ví dụ 2.18

Quản lý tràn bộ nhớ set_new_handler

```

#include <iostream>

main()
{
void outof();
set_new_handler(&outof);
long taille;
int *adr;
int nbloc;
cout<<"Kich thuoc can nhap? ";

```

```

cin >>taille;
for(nbloc=1;nbloc++)
{
    adr =new int [taille];
    cout <<"Cap phat bloc so : "<<nbloc<<"\n";
}
}

void outof()//ham được gọi khi thiếu bộ nhớ
{
    cout <<"Het bo nho -Ket thuc \n";
    exit(1);
}

```

9. TÓM TẮT

9.1 Ghi nhớ

C++ là một sự mở rộng của C(superset), do đó có thể sử dụng một chương trình biên dịch C++ để dịch và thực hiện các chương trình nguồn viết bằng C.

C yêu cầu các chú thích nằm giữa /* và */. C++ còn cho phép tạo một chú thích bắt đầu bằng “//” cho đến hết dòng.

C++ cho phép khai báo khá tuỳ ý. thậm chí có thể khai báo biến trong phần khởi tạo của câu lệnh lặp for.

C++ cho phép truyền tham số cho hàm bằng tham chiếu. Điều này tương tự như truyền tham biến cho chương trình con trong ngôn ngữ PASCAL. Trong lời gọi hàm ta dùng tên biến và biến đó sẽ được truyền cho hàm qua tham chiếu. Điều đó cho phép thao tác trực tiếp trên biến được truyền chứ không phải gián tiếp qua biến trỏ.

Toán tử **new** và **delete** trong C++ được dùng để quản lý bộ nhớ động thay vì các hàm cấp phát động của C.

C++ cho phép người viết chương trình mô tả các giá trị ngầm định cho các tham số của hàm, nhờ đó hàm có thể được gọi với một danh sách các tham số không đầy đủ.

Toán tử “::” cho phép truy nhập biến toàn cục khi đồng thời sử dụng biến cục bộ và toàn cục trùng tên.

Có thể định nghĩa các hàm cùng tên với các tham số khác nhau. Hai hàm cùng tên sẽ được phân biệt nhờ giá trị trả về và danh sách kiểu các tham số.

9.2 Các lỗi thường gặp

Quên đóng */ cho các chú thích.

Khai báo biến sau khi biến được sử dụng.

Sử dụng lệnh **return** để trả về giá trị nhưng khi định nghĩa lại mô tả hàm kiểu **void** hoặc ngược lại, quên câu lệnh này trong trường hợp hàm yêu cầu có giá trị trả về.

Không có hàm nguyên mẫu cho các hàm.

Bỏ qua việc khởi tạo cho các tham chiếu.

Thay đổi giá trị của biến hằng.

Tạo các hàm cùng tên, cùng tham số.

9.3 Một số thói quen lập trình tốt

Sử dụng “//” để tránh lối không đóng */ khi chú thích nằm gọn trong một dòng.

Sử dụng các khả năng vào ra mới của C++ để chương trình dễ đọc hơn.

Đặt các khai báo biến lên đầu các khối lệnh.

Chỉ dùng từ khoá inline với các hàm “nhỏ”, “không phức tạp”

Sử dụng con trỏ để truyền tham số cho hàm khi cần thay đổi giá trị tham số, còn tham chiếu dùng để truyền các tham số có kích thước lớn mà không có nhu cầu thay đổi nội dung.

Tránh sử dụng biến cùng tên cho nhiều mục đích khác nhau trong chương trình.

10. BÀI TẬP

Bài tập 2.1

Sử dụng cin và cout viết chương trình nhập vào một dãy số nguyên rồi sắp xếp dãy số đó tăng dần.

Bài tập 2.2

Sử dụng new và delete thực hiện các thao tác cấp phát bộ nhớ cho một mảng động hai chiều. Hoàn thiện chương trình bằng cách thực hiện các thao tác liên quan đến ma trận vuông.

Bài tập 2.3

Lập chương trình mô phỏng các hoạt động trên một ngăn xếp chứa các số nguyên.

ĐỐI TƯỢNG VÀ LỚP (Object & Class)

Mục đích chương này:

1. Khái niệm về đóng gói dữ liệu.
2. Khai báo và sử dụng một lớp.
3. Khai báo và sử dụng đối tượng, con trỏ đối tượng, tham chiếu đối tượng.
4. Hàm thiết lập và hàm huỷ bỏ.
5. Khai báo và sử dụng hàm thiết lập sao chép.
6. Vai trò của hàm thiết lập ngầm định.

1. ĐỐI TƯỢNG

Đối tượng là một khái niệm trong lập trình hướng đối tượng biểu thị sự liên kết giữa dữ liệu và các thủ tục (gọi là các phương thức) thao tác trên dữ liệu đó. Ta có công thức sau:

$$\text{Đối tượng} = \text{Dữ liệu} + \text{Phương thức}$$

Ở đây chúng ta hiểu rằng đối tượng chính là công cụ hỗ trợ cho sự đóng gói. Sự đóng gói là cơ chế liên kết các lệnh thao tác và dữ liệu có liên quan, giúp cho cả hai được an toàn tránh được sự can thiệp từ bên ngoài và việc sử dụng sai. Nhìn chung định nghĩa một đối tượng phức tạp hơn so với định nghĩa các biến cấu trúc thông thường, bởi lẽ ngoài việc mô tả các thành phần dữ liệu, ta còn phải xác định được các thao tác tác động lên đối tượng đó. Hình 3.1 mô tả các đối tượng điểm trên mặt phẳng.

Mỗi đối tượng được xác định bởi hai thành phần toạ độ được biểu diễn bởi hai biến nguyên. Các thao tác tác động lên điểm bao gồm việc xác định toạ độ một điểm trên mặt phẳng toạ độ (thể hiện bằng việc gán giá trị cho hai thành phần toạ độ), thay đổi toạ độ và hiển thị kết quả lên trên mặt phẳng toạ độ (tương tự như việc chấm điểm trên mặt phẳng đó).

Lợi ích của việc đóng gói là khi nhìn từ bên ngoài, một đối tượng chỉ được biết tới bởi các mô tả về các phương thức của nó, cách thức cài đặt các dữ liệu không quan trọng đối với người sử dụng. Với một đối tượng điểm, người ta chỉ quan tâm đến việc có thể thực hiện được thao tác gì trên nó mà không cần biết các thao tác đó được thực hiện như thế nào, cũng như điều gì xảy ra bên trong bản thân đối tượng.

đó. Ta thường nói đó là “sự trìn̄u tượng hoá dữ liệu” (khi các chi tiết cài đặt cụ thể được giấu đi).

```
Mô tả đối tượng điểm {
//dữ liệu
int x, y;
//phương thức
void init(int ox, int oy);
void move(int dx, int dy);
void display();
};
```

Hình 3.1 Mô tả các đối tượng điểm.

Đóng gói có nhiều lợi ích góp phần nâng cao chất lượng của chương trình. Nó làm cho công việc bảo trì chương trình thuận lợi hơn rất nhiều: một sự thay đổi cấu trúc của một đối tượng chỉ ảnh hưởng tới bản thân đối tượng; người sử dụng đối tượng không cần biết đến thay đổi này (với lập trình cấu trúc thì người lập trình phải tự quản lý sự thay đổi đó). Chẳng hạn có thể biểu diễn tọa độ một điểm dưới dạng số thực, khi đó chỉ có người thiết kế đối tượng phải quan tâm để sửa lại định nghĩa của đối tượng trong khi đó người sử dụng không cần hay biết về điều đó, miễn là những thay đổi đó không tác động đến việc sử dụng đối tượng điểm.

Tương tự như vậy, ta có thể bổ sung thêm thuộc tính màu và một số thao tác lên một đối tượng điểm, để có được một đối tượng điểm màu. Rõ ràng là đóng gói cho phép đơn giản hóa việc sử dụng một đối tượng.

Trong lập trình hướng đối tượng, đóng gói cho phép dữ liệu của đối tượng được che lấp khi nhìn từ bên ngoài, nghĩa là nếu người dùng muốn tác động lên dữ liệu của đối tượng thì phải gửi đến đối tượng các thông điệp(message). Ở đây các phương thức đóng vai trò là giao diện bắt buộc giữa các đối tượng và người sử dụng. Ta có nhận xét: “*Lời gọi đến một phương thức là truyền một thông báo đến cho đối tượng*”.

Các thông điệp gửi tới đối tượng nào sẽ gắn chặt với đối tượng đó và chỉ đối tượng nào nhận được thông điệp mới phải thực hiện theo thông điệp đó; chẳng hạn các đối tượng điểm độc lập với nhau, vì vậy thông điệp thay đổi tọa độ đối tượng điểm p chỉ làm ảnh hưởng đến các thành phần tọa độ trong p chứ không thể thay đổi được nội dung của một đối tượng điểm q khác.

So với lập trình hướng đối tượng thuần tuý, các cài đặt cụ thể của đối tượng trong C++ linh động hơn một chút, bằng cách cho phép chỉ che dấu một bộ phận dữ liệu của đối tượng và mở rộng hơn khả năng truy nhập đến các thành phần riêng của đối tượng. Khái niệm lớp chính là cơ sở cho các linh động này.

Lớp là một mô tả trìn̄u tượng của nhóm các đối tượng có cùng bản chất. Trong một lớp người ta đưa ra các mô tả về tính chất của các thành phần dữ liệu, cách thức

thao tác trên các thành phần này (hành vi của các đối tượng), ngược lại mỗi một đối tượng là một thể hiện cụ thể cho những mô tả trắc tượng đó. Trong các ngôn ngữ lập trình, lớp đóng vai trò một kiểu dữ liệu được người dùng định nghĩa và việc tạo ra một đối tượng được ví như khai báo một biến có kiểu lớp.

2. LỚP

2.1 Khai báo lớp

Từ quan điểm của lập trình cấu trúc, lớp là một kiểu dữ liệu tự định nghĩa. Trong lập trình hướng đối tượng, chương trình nguồn được phân bố trong khai báo và định nghĩa của các lớp.

Sau đây là một ví dụ điển hình về cú pháp khai báo lớp. Kinh nghiệm cho thấy mọi kiểu khai báo khác đều có thể chuẩn hóa để đưa về dạng này.

```
class <tên lớp> {
    private:
        <khai báo các thành phần riêng trong từng đối tượng>
    public:
        <khai báo các thành phần công cộng của từng đối tượng>
    };
<định nghĩa của các hàm thành phần chưa được định nghĩa bên trong khai báo lớp>
...
}
```

Các chi tiết liên quan đến khai báo lớp sẽ lần lượt được đề cập đến trong các phần sau. Để dễ hình dung xét một ví dụ về khai báo lớp điểm trong mặt phẳng. Trong ví dụ này ta có đề cập đến một vài khía cạnh liên quan đến khai báo lớp, đối tượng và sử dụng chúng.

Ví dụ 3.1

```
/*point.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    /*khai báo các thành phần dữ liệu riêng*/
    private:
        int x,y;
```

```

/*khai báo các hàm thành phần công cộng*/
public:
    void init(int ox, int oy);
    void move(int dx, int dy);
    void display();
/*định nghĩa các hàm thành phần bên ngoài khai báo lớp*/
void point::init(int ox, int oy) {
    cout<<"Ham thanh phan init\n";
    x = ox; y = oy; /*x,y là các thành phần của đối tượng gọi hàm thành phần*/
}
void point::move(int dx, int dy) {
    cout<<"Ham thanh phan move\n";
    x += dx; y += dy;
}
void point::display() {
    cout<<"Ham thanh phan display\n";
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
}
void main() {
    clrscr();
    point p;
    p.init(2,4); /*gọi hàm thành phần từ đối tượng*/
    p.display();
    p.move(1,2);
    p.display();
    getch();
}

```

```

Ham thanh phan init
Ham thanh phan display
Toa do: 2 4
Ham thanh phan move

```

Hàm thành phần display

Tọa độ: 3 6

Nhận xét

- Có thể khai báo trực tiếp các hàm thành phần bên trong khai báo lớp. Tuy vậy điều đó đôi khi làm mất mỹ quan của chương trình nguồn, do vậy người ta thường sử dụng cách khai báo các hàm thành phần ở bên ngoài khai báo lớp. Khi đó ta sử dụng cú pháp:

```
<ten kiểu giá trị trả lại> <ten lớp>::<ten hàm> (<danh sách tham số>) {  
    <nội dung>  
}
```

- Gọi hàm thành phần của lớp từ một đối tượng chính là truyền thông điệp cho hàm thành phần đó. Cú pháp như sau:

```
<ten đối tượng>.<ten hàm thành phần>(<danh sách các tham số nếu có>);
```

2.1.1 Tạo đối tượng

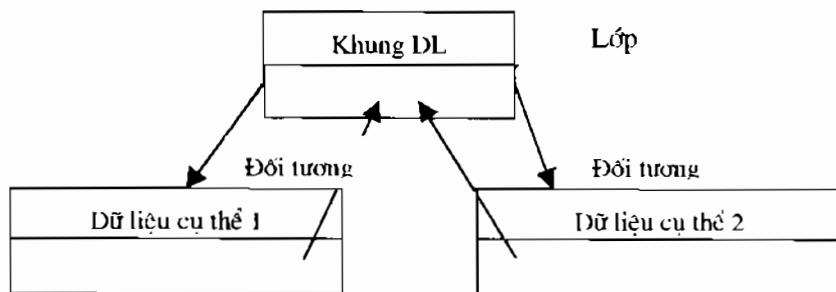
Trong C++, một đối tượng có thể được xác lập thông qua một biến/hằng số kiểu lớp.

```
<ten lớp> <ten đối tượng>;
```

Do đó vùng nhớ được cấp phát cho một biến kiểu lớp sẽ cho ta một khung của đối tượng bao gồm dữ liệu là các thể hiện cụ thể của các mô tả dữ liệu trong khai báo lớp cùng với các thông điệp gửi tới các hàm thành phần.

Mỗi đối tượng sở hữu một tập các biến tương ứng với tên và kiểu của các thành phần dữ liệu định nghĩa trong lớp. Ta gọi chúng là các biến thể hiện của đối tượng. Tuy nhiên tất cả các đối tượng cùng một lớp chung nhau định nghĩa của các hàm thành phần.

Lớp là một kiểu dữ liệu vì vậy có thể khai báo con trỏ hay tham chiếu đến một đối tượng thuộc lớp và bằng cách ấy có thể truy nhập gián tiếp đến đối tượng. Nhưng chú ý là con trỏ và tham chiếu không phải là một thể hiện của lớp.



Hình 3.2 Đối tượng là một thể hiện của lớp.

2.1.2 Các thành phần dữ liệu

Cú pháp khai báo các thành phần dữ liệu giống như khai báo biến:

```
<tên kiểu> <tên thành phần>;
```

Một thành phần dữ liệu có thể là một biến kiểu cơ sở (**int**, **float**, **double**, **char**, **char***), kiểu trường bit, kiểu liệt kê (**enum**) hay các kiểu do người dùng định nghĩa. Thậm chí, thành phần dữ liệu còn có thể là một đối tượng thuộc lớp đã được khai báo trước đó. Tuy nhiên không thể dùng **trực tiếp** các lớp để khai báo kiểu thành phần dữ liệu thuộc vào bản thân lớp đang được định nghĩa. Muốn vậy, trong khai báo của một lớp có thể dùng các con trỏ hoặc tham chiếu đến các đối tượng của chính lớp đó.

Trong khai báo của các thành phần dữ liệu, có thể sử dụng từ khoá **static** nhưng không được sử dụng các từ khoá **auto**, **register**, **extern** trong khai báo các thành phần dữ liệu. Cũng không thể khai báo và khởi đầu giá trị cho các thành phần đó.

2.1.3 Các hàm thành phần

Hàm được khai báo trong định nghĩa của lớp được gọi là hàm thành phần hay phương thức của lớp (hàm thành phần là thuật ngữ của C++, còn phương thức là thuật ngữ trong lập trình hướng đối tượng nói chung). Các hàm thành phần có thể truy nhập đến các thành phần dữ liệu và các hàm thành phần khác trong lớp. Như trên đã nói, C++ cho phép hàm thành phần truy nhập tới các thành phần của các đối tượng cùng lớp, miễn là chúng được khai báo bên trong định nghĩa hàm (như là một đối tượng cục bộ hay một tham số hình thức của hàm thành phần). Phần tiếp sau sẽ có các ví dụ minh họa cho khả năng này.

Trong chương trình point.cpp, trong khai báo của lớp **point** có chứa các khai báo các hàm thành phần của lớp. Các khai báo này cũng tuân theo cú pháp khai báo cho các hàm bình thường. Định nghĩa của các hàm thì có thể đặt ở bên trong hay bên ngoài khai báo lớp. Khi định nghĩa hàm thành phần đặt trong khai

báo lốp (nếu hàm thành phần đơn giản, không chứa các cấu trúc lặp¹) không có gì khác so với định nghĩa của hàm thông thường. Chương trình point1.cpp sau đây là một cách viết khác của point.cpp trong đó hàm thành phần init() được định nghĩa ngay bên trong khai báo lớp.

Ví dụ 3.2

```
/*point1.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    /*khai báo các thành phần dữ liệu private*/
private:
    int x,y;
    /*khai báo các hàm thành phần public*/
public:
    /*Định nghĩa hàm thành phần bên trong khai báo lớp*/
    void init(int ox, int oy) {
        cout<<"Hàm thành phần init\n";
        x = ox; y = oy; /*x,y là các thành phần của đối tượng gọi hàm thành phần*/
    }
    void move(int dx, int dy);
    void display();
};

/*định nghĩa các hàm thành phần bên ngoài khai báo lớp*/
void point::move(int dx, int dy) {
    cout<<"Hàm thành phần move\n";
    x += dx; y += dy;
}

void point::display() {
    cout<<"Hàm thành phần display\n";
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
}
```

¹ Hàm thành phần định nghĩa trong khai báo lớp được chương trình dịch hiểu là hàm **inline**, nên không được quá phức tạp.

```

}

void main() {
    clrscr();
    point p;
    p.init(2,4); /*gọi hàm thành phần từ đối tượng*/
    p.display();
    p.move(1,2);
    p.display();
    getch();
}

```

```

Ham thanh phan init
Ham thanh phan display
Toa do: 2 4
Ham thanh phan move
Ham thanh phan display
Toa do: 3 6

```

Khi định nghĩa hàm thành phần ở ngoài lớp, dòng tiêu đề của hàm thành phần phải chứa tên của lớp có hàm là thành viên tiếp theo là toán tử định phạm vi “::”. Đó là cách để phân biệt hàm thành phần với các hàm tự do, đồng thời còn cho phép hai lớp khác nhau có thể có các hàm thành phần cùng tên.

Có thể đặt định nghĩa hàm thành phần trong cùng tập tin khai báo lớp hoặc trong một tập tin khác. Ví dụ sau đây sau đây là một cải biến khác từ `point.cpp`, trong đó ta đặt riêng khai báo lớp `point` trong một tệp tiêu đề. Tệp tiêu đề sẽ được tham chiếu tới trong tệp chương trình `point2.cpp` chứa định nghĩa các hàm thành phần của lớp `point`.

Ví dụ 3.3

Tệp tiêu đề

```

/*point.h*/
/* đây là tập tin tiêu đề khai báo lớp point được gộp vào tệp point2.cpp */
#ifndef point_h
#define point_h
#include <iostream.h>
class point {

```

```

/*khai báo các thành phần dữ liệu private*/
private:
    int x,y;
/*khai báo các hàm thành phần public*/
public:
/*Định nghĩa hàm thành phần bên trong khai báo lớp*/
    void init(int ox, int oy);
    void move(int dx, int dy);
    void display();
};

#endif

```

Tệp chương trình nguồn

```

/*point2.cpp*/
/*Tập tin chương trình, định nghĩa và sử dụng các hàm thành phần trong lớp point được
khai báo trong tập tin tiêu đề point.h */
#include "point.h"/*chèn định nghĩa lớp point vào chương trình*/
#include <conio.h>

/*định nghĩa các hàm thành phần bên ngoài khai báo lớp*/
void point::init(int ox, int oy) {
    cout<<"Hàm thành phần init\n";
    x = ox; y = oy; /*x,y là các thành phần của đối tượng gọi hàm thành phần*/
}

void point::move(int dx, int dy) {
    cout<<"Hàm thành phần move\n";
    x += dx; y += dy;
}

void point::display() {
    cout<<"Hàm thành phần display\n";
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
}

void main() {
    clrscr();
}

```

```
point p;  
p.init(2,4); /*gọi hàm thành phần từ đối tượng*/  
p.display();  
p.move(1,2);  
p.display();  
getch();  
}
```

Hàm thành phần init

Hàm thành phần display

Tọa độ: 2-4

Hàm thành phần move

Hàm thành phần display

Tọa độ: 3-6

2.1.4 Tham số ngầm định trong lời gọi hàm thành phần

Ở đây không nên nhầm lẫn khái niệm này với lời gọi hàm với tham số có giá trị ngầm định. Lời gọi hàm thành phần luôn có một và chỉ một tham số ngầm định là đối tượng thực hiện lời gọi hàm. Như thế các biến x, y trong định nghĩa của các hàm point::init(), point::display(), hay point::move() chính là các biến thể hiện của đối tượng dùng làm tham số ngầm định trong lời gọi hàm. Do vậy lời gọi hàm thành phần:

```
p.init(2,4)
```

sẽ gán 2 cho p.x còn p.y sẽ có giá trị 4.

Tất nhiên, theo nguyên tắc đóng gói, không gán trị cho các thành phần dữ liệu của đối tượng một cách trực tiếp.

```
p.x = 2;
```

```
p.y = 4;
```

Hơn nữa, không thể thực hiện lời gọi tới hàm thành phần nếu không chỉ rõ đối tượng được tham chiếu. Chỉ thi:

```
init(5,2);
```

trong hàm main sẽ có thể gây lỗi biên dịch nếu trong chương trình không có hàm tự do với tên init.

2.1.5 Phạm vi lớp

Phạm vi chỉ ra phần chương trình trong đó có thể truy xuất đến một đối tượng nào đó. Trong C có bốn kiểu phạm vi liên quan đến cách thức và vị trí khai báo biến: phạm vi khôi lệnh, phạm vi tệp, phạm vi chương trình và phạm vi hàm nguyên mẫu, trong đó thường dùng nhất là phạm vi toàn cục (tệp, chương trình) và phạm vi cục bộ (khôi lệnh, hàm). Mục đích của phạm vi là để kiểm soát việc truy xuất đến các biến/hàng/hàm.

Để kiểm soát truy nhập đến các thành phần (dữ liệu, hàm) của các lớp, C++ đưa ra khái niệm phạm vi lớp. Tất cả các thành phần của một lớp sẽ được coi là thuộc phạm vi lớp; trong định nghĩa hàm thành phần của lớp có thể tham chiếu đến bất kỳ một thành phần nào khác của cùng lớp đó. Tuân theo ý tưởng đóng gói, C++ coi tất cả các thành phần của một lớp có liên hệ với nhau. Ngoài ra, C++ còn cho phép mở rộng phạm vi lớp đến các lớp con cháu, bạn bè và họ hàng (Xem thêm chương 5 - Kế thừa và các mục tiếp sau để hiểu rõ hơn).

2.1.6 Từ khoá xác định thuộc tính truy xuất

Trong phần này ta nói tới vai trò của hai từ khoá **private** và **public** - dùng để xác định thuộc tính truy xuất của các thành phần lớp.

Trong định nghĩa của lớp ta có thể xác định khả năng truy xuất thành phần của một lớp nào đó từ bên ngoài phạm vi lớp. Trong lớp point có hai thành phần dữ liệu và ba thành phần hàm. Các thành phần dữ liệu được khai báo với nhãn là **private**, còn các hàm thành với nhãn **public**. **private** và **public** là các từ khoá xác định thuộc tính truy xuất. Mọi thành phần được liệt kê trong phần **public** đều có thể truy xuất trong bất kỳ hàm nào. Những thành phần được liệt kê trong phần **private** chỉ được truy xuất bên trong phạm vi lớp, bởi chúng thuộc sở hữu riêng của lớp, trong khi đó các thành phần **public** thuộc sở hữu chung của mọi thành phần trong chương trình.

Với khai báo lớp point ta thấy rằng các thành phần **private** được tính từ chỗ nó xuất hiện cho đến trước nhãn **public**. Trong lớp có thể có nhiều nhãn **private** và **public**. Mỗi nhãn này có phạm vi ảnh hưởng cho đến khi gặp một nhãn kế tiếp hoặc hết khai báo lớp. Xem chương trình tamgiac.cpp sau đây:

Ví dụ 3.4

```
/*tamgiac.cpp*/
#include <iostream.h>
#include <math.h>
#include <conio.h>
/*khai báo lớp tam giác*/
class tamgiac{
private:
    float a,b,c; /*độ dài ba cạnh*/
public:
    void nhap(); /*nhập vào độ dài ba cạnh*/
    void in(); /*in ra các thông tin liên quan đến tam giác*/
private:
    int loaitg(); /*cho biết kiểu của tam giác: 1-d,2-vc,3-c,4-v,5-t*/
    float dientich(); /*tính diện tích của tam giác*/
};

/*định nghĩa hàm thành phần*/
void tamgiac::nhap() {
    /*nhập vào ba cạnh của tam giác, có kiểm tra điều kiện*/
    do {
        cout<<"Canh a : "; cin>>a;
        cout<<"Canh b : "; cin>>b;
        cout<<"Canh c : "; cin>>c;
    }while(a+b<=c || b+c<=a || c+a<=b);
}

void tamgiac::in() {
    cout<<"Do dai ba canh :"<<a<<" "<<b<<" "<<c<<"\n";
    /* gọi hàm thành phần bên trong một hàm thành phần khác cùng lớp */
    cout<<"Dien tich tam giac : "<<dientich()<<"\n";
    switch(loaitg()) {
        case 1: cout<<"Tam giac deu\n";break;
```

```

case 2: cout<<"Tam giac vuong can\n";break;
case 3: cout<<"Tam giac can\n";break;
case 4: cout<<"Tam giac vuong\n";break;
default:cout<<"Tam giac thuong\n";break;
}
}

float tamgiac::dientich() {
    return (0.25*sqrt((a+b+c)*(a+b-c)*(a-b+c)*(-a+b+c)));
}

int tamgiac::loaitg() {
    if (a==b||b==c||c==a)
        if (a==b && b==c)
            return 1;
        else if (a*a==b*b+c*c||b*b==a*a+c*c||c*c==a*a+b*b)
            return 2;
        else return 3;
    else if (a*a==b*b+c*c||b*b==a*a+c*c||c*c==a*a+b*b)
        return 4;
    else return 5;
}

void main() {
    clrscr();
    tamgiac tg;
    tg.nhap();
    tg.in();
    getch();
}

```

Canh a : 3
 Canh b : 3
 Canh c : 3
 Do dai ba canh : 3 3 3
 Dien tich tam giac : 3.897114

Tam giác đều

Canh a : 3

Canh b : 4

Canh c : 5

Độ dài ba cạnh : 3 4 5

Diện tích tam giác : 6

Tam giác vuông

Các thành phần trong một lớp có thể được sắp xếp một cách hết sức tùy ý. Do đó có thể sắp xếp lại các khai báo hàm thành phần để cho các thành phần **private** ở trên, còn các thành phần **public** ở dưới trong khai báo lớp. Chẳng hạn có thể đưa ra một khai báo khác cho lớp tamgiac trong tamgiac.cpp như sau:

```
class tamgiac{
    private:
        float a,b,c; /*độ dài ba cạnh*/
        int loaitg(); /*cho biết kiểu của tam giác: 1-d,2-vc,3-c,4-v,5-t*/
        float dientich(); /*tính diện tích của tam giác*/
    public:
        void nhap(); /*nhập vào độ dài ba cạnh*/
        void in(); /*in ra các thông tin liên quan đến tam giác*/
};
```

Ngoài ra, còn có thể bỏ nhãn **private** đi vì C++ ngầm hiểu rằng các thành phần trước nhãn **public** đầu tiên là **private** (ở đây chúng ta tạm thời chưa bàn đến từ khoá **protected**). Tóm lại, khai báo “súc tích” nhất cho lớp tam giác như sau:

```
class tamgiac {
    float a,b,c; /*độ dài ba cạnh*/
    int loaitg(); /*cho biết kiểu của tam giác: 1-d,2-vc,3-c,4-v,5-t*/
    float dientich(); /*tính diện tích của tam giác*/
public:
    void nhap(); /*nhập vào độ dài ba cạnh*/
    void in(); /*in ra các thông tin liên quan đến tam giác*/
};
```

2.1.7 Gọi một hàm thành phần trong một hàm thành phần khác

Khi khai báo lớp, có thể gọi hàm thành phần từ một hàm thành phần khác trong cùng lớp đó. Khi muốn gọi một hàm tự do trùng tên và danh sách tham số ta phải sử dụng toán tử phạm vi “::”. Bạn đọc có thể kiểm nghiệm điều này bằng cách định nghĩa một hàm tự do tên `loaitg` và gọi nó trong định nghĩa của hàm `tamgiac::in()`.

Nhận xét

1. Nếu tất cả các thành phần của một lớp là **public**, lớp sẽ hoàn toàn tương đương với một cấu trúc, không có phạm vi lớp. C++ cũng cho phép khai báo các cấu trúc với các hàm thành phần. Hai khai báo sau là tương đương nhau:

<pre>struct point { int x, y; void init(int, int); void move(int, int); void display(); };</pre>	<pre>class point { public: int x, y; void init(int, int); void move(int, int); void display(); };</pre>
--	---

2. Ngoài **public** và **private**, còn có từ khoá **protected** (được bảo vệ) dùng để chỉ định trạng thái của các thành phần trong một lớp. Trong phạm vi của lớp hiện tại một thành phần **protected** có tính chất giống như thành phần **private**.

2.2 Khả năng của các hàm thành phần

2.2.1 Định nghĩa chồng các hàm thành phần.

Các hàm thành phần có thể có trùng tên nhưng phải khác nhau ở kiểu giá trị trả về, danh sách kiểu các tham số. Hàm thành phần được phép gọi tới các hàm thành phần khác, thậm chí trùng tên. Chương trình `point3.cpp` sau đây là một cải biến mới của `point.cpp`:

Ví dụ 3.5

```
/*point3.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    int x,y;
```

```
public:  
    /*định nghĩa chống các hàm thành phần init và display*/  
    void init();  
    void init (int);  
    void init (int,int);  
    void display();  
    void display(char *);  
};  
void point::init() {  
    x=y=0;  
}  
void point::init(int abs) {  
    x=abs;y=0;  
}  
void point::int(int abs,int ord) {  
    x=abs;  
    y=ord;  
}  
void point::display()  
{  
    cout<<"Toa do : "<<x<<" "<<y<<"\n";  
}  
void point::display(char *msg) {  
    cout<<msg;  
    display();  
}  
void main() {  
    clrscr();  
    point a; a.init();/*point::init()*/  
    a.display();/*point::display()*/  
    point b; b.init(5); /*point::init(int )*/  
    b.display("point b = ");/*point::display(char *)*/
```

```

point c; c.init(3,12); /*point::init(int,int)*/
c.display("Hello ----");
getch();
}

```

```

Toa do : 0 0
point b - Toa do : 5 0
Hello ----Toa do : 3 12

```

2.2.2 Các tham số với giá trị ngầm định

Giống như các hàm thông thường, lời gọi hàm thành phần có thể sử dụng giá trị ngầm định cho các tham số. Giá trị ngầm định này sẽ được khai báo trong định nghĩa hàm thành phần hay trong khai báo (trong khai báo lớp) của nó. Chương trình point4.cpp sau đây được cải tiến từ point3.cpp ngắn gọn hơn nhưng vẫn giữ được tất cả các khả năng như trong point3.cpp

Ví dụ 3.6

```

/*point4.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    int x,y;
public:
    void init (int=0,int=0);
    void display(char *="");
};
void point::int(int abs,int ord) {
    x=abs;
    y=ord;
}
void point::display(char *mesg) {
    cout<<mesg;
    display();
}
void main() {

```

```

clrscr();
point a; a.init(); /*a.init(0,0)*/
a.display(); /*a.display(□□);*/
point b; b.init(5); /*b.init(5,0)*/
b.display("point b - ");
point c; c.init(3,12); /*c.init(3,12)*/
c.display("Hello ----");
getch();
}

```

```

Toa do : 0 0
point b - Toa do : 5 0
Hello ----Toa do : 3 12

```

2.2.3 Sử dụng đối tượng như tham số của hàm thành phần

Ở đây đề cập đến khả năng mở rộng phạm vi lớp đối với các đối tượng “hàng”.

2.2.3.1 Truy nhập đến các thành phần private trong đối tượng

Hàm thành phần có quyền truy nhập đến các thành phần **private** của đối tượng gọi nó. Xem định nghĩa hàm thành phần `point::init()`:

```

void point::int(int abs,int ord)
{
    x=abs;
    y=ord;
}

```

2.2.3.2 Truy nhập đến các thành phần **private** trong các tham số là đối tượng truyền cho hàm thành phần.

Hàm thành phần có quyền truy nhập đến tất cả các thành phần **private** của các đối tượng, tham chiếu đối tượng hay con trả đối tượng có cùng kiểu lớp khi được dùng là tham số hình thức của nó.

```

class point {
    int x,y;
}

```

```

public:
...
/* Các đối tượng được truyền theo giá trị của chúng */
int coincide(point pt)
{
    return(x==pt.x && y==pt.y);
}
/* Các đối tượng được truyền bằng địa chỉ */
int coincide(point *pt)
{
    return(x==pt->x && y==pt->y);
}
/* Các đối tượng được truyền bằng tham chiếu */
int coincide(point &pt)
{
    return(x==pt.x && y==pt.y);
}
}

```

2.2.3.3 Dùng đối tượng như giá trị trả về của hàm thành phần hàm trong cùng lớp

Hàm thành phần có thể truy nhập đến các thành phần **private** của các đối tượng, con trỏ đối tượng, tham chiếu đối tượng định nghĩa bên trong nó.

```

class point
{
    int x,y;
public:
...
point symetry()
{
    point res;
    res.x=-x;res.y=-y;
    return res;
}
};

```

2.2.4 Con trỏ this

Từ khoá **this** trong định nghĩa của các hàm thành phần lớp dùng để xác định địa chỉ của đối tượng dùng làm tham số ngầm định cho hàm thành phần. Nói cách khác, con trỏ **this** tham chiếu đến đối tượng đang gọi hàm thành phần. Như vậy, có thể truy nhập đến các thành phần của đối tượng gọi hàm thành phần gián tiếp thông qua **this**. Sau đây là một cách viết khác cho định nghĩa của các hàm `point::coincide()` và `point::display()`:

```
int point::coincide(point pt)
{
    return(this->x==pt.x && this->y==pt.y);
}

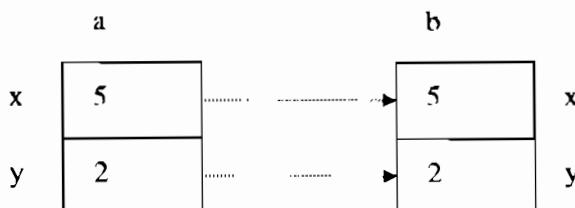
void point::display()
{
    cout<<"Dia chi : "<<this<<"Toa do : "<<x<<" "<<y<<"\n";
}
```

3. PHÉP GÁN CÁC ĐỐI TƯỢNG

Có thể thực hiện phép gán giữa hai đối tượng cùng kiểu. Chẳng hạn, với lớp `point` khai báo ở trên:

```
point a, b;
a.init(5,2);
b=a;
```

Về thực chất đó là việc sao chép giá trị các thành phần dữ liệu (`x, y`) từ đối tượng `a` sang đối tượng `b` tương ứng từng đối một (không kể đó là các thành phần **public** hay **private**).



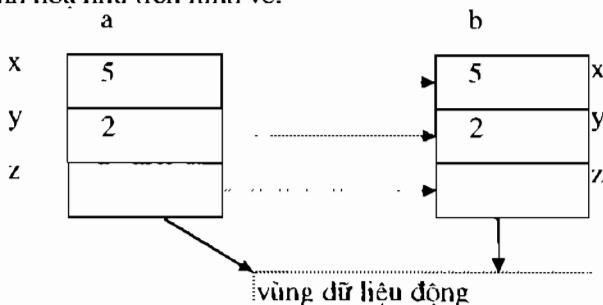
Chú ý

Khi các đối tượng trong phép gán chứa các thành phần dữ liệu động, việc sao chép lại không liên quan đến các vùng dữ liệu đó (Người ta nói rằng đó là sự “sao chép bề mặt”). Chẳng hạn, nếu hai đối tượng *a* và *b* cùng kiểu, có các thành phần dữ liệu *x*, *y*(tĩnh) và *z* là một con trỏ chỉ đến một vùng nhớ được cấp phát động.

Phép gán

a = b;

được minh họa như trên hình vẽ:



Điều này có thể ít nhiều gây khó khăn cho việc quản lý cấp phát động. Thứ nhất, vùng nhớ động trước đây trong *a* (nếu có) bây giờ không thể kiểm soát được nữa. Thứ hai, vùng nhớ động của *b* bây giờ sẽ được truy nhập bởi các hàm thành phần của cả *a* và *b* và như vậy tính “riêng tư” dữ liệu của các đối tượng đã bị vi phạm.

4. HÀM THIẾT LẬP (constructor) VÀ HÀM HỦY BỎ (destructor)

4.1 Hàm thiết lập

4.1.1 Chức năng của hàm thiết lập

Hàm thiết lập là một hàm thành phần đặc biệt không thể thiếu được trong một lớp. Nó được gọi tự động mỗi khi có một đối tượng được khai báo. Chức năng của hàm thiết lập là khởi tạo các giá trị thành phần dữ liệu của đối tượng, xin cấp phát bộ nhớ cho các thành phần dữ liệu động. Chương trình point5.cpp sau đây là một phiên bản mới của point.cpp trong đó thay thế hàm thành phần init bởi hàm thiết lập.

Ví dụ 3.7

```

/*point5.cpp*/
#include <iostream.h>
#include <conio.h>
/*định nghĩa lớp point*/
class point
{
    /*khai báo các thành phần dữ liệu*/
    int x;
    int y;
public:
    /*khai báo các thành phần hàm*/
    point(int ox,int oy) {x=ox;y=oy;}/*hàm thiết lập*/
    void move(int dx,int dy);
    void display();
};

void point::move(int dx,int dy)
{
    x+=dx;
    y+=dy;
}

void point::display()
{
    cout<<"Toa do : "<<x<<" "<<y<<"\n";
}

void main() {
    clrscr();
    point a(5,2); /*Sử dụng hàm thiết lập*/
    a.display();
    a.move(-2,4); a.display();
    point b.init(1,-1);b.display();
    clrscr();
}

```

Toa do : 5 2

Toa do : 3 6

Toa do : 1 -1

4.1.2 Một số đặc điểm quan trọng của hàm thiết lập

1. Hàm thiết lập có cùng tên với tên của lớp.
2. Hàm thiết lập phải có thuộc tính **public**.
3. Hàm thiết lập không có giá trị trả về. Và không cần khai báo void².
4. Có thể có nhiều hàm thiết lập trong cùng lớp (chỗng các hàm thiết lập).
5. Khi một lớp có nhiều hàm thiết lập, việc tạo các đối tượng phải kèm theo các tham số phù hợp với một trong các hàm thiết lập đã khai báo. Ví dụ:

```
/*định nghĩa lại lớp point*/
class point {
    int x,y;
public:
    point() {x=0;y=0;}
    point(int ox, int oy) {x=ox;y=oy;} /*hàm thiết lập có hai tham số*/
    void move(int,int);
    void display();
}
```

point a(1); /* Lỗi vì tham số không phù hợp với hàm thiết lập */

point b; /*Đúng, tham số phù hợp với hàm thiết lập không tham số*/

point c(2,3); /*Đúng, tham số phù hợp với hàm thiết lập thứ hai, có hai tham số*/

6. Hàm thiết lập có thể được khai báo với các tham số có giá trị ngầm định. Xét ví dụ sau:

```
/*Định nghĩa lại lớp point*/
```

²Sự giới hạn này là không thể tránh được vì hàm thiết lập thường được gọi vào lúc định nghĩa một đối tượng mới, mà lúc đó thì không có cách nào để lấy lại hoặc xem xét giá trị trả về của hàm thiết lập cả. Điều này có thể trở thành một vấn đề khi hàm thiết lập cần phải trả về một trạng thái lỗi. Giải quyết vấn đề này người ta dùng đến khả năng kiểm soát lỗi sẽ được trình bày trong phần lục 2.

```
class point {  
    int x,y;  
public:  
    point(int ox, int oy = 0) {x=ox;y=oy;} /*hàm thiết lập có hai tham số*/  
    void move(int,int);  
    void display();  
};  
point a; /*Lỗi: không có hàm thiết lập ngầm định hoặc hàm thiết lập với các tham số có  
giá trị ngầm định*/  
point b(1); //Đối số thứ hai nhận giá trị 0  
point c(2,3); //Đúng
```

Nhận xét

Trong ví dụ trên, chỉ thi:

```
point b(1);
```

có thể được thay thế bằng cách viết khác như sau:

```
point b=1;
```

Cách viết thứ hai hàm ý rằng đã có chuyển kiểu ngầm định từ số nguyên 1 thành đối tượng kiểu point. Chúng ta sẽ đề cập vấn đề này một cách đầy đủ hơn trong chương 4.

4.1.3 Hàm thiết lập ngầm định

Hàm thiết lập ngầm định do chương trình dịch cung cấp khi trong khai báo lớp không có định nghĩa hàm thiết lập nào. Lớp point định nghĩa trong chương trình point.cpp là một ví dụ trong đó chương trình biên dịch tự bổ sung một hàm thiết lập ngầm định cho khai báo lớp. Dĩ nhiên hàm thiết lập ngầm định đó không thực hiện bất cứ nhiệm vụ nào ngoài việc “lắp chỗ trống”.

Đôi khi người ta cũng gọi hàm thiết lập không có tham số do người sử dụng định nghĩa là hàm thiết lập ngầm định.

Cần phải có hàm thiết lập ngầm định khi cần khai báo mảng các đối tượng. Ví dụ, trong khai báo:

```
X a[10];
```

bắt buộc trong lớp X phải có một hàm thiết lập ngầm định.

Ta minh họa nhận xét này bằng hai ví dụ sau:

a. Trong trường hợp thứ nhất không dùng hàm thiết lập không tham số:

Ví dụ 3.8

```
/*point6.cpp*/
#include <iostream.h>
/*định nghĩa lớp point*/
class point {
    /*khai báo các thành phần dữ liệu*/
    int x;
    int y;
public:
    /*khai báo các hàm thành phần */
    point(int ox,int oy) {x=ox;y=oy;}
    void move(int dx,int dy) ;
    void display();
};

/*phân biệt các hàm thành phần với các hàm thông thường nhờ tên lớp và toán tử ::*/
void point::move(int dx,int dy) {
    x+=dx;
    y+=dy;
}

void point::display() {
    cout<<"Toa do : "<<x<< " "<<y<<"\n";
}

void main() {
    point a(5,2); //OK
    a.display();
    a.move(-2,4); a.display();
    point b[10];//lỗi vì không cung cấp thông số cần thiết cho hàm thiết lập
}
```

Trong chương trình point6.cpp, lỗi xảy ra vì ta muốn tạo ta mười đối tượng nhưng không cung cấp đủ các tham số cho hàm thiết lập có như đã định nghĩa (ở đây ta chưa đề cập đến hàm thiết lập sao chép ngầm định, nó sẽ được trình bày trong phần sau). Giải quyết tình huống này bằng hai cách: hoặc bỏ luôn hàm thiết lập hai tham số trong khai báo lớp nhưng khi đó, khai báo của đối tượng a sẽ không còn đúng nữa. Dó vậy ta thường sử dụng giải pháp định nghĩa thêm một hàm thiết lập không tham số:

b. Định nghĩa hàm thiết lập không tham số

Ví dụ 3.9

```
/*point7.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    /*khai báo các thành phần dữ liệu*/
    int x;
    int y;
public:
    /*khai báo các hàm thành phần*/
    point(int ox,int oy) {x=ox;y=oy;}
    /*định nghĩa thêm hàm thiết lập không tham số*/
    point() {x = 0; y = 0;}
    void move(int dx,int dy) ;
    void display();
}
/*phân biệt các thành phần hàm với các hàm thông thường nhờ tên lớp và toán tử ::*/
void point::move(int dx,int dy) {
    x+=dx;
    y+=dy;
}
void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<"\n";
}
void main() {
    clrscr();
```

```

point a(5,2); //OK
a.display();
a.move(-2,4); a.display();
point b[10];/*Hết lỗi vì hàm thiết lập không tham số được gọi để tạo các đối tượng
            thành phần của */
getch();
}

```

Còn một giải pháp khác không cần định nghĩa thêm hàm thiết lập không tham số. Khi đó cần khai báo giá trị ngầm định cho các tham số của hàm thiết lập hai tham số.

Ví dụ 3.10

```

/*point8.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
/*khai báo các thành phần dữ liệu*/
    int x;
    int y;
public:
/*khai báo các hàm thành phần */
    point(int ox = 1,int oy =0) {x=ox;y=oy;}
    void move(int dx,int dy) ;
    void display();
};

void point::move(int dx,int dy) {
    x+=dx;
    y+=dy;
}

void point::display()  {
    cout<<"Toa do : "<<x<<" "<<y<<"\n";
}

```

```

void main() {
    clrscr();
    point a(5,2); //OK
    a.display();
    a.move(-2,4); a.display();
    point b[10]; /*Trong trường hợp này các đối tượng thành phần của b được tạo ra nhờ
                  hàm thiết lập được gọi với hai tham số có giá trị ngầm định là 1 và 0.*/
    getch();
}

```

4.1.4 Con trỏ đối tượng

Con trỏ đối tượng được khai báo như sau:

```
point *ptr;
```

Con trỏ đối tượng có thể nhận giá trị là địa chỉ của các đối tượng có cùng kiểu lớp:

```
ptr = &a;
```

Khi đó có thể gọi các hàm thành phần của lớp point thông qua con trỏ như sau:

```
ptr->display();
ptr->move(-2,3);
```

Khi dùng toán tử **new** cấp phát một đối tượng động, hàm thiết lập cũng được gọi, do vậy cần cung cấp danh sách các tham số. Chẳng hạn, giả sử trong lớp point có một hàm thiết lập hai tham số, khi đó câu lệnh sau:

```
ptr = new point(3,2);
```

sẽ xin cấp phát một đối tượng động với hai thành phần x và y nhận giá trị tương ứng là 2 và 3. Kết quả này được minh chứng qua lời gọi hàm:

```
ptr->display();
```

Tọa độ : 2 3

Ta xét chương trình ví dụ sau:

Ví dụ 3.11

```
/*point9.cpp*/
#include <conio.h>
#include <iostream.h>
class point
{
    /*khai báo các thành phần dữ liệu*/
    int x;
    int y;
public:
    /*khai báo các hàm thành phần */
    point(int ox = 1,int oy =0) {x=ox;y=oy;}
    void move(int dx,int dy) ;
    void display();
};

void point::move(int dx,int dy){
    x+=dx;
    y+=dy;
}

void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<"\n";
}

void main() {
    clrscr();
    point a(5,2);
    a.display();
    point *ptr = &a;
    ptr->display();
    a.move(-2,4);
    ptr->display();
}
```

```

ptr->move(2,-4);
a.display();

ptr = new point; /*Gọi hàm thiết lập với hai tham số có giá trị ngầm định*/
ptr->display();
delete ptr;

ptr = new point(3); /*Tham số thứ hai của hàm thiết lập có giá trị ngầm định*/
ptr->display();
delete ptr;

ptr = new point(3,5); /*Hàm thiết lập được cung cấp hai tham số tương ứng*/
ptr->display();
delete ptr;
point b[10];
getch();
}

```

```

Toa do : 5 2
Toa do : 5 2
Toa do : 3 6
Toa do : 5 2
Toa do : 1 0
Toa do : 3 0
Toa do : 3 5

```

4.1.5 Khai báo tham chiếu đối tượng

Khi đối tượng là nội dung một biến có kiểu lớp, ta có thể gán cho nó các “bí danh”; nghĩa là có thể khai báo các tham chiếu đến chúng. Một tham chiếu đối tượng chỉ có ý nghĩa khi tham chiếu tới một đối tượng nào đó đã được khai báo trước đó. Chẳng hạn:

```

point a(2,5);
point &ra=a;
a.display();
ra.display();
ra.move(2,3);
a.display();

```

```
Toa do : 2 5
```

```
Toa do : 2 5
```

```
Toa do : 4 8
```

4.2 Hàm huỷ bỏ

4.2.1 Chức năng của hàm huỷ bỏ

Ngược với hàm thiết lập, hàm huỷ bỏ được gọi khi đối tượng tương ứng bị xoá khỏi bộ nhớ. Ta xét chương trình ví dụ sau:

Ví dụ 3.12

```
/*test.cpp*/
#include <iostream.h>
#include <conio.h>
int line=1;
class test {
public:
    int num;
    test(int);
    ~test();
};
test::test(int n) {
    num = n;
    cout<<line++<<".";;
    cout<<"++ Gọi ham thiet lap voi num = "<<num<<"\n";
}
test::~test() {
    cout<<line++<<".";;
    cout<<"-- Gọi ham huy bo voi num = "<<num<<"\n";
}
void main() {
    clrscr();
    void fct(int);
    test a(1);
```

```

for(int i=1; i<= 2; i++) fct(i); // Lỗi này sẽ báo lỗi mà sau đó xuất hiện sau
}

```

Lỗi này xảy ra do lỗi về cách định nghĩa hàm fct(). Khi ta gọi fct(2) và fct(4) sau khi đã định nghĩa fct(p) với p là một biến số. Khi ta gọi fct(2) và fct(4) sau khi đã định nghĩa fct(p) với p là một biến số. Khi ta gọi fct(2) và fct(4) sau khi đã định nghĩa fct(p) với p là một biến số. Khi ta gọi fct(2) và fct(4) sau khi đã định nghĩa fct(p) với p là một biến số. Khi ta gọi fct(2) và fct(4) sau khi đã định nghĩa fct(p) với p là một biến số. Khi ta gọi fct(2) và fct(4) sau khi đã định nghĩa fct(p) với p là một biến số.

1. ++ Gọi hàm thiết lập với num = 1
2. ++ Gọi hàm thiết lập với num = 2
3. -- Gọi hàm hủy bỏ với num = 2
4. ++ Gọi hàm thiết lập num = 4
5. -- Gọi hàm hủy bỏ với num = 4
6. -- Gọi hàm hủy bỏ với num = 1

Ta lý giải như sau: trong chương trình chính, dòng thứ nhất tạo ra đối tượng a có kiểu lớp test, do đó có dòng thông báo số 1. Vòng lặp **for** hai lần gọi tới hàm fct(). Mỗi lời gọi hàm fct() kéo theo việc khai báo một đối tượng cục bộ x trong hàm. Vì là đối tượng cục bộ bên trong hàm fct(), nên x bị xoá khỏi vùng bộ nhớ ngăn xếp (dùng để cấp phát cho các biến cục bộ khi gọi hàm) khi kết thúc thực hiện hàm. Do đó, mỗi lời gọi tới fct() sinh ra một cặp dòng thông báo, tương ứng với lời gọi hàm thiết lập, hàm huỷ bỏ (các dòng thông báo 2, 3, 4, 5 tương ứng). Cuối cùng, khi hàm main() kết thúc thực hiện, đối tượng a được giải phóng, hàm huỷ bỏ đối với a sẽ cho ra dòng thông báo thứ 6.

4.2.2 Một số qui định đối với hàm huỷ bỏ

1. Tên của hàm huỷ bỏ bắt đầu bằng dấu ~ theo sau là tên của lớp tương ứng. Chẳng hạn lớp test thì sẽ hàm huỷ bỏ tên là ~test.
2. Hàm huỷ bỏ phải có thuộc tính **public**
3. Nói chung hàm huỷ bỏ không có tham số, mỗi lớp chỉ có một hàm huỷ bỏ (Trong khi đó có thể có nhiều các hàm thiết lập).
4. Khi không định nghĩa hàm huỷ bỏ, chương trình dịch tự động sản sinh một hàm như vậy (hàm huỷ bỏ ngầm định), hàm này không làm gì ngoài việc “lắp chỗ trống”. Đối với các lớp không có khai báo các thành phần bộ nhớ động, có thể dùng hàm huỷ bỏ ngầm định. Trái lại, phải khai báo hàm huỷ bỏ tường minh để đảm bảo quản lý tốt việc giải phóng bộ nhớ động do các đối tượng chiếm giữ chiếm giữ khi chừng hết thời gian làm việc.
5. Giống như hàm thiết lập, hàm huỷ bỏ không có giá trị trả về.

4.3 Sự cần thiết của các hàm thiết lập và huỷ bỏ -lớp vector trong không gian n chiều

Trên thực tế, với các lớp không có các thành phần dữ liệu động chỉ cần sử dụng hàm thiết lập và huỷ bỏ ngầm định là đủ. Hàm thiết lập và huỷ bỏ do người lập trình tạo ra rất cần thiết khi các lớp chứa các thành phần dữ liệu động. Khi tạo đối tượng hàm thiết lập đã xin cấp phát một khối bộ nhớ động, do đó hàm huỷ bỏ phải giải phóng vùng nhớ đã được cấp phát trước đó. Ví dụ sau đây minh họa vai trò của hàm huỷ bỏ trong trường hợp lớp có các thành phần cấp phát động.

Ví dụ 3.13

```
/*vector.cpp*/
#include <iostream.h>
#include <conio.h>
class vector {
    int n; //số chiều
    float *v; //vùng nhớ toạ độ
public:
    vector(); //Hàm thiết lập không tham số
    vector(int size); //Hàm thiết lập một tham số
    vector(int size, float *a);
    ~vector(); //Hàm huỷ bỏ, luôn luôn không có tham số
    void display();
};

vector::vector() {
    int i;
    cout<<"Tạo đối tượng tại "<<this<<endl;
    cout<<"Số chiều :">>n;
    v= new float [n];
    cout<<"Xin cấp phát vùng nhớ "<<n<<" số thực tại "<<v<<endl;
    for(i=0;i<n;i++) {
        cout<<"Toạ độ thứ "<<i+1<<" : ";
        cin>>v[i];
    }
}

vector::vector(int size) {
```

```
int i;
cout<<"Su dung ham thiet lap 1 tham so\n";
cout<<"Tao doi tuong tai "<<this<<endl;
n=size;
cout<<"So chieu :"<<size<<endl;
v= new float [n];
cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
}
}

vector::vector(int size, float *a ) {
int i;
cout<<"Su dung ham thiet lap 2 tham so\n";
cout<<"Tao doi tuong tai "<<this<<endl;
n=size;
cout<<"So chieu :"<<n<<endl;
v= new float [n];
cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
for(i=0;i<n;i++)
    v[i] = a[i];
}

vector::~vector() {
    cout<<"Giai phong "<<v<<"cua doi tuong tai"<<this<<endl;
    delete v;
}

//Hiển thị kết quả
void vector::display() {
int i;
cout<<"Doi tuong tai :"<<this<<endl;
cout<<"So chieu :"<<n<<endl;
```

```

for(i=0;i<n;i++)
    cout <<v[i] <<" ";
cout <<"\n";
}
void main() {
    clrscr();
    vector s1;
    s1.display();
    vector s2(4);
    s2.display();
    float a[3]={1,2,3};
    vector s3(3,a);
    s3.display();
    getch();
}

```

Tao doi tuong tai 0xffff2

So chieu :3

Xin cap phat vung bo nho 3 so thuc tai0x13cc

Toa do thu 1 : 2

Toa do thu 2 : 3

Toa do thu 3 : 2

Doi tuong tai :0xffff2

So chieu :3

2 3 2

Sử dụng hàm thiết lập 1 tham số

Tao doi tuong tai 0xffee

So chieu :4

Xin cap phat vung bo nho 4 so thuc tai0x13dc

Toa do thu 1 : 3

Toa do thu 2 : 2

Toa do thu 3 : 3

Toa do thu 4 : 2

Doi tuong tai :0xffee

```

So chieu :4
3 2 3 2

Su dung ham thiet lap 2 tham so

Tao doi tuong tai 0xffffea
So chieu :3

Xin cap phat vung bo nho 3 so thuc tai0x13f0
Doi tuong tai :0xffffea
So chieu :3
1 2 3

Doi tuong tai :0xffff2
So chieu :3
2 3 2

Giai phong 0x13f0cua doi tuong tai0xffffea
Giai phong 0x13dccua doi tuong tai0xffffee
Giai phong 0x13cccua doi tuong tai0xffff2

```

Chú ý

Không được lẫn lộn giữa cấp phát bộ nhớ động trong hàm thành phần của đối tượng (thông thường là hàm thiết lập) với việc cấp phát động cho một đối tượng. Khi cấp phát bộ nhớ động trong hàm thành phần, ta chỉ cần gán địa chỉ của biến mà không cần gán giá trị ban đầu. Khi cấp phát động cho một đối tượng, ta cần gán giá trị ban đầu và sau đó mới gán giá trị ban đầu.

4.4 Hàm thiết lập sao chép (COPY CONSTRUCTOR)**4.4.1 Các tình huống sử dụng hàm thiết lập sao chép**

Xét các chỉ thị khai báo và khởi tạo giá trị cho một biến nguyên:

```

int p;
int x = p;

```

Chỉ thị thứ hai khai báo một biến nguyên x và gán cho nó giá trị của biến nguyên p. Tương tự, ta cũng có thể khai báo một đối tượng và gán cho nó nội dung của một đối tượng cùng lớp đã tồn tại trước đó. Chẳng hạn:

```

point p(2, 3); /*giai thiet lop point co ham thiet lap hai tham so*/
point q = p;

```

Đi nhiên hai đối tượng, mới q và cũ p có cùng nội dung. Khi một đối tượng được tạo ra (khai báo) thì một hàm thiết lập của lớp tương ứng sẽ được gọi. Hàm

thiết lập được gọi khi khai báo và khởi tạo nội dung một đối tượng thông qua một đối tượng khác, gọi là hàm thiết lập sao chép. Nhiệm vụ của hàm thiết lập sao chép là tạo ra một đối tượng giống hệt một đối tượng đã có. Thoạt nhìn hàm thiết lập sao chép có vẻ thực hiện các công việc giống như phép gán, nhưng nếu để ý sẽ thấy giữa chúng có chút ít khác biệt; phép gán thực hiện việc sao chép nội dung từ đối tượng này sang đối tượng khác, do vậy cả hai đối tượng trong phép gán đều đã tồn tại:

```
point p(2, 3); // giả thiết lớp point có hàm thiết lập hai tham số
point q; // giả thiết lớp point có hàm thiết lập không tham số
q = p;
```

Ngược lại, hàm thiết lập thực hiện đồng thời hai nhiệm vụ: tạo đối tượng và sao chép nội dung từ một đối tượng đã có sang đối tượng mới tạo ra đó.

Ngoài tình huống trên đây, còn có hai trường hợp cần dùng hàm thiết lập sao chép: truyền đối tượng cho hàm bằng tham trị hoặc hàm trả về một đối tượng nhằm tạo một đối tượng giống hệt một đối tượng cùng lớp đã có trước đó. Trong phần sau chúng ta sẽ có ví dụ minh họa cho các trình bày này.

4.4.2 Hàm thiết lập sao chép ngầm định

Giống như hàm thiết lập ngầm định (hàm thiết lập không tham số), nếu không được mô tả tường minh, sẽ có một hàm thiết lập sao chép ngầm định do chương trình dịch cung cấp nhằm đảm bảo tính đúng đắn của chương trình trong các tình huống cần đến hàm thiết lập. Như vậy, trong khai báo của một lớp có ít nhất hai hàm thiết lập ngầm định: hàm thiết lập ngầm định và hàm thiết lập sao chép ngầm định.

Do là một hàm được tạo ra tự động nên hàm thiết lập sao chép ngầm định cũng chỉ thực hiện những thao tác tối thiểu ("ngầm định"): tạo giá trị của các thuộc tính trong đối tượng mới bằng các giá trị của các thuộc tính tương ứng trong đối tượng cũ. Bạn đọc có thể xem lại phần 3 của chương để hiểu rõ hơn. Nói chung, với các lớp không khai báo các thành phần dữ liệu động thì chỉ cần dùng hàm thiết lập sao chép ngầm định là đủ. Vấn đề sẽ khác đi khi cần đến các thao tác quản lý bộ nhớ động trong các đối tượng. Trong trường hợp này không được dùng hàm thiết lập sao chép ngầm định mà phải gọi hàm thiết lập sao chép tường minh.

4.4.3 Khai báo và định nghĩa hàm thiết lập sao chép tường minh

Dạng của hàm thiết lập sao chép

Xét các đối tượng thuộc lớp point. Câu lệnh

```
point q=p;
```

sẽ gọi đến hàm thiết lập sao chép.

Như nhận xét trong phần trên ta có thể viết theo cách khác như sau:

point q(p);

Từ cách viết trên có thể cho rằng dạng của hàm thiết lập sao chép cho lớp point có thể là:

point (point);

hoặc

point (point &);

Ta nhận thấy dạng thứ nhất không dùng được vì việc gọi nó đòi hỏi phải truyền cho hàm một đối tượng như một tham trị, do đó gây ra **đè quy vô hạn** lần.

Với dạng thứ hai ta đã thiết lập một tham chiếu tới đối tượng như một tham số hình thức truyền cho hàm, nên có thể chấp nhận được.

Dạng khai báo của hàm thiết lập là:

point (point &); hoặc point(const point &);

trong đó từ khoá **const** trong khai báo tham số hình thức chỉ nhằm ngăn cấm mọi thay đổi nội dung của tham số truyền cho hàm.

Chương trình point10.cpp sau đây bổ sung thêm hàm thiết lập sao chép vào lớp point.

Ví dụ 3.14

```
/*point10.cpp*/
#include <conio.h>
#include <iostream.h>
/*Khinh nghĩa lớp point*/
class point {
    /*Khai báo các thành phần dữ liệu*/
    int x;
    int y;
public:
    /*Khai báo các thành phần hàm*/
    point(int ox = 1,int oy =0) {
        cout<<"Tao doi tuong : "<<this<<endl;
```

```

cout<<"Dung ham thiet lap hai tham so\n";
x=ox; y=oy;
}

/*Hàm thiết lập sao chép*/
point(point &p) {
    cout<<"Tao doi tuong : "<<this<<endl;
    cout<<"Dung ham thiet lap sao chep\n";
    x = p.x; y = p.y;
}

void move(int dx, int dy) {
    x+=dx; y+=dy;
}

void display();
};

void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<"\n";
}

point fct(point a) {
    point b=a;
    b.move(2,3);
    return b;
}

void main(){
    clrscr();
    point a(5,2);
    a.display();
    point b=fct(a);
    b.display();
    getch();
}

```

Tao doi tuong : 0xffff2

Dung ham thiet lap hai tham so

```
Tao do : 5 2
Tao doi tuong : 0xffffea
Dung ham thiet lap sao chep
Tao doi tuong : 0xffffde
Dung ham thiet lap sao chep
Tao doi tuong : 0xffffee
Dung ham thiet lap sao chep
Tao do : 7 5
```

4.4.4 Hàm thiết lập sao chép cho lớp vector

Chương trình ví dụ sau giới thiệu cách định nghĩa hàm thiết lập khi đối tượng có các thành phần dữ liệu động.

Ví dụ 3.15

```
/*vector2.cpp*/
#include <iostream.h>
#include <conio.h>
class vector {
    int n; //số chiều của vector
    float *v; //vùng nhớ chứa các tọa độ
public:
    vector();
    vector(int size);
    vector(int size, float *a);
    vector(vector &); //hàm thiết lập sao chép
    ~vector();
    void display();
};
vector::vector() {
    int i;
    cout<<"Tao doi tuong tai "<<this<<endl;
    cout<<"So chieu :">>n;
    v= new float [n];
}
```

```

cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
}
}

vector::vector(int size) {
    int i;
    cout<<"Su dung ham thiet lap 1 tham so\n";
    cout<<"Tao doi tuong tai "<<this<<endl;
    n=size;
    cout<<"So chieu :"<<size<<endl;
    v= new float [n];
    cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
    for(i=0;i<n;i++) {
        cout<<"Toa do thu "<<i+1<<" : ";
        cin>>v[i];
    }
}

vector::vector(int size,float *a ) {
    int i;
    cout<<"Su dung ham thiet lap 2 tham so\n";
    cout<<"Tao doi tuong tai "<<this<<endl;
    n=size;
    cout<<"So chieu :"<<n<<endl;
    v= new float [n];
    cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
    for(i=0;i<n;i++)
        v[i] = a[i];
}

vector::vector(vector &b) {
    int i;

```

```

cout<<"Su dung ham thiet lap sao chep\n";
cout<<"Tao doi tuong tai "<<this<<endl;
/*xin cap phat mot vung nhor dong bang kich thuoc co trong doi tuong cu*/
v= new float [n=b.n];
cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
for(i=0;i<n;i++)
/*gan noi dung vung nhor dong cua doi tuong cu sang doi tuong moi*/
v[i] = b.v[i];
}

vector::~vector() {
    cout<<"Giai phong "<<v<<"cua doi tuong tai"<<this<<endl;
    delete v;
}

//hiển thị kết quả
void vector::display() {
    int i;
    cout<<"Doi tuong tai :"<<this<<endl;
    cout<<"So chieu :"<<n<<endl;
    for(i=0;i<n;i++) cout <<v[i] << " ";
    cout <<"\n";
}

void main() {
    clrscr();
    vector s1;//goi ham thiet lap khong tham so
    s1.display();
    vector s2(4); //4 giá trị
    s2.display();
    float a[3]={1,2,3};
    vector s3(3,a);
    s3.display();
    vector s4 = s1;//ham thiet lap sao chep
    s4.display();
}

```

```

getch();
}

Tao doi tuong tai 0xffff2
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x142c
Toa do thu 1 : 1
Toa do thu 2 : 2
Toa do thu 3 : 3
Doi tuong tai :0xffff2
So chieu :3
1 2 3
Su dung ham thiet lap 1 tham so
Tao doi tuong tai 0xffee
So chieu :4
Xin cap phat vung bo nho 4 so thuc tai0x143c
Toa do thu 1 :
Su dung ham thiet lap 1 tham so
Tao doi tuong tai 0xffee
So chieu :4
Xin cap phat vung bo nho 4 so thuc tai0x143c
Toa do thu 1 : 2
Toa do thu 2 : 3
Toa do thu 3 : 4
Toa do thu 4 : 5
Doi tuong tai :0xffee
So chieu :4
2 3 4 5
Su dung ham thiet lap 2 tham so
Tao doi tuong tai 0xffea
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x1450
Doi tuong tai :0xffea

```

```

So chieu :3
1 2 3
Su dung ham thiet lap sac chep
Tao doi tuong tai 0xffe6
Xin cap phat vung bo nho 3 so thuc tai0x1460
Doi tuong tai :0xffe6
So chieu :3
1 2 3
Giai phong 0x1460cua doi tuong tai0xffe6
Giai phong 0x1450cua doi tuong tai0xffea
Giai phong 0x143ccua doi tuong tai0xffee
Giai phong 0x142ccua doi tuong tai0xffff2

```

5. CÁC THÀNH PHẦN TĨNH (static)

5.1 Thành phần dữ liệu static

Thông thường, trong cùng một chương trình các đối tượng thuộc cùng một lớp chỉ sở hữu các thành phần dữ liệu của riêng nó. Ví dụ, nếu chúng ta định nghĩa lớp exple1 bằng:

```

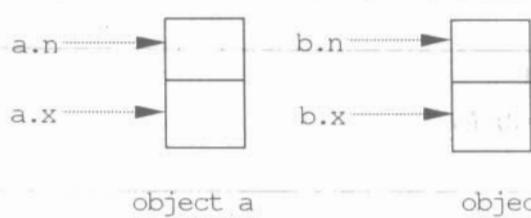
class exple1
{
    int n;
    float x;
};

khai báo :

```

```
exple1 a,b;
```

sẽ tạo ra hai đối tượng a, b sở hữu riêng biệt hai vùng dữ liệu khác nhau như hình vẽ:



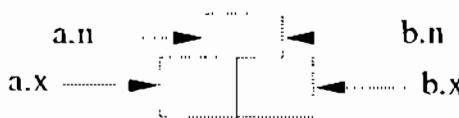
Có thể cho phép nhiều đối tượng cùng chia sẻ dữ liệu bằng cách đặt từ khoá **static** trước khai báo thành phần dữ liệu tương ứng. Ví dụ, nếu ta định nghĩa lớp **exple2** bằng:

```
class exple2
{
    static int n;
    float x;
    ...
};
```

thì khai báo

```
exple2 a,b;
```

tạo ra hai đối tượng có chung thành phần n:



Như vậy, việc chỉ định **static** đối với một thành phần dữ liệu có ý nghĩa là trong toàn bộ lớp, chỉ có một thể hiện duy nhất của thành phần đó. Thành phần **static** được dùng chung cho tất cả các đối tượng của lớp đó và do đó vẫn chiếm giữ vùng nhớ ngay cả khi không khai báo bất kỳ đối tượng nào. Có thể nói rằng các thành phần dữ liệu tĩnh giống như các biến toàn cục trong phạm vi lớp. Các phần tiếp sau sẽ làm nổi bật nhận xét này.

5.2 Khởi tạo các thành phần dữ liệu tĩnh

Các thành phần dữ liệu **static** chỉ có một phiên bản trong tất cả các đối tượng. Như vậy không thể khởi tạo chúng bằng các hàm thiết lập của một lớp.

Cũng không thể khởi tạo lúc khai báo các thành phần dữ liệu **static** như trong ví dụ sau:

```
class exple2{
    static int n=2;//lỗi
};
```

Một thành phần dữ liệu **static** phải được khởi tạo một cách tường minh bên ngoài khai báo lớp bằng một chỉ thị như sau:

```
int exple2::n=5;
```

Trong C++ việc khởi tạo giá trị như thế này không vi phạm tính riêng tư của các đối tượng. Chú ý rằng cần phải có tên lớp và toán tử phạm vi để chỉ định các thành phần của lớp được khởi tạo.

Ngoài ra, khác với các biến toàn cục thông thường, các thành phần dữ liệu **static** không được khởi tạo ngầm định là 0. Chương trình counter.cpp sau đây minh họa cách sử dụng và thao tác với thành phần dữ liệu **static**, dùng để đếm số đối tượng hiện đang được sử dụng:

Ví dụ 3.16

```
/*counter.cpp*/
#include <iostream.h>
#include <conio.h>
class counter {
    static int count; //đếm số đối tượng được tạo ra
public :
    counter ();
    ~counter ();
};

int counter::count=0;//khởi tạo giá trị cho thành phần static
//hàm thiết lập
counter:: counter () {
    cout<<"++Tao : bay gio co "<<++count<<" doi tuong\n";
}
counter:: ~counter () {
    cout<<"--Xoa : bay gio con "<<--count<<" doi tuong\n";
}
void main() {
    clrscr();
    void fct();
    counter a;
```

```

    fct();
    counter b;
}
}

void fct() {
    counter u,v;
}

```

```

++Tao : bay gio co 1 doi tuong
++Tao : bay gio co 2 doi tuong
++Tao : bay gio co 3 doi tuong
--Xoa : bay gio con 2 doi tuong
--Xoa : bay gio con 1 doi tuong
++Tao : bay gio co 2 doi tuong
--Xoa : bay gio con 1 doi tuong
--Xoa : bay gio con 0 doi tuong

```

Nhận xét

1. Thành phần dữ liệu tĩnh có thể là **private** hay **public**.
2. Trong C thuật ngữ **static** có nghĩa là: "lớp lưu trữ cố định" hay có phạm vi giới hạn bởi file nguồn. Trong C++, các thành phần dữ liệu **static** còn có thêm ý nghĩa khác: "không phụ thuộc vào bất kỳ thể hiện nào của lớp".

Trong phần sau chúng ta sẽ đề cập đến các hàm thành phần **static**.

5.3 Các hàm thành phần static

Một hàm thành phần được khai báo bắt đầu với từ khoá **static** được gọi là **hàm thành phần static**, hàm thành phần **static** cũng độc lập với bất kỳ đối tượng nào của lớp. Nói cách khác hàm thành phần **static** không có tham số ngầm định. Vì không đòi hỏi đối tượng làm tham số ngầm định nên không thể sử dụng con trỏ **this** trong định nghĩa của hàm thành phần **static**. Các hàm thành phần **static** của một lớp có thể được gọi, cho dù có khai báo các đối tượng của lớp đó hay không.

Cú pháp gọi hàm trong trường hợp này là:

<tên lớp>::<tên hàm thành phần>(<các tham số nếu có>)

Tất nhiên vẫn có thể gọi các hàm thành phần **static** thông qua các đối tượng. Tuy nhiên cách gọi thông qua tên lớp trực quan hơn vì phản ánh được bản chất của hàm thành phần **static**.³

Thông thường, các hàm thành phần **static** được dùng để xử lý chung trên tất cả các đối tượng của lớp, chẳng hạn để hiện thị các thông tin liên quan đến các thành phần dữ liệu **static**.

Chương trình counter1.cpp sau đây được cải tiến từ counter.cpp bằng cách thêm một hàm thành phần **static** trong lớp counter.

Ví dụ 3.17

```
/*counter1.cpp*/
#include <iostream.h>
#include <conio.h>
class counter {
    static int count; //đếm số đối tượng được tạo ra
public :
    counter ();
    ~ counter ();
    static void counter_display();
};

int counter::count=0; //khởi tạo giá trị cho thành phần static
void counter::counter_display() {
    cout<<"Hien dang co "<<count<<" doi tuong \n";
}
counter:: counter () {
    cout<<"++Tao : bay gio co "<<++count<<" doi tuong\n";
}
counter:: ~counter () {
```

³Có một số phiên bản chương trình dịch C++ không chấp nhận cách gọi hàm thành phần static qua đối tượng.

```

cout<<"--Xoa : bay gio con "<<--count<<" doi tuong\n";
}

void main() {
    clrscr();
    void fct();
    counter a;
    fct();
    counter::counter_display(); // gọi qua tên lớp
    counter b;
}
void fct() {
    counter u;
    counter::counter_display(); // gọi qua tên lớp
    counter v;
    v.counter_display(); // gọi qua đối tượng
}

```

```

++Tao : bay gio co 1 doi tuong
++Tao : bay gio co 2 doi tuong
Hien dang co 2 doi tuong
++Tao : bay gio co 3 doi tuong
Hien dang co 3 doi tuong
--Xoa : bay gio con 2 doi tuong
--Xoa : bay gio con 1 doi tuong
Hien dang co 1 doi tuong
++Tao : bay gio co 2 doi tuong
--Xoa : bay gio con 1 doi tuong
--Xoa : bay gio con 0 doi tuong

```

đoạn mã trên đây có thể thấy rằng ++ & -- chỉ được áp dụng cho biến số của lớp và không thể áp dụng cho biến

6. ĐỐI TƯỢNG HÀNG (CONSTANT)

6.1 Đối tượng hàng

Cũng như các phần tử dữ liệu khác, một đối tượng có thể được khai báo là **hàng**. Trừ khi có các chỉ định cụ thể, phương thức duy nhất sử dụng các đối tượng hàng là các hàm thiết lập và hàm huỷ bỏ. Bởi lẽ các đối tượng hàng không thể thay đổi, mà chỉ được tạo ra hoặc huỷ bỏ đi. Tuy nhiên, ta có thể tạo ra các phương thức khác để xử lý các đối tượng hàng. Mỗi hàm thành phần nào có từ khoá **const** đứng ngay sau danh sách các tham số hình thức trong dòng khai báo (ta gọi là hàm thành phần **const**) đều có thể sử dụng các đối tượng hàng trong lớp. Nói cách khác, ngoài hàm thiết lập và huỷ bỏ, các đối tượng hàng chỉ có thể gọi được các hàm thành phần được khai báo với từ khoá **const**. Xét ví dụ sau đây:

```
class point {
    int x,y;
public:
    point(...);
    void display() const;
    void move(...);
};
```

6.2 Hàm thành phần **const**

Hàm thành phần của lớp được khai báo với từ khoá **const** đứng ngay sau danh sách các tham số hình thức được gọi là **hàm thành phần const**.

Hàm thành phần **const** thì không thể thay đổi nội dung một đối tượng. Một hàm thành phần **const** phải được mô tả cả trong khai báo và khi định nghĩa. Hàm thành phần **const** có thể được định nghĩa chồng hàng một hàm khác không phải **const**.

7. HÀM BẠN VÀ LỚP BẠN

7.1 Đạt vấn đề

Trong các phần trước ta thấy rằng nguyên tắc đóng gói dữ liệu trong C++ bị “vi phạm” tí chút; các thành phần **private** của đối tượng chỉ có thể truy nhập bởi các hàm thành phần của chính lớp đó. Ngoài ra, hàm thành phần còn có thể truy nhập đến tất cả thành phần **private** của các đối tượng cùng lớp được khai báo cục bộ bên trong hàm thành phần đó hoặc được truyền như là tham số của hàm thành phần (có thể bằng tham trị, bằng tham chiếu hay bằng tham trả). Có thể thấy điều này trong định nghĩa của hàm thành phần **point::coincide()** và hàm thành phần

`point::symetric()` đã tinh hay ở trên. Những “vi phạm” trên đây đều chấp nhận được do làm tăng khả năng của ngôn ngữ và nâng cao tốc độ thực hiện chương trình.

Khi cho phép các hàm bạn, lớp bạn nguyên tắc đóng gói dữ liệu lại bị “vi phạm”. Sự “vi phạm” đó hoàn toàn chấp nhận được và tỏ ra rất hiệu quả trong một số tình huống đặc biệt: giả sử chúng ta đã có định nghĩa lớp vec tơ vector, lớp ma trận matrix. Và muốn định nghĩa hàm thực hiện nhân một ma trận với một vec tơ. Với những kiến thức về C++ cho đến nay, ta không thể định nghĩa hàm này bởi nó không thể là hàm thành phần của lớp vector, cũng không thể là hàm thành phần của lớp matrix và càng không thể là một hàm tự do (có nghĩa là không của lớp nào). Tuy nhiên, có thể khai báo tất cả các thành phần dữ liệu trong hai lớp vector và matrix là **public**, nhưng điều đó sẽ làm mất đi khả năng bảo vệ chúng. Một biện pháp khác là định nghĩa các hàm thành phần **public** cho phép truy nhập dữ liệu, tuy nhiên giải pháp này khá phức tạp và chi phí thời gian thực hiện không nhỏ. Khái niệm “hàm bạn” - friend function đưa ra một giải pháp tối ưu hơn nhiều cho vấn đề đặt ra ở trên. Khi định nghĩa một lớp, có thể khai báo rằng một hay nhiều hàm “Bạn” (bên ngoài lớp); khai báo bạn bè như thế cho phép các hàm này truy xuất được tới các thành phần **private** của lớp giống như các hàm thành phần của lớp đó. Ưu điểm của phương pháp này là kiểm soát các truy nhập ở cấp độ lớp: không thể áp đặt hàm bạn cho một lớp nếu điều đó không được dự trù trước trong khai báo của lớp. Điều này có thể ví như việc cấp thẻ vào ở một số cơ quan, không phải ai muốn đều được cấp thẻ mà chỉ những người có quan hệ đặc biệt với cơ quan mới được cấp.

Có nhiều kiểu bạn bè:

1. Hàm tự do là bạn của một lớp.
2. Hàm thành phần của một lớp là bạn của một lớp khác.
3. Hàm bạn của nhiều lớp.
4. Tất cả các hàm thành phần của một lớp là bạn của một lớp khác.

Sau đây chúng ta sẽ xem xét cụ thể cách khai báo, định nghĩa và sử dụng một hàm bạn cùng các tình huống đã nêu ở trên.

7.2 Hàm tự do bạn của một lớp

Trở lại ví dụ định nghĩa hàm `point::coincide()` kiểm tra sự trùng nhau của hai đối tượng kiểu point. Trước đây chúng ta định nghĩa nó như một hàm thành phần của lớp point:

```
class point {
    int x,y;
public:
    ...
    int coincide (point p); };
}
```

Ở đây còn có một cách khác định nghĩa coincide như một hàm tự do bạn của lớp point. Trước hết, cần phải đưa ra trong lớp point khai báo bạn bè:

```
friend int coincide (point , point);
```

Trong trường hợp hàm coincide() này là hàm tự do và không còn là hàm thành phần của lớp point nên chúng ta phải đưa hai tham số kiểu point cho coincide. Việc định nghĩa hàm coincide giống như một hàm thông thường. Sau đây là một ví dụ của chương trình:

Ví dụ 3.18

```
/*friend1.cpp*/
#include <iostream.h>
class point {
    int x, y;
public:
    point(int abs = 0, int ord = 0) {
        x = abs; y = ord;
    }
    friend int coincide (point,point);
};
int coincide (point p, point q) {
if ((p.x == q.x) && (p.y == q.y)) return 1;
else return 0;
}
void main() {
    point a(1,0),b(1),c;
    if (coincide (a,b)) cout <<"a trung voi b\n";
    else cout<<"a va b khac nhau\n";
    if (coincide(a,c)) cout<<"a trung voi c\n";
    else cout<<"a va c khac nhau\n";
}
```

a trung voi b

a va c khac nhau

Nhận xét

1. Vị trí của khai báo “bạn bè” trong lớp point hoàn toàn tuỳ ý.
2. Trong hàm bạn, không còn tham số ngầm định **this** như trong hàm thành phần.

Cũng giống như các hàm thành phần khác danh sách “tham số” của hàm bạn gắn với định nghĩa chống các toán tử. Hàm bạn của một lớp có thể có một hay nhiều tham số, hoặc có giá trị trả về thuộc kiểu lớp đó. Tuy rằng điều này không bắt buộc.

Có thể có các hàm truy xuất đến các thành phần riêng của các đối tượng cục bộ trong hàm. Khi hàm bạn của một lớp trả giá trị thuộc lớp này, thường đó là giá trị dữ liệu của đối tượng cục bộ bên trong hàm, việc truyền tham số phải thực hiện bằng tham trị, hở lẽ truyền bằng tham chiếu (hoặc hằng địa chỉ) hàm gọi sẽ nhận địa chỉ của một vùng nhớ bị giải phóng khi hàm kết thúc.

7.3 Các kiểu bạn bè khác

7.3.1 Hàm thành phần của lớp là bạn của lớp khác

Có thể xem đây như là một trường hợp đặc biệt của tình huống trên, chỉ khác ở cách mô tả hàm. Người ta sử dụng tên đầy đủ của hàm thành phần bao gồm tên lớp, toán tử phạm vi và tên hàm thành phần bạn bè.

Giả thiết có hai lớp A và B, trong B có một hàm thành phần f khai báo như sau:

```
int f(char , A);
```

Nếu f có nhu cầu truy xuất vào các thành phần riêng của A thì f cần phải được khai báo là bạn của A ở trong lớp A bằng câu lệnh:

```
friend int B::f(char , A);
```

Ta có các nhận xét quan trọng sau:

để biên dịch được các khai báo của lớp A có chứa khai báo bạn bè kiểu như:

```
friend int B::f(char, A);
```

chương trình dịch cần phải biết được nội dung của lớp B; nghĩa là khai báo của B (không nhất thiết định nghĩa của các hàm thành phần) phải được biên dịch trước khai báo của A.

Ngược lại, khi biên dịch khai báo:

```
int f(char, A) ;
```

bên trong lớp B, chương trình dịch không nhất thiết phải biết chi tiết nội dung của A, nó chỉ cần biết rằng là một lớp. Để có được điều này ta dùng chỉ thị sau:

```
class A;
```

trước khai báo lớp B. Việc biên dịch định nghĩa hàm f cần các thông tin đầy đủ về các thành phần của A và B; như vậy các khai báo của A và B phải có trước định nghĩa đầy đủ của f. Tóm lại, sơ đồ khai báo và định nghĩa phải như sau:

```
class A;
class B {
...
int f(char, A);
...
}; class A {
...
friend int B::f(char, A);
...
}; int B::f(char ...,A ...) {
...
}
```

Đề nghị bạn đọc thử nghiệm trường hợp "bạn bè chéo nhau", nghĩa là đồng thời hàm thành phần của lớp này là bạn của lớp kia và một hàm thành phần của lớp kia là bạn của lớp này.

7.3.2 Hàm bạn của nhiều lớp

Về nguyên tắc, mọi hàm (hàm tự do hay hàm thành phần) đều có thể là bạn của nhiều lớp khác nhau. Sau đây là một ví dụ một hàm là bạn của hai lớp A và B.

```
class A {
...
friend void f(A, B);
...
};
class B {
...
friend void f(A,B);
...
};
```

```
void f(A...,B...) {
//truy nhập vào các thành phần riêng của hai lớp bất kỳ A và B
}
```

Nhận xét

Ở đây, khai báo của A có thể được biên dịch không cần khai báo của B nếu có chỉ thị class B; đứng trước.

Tương tự, khai báo của lớp B cũng được biên dịch mà không cần đến A nếu có chỉ thị class A; đứng trước.

Nếu ta muốn biên dịch cả hai khai báo của A và của B, thì cần phải sử dụng một trong hai chỉ thị đã chỉ ra ở trên. Còn định nghĩa của hàm f cần phải có đầy đủ cả hai khai báo của A và của B đứng trước. Sau đây là một ví dụ minh họa:

```
class B;
class A {
...
friend void f(A, B);
...
};

class B {
...
friend void f(A,B);
...
};

void f(A...,B...) {
//truy nhập vào các thành phần riêng của hai lớp bất kỳ A và B
}
```

7.3.3 Tất cả các hàm của lớp là bạn của lớp khác

Đây là trường hợp tổng quát trong đó có thể khai báo lớp hạn hẹp với các hàm. Mọi vấn đề sẽ đơn giản hơn nếu ta đưa ra một khai báo tổng thể để nói rằng tất cả các hàm thành phần của lớp B là bạn của lớp A. Muốn vậy ta sẽ đặt trong khai báo lớp A chỉ thị:

```
friend class B;
```

Nhận xét: Trong trường hợp này, để biên dịch khai báo của lớp A, chỉ cần đặt trước nó chỉ thị: class B; kiểu khai báo lớp bạn cho phép không phải khai báo tiêu đề của các hàm có liên quan.

7.4 Bài toán nhân ma trận với vector

Trong phần này ta sẽ giải quyết bài toán xây dựng một hàm nhân ma trận (đối tượng thuộc lớp matrix) với vector (đối tượng thuộc lớp vect). Để đơn giản, ta giới hạn chỉ có các hàm thành phần:

- (i) một constructor cho vect và matrix,
- (ii) một hàm hiển thị (display) cho vect.

Ta trình bày hai giải pháp dựa trên việc định nghĩa prod có hai đối số, một là đối tượng matrix và một là đối tượng vect:

1. prod là hàm tự do và là bạn của hai lớp vect và matrix,
2. prod là hàm thành phần của matrix và là bạn của vect.

Giải pháp thứ nhất - prod là hàm bạn tự do

Ví dụ 3.19

```
/*vectmat1.cpp*/
#include <iostream.h>

class matrix;
//****class vect

class vect {
double v[3]; //vector có ba thành phần
public:
vect (double v1=0, double v2=0, double v3=0)
{ v[0] = v1; v[1] = v2; v[2] = v3;
}
friend vect prod(matrix, vect);
void display () {
int i;
for (int i=0; i<3; i++) cout<<v[i]<<" ";
cout<<endl;
}
};
```

```

*****class matrix
class matrix {
double mat[3][3];
public:
matrix(double t[3][3])
{ int i; int j;
for(i=0; i<3; i++)
for(j=0; j<3; j++)
mat[i][j] = t[i][j];
}
friend vect prod (matrix, vect);
};

*****Hàm prod
vect prod (matrix m, vect x) {
int i,j;
double sum;
vect res; //kết quả
for(i=0; i<3; i++) {
for(j=0,sum=0; j<3; j++)
sum += m.mat[i][j]*x.v[j];
res.v[i] = sum;
}
return res;
}

***** chương trình kiểm tra
void main() {
vect w1,2,3);
vect res;
double tb[3][3] = {1,2,3,4,,5,6,7,8,9};
matrix a=tb;
res=prod(a,w); res.display();
}

```

14 32 50

Giải pháp thứ hai- prod là hàm thành phần của lớp matrix và là bạn của vect

Ví dụ 3.20

```
/*vectmat2.cpp*/
#include <iostream.h>
class vect;
//***class matrix
class matrix {
double mat[3][3];
public:
matrix(double t[3][3])
{ int i; int j;
for(i=0; i<3; i++)
for(j=0; j<3; j++)
mat[i][j] = t[i][j];
}
vect prod (vect);
};

//*** class vect
class vect {
double v[3];
public:
vect (double v1=0, double v2=0, double v3=0)
{ v[0] = v1; v[1] = v2; v[2] = v3;
}
friend vect matrix::prod(vect);
void display () {
int i;
for (int i=0; i<3; i++) cout<<v[i]<<" ";
cout<<endl;
}
```

```

};

//***Hàm matrix::prod
vect matrix::prod (vect x) {
    int i,j;
    double sum;
    vect res;
    for(i=0; i<3; i++) {
        for(j=0, sum=0; j<3; j++)
            sum += mat[i][j]*x.v[j];
        res.v[i] = sum;
    }
    return res;
}

//*** chương trình kiểm tra
void main() {
    vect w{1,2,3};
    vect res;
    double tb[3][3] = {1,2,3,4,5,6,7,8,9};
    matrix a=tb;
    res=a.prod(w); res.display();
}

```

14 32 50

8. VÍ DỤ TỔNG HỢP

Chương trình vectmat3.cpp sau đây được phát triển dựa trên các chương trình vector2.cpp , vectmat1.cpp và vectmat2.cpp.

Ví dụ 3.21

```

/*vectmat2.cpp*/
#include <iostream.h>
#include <conio.h>
/*lớp vector*/
class matrix; /*khai báo trước lớp matrix*/

```

```
/*khai báo lớp vector*/
class vector{
    static int n; //số chiều của vector
    float *v; //vùng nhớ chứa các toạ độ
public:
    vector();
    vector(float *);
    vector(vector &); //hàm thiết lập sao chép
    ~vector();
    void display();
    static int & Size() {return n;}
    friend vector prod(matrix &, vector &);
    friend class matrix;
};

int vector::n = 0;

/*các hàm thành phần của lớp vector*/
vector::vector()
{
    int i;
    v= new float [n];
    for(i=0;i<n;i++) {
        cout<<"Toa do thu "<<i+1<<" : ";
        cin>>v[i];
    }
}

vector::vector(float *a) {
    for(int i =0; i<n; i++)
        v[i]=a[i];
}

vector::vector(vector &b)
{
    int i;
```

```

for(i=0;i<n;i++)
    v[i] = b.v[i];
}

vector::~vector(){
    delete v;
}

void vector::display() //hiển thị kết quả
{
    for(int i=0;i<n;i++)
        cout <<v[i] <<" ";
    cout <<"\n";
}

/*khai báo lớp matrix*/
class matrix{
    static int n; //số chiều của vector
    vector *m; //vùng nhớ chứa các toạ độ
public:
    matrix();
    ~matrix();
    void display();
    static int &Size() {return n;}
    friend vector prod(matrix &, vector &);
};

int matrix::n =0;
/*hàm thành phần của lớp matrix*/
matrix::matrix(){
    int i;
    m= new vector [n];
}

matrix::~matrix() {
    delete m;
}

```

```
void matrix::display() //hiển thị kết quả
{
for (int i=0; i<n; i++)
    m[i].display();
}

/*hàm prod*/
vector prod(matrix &m, vector &v) {
    float *a = new float [vector::Size()];
    int i,j;
    for (i=0; i<matrix::Size(); i++) {
        a[i]=0;
        for(j=0; j<vector::Size(); j++)
            a[i]+=m.m[i].v[j]*v.v[j];
    }
    return vector(a);
}

void main()
{
    clrscr();
    int size;
    cout<<"Kích thước của vector "; cin>>size;
    vector::Size() = size;
    matrix::Size() = size;
    cout<<"Tạo một vector \n";
    vector v;
    cout<<" v= \n";
    v.display();
    cout<<"Tạo một ma trận \n";
    matrix m;
    cout<<" m = \n";
    m.display();
}
```

```

cout<<"Tich m*v \n";
vector u = prod(m,v);
u.display();
getch();
}

```

Kich thuoc cua vector 3

Tao mot vector

Toa do thu 1 : i

Toa do thu 2 : i

Toa do thu 3 : i

v=

1 1 1

Tao mot ma tran

Toa do thu 1 : 2

Toa do thu 2 : 3

Toa do thu 3 : 2

Toa do thu 1 : 1

Toa do thu 2 : 2

Toa do thu 3 : 3

Toa do thu 1 : 2

Toa do thu 2 : 3

Toa do thu 3 : 2

m =

2 3 2

1 2 3

2 3 2

Tich m*v

7 6 7

9. TÓM TẮT

9.1 Ghi nhớ

Trong C++, tên cấu trúc là một kiểu dữ liệu không cần phải kèm theo từ khoá **struct**.

Lớp cho phép người lập trình mô tả các đối tượng thực tế với các thuộc tính và hành vi. Trong C++ thường sử dụng từ khoá **class** để khai báo một lớp. Tên lớp là một kiểu dữ liệu dùng khi khai báo các đối tượng thuộc lớp (các thể hiện cụ thể của lớp).

Thuộc tính của đối tượng trong một lớp được mô tả dưới dạng các biến thể hiện. Các hành vi là các hàm thành phần bên trong lớp.

Có hai cách định nghĩa các hàm thành phần của một lớp; khi định nghĩa hàm thành phần bên ngoài khai báo lớp phải đặt trước tên hàm thành phần tên của lớp và toán tử “::” để phân biệt với các hàm tự do cùng tên. Chỉ nên định nghĩa hàm thành phần bên trong lớp khi nó không quá phức tạp để cho chương trình dễ đọc.

Có thể khai báo và sử dụng các con trỏ đối tượng, tham chiếu đối tượng.

Hai từ khoá **public** và **private** dùng để chỉ định thuộc tính truy nhập cho các thành phần (dữ liệu/hàm) khai báo bên trong lớp.

Thành phần bên trong lớp được khai báo **public** có thể truy nhập từ mọi hàm khai báo một đối tượng thuộc lớp đó.

Thành phần **private** trong một đối tượng chỉ có thể truy nhập được bởi các hàm thành phần của đối tượng hoặc các hàm thành phần của lớp dùng để tạo đối tượng (ở đây tính cả trường hợp đối tượng là tham số của hàm thành phần).

Hai hàm thành phần đặc biệt của một lớp gọi là hàm thiết lập và hàm huỷ bỏ. Hàm thiết lập được gọi tự động (ngầm định) mỗi khi một đối tượng được tạo ra và hàm huỷ bỏ được gọi tự động khi đối tượng hết thời gian sử dụng.

Hàm thiết lập có thuộc tính **public**, cùng tên với tên lớp nhưng không có giá trị trả về.

Một lớp có ít nhất hai hàm thiết lập: hàm thiết lập sao chép ngầm định và hàm thiết lập do người lập trình thiết lập (nếu không được mô tả tường minh thì đó là hàm thiết lập ngầm định).

Hàm huỷ bỏ cũng có thuộc tính **public**, không tham số, không giá trị trả về và có tên bắt đầu bởi ~ theo sau là tên của lớp.

Bên trong phạm vi lớp (định nghĩa của các hàm thành phần), các thành phần của lớp được gọi theo tên. Trường hợp có một đối tượng toàn cục cùng tên, muốn xác định đối tượng ấy phải sử dụng toán tử “::”

Lớp có thể chứa các thành phần dữ liệu là các đối tượng có kiểu lớp khác. Các đối tượng này phải được khởi tạo trước đối tượng tương ứng của lớp bao.

Mỗi đối tượng có một con trỏ chỉ đến bản thân nó, ta gọi đó là con trỏ **this**. Con trỏ này có thể được sử dụng tường minh hoặc ngầm định để tham xác định các thành phần bên trong đối tượng. Thông thường người ta sử dụng this dưới dạng ngầm định.

Toán tử **new** tự động tạo ra một đối tượng với kích thước thích hợp và trả về con trỏ có kiểu lớp. Để giải phóng vùng nhớ cấp phát cho đối tượng này sử dụng toán tử **delete**.

Thành phần dữ liệu tĩnh biểu thị các thông tin dùng chung trong tất cả các đối tượng thuộc lớp. Khai báo của thành phần tĩnh bắt đầu bằng từ khoá **static**.

Có thể truy nhập tới các thành phần tĩnh thông qua các đối tượng kiểu lớp hoặc bằng tên lớp nhờ sử dụng toán tử phạm vi.

Hàm thành phần có thể được khai báo là tĩnh nếu nó chỉ truy nhập đến các thành phần dữ liệu tĩnh.

Hàm bạn của một lớp là hàm không thuộc lớp nhưng có quyền truy nhập tới các thành phần private của lớp.

Khai báo bạn bè có thể đặt bất kỳ nơi nào trong khai báo lớp.

9.2 Các lỗi thường gặp

Quên dấu ";" ở cuối khai báo lớp.

Khởi tạo giá trị cho các thành phần dữ liệu trong khai báo lớp.

Định nghĩa chống một hàm thành phần bằng một hàm không thuộc lớp.

Truy nhập đến các thành phần riêng của lớp từ bên ngoài phạm vi lớp

Khai báo giá trị trả về cho hàm thiết lập và hàm huỷ bỏ.

Khai báo hàm huỷ bỏ có tham số, định nghĩa chống hàm huỷ bỏ.

Gọi tường minh hàm thiết lập và hàm huỷ bỏ.

Gọi các hàm thành phần bên trong hàm thiết lập

Định nghĩa một hàm thành phần **const** thay đổi các thành phần dữ liệu của một đối tượng.

Định nghĩa một hàm thành phần const gọi tới một hàm thành phần không phải **const**.

Gọi các hàm thành phần không phải const từ các đối tượng **const**.

Thay đổi nội dung một đối tượng **const**.

Nhầm lẫn giữa **new** và **delete** với **malloc** và **free**.

Sử dụng **this** bên trong các hàm thành phần tĩnh.

9.3 Một số thói quen lập trình tốt

Nhóm tất cả các thành phần có cùng thuộc tính truy nhập ở một nơi trong khai báo lớp, nhờ vậy mỗi từ khoá mô tả truy nhập chỉ được xuất hiện một lần. Khai báo lớp vì vậy dễ đọc hơn. Theo kinh nghiệm, để các thành phần **private** trước tiên rồi đến các thành phần **protected**, cuối cùng là các thành phần **public**.

Định nghĩa tất cả các hàm thành phần bên ngoài khai báo lớp. Điều này nhằm phân biệt giữa hai phần giao diện và phần cài đặt của lớp.

Sử dụng các chỉ thị tiền xử lý `#ifndef`, `#define`, `#endif` để cho các tập tin tiêu đề chỉ xuất hiện một lần bên trong các chương trình nguồn.

Phải định nghĩa các hàm thiêt lập để đảm bảo rằng các đối tượng đều được khởi tạo nội dung một cách đúng đắn.

Khai báo là **const** tất cả các hàm thành phần chỉ để sử dụng với các đối tượng **const**.

Nên sử dụng **new** và **delete** thay vì **malloc** và **free**.

10. BÀI TẬP

Bài tập 3.1

Số sánh ý nghĩa của struct và class trong C++

Bài tập 3.2

Tạo một lớp gọi là Complex để thực hiện các thao tác số học với các số phức. Viết một chương trình để kiểm tra lớp này.

Số phức có dạng

$\langle\text{Phần thực}\rangle + \langle\text{Phần ảo}\rangle * j$

Sử dụng các biến thực để biểu diễn các thành phần dữ liệu riêng của lớp. Cung cấp một hàm thiết lập để tạo đối tượng. Hàm thiết lập sử dụng các tham số có giá trị ngầm định. Ngoài ra còn có các hàm thành phần **public** để thực hiện các công việc sau:

- + Cộng hai số phức: các phần thực được cộng với nhau và các phần ảo được cộng với nhau.
- + Trừ hai số phức: Phần thực của số phức thứ hai được trừ cho phần thực của số phức thứ nhất. Tương tự cho phần ảo.
- + In số phức ra màn hình dưới dạng (a, b) trong đó a là phần thực và b là phần ảo.

Bài tập 3.3

Tạo một lớp gọi là PS để thực hiện các thao tác số học với phân số. Viết chương trình để kiểm tra lớp vừa tạo ra.

Sử dụng các biến nguyên để biểu diễn các thành phần dữ liệu của lớp-tử số và mẫu số. Viết định nghĩa hàm thiết lập để tạo đối tượng sao cho phần ảo phải là số nguyên dương. Ngoài ra còn có các hàm thành phần khác thực hiện các công việc cụ thể:

- + Cộng hai phân số. Kết quả phải được tối giản.
- + Trừ hai phân số. Kết quả phải được tối giản.
- + Nhận hai phân số. Kết quả dưới dạng tối giản.
- + Chia hai phân số. Kết quả dưới dạng tối giản.
- + In ra màn hình phân số dưới dạng a/b trong đó a là tử số, còn b là mẫu số.
- + In phân số dưới dạng số thập phân.

Bài tập 3.4

Khai báo, định nghĩa và sử dụng lớp time mô tả các thông tin về giờ, phút và giây với các yêu cầu như sau:

Tạo tập tin tiêu đề TIME.H chứa khai báo của lớp time với các thành phần dữ liệu mô tả giờ, phút và giây: hour, minute, second.

Trong lớp time khai báo :

- + một hàm thiết lập ngầm định, dùng để gán cho các thành phần dữ liệu giá trị 0.
- + hàm thành phần set (int, int, int) với ba tham số tương ứng mang giá trị của ba thành phần dữ liệu.
- + hàm hiển thị trong đó giờ được hiển thị với giá trị 0 tới 24.
- + hàm hiển thị chuẩn có phân biệt giờ trước và sau buối trưa.

Tạo tập tin chương trình TIME . CPP chứa định nghĩa của các hàm thành phần trong lớp time, và chương trình minh họa cách sử dụng lớp time.

Bài tập 3.5

Tương tự như bài 3.4 nhưng ở đây hàm thiết lập có ba tham số có giá trị ngầm định bằng 0.

Bài tập 3.6

Vẫn dựa trên lớp time, nhưng ở đây ta bổ sung thêm các hàm thành phần để thiết lập riêng rẽ giờ, phút, giây:

- + void setHour(int)
- + void setMinute(int)
- + void setSecond(int)

Đồng thời có các hàm để lấy từng giá trị đó của từng đối tượng:

- + int getHour()
- + int getMinute();
- + int getSecond()

Bài tập 3.7

Thêm một hàm thành phần `tick()` vào lớp `time` để tăng thời gian trong một đối tượng `time` mỗi lần một giây. Lưu ý các trường hợp, tăng sang phút tiếp theo, tăng sang giờ tiếp theo, tăng sang ngày tiếp theo.

Bài tập 3.8

Khai báo lớp `date` mô tả thông tin về ngày tháng năm: `month`, `day`, `year`.

Lớp `date` có hai hàm thành phần:

- + hàm thiết lập với ba tham số có giá trị ngầm định.
- + hàm `print()` in thông tin về ngày tháng dưới dạng quen thuộc `mm-dd-yy`.

Viết chương trình kiểm tra việc sử dụng phép gán cho các đối tượng thuộc lớp `date`.

Bài tập 3.9

Dựa trên lớp `date` của bài 3.8 người ta thực hiện một số thay đổi để kiểm soát lỗi trên giá trị các tham số của hàm thiết lập. Đồng thời bổ sung thêm hàm thành phần `nextDay()` để tăng `date` từng ngày một.

Bài tập 3.10

Kết hợp lớp `time` trong bài 3.7 và lớp `date` trong bài 3.9 để tạo nên một lớp `date_time` mô tả đồng thời thông tin về ngày, giờ. Thay đổi hàm thành phần `tick()` để gọi tới hàm `tăng ngày` mỗi khi cần thiết. Thêm các hàm hiển thị thông tin về ngày giờ. Hoàn thiện chương trình để kiểm tra hoạt động của lớp.

Bài tập 3.11

Viết chương trình khai báo lớp mô tả hoạt động của một ngăn xếp hoặc hàng đợi chứa các số nguyên.

ĐỊNH NGHĨA TOÁN TỬ TRÊN LỚP

(class operators)

Mục đích chương này :

1. Cách định nghĩa các phép toán cho kiểu dữ liệu lớp và cấu trúc
2. Các toán tử chuyển kiểu áp dụng cho kiểu dữ liệu lớp

1. GIỚI THIỆU CHUNG

Thực ra, vấn đề định nghĩa chồng toán tử đã từng có trong C, ví dụ trong biểu thức:

$a + b$

ký hiệu + tuỳ theo kiểu của a và b có thể biểu thị:

1. phép cộng hai số nguyên,
2. phép cộng hai số thực độ chính xác đơn (**float**)
3. phép cộng hai số thực chính xác đôi (**double**)
4. phép cộng một số nguyên vào một con trả.

Trong C++, có thể định nghĩa chồng đối với hầu hết các phép toán (một ngôi hoặc hai ngôi) trên các lớp, nghĩa là một trong số các toán hạng tham gia phép toán là các đối tượng. Đây là một khả năng mạnh vì nó cho phép xây dựng trên các lớp các toán tử cần thiết, làm cho chương trình được viết ngắn gọn dễ đọc hơn và có ý nghĩa hơn. Chẳng hạn, khi định nghĩa một lớp **complex** để biểu diễn các số phức, có thể viết trong C++: $a+b$, $a-b$, $a*b$, a/b với a, b là các đối tượng **complex**.

Để có được điều này, ta định nghĩa chồng các phép toán +, -, * và / bằng cách định nghĩa hoạt động của từng phép toán giống như định nghĩa một hàm, chỉ khác là đây là hàm toán tử (operator function). Hàm toán tử có tên được ghép bởi từ khoá **operator** và ký hiệu của phép toán tương ứng. Bảng 4.1 đưa ra một số ví dụ về tên hàm toán tử.

Hàm toán tử có thể dùng như là một hàm thành phần của một lớp hoặc là hàm tự do; khi đó hàm toán tử phải được khai báo là bạn của các lớp có các đối tượng mà hàm thao tác.

Tên hàm	Dùng để
operator+	định nghĩa phép +
operator*	định nghĩa phép nhân *
operator/	định nghĩa phép chia /
operator+=	định nghĩa phép tự cộng +=
operator!=	định nghĩa phép so sánh khác nhau

Bảng 4.1 Một số tên hàm toán tử quen thuộc.

2. VÍ DỤ TRÊN LỚP SỐ PHỨC

2.1 Hàm toán tử là hàm thành phần

Trong chương trình complex1.cpp toán tử + giữa hai đối tượng complex được định nghĩa như một hàm thành phần. Hàm toán tử thành phần có một tham số ngầm định là đối tượng gọi hàm nên chỉ có một tham số tường minh.

Ví dụ 4.1

```
/*complex1.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
public:
    complex(float r=0, float i =0) {
        real = r; image = i;
    }
    void display() {
        cout<<real<<(image)>=0?'+':'-'<<"j*"<<fabs(image)<<endl;
    }
/*hàm operator+ định nghĩa phép toán + hai ngôi trên lớp số phức complex*/
complex operator+(complex b) {
    complex c;
    c.real = real+b.real;
```

```

        c.image =image+b.image; // gán giá trị của biến c bằng tổng
        // của biến a và biến b, trước đó phải là để kế thừa, vì biến a là
        // biến của lớp mà biến c là biến con kế thừa nó nên có thể copy
    }
}

int main()
{
    clrscr();
    complex a(-2,5);
    complex b(3,4);
    cout<<"Hai so phuc:\n";
    a.display();
    b.display();
    cout<<"Tong hai so phuc:\n";
    complex c;
    c=a+b;//a.operator+(b)
    c.display();
    getch();
}

```

```

Hai so phuc:
-2+j*5
3+j*4
Tong hai so phuc:
1+j*9

```

Chỉ thị

`c = a+b;`

trong ví dụ trên được chương trình dịch hiểu là:

`c = a.operator+(b);`

Nhận xét

- Thực ra cách viết `a+b` chỉ là một quy ước của chương trình dịch cho phép người sử dụng viết gọn lại, nhờ đó cảm thấy tự nhiên hơn.
- Hàm toán tử `operator+` phải có thuộc tính **public** vì nếu không chương trình dịch không thể thực hiện được nó ở ngoài phạm vi lớp.

3. Trong lời gọi `a.operator+(b)`, a đóng vai trò của tham số ngầm định của hàm thành phần và b là tham số tường minh. Số tham số tường minh cho hàm toán tử thành phần luôn ít hơn số ngôi của phép toán là 1 vì có một tham số ngầm định là đối tượng gọi hàm toán tử.
4. Chương trình dịch sẽ không thể hiểu được biểu thức `3+a` vì cách viết tương ứng `3.operator+(a)` không có ý nghĩa. Để giải quyết tình huống này ta dùng hàm bạn để định nghĩa hàm toán tử.

2.2 Hàm toán tử là hàm bạn

Chương trình `complex2.cpp` được phát triển từ `complex1.cpp` bằng cách thêm hàm toán tử cộng thêm một số thực `float` vào phần thực của một đối tượng `complex`, được hiểu là bởi phép cộng với số thực `float` là toán hạng thứ nhất, còn đối tượng `complex` là toán hạng thứ hai. Trong trường hợp này không thể dùng phép cộng như hàm thành phần vì tham số thứ nhất của hàm toán tử không còn là một đối tượng.

Ví dụ 4.2

```
/*complex2.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
public:
    complex(float r=0, float i =0) {
        real = r; image = i;
    }
    void display() {
        cout<<real<<(image>=0?'+' :'-')<<" ] * " <<fabs(image)<<endl ;
    }
/*hàm thành phần operator+ định nghĩa phép toán + hai ngôi trên lớp số phức complex*/
complex operator+(complex b) {
    cout<<"Gọi tới complex::operator+(float, complex)\n";
    complex c;
    c.real = real+b.real;
    c.image =image+b.image;
```

```
return c;
}

/*hàm tự do operator+ định nghĩa phép toán + giữa một số thực và một đối tượng số phức*/
friend complex operator+(float x, complex b);
};

complex operator+(float x, complex b) {
    cout<<"Goi toi operator+(float, complex)\n";
    complex c;
    c.real = x+b.real;
    c.image = b.image;
    return c;
}

void main() {
    clrscr();
    complex a(-2,5);
    complex b(3,4);
    cout<<"Hai so phuc:\n";
    cout<<"a = ";
    a.display();
    cout<<"b = ";
    b.display();
    cout<<"Tong hai so phuc:\n";
    complex c;
    c=a+b; //a.operator+(b);
    cout<<"c = ";
    c.display();
    cout<<"Tang them phan thuc cua a 3 don vi\n";
    complex d;
    d=3+a; //operator+(3,a);
    cout<<"d = ";
    d.display();
    getch();
}
```

```

}

Hai so phuc:
a = -2+j*5
b = 3+j*4
Tong hai so phuc:
Goi toi complex::operator+(complex)
c = 1+j*9
Tang them phan thuc cua a 3 don vi
Goi toi operator+(float, complex)
d = 1+j*5

```

Trong chương trình trên, biểu thức $a+b$ được chương trình hiểu là lời gọi hàm thành phần $a.\text{operator}+(b)$, trong khi đó với biểu thức $3+a$, chương trình dịch sẽ thực hiện lời gọi hàm tự do $\text{operator}+(3, a)$.

Số tham số trong hàm toán tử tự do $\text{operator}+(\dots)$ đúng bằng số ngôi của phép + mà nó định nghĩa. Trong định nghĩa của hàm toán tử tự do, tham số thứ nhất có thể có kiểu bất kỳ chứ không nhất thiết phải có kiểu lớp nào đó.

Với một hàm $\text{operator}+$ nào đó chỉ có thể thực hiện được phép + tương ứng giữa hai toán hạng có kiểu như đã được mô tả trong tham số hình thức, nghĩa là muốn có được phép cộng “vạn năng” áp dụng cho mọi kiểu toán hạng ta phải định nghĩa rất nhiều hàm toán tử $\text{operator}+$ (định nghĩa chống các hàm toán tử).

Vấn đề bảo toàn các tính chất tự nhiên của các phép toán không được C++ đê cập, mà nó phụ thuộc vào cách cài đặt cụ thể trong chương trình dịch C++ hoặc bản thân người sử dụng khi định nghĩa các hàm toán tử. Chẳng hạn, phép gán:

$c = a+b;$

được chương trình dịch hiểu như là: $c = a.\text{operator}+(b)$; trong khi đó với phép gán:

$d = a+b+c;$

ngôn ngữ C++ không đưa ra diễn giải nghĩa duy nhất. Một số chương trình biên dịch sẽ tạo ra đối tượng trung gian t:

$t=a.\text{operator}+(b);$

và

$d=t.\text{operator}+(c);$

Chương trình complex3.cpp sau đây minh họa lý giải này:

Ví dụ 4.3

```
/*complex3.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
public:
    complex(float r=0, float i =0) {
        cout<<"Tao doi tuong :"<<this<<endl;
        real = r; image = i;
    }
    void display() {
        cout<<real<<(image>=0?'+' : '-')<<"j*"<<fabs(image)<<endl;
    }
}
complex operator+(complex b) {
    cout<<"Goi toi complex::operator+(complex)\n";
    cout<<this<<endl;
    complex c;
    c.real=real+b.real;
    c.image=image+b.image;
    return c;
}
friend complex operator+(float x, complex b);
complex operator+(float x, complex b) {
    cout<<"Goi toi operator+(float, complex)\n";
    complex c;
    c.real = x+b.real;
    c.image = b.image;
    return c;
}
```

```

void main() {
    clrscr();
    cout<<"so phuc a \n";
    complex a(-2,5);
    cout<<"so phuc b \n";
    complex b(3,4);
    cout<<"Hai so phuc:\n";
    cout<<"a = ";
    a.display();
    cout<<"b = ";
    b.display();
    complex c(2,3);
    cout<<"Cong a+b+c\n";
    cout<<"so phuc d \n";
    complex d;
    d = a+b+c;
    cout<<"a = ";a.display();
    cout<<"b = ";b.display();
    cout<<"c = ";c.display();
    cout<<"d = a+b+c : ";
    d.display();
    getch();
}

```

Ex 1b (7)

so phuc a
Tao doi tuong :0xffffe
so phuc b
Tao doi tuong :0xffffe6
Hai so phuc:
a = -2+j*5
b = 3+j*4
Tao doi tuong :0xffde
Cong a+b+c

```

so phuc d
Tao doi tuong :0xffffd6
Goi toi complex::operator+(complex)
0xffffe
Tao doi tuong :0xffffa0
Goi toi complex::operator+(complex)
0xffffce
Tao doi tuong :0xffffa8
a = -2+j*5
b = 3+j*4
c = 2+j*3
d = a+b+c : 3+j*12

```

Cũng có thể làm như sau: trong định nghĩa của hàm toán tử, ta trả về tham chiếu đến một trong hai đối tượng tham gia biểu thức (chẳng hạn a). Khi đó a+b+c được hiểu là a.operator+(b) và sau đó là a.operator+(c). Tất nhiên trong trường hợp này nội dung của đối tượng a bị thay đổi sau mỗi phép cộng. Xét chương trình sau:

Ví dụ 4.4

```

/*complex4.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
public:
    complex(float r=0, float i =0) {
        cout<<"Tao doi tuong :"<<this<<endl;
        real = r; image = i;
    }
    void display() {
        cout<<real<<(image>=0?'+'':'-')<<"j*"<<fabs(image)<<endl;
    }
    complex & operator+(complex b) {

```

```

cout<<"Gọi tới complex::operator+(complex)\n";
cout<<this<<endl;

real+=b.real;
image+=b.image;
return *this;
}

friend complex operator+(float x, complex b);
};

complex operator+(float x, complex b) {
    cout<<"Gọi tới operator+(float, complex)\n";
    complex c;
    c.real = x+b.real;
    c.image = b.image;
    return c;
}

void main() {
    clrscr();
    cout<<"so phuc a \n";
    complex a(-2,5);
    cout<<"so phuc b \n";
    complex b(3,4);
    cout<<"Hai so phuc:\n";
    cout<<"a = ";
    a.display();
    cout<<"b = ";
    b.display();
    cout<<"so phuc c \n";
    complex c;
    c=a+b; //a.operator+(b);
    cout<<"c = a+b: ";
    c.display();
    cout<<"a = ";

```

```

a.display();
cout<<"Cong a+b+c\n";
cout<<"so phuc d \n";
complex d;
d = a+b+c;//a.operator+(b);a.operator+(c);
cout<<"a = ";a.display();
cout<<"b = ";b.display();
cout<<"c = ";c.display();
cout<<"d = a+b+c : ";
d.display();
getch();
}

```

```

so phuc a
Tao doi tuong :0xffffe
so phuc b
Tao doi tuong :0xfffe6
Hai so phuc:
a = -2+j*5
b = 3+j*4
so phuc c
Tao doi tuong :0xfffd8
Goi toi complex::operator+(complex)
0xffffe
c = a+b: 1+j*9
a = 1+j*9
Cong a+b+c
so phuc d
Tao doi tuong :0xfffd6
Goi toi complex::operator+(complex)
0xffffe
Goi toi complex::operator+(complex)
0xffffe

```

```
a = 5+j*22
b = 3+j*4
c = 1+j*9
d = a+b+c : 5+j*22
```

Trong hai ví dụ trên, việc truyền các đối số và giá trị trả về của hàm toán tử được thực hiện bằng giá trị. Với các đối tượng có kích thước lớn, người ta thường dùng tham chiếu để truyền đối cho hàm.

```
complex operator+(float , complex &);
```

Tuy nhiên việc dùng tham chiếu như là giá trị trả về của hàm toán tử, có nhiều điều đáng nói. Biểu thức nằm trong lệnh **return** bắt buộc phải tham chiếu đến một vùng nhớ tồn tại ngay cả khi thực hiện xong biểu thức tức là khi hàm toán tử kết thúc thực hiện. Vùng nhớ ấy có thể là một biến được cấp tĩnh **static** (các biến toàn cục hay biến cục bộ **static**), một biến thể hiện (một thành phần dữ liệu) của một đối tượng nào đó ở ngoài hàm. Bạn đọc có thể xem chương trình **vecmat3.cpp** trong chương 3 để hiểu rõ hơn. Vấn đề tương tự cũng được đề cập khi giá trị trả về của hàm toán tử là địa chỉ; trong trường hợp này, một đối tượng được tạo ra nhờ cấp phát động trong vùng nhớ heap dùng độc lập với vùng nhớ ngăn xếp dùng để cấp phát biến, đối tượng cục bộ trong chương trình, do vậy vẫn còn lưu lại khi hàm toán tử kết thúc công việc.

Hàm toán tử cũng có thể trả về kiểu **void** khi ảnh hưởng chỉ tác động lên một trong các toán hạng tham gia biểu thức. Xem định nghĩa của hàm đảo dấu số phức trong ví dụ sau:

Ví dụ 4.5

```
/*complex5.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>

class complex {
    float real, image;
public:
    complex(float r=0, float i =0) {
        real = r; image = i;
    }
    void display() {
```

```

cout<<real<<(image>=0?'+' :'-')<<"j*"<<fabs(image)<<endl;
}

/*Hàm đảo dấu chỉ tác động lên toán hạng, không sử dụng được trong các biểu thức*/
void operator-() {
    real = -real;
    image = -image;
}

complex operator+(complex b) {
    complex c;
    c.real=real+b.real;
    c.image=image+b.image;
    return c;
}

friend complex operator+(float x, complex b);
};

complex operator+(float x, complex b) {
    cout<<"Gọi tới operator+(float, complex)\n";
    complex c;
    c.real = x+b.real;
    c.image = b.image;
    return c;
}

void main() {
    clrscr();
    cout<<"so phuc a \n";
    complex a(-2,5);
    cout<<"so phuc b \n";
    complex b(3,4);
    cout<<"Hai so phuc:\n";
    cout<<"a = ";
    a.display();
    cout<<"b = ";
}

```

```
b.display(); //breen = -2+5j
complex c;
-a; //một số phức a và -a là hai số phức đối nhau
cout<<"a = ";a.display();
getch();
}
```

```
so phuc a
so phuc b
Hai so phuc:
a = -2+j*5
b = 3+j*4
a = 2-j*5
```

Chú ý:

Câu lệnh

`complex c;`

`c=-a+b`

sẽ gây lỗi vì `-a` có giá trị `void`.

3. KHẢ NĂNG VÀ GIỚI HẠN CỦA ĐỊNH NGHĨA CHỒNG TOÁN TỬ

Phân lớn toán tử trong C++ đều có thể định nghĩa chồng

Ký hiệu đứng sau từ khoá **operator** phải là một trong số các ký hiệu toán tử áp dụng cho các kiểu dữ liệu cơ sở, không thể dùng các ký hiệu mới. Một số toán tử không thể định nghĩa chồng (chẳng hạn toán tử truy nhập thành phần cấu trúc “.”, toán tử phạm vi “::”, toán tử điều kiện “? :”) và có một số toán tử ta phải tuân theo các ràng buộc sau:

- phép `=`, `[]` nhất định phải được định nghĩa như hàm thành phần của lớp.
- phép `<<` và `>>` dùng với `cout` và `cin` phải được định nghĩa như hàm bạn.
- hai phép toán `++` và `--` có thể sử dụng theo hai cách khác nhau ứng với dạng tiền tố `++a`, `--b` và dạng hậu tố `a++`, `b--`. Điều này đòi hỏi hai hàm toán tử khác nhau.

Các toán tử được định nghĩa chồng phải bảo toàn số ngôi của chính toán tử đó theo cách hiểu thông thường, ví dụ: có thể định nghĩa toán tử `-` một ngôi và hai ngôi trên lớp tương ứng với phép đảo dấu (một ngôi) và phép trừ số học (hai ngôi),

nhưng không thể định nghĩa toán tử gán một ngôi, còn ++ lại cho hai ngôi. Nếu làm vậy, chương trình dịch sẽ hiểu là tạo ra một ký hiệu phép toán mới.

Khi định nghĩa chồng toán tử, phải tuân theo nguyên tắc là **Một trong số các toán hạng phải là đối tượng**. Nói cách khác, hàm toán tử phải :

- (i) hoặc là hàm thành phần, khi đó, hàm đã có một tham số ngầm định có kiểu lớp chính là đối tượng gọi hàm. Tham số ngầm định này đóng vai trò toán hạng đầu tiên (đối với phép toán hai ngôi) hay toán hạng duy nhất (đối với phép toán một ngôi). Do vậy, nếu toán tử là một ngôi thì hàm toán tử thành phần sẽ không chứa một tham số nào khác. Ngược lại khi toán tử là hai ngôi, hàm sẽ có thêm một đối số tường minh.
- (ii) hoặc là một hàm tự do. Trong trường hợp này, ít nhất tham số thứ nhất hoặc tham số thứ hai (nếu có) phải có kiểu lớp.

Hơn nữa, mỗi hàm toán tử chỉ có thể áp dụng với kiểu toán hạng nhất định; cần chú ý rằng các tính chất vốn có, chẳng hạn tính giao hoán của toán tử không thể áp dụng một cách tuỳ tiện cho các toán tử được định nghĩa chồng. Ví dụ:

$a+3.5$

khác với

$3.5+a$

ở đây a là một đối tượng `complex` nào đó.

Cần lưu ý rằng không nên định nghĩa những hàm toán tử khác nhau cùng làm những công việc giống nhau vì dễ xảy ra nhập nhằng. Chẳng hạn, đã có một hàm `operator+` là một hàm thành phần có tham số là đối tượng `complex` thì không được định nghĩa thêm một hàm `operator+` là một hàm tự do có hai tham số là đối tượng `complex`.

Trường hợp các toán tử `++` và `--`

Hàm cho dạng tiền tố	Hàm cho dạng hậu tố
<code>operator++()</code>	<code>operator++(int)</code>
<code>operator--()</code>	<code>operator--(int)</code>

Lưu ý rằng tham số `int` trong dạng hậu tố chỉ mang ý nghĩa lượng trừng (dump type).

Lựa chọn giữa hàm thành phần và hàm bạn

Phải tuân theo các quy tắc sau đây:

- (i) Lưu ý đến hạn chế của chương trình dịch, xem dạng nào được phép.
- (ii) Nếu đối số đầu tiên là một đối tượng, có thể một trong hai dạng. Ngược lại phải dùng hàm bạn.
- (iii) Trái lại, phải dùng hàm bạn.

4. CHIẾN LƯỢC SỬ DỤNG HÀM TOÁN TỬ

Về nguyên tắc, định nghĩa không một phép toán là khá đơn giản, nhưng việc sử dụng phép toán định nghĩa không lại không phải dễ dàng và đòi hỏi phải cân nhắc bởi lẽ nếu bị lạm dụng sẽ làm cho chương trình khó hiểu.

Phải làm sao để các phép toán vẫn giữ được ý nghĩa trực quan nguyên thuỷ của chúng. Chẳng hạn không thể định nghĩa cộng “+” như phép trừ “-” hai giá trị. Phải xác định trước ý nghĩa các phép toán trước khi viết định nghĩa của các hàm toán tử tương ứng.

Các phép toán một ngôi

Các phép toán một ngôi là:

`*`, `&`, `~`, `!`, `++`, `--`, `sizeof` (kiểu)

Các hàm toán tử tương ứng chỉ có một đối số và phải trả về giá trị cùng kiểu với toán hạng, riêng `sizeof` có giá trị trả về kiểu nguyên không dấu và toán tử (kiểu) dùng để trả về một giá trị có kiểu như đã ghi trong dấu ngoặc.

Các phép toán hai ngôi

Các phép toán hai ngôi như:

`*`, `/`, `+`, `-`, `<<`, `>>`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&`, `^`, `&&`, `||`

Hai toán hạng tham gia các phép toán không nhất thiết phải cùng kiểu, mặc dù trong thực tế sử dụng thì thường là như vậy. Như vậy chỉ cần một trong hai đối số của hàm toán tử tương ứng là đối tượng là đủ.

Các phép gán

Các toán tử gán gồm có:

`=`, `+=`, `-=`, `*=`, `/=`, `^=`, `>>=`, `<<=`, `&=`, `^=`, `|=`

Do các toán tử gán được định nghĩa dưới dạng hàm thành phần, nên chỉ có một tham số tường minh và không có ràng buộc gì về kiểu đối số và kiểu giá trị trả về của các phép gán.

Toán tử truy nhập thành phần “->”

Phép toán này được dùng để truy xuất các thành phần của một cấu trúc hay một lớp và cần phân biệt với những cách sử dụng khác để tránh dẫn đến sự nhầm lẫn. Có thể định nghĩa phép toán lấy thành phần giống như đổi với các phép toán một ngôi.

Toán tử truy nhập thành phần theo chỉ số

Toán tử lấy thành phần theo chỉ số được dùng để xác định một thành phần cụ thể trong một khối dữ liệu (cấp phát động hay tĩnh). Thông thường phép toán này được dùng với mảng, nhưng cũng có thể định nghĩa lại nó khi làm việc với các kiểu dữ liệu khác. Chẳng hạn với kiểu dữ liệu vector có thể định nghĩa phép lấy theo chỉ số để trả về một thành phần toạ độ nào đó vector. Và phải được định nghĩa như hàm thành phần có một đối số tương ứng.

Toán tử gọi hàm

Đây là một phép toán thú vị nhưng nói chung rất khó đưa ra một ví dụ cụ thể.

5. MỘT SỐ VÍ DỤ TIÊU BIỂU**5.1 Định nghĩa chông phép gán “=”**

Việc định nghĩa chông phép gán chỉ cần khi các đối tượng có các thành phần dữ liệu động (chương 3 đã đề cập vấn đề này). Chúng ta xét vấn đề này qua phân tích định nghĩa chông phép gán “=” áp dụng cho lớp vector.

Điểm đầu tiên cần lưu ý là hàm operator= nhất thiết phải được định nghĩa như là hàm thành phần của lớp vector. Như vậy hàm operator= sẽ chỉ có một tham số tương ứng (toán hạng bên phải dấu =).

Giả sử a và b là hai đối tượng thuộc lớp vector, khi đó

a=b;

được hiểu là

a.operator=(b);

do đó b được truyền cho hàm dưới dạng tham trị hoặc tham chiếu. Việc truyền bằng tham trị đòi hỏi sự có mặt của hàm thiết lập sao chép, hơn thế nữa sẽ làm cho chương trình chạy chậm vì mất thời gian sao chép một lượng lớn dữ liệu. Vì vậy, b sẽ được truyền cho hàm operator= dưới dạng tham chiếu.

Giá trị trả về của hàm operator= phụ thuộc vào mục đích sử dụng của biểu thức gán. Chúng ta chọn giải pháp trả về tham chiếu của đối tượng đứng bên trái dấu bằng nhằm giữ hai tính chất quan trọng của biểu thức gán: (i) trật tự kết hợp từ bên phải sang trái, (ii) có thể sử dụng kết quả biểu thức gán trong các biểu thức

khác. Ngoài ra giải pháp này cũng hạn chế việc sao chép dữ liệu từ nơi này đi nơi khác trong bộ nhớ.

Chúng ta phân biệt hai trường hợp:

Trường hợp 1

```
a=a;
```

Với hai toán hạng là một. Trong trường hợp này hàm operator= không làm gì, ngoài việc trả về tham chiếu đến a.

Trường hợp 2

```
a=b;
```

khi hai đối tượng tham gia biểu thức gán hoàn toàn khác nhau, việc đầu tiên là phải giải phóng vùng nhớ động chiếm giữ trước đó trong a, trước khi xin cấp phát một vùng nhớ động khác bằng kích thước vùng nhớ động có trong b, cuối cùng sao chép nội dung từ vùng nhớ động trong b sang a. Và không quên "sao chép" giá trị của các thành phần "không động" còn lại.

Ta xét chương trình minh họa.

Ví dụ 4.6

```
/*vector4.cpp*/
#include <iostream.h>
#include <conio.h>
class vector{
    int n; //số lượng của vector
    float *v; //con trỏ tới vùng nhớ toạ độ
public:
    vector(); //hàm thiết lập không tham số
    vector(int size); //hàm thiết lập 1 tham số
    vector(int size, float *a);
    vector(vector &);
    vector & operator=(vector & b);
    ~vector();
    void display();
};
vector::vector()
```

```

{
int i;

cout<<"Tao doi tuong tai "<<this<<endl;
cout<<"So chieu :">>n;
v= new float [n];
cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
}
}

vector::vector(int size)
{
int i;
cout<<"Su dung ham thiet lap 1 tham so\n";
cout<<"Tao doi tuong tai "<<this<<endl;
n=size;
cout<<"So chieu :"<<size<<endl;
v= new float [n];
cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
}
}

vector::vector(int size, float *a )  {
int i;
cout<<"Su dung ham thiet lap 2 tham so\n";
cout<<"Tao doi tuong tai "<<this<<endl;
n=size;
cout<<"So chieu :"<<n<<endl;
v= new float [n];
}

```

```

cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
for(i=0;i<n;i++)
    v[i] = a[i];
}

vector::vector(vector &b) {
    int i;
    cout<<"Su dung ham thiet lap sao chep\n";
    cout<<"Tao doi tuong tai "<<this<<endl;
    v= new float [n=b.n];
    cout<<"Xin cap phat vung bo nho "<<n<<" so thuc tai"<<v<<endl;
    for(i=0;i<n;i++)
        v[i] = b.v[i];
}

vector::~vector() {
    cout<<"Giai phong "<<v<<"cua doi tuong tai"<<this<<endl;
    delete v;
}

vector & vector::operator=(vector & b) {
    cout<<"Goi operator=() cho "<<this<<" va "<<&b<<endl;
    if (this !=&b){
        /*xoá vùng nhớ động đã có trong đối tượng về trùi */
        cout<<"xoá vùng nhớ động"<<v<<" trong "<<this<<endl;
        delete v;
        /*cấp phát vùng nhớ mới có kích thước như trong b*/
        v=new float [n=b.n];
        cout<<"cap phat vung nho dong moi"<<v<<" trong "<<this<<endl;
        for(int i=0;i<n;i++) v[i]=b.v[i];
    }
    /*khi hai doi tuong giống nhau, không làm gì */
    else cout<<"Hai doi tuong la mot\n";
    return *this;
}

```

```

void vector::display() {
    int i;
    cout<<"Doi tuong tai :"<<this<<endl;
    cout<<"So chieu :"<<n<<endl;
    for(i=0;i<n;i++) cout <<v[i] << " ";
    cout <<"\n";
}

```

```

void main() {
    clrscr();
    vector s1;//gọi hàm thiết lập không tham số
    s1.display();
    vector s2 = s1;//gọi hàm thiết lập sao chép
    s2.display();
    vector s3(0);
    s3=s1;//gọi hàm toán tử vector::operator=(...)
    s1=s1;
    getch();
}

```

```

Tao doi tuong tai 0xffff2
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x148c
Toa do thu 1 : 2
Toa do thu 2 : 3
Toa do thu 3 : 2
Doi tuong tai :0xffff2
So chieu :3
2 3 2
Su dung ham thiet lap sao chep
Tao doi tuong tai 0xffee
Xin cap phat vung bo nho 3 so thuc tai0x149c
Doi tuong tai :0xffee
So chieu :3

```

```

2 3 2
Su dung ham thiet lap 1 tham so
Tao doi tuong tai 0xffffea
So chieu :0
Xin cap phat vung bo nho 0 so thuc tai0x14ac
Goi operator=() cho 0xffffea va 0xffff2
xoa vung nho dong0x14ac trong 0xffffea
cap phat vung nho dong moi0x14ac trong 0xffffea
Goi operator=() cho 0xffff2 va 0xffff2
Hai doi tuong la mot

```

5.2 Định nghĩa chồng phép "[]"

Xét chương trình sau:

Ví dụ 4.7

```

/*vector5.cpp*/
#include <iostream.h>
#include <conio.h>
class vector{
    int n; //số giá trị
    float *v; //con trỏ tới vùng nhớ toạ độ
public:
    vector(); //hàm thiết lập không tham số
    vector(vector &);
    int length() { return n; }
    vector & operator=(vector &);
    float & operator[](int i) {
        return v[i];
    }
    ~vector();
};
vector::vector() {
    int i;

```

```

cout<<"So chieu :">>n;
v= new float [n];
}

vector::vector(vector &b) {
    int i;
    v= new float [n=b.n];
    for(i=0;i<n;i++)
        v[i] = b.v[i];
}

vector::~vector() {
    delete v;
}

vector & vector::operator=(vector & b) {
    cout<<"Goi operator=( ) cho "<<this<<" va "<<b<<endl;
    if (this !=&b) {
        /*xoá vùng nhớ dòng đã có trong đối tượng về trái*/
        cout<<"xoá vùng nhớ dòng "<<v<<" trong "<<this<<endl;
        delete v;
        /*cấp phát vùng nhớ mới có kích thước như trong b*/
        v=new float [n=b.n];
        cout<<"cap phat vung nho dong moi "<<v<<" trong "<<this<<endl;
        for(int i=0;i<n;i++) v[i]=b.v[i];
    }
    /*khi hai doi tuong giống nhau, không làm gì */
    else cout<<"Hai doi tuong la mot\n";
    return *this;
}

void Enter_Vector(vector &s) {
    for (int i=0; i<s.length();i++) {
        cout<<"Toa do thu "<<i+1<<" : ";
        cin>>s[i];
    }
}

```

```

    }

void Display_Vector(vector &s) {
    cout<<"So chieu : "<<s.length()<<endl;
    for(int i=0; i<s.length(); i++)
        cout<<s[i]<<" ";
    cout<<endl;
}

void main() {
    clrscr();
    cout<<"Tao doi tuong s1\n";
    vector s1;
    /*Nhập các tọa độ cho vector s1*/
    cout<<"Nhập các tọa độ của s1\n";
    Enter_Vector(s1);
    cout<<"Thông tin về vector s1\n";
    Display_Vector(s1);
    vector s2 = s1;
    cout<<"Thông tin về vector s2\n";
    Display_Vector(s2);
    getch();
}

```

Tạo đối tượng s1
 Số chiều : 4
 Nhập các tọa độ của s1
 Tọa độ thứ 1 : 2
 Tọa độ thứ 2 : 3
 Tọa độ thứ 3 : 2
 Tọa độ thứ 4 : 3
 Thông tin về vector s1
 Số chiều : 4
 2 3 2 3
 Thông tin về vector s2

Số chiều i: 4

2 3 2 3

Nhận xét

- Nhờ giá trị trả về của hàm **operator[]** là tham chiếu đến một thành phần toạ độ của vùng nhớ động nên ta có thể đọc/ghi các thành phần toạ độ của mỗi đối tượng vector. Như vậy có thể sử dụng các đối tượng vector giống như các biến mảng. Trong ví dụ trên chúng ta cũng không cần đến hàm thành phần vector: **:display()** để in ra các thông tin của các đối tượng.
- Có thể cải tiến hàm toán tử **operator[]** bằng cách bổ sung thêm phân kiêm tra tràn chỉ số.

5.3 Định nghĩa chồng << và >>

Có thể định nghĩa chồng hai toán tử vào/ra << và >> cho phép các đối tượng đứng bên phải chúng khi thực hiện các thao tác vào ra. Chương trình sau đưa ra một cách định nghĩa chồng hai toán tử này.

Ví dụ 4.8

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

class complex {
    float real, image;
public:
    friend ostream & operator<<(ostream &o, complex &b);
    friend istream & operator>>(istream &i, complex &b);
};

ostream & operator<<(ostream &os, complex &b) {
    os<<b.real<<(b.image>=0?'+' : '-')<<"j*"<<fabs(b.image)<<endl;
    return os;
}

istream & operator>>(istream &is, complex &b) {
    cout<<"Phan thuc : ";
    is>>b.real;
    cout<<"Phan ao : ";
    is>>b.image;
    return is;
}
```

```

    }
void main() {
    clrscr();
    cout<<"Tao so phuc a\n";
    complex a;
    cin>>a;
    cout<<"Tao so phuc b\n";
    complex b;
    cin >>b;
    cout<<"In hai so phuc\n";
    cout<<"a = "<<a;
    cout<<"b = "<<b;
    getch();
}

```

Tạo số phức a

Phan thuc : 3

Phan ao : 4

Tạo số phức b

Phan thuc : 5

Phan ao : 3

In hai so phuc

a = 3+j*4

b = 5+j*3

Nhận xét

- Trong chương trình trên, ta không thấy các hàm thiết lập tường minh để gán giá trị cho các đối tượng. Thực tế, việc gán các giá trị cho các đối tượng được đảm nhiệm bởi hàm toán tử **operator>>**.
- Việc hiển thị nội dung của các đối tượng số phức có trước đây do hàm thành phần **display()** đảm nhiệm thì nay đã có thể thay thế nhờ hàm toán tử **operator<<**.
- Hai hàm **operator<<** và **operator>>** cho phép sử dụng **cout** và **cin** cùng lúc với nhiều đối tượng khác nhau: giá trị số nguyên, số thực, xâu ký tự, ký tự và các đối tượng của lớp **complex**. Có thể thử nghiệm các cách khác để thấy được rằng giải pháp đưa ra trong chương trình trên là tốt nhất.

5.4 Định nghĩa chông các toán tử **new** và **delete**

Các toán tử **new** và **delete** được định nghĩa cho từng lớp và chúng chỉ có ảnh hưởng đối với các lớp liên quan, còn các lớp khác vẫn sử dụng các toán tử **new** và **delete** như bình thường.

Định nghĩa chông toán tử **new** buộc phải sử dụng hàm thành phần và đáp ứng các ràng buộc sau:

- (i) có một tham số kiểu **size_t** (trong tệp tiêu đề **stddef.h**). Tham số này tương ứng với kích thước (tính theo byte) của đối tượng xin cấp phát. Lưu ý rằng đây là tham số giả (dummy argument) vì nó sẽ không được mô tả khi gọi tới toán tử **new**, mà do chương trình biên dịch tự động tính dựa trên kích thước của đối tượng liên đới.
- (ii) trả về một giá trị kiểu **void *** tương ứng với địa chỉ vùng nhớ động được cấp phát.

Khi định nghĩa chông toán **delete** ta phải sử dụng hàm thành phần, tuân theo các quy tắc sau đây:

- (i) nhận một tham số kiểu con trỏ tới lớp tương ứng; con trỏ này mang địa chỉ vùng nhớ động đã được cấp phát cần giải phóng,
- (ii) không có giá trị trả về (trả về **void**)

Nhận xét

Có thể gọi được các toán tử **new** và **delete** chuẩn (ngay cả khi chúng đã được định nghĩa chông) thông qua toán tử **phạm vi**.

Các toán tử **new** và **delete** là các hàm thành phần **static** của các lớp bởi vì chúng không có tham số ngầm định.

Sau đây giới thiệu ví dụ định nghĩa chông các toán tử **new** và **delete** trên lớp **point**. Ví dụ cũng chỉ ra cách gọi lại các toán tử **new** và **delete** truyền thống.

Ví dụ 4.9

```
/*newdelete.cpp*/
#include <iostream.h>
#include <stddef.h>
#include <conio.h>
class point {
    static int npt; /*số điểm tĩnh*/
    static int npt_dyn; /*số điểm động*/
    int x, y;
```

```

public:
    point(int ox=0, int oy = 0) {
        x = ox; y = oy;
        npt++;
        cout<<"++Tong so diem : "<<npt<<endl;
    }

    ~point () {
        npt--;
        cout<<"--Tong so diem : "<<npt<<endl;
    }

    void * operator new (size_t sz) {
        npt_dyn++;
        cout<<"      Co "<<npt_dyn<<" diem dong "<<endl;
        return ::new char [sz];
    }

    void operator delete (void *dp) {
        npt_dyn--;
        cout<<"      Co "<<npt_dyn<<" diem dong "<<endl;
        ::delete (dp);
    }
};

int point::npt = 0;
int point::npt_dyn = 0;

void main() {
    clrscr();
    point * p1, *p2;
    point a(3,5);
    p1 = new point(1,3);
    point b;
    p2 = new point (2,0);
    delete p1;
    point c(2);
}

```

```

    delete p2;
    getch();
}

```

```

++Tong so diem : 1
    Co 1 diem dong
++Tong so diem : 2
++Tong so diem : 3
    Co 2 diem dong
++Tong so diem : 4
--Tong so diem : 3
    Co 1 diem dong
++Tong so diem : 4
--Tong so diem : 3
    Co 0 diem dong

```

Nhận xét

Dù cho **new** có được định nghĩa chồng hay không, lời gọi tới **new** luôn luôn cần đến các hàm thiết lập.

5.5 Phép nhận ma trận véc tơ

Chương trình vectmat4.cpp sau đây được cải tiến từ vectmat3.cpp trong đó hàm prod() được thay thế bởi **operator***.

Ví dụ 4.10

```

/*vectmat4.cpp*/
#include <iostream.h>
#include <conio.h>
class matrix; /*khai báo trước lớp matrix*/
/*lớp vector*/
class vector{
    static int n; //số chiều của vector
    float *v; //vùng nhớ chứa các toạ độ
public:
    vector();

```

```

vector(float *);
vector(vector &); //hàm thiết lập sao chép
~vector();
void display();
static int & Size() {return n;}
friend vector operator*(matrix &, vector &);
friend class matrix;
};

int vector::n = 0;
/*các hàm thành phần của lớp vector*/
vector::vector() {
    int i;
    v= new float [n];
    for(i=0;i<n;i++) {
        cout<<"Toa do thu "<<i+1<<" : ";
        cin>>v[i];
    }
}

vector::vector(float *a) {
    for(int i =0; i<n; i++)
        v[i]=a[i];
}

vector::vector(vector &b) {
    int i;
    for(i=0;i<n;i++)
        v[i] = b.v[i];
}

vector::~vector() {
    delete v;
}

void vector::display() //hiển thị kết quả
{

```

```
for(int i=0;i<n;i++)
    cout <<v[i] <<" ";
cout <<"\n";
}

/* lớp matrix*/
class matrix {
    static int n; //số chiều của vector.
    vector *m; //vùng nhớ chứa các tog độ
public:
    matrix();
    matrix(matrix &); //hàm thiết lập sao chép
    ~matrix();
    void display();
    static int &Size() {return n;}
    friend vector operator*(matrix &, vector &);
};

int matrix::n =0;
/*hàm thành phần của lớp matrix*/
matrix::matrix(){
    int i;
    m= new vector [n];
}

matrix::matrix(matrix &b) {
    int i,j;
    m= new vector[n];
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            m[i].v[j]=b.m[i].v[j];
}

matrix::~matrix() {
    delete m;
```

```
    }
void matrix::display()  {
for (int i=0; i<n; i++)
    m[i].display();
}
/*hàm toán tử*/
vector operator*(matrix &m, vector &v) {
    float *a = new float [vector::Size()];
    int i,j;
    for (i=0; i<matrix::Size(); i++) {
        a[i]=0;
        for(j=0; j<vector::Size(); j++)
            a[i]+=m.m[i].v[j]*v.v[j];
    }
    return vector(a);
}
void main() {
    clrscr();
    int size;
    cout<<"Kich thuoc cua vector "; cin>>size;
    vector::Size() = size;
    matrix::Size() = size;
    cout<<"Tao mot vector \n";
    vector v;
    cout<<" v= \n";
    v.display();
    cout<<"Tao mot ma tran \n";
    matrix m;
    cout<<" m = \n";
    m.display();
    cout<<"Tich m*v \n";
    vector u = m*v; /* opertaor*(m,v) */
```

```
u.display();
getch();
}
```

Kich thuoc cua vector 4

Tao mot vector

Toa do thu 1 : 1

Toa do thu 2 : 2

Toa do thu 3 : 3

Toa do thu 4 : 4

v=

1 2 3 4

Tao mot ma tran

Toa do thu 1 : 3

Toa do thu 2 : 2

Toa do thu 3 : 1

Toa do thu 4 : 2

Toa do thu 1 : 3

Toa do thu 2 : 2

Toa do thu 3 : 3

Toa do thu 4 : 2

Toa do thu 1 : 3

Toa do thu 2 : 2

Toa do thu 3 : 3

Toa do thu 4 : 2

Toa do thu 1 : 2

Toa do thu 2 : 3

Toa do thu 3 : 2

Toa do thu 4 : 3

m =

3 2 1 2

3 2 3 2

3 2 3 2

2	3	2	3
---	---	---	---

Tích m*v

18	24	24	26
----	----	----	----

6. CHUYỂN ĐỔI KIỂU

Với các kiểu dữ liệu chuẩn, ta có thể thực hiện các phép chuyển kiểu ngầm định, chẳng hạn có thể gán một giá trị **int** vào một biến **long**, hoặc cộng giá trị **long** vào một biến **float**. Thường có hai kiểu chuyển kiểu: chuyển kiểu ngầm định(tự động) và chuyển kiểu tường minh (ép kiểu). Phép chuyển kiểu ngầm định được thực hiện bởi chương trình biên dịch. Phép chuyển kiểu tường minh xảy ra khi sử dụng phép ép kiểu bắt buộc. Phép ép kiểu thường được dùng trong các câu lệnh gọi hàm để gửi các tham số có kiểu khác với các tham số hình thức tương ứng.

Các kiểu lớp không thể thoải mái chuyển sang các kiểu khác được mà phải do người tự làm lấy. C++ cũng cung cấp cách thức định nghĩa phép chuyển kiểu ngầm định và tường minh. Phép chuyển kiểu ngầm định được định nghĩa bằng một hàm thiết lập chuyển kiểu (conversion constructor), còn phép chuyển kiểu tường minh được xác định thông qua toán tử chuyển kiểu hoặc ép kiểu (cast operator).

Phép chuyển kiểu ngầm định được định nghĩa thông qua một hàm thiết lập chuyển kiểu cho lớp. Với đối số có kiểu cần phải chuyển thành một đối tượng của lớp đó. Tham số này có thể có kiểu cơ sở hay là một đối tượng thuộc lớp khác. Hàm thiết lập một tham số trong lớp point trong các chương trình point?.cpp ở chương trước là ví dụ cho hàm thiết lập chuyển kiểu.

Trong chỉ thị

```
point p=2;
```

đã chuyển kiểu từ giá trị nguyên 2 sang một đối tượng point. Thực tế ở đây chương trình dịch gọi tới hàm thiết lập một tham số. Đây là sự chuyển kiểu một chiều, nhận giá trị hoặc đối tượng nào đó và chuyển nó thành đối tượng của lớp. Các hàm thiết lập chuyển kiểu không thể sử dụng để chuyển các đối tượng của lớp mình sang các kiểu khác và chúng chỉ có thể được sử dụng trong các phép gán và phép khởi tạo giá trị.

Tuy nhiên, các toán tử chuyển kiểu có thể được dùng để chuyển các đối tượng sang các kiểu khác và cũng có thể được dùng cho các mục đích khác ngoài phép gán và khởi tạo giá trị. C++ qui định rằng một hàm toán tử chuyển kiểu như thế buộc phải là hàm thành phần của lớp liên quan và không có tham số hoặc kiểu trả về. Tên của nó được cho theo dạng như sau:

- operator type();

trong đó **type** là tên của kiểu dữ liệu mà một đối tượng sẽ được chuyển sang; có thể là kiểu dữ liệu cơ sở (khi đó ta phải chuyển kiểu từ đối tượng sang kiểu cơ

sở) hay một kiểu lớp khác (khi đó ta phải chuyển kiểu từ đối tượng lớp này sang lớp khác).

6.1 Hàm toán tử chuyển kiểu ép buộc

Chương trình complex6.cpp sau đây minh họa cách cài đặt các hàm toán tử chuyển kiểu ngầm định và chuyển kiểu từ lớp complex sang một số thực. Vấn đề chuyển kiểu từ lớp này sang lớp khác sẽ được giới thiệu sau.

Ví dụ 4.11

```
/*complex6.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
public:
    complex( ) {
        real = 0; image = 0;
    }
    /*hàm thiết lập đóng vai trò một hàm toán tử chuyển kiểu tự động*/
    complex(float r) {
        real = r; image = 0;
    }
    complex(float r, float i ) {
        real = r; image = i;
    }
    /*Hàm toán tử chuyển kiểu ép buộc*/
    operator float() {
        return real;
    }
    void display() {
        cout<<real<<(image>=0?'+' : '-')<<"j*"<<fabs(image)<<endl;
    }
    /*hàm operator+ định nghĩa phép toán + hai ngôi trên lớp số phức complex*/
}
```

```

    complex operator+(complex b){ //operator+ là tên của lớp mà chúng ta định nghĩa
        complex c;
        c.real = real+b.real;
        c.image = image+b.image;
        return c;
    }
};

void main() {
    clrscr();
    cout<<"a = ";
    complex a(-2,5);
    a.display();
    cout<<"b = ";
    complex b(3,4);
    b.display();
    cout<<"c = a+b : ";
    complex c;
    c=a+b;//a.operator+(b), với a là đối số và b là tham số
    c.display();
    cout<<"d = 3+c : ";
    complex d;
    d= complex(3)+c;
    d.display();
    cout<<"float x = a : ";
    float x = a;//float(complex)
    cout<<x<<endl;
    getch();
}

```

```

a = -2+j*5
b = 3+j*4
c = a+b : 1+j*9
d = 3+c : 4+j*9

```

```
float x = a : -2
```

Chú ý:

- Phép cộng $3+c$ khác với $\text{complex}(3)+c$ vì trong phép thứ nhất người ta thực hiện chuyển đổi c thành số thực và phép cộng đó thực hiện giữa hai số thực. Có thể thử nghiệm để kiểm tra lại kết quả sau:
- Đoạn chương trình

```
d = 3 + c;
```

```
d.display()
```

cho ra kết quả

```
4+j*0
```

Các phép toán nguyên thuỷ có độ ưu tiên hơn so với các phép toán được định nghĩa chồng.

6.1.1 Hàm toán tử chuyển kiểu trong lời gọi hàm

Trong chương trình dưới đây ta định nghĩa hàm `fct()` với một tham số thực (`float`) và sẽ gọi hàm này hai lần: lần thứ nhất với một tham số thực, lần thứ hai với một tham số kiểu `complex`.

Trong lớp `complex` còn có một hàm thiết lập sao chép, không được gọi khi ta truyền đối tượng `complex` cho hàm `fct()` vì ở đây xảy ra sự chuyển đổi kiểu dữ liệu.

Ví dụ 4.12

```
/*complex7.cpp*/
#include <iostream.h>
#include <conio.h>
class complex {
    float real, image;
public:
    complex(float r, float i) {
        real = r; image = i;
    }
    complex(complex &b) {
        cout<<"Ham thiet lap sao chep\n";
        real = b.r; image = b.i;
    }
    void display() {
        cout<<"(" << real << ", " << image << ")\n";
    }
}
```

```

    }

/*Hàm toán tử chuyển kiểu ép buộc*/
operator float() {
    float real;
    cout<<"Gọi float() cho complex\n";
    real = image;
    return real;
}

void display() {
    cout<<real<<(image>=0?'+':'-')<<"j"<<fabs(image)<<endl;
}

};

void fct(float n) {
    cout<<"Gọi fct voi tham so : "<<n<<endl;
}

void main() {
    clrscr();
    complex a(3,4);
    fct(6); //lời gọi hàm thông thường
    fct(a); //lời gọi hàm có xảy ra chuyển đổi kiểu dữ liệu
    getch();
}

```

```

Gọi fct voi tham so : 6
Gọi float() cho complex
Gọi fct voi tham so : 3

```

Trong chương trình này, lời gọi hàm

`fct(a)`

đã được chương trình dịch chuyển thành các thao tác:

- (i) chuyển đổi đối tượng thành **float**,
- (ii) lời gọi hàm `fct()` với tham số là giá trị thu được sau chuyển đổi.

Sự chuyển đổi được thực hiện khi gọi hàm do đó không xảy ra việc sao chép lại đối tượng `a`.

6.1.2 Hàm toán tử chuyển kiểu trong biến thức

Chương trình dưới đây cho ta biết biến thức dạng $a+b$ hoặc $a+b$ được tính như thế nào với a , b là các đối tượng kiểu complex.

Ví dụ 4.13

```
/*complex8.cpp*/
#include <iostream.h>
#include <conio.h>

class complex {
    float real, image;
public:
    complex(float r, float i) {
        real = r; image = i;
    }
    /*Hàm toán tử chuyển kiểu ép buộc*/
    operator float() {
        cout<<"Gọi float() cho complex\n";
        return real;
    }
    void display() {
        cout<<real<<(image>=0?'+' : '-')<<"j*"<<fabs(image)<<endl;
    }
};

void main() {
    clrscr();
    complex a(3,4);
    complex b(5,7);
    float n1, n2;
    n1 = a+3; cout<<"n1 = "<<n1<<endl;
    n2 = a + b; cout<<"n2 = "<<n2<<endl;
    double z1, z2;
    z1 = a+3; cout<<"z1 = "<<z1<<endl;
    z2 = a + b; cout<<"z2 = "<<z2<<endl;
}
```

```

getch();
}

tín chất xanh +, và điều gì sẽ happen nếu ta chỉ gán số 1 cho z1 và 2 cho z2?
Gọi float() cho complex
n1 = 6
Gọi float() cho complex
Gọi float() cho complex
n2 = 8
Gọi float() cho complex
z1 = 6
Gọi float() cho complex
Gọi float() cho complex
z2 = 8

```

Khi gặp biểu thức dạng $a+3$ với phép toán $+$ được định nghĩa với các toán hạng có kiểu lớp `complex` và số thực, chương trình dịch trước hết đi tìm xem đã có một toán tử $+$ được định nghĩa chồng tương ứng với các kiểu dữ liệu của các toán hạng này hay chưa. Trong trường hợp này vì không có, nên chương trình dịch sẽ chuyển đổi kiểu dữ liệu của các toán hạng để phù hợp với một trong số các phép toán đã định nghĩa, cụ thể là chuyển đổi từ đối tượng `a` sang `float`.

6.2 Hàm toán tử chuyển đổi kiểu cơ sở sang kiểu lớp

Trở lại chương trình `complex6.cpp`, ta có thể thực hiện các chỉ thị kiểu như:

`complex e=10;`

hoặc

`a=1;`

Chỉ thị thứ nhất nhằm tạo một đối tượng tạm thời có kiểu `complex` tương ứng với phần thực bằng 10, phần ảo bằng 0 rồi sao chép sang đối tượng `e` mới được khai báo. Trong chỉ thị thứ hai, cũng có một đối tượng tạm thời kiểu `complex` được tạo ra và nội dung của nó (phần thực 1, phần ảo 0) được gán cho `a`. Như vậy, trong cả hai trường hợp đều phải gọi tới hàm thiết lập một tham số của lớp `complex`.

Tương tự, nếu có hàm `fct()` với khai báo:

`fct(complex)`

thì lời gọi

`fct(4)`

sẽ đòi hỏi phải chuyển đổi từ giá trị nguyên 4 thành một đối tượng tạm thời có kiểu complex, để truyền cho fct(). Sau đây là chương trình nhận được do sửa đổi từ complex6.cpp.

Ví dụ 4.14

```
/*complex9.cpp*/
#include <iostream.h>
#include <conio.h>
class complex {
    float real, image;
public:
    complex(float r) {
        cout<<"Ham thiet lap dong vai tro cua ham toan tu chuyen kieu ngam
dinh\n";
        real = r; image = 0;
    }
    complex(float r, float i ) {
        cout<<"Ham thiet lap \n";
        real = r; image = i;
    }
    complex(complex &b) {
        cout<<"Ham thiet lap sao chep lai "<<&b<<" Sang "<<this<<endl;
        real = b.real; image = b.image;
    }
};

void fct(complex p) {
    cout<<"Goi fct \n";
}

void main() {
    clrscr();
    complex a(3,4);
    a = complex(12);
    a = 12;
    fct(4);
```

```
getch();  
}
```

Hàm thiết lập

Hàm thiết lập đóng vai trò của hàm toán tử chuyển kiểu ngầm định

Hàm thiết lập đóng vai trò của hàm toán tử chuyển kiểu ngầm định

Hàm thiết lập đóng vai trò của hàm toán tử chuyển kiểu ngầm định

Gọi fct

6.2.1 Hàm thiết lập trong các chuyển đổi kiểu liên tiếp

Trong lớp số phức complex hàm thiết lập với một tham số cho phép thực hiện chuyển đổi float -->complex đồng thời cả chuyển đổi ngầm định

int --> float tức là nó có thể cho phép một chuỗi các chuyển đổi:

int --> float -->complex

Chẳng hạn khi gặp phép gán kiểu như:

a = 2;

Cần chú ý khả năng chuyển đổi này phải dựa trên các quy tắc chuyển đổi thông thường như đã nói ở trên.

6.2.2 Lựa chọn giữa hàm thiết lập và phép toán gán

Với chỉ thị gán:

a=12;

trong chương trình complex9.cpp không có định nghĩa toán tử gán một số nguyên cho một đối tượng complex. Giả sử có một toán tử gán như vậy thì có thể sẽ xảy ra xung đột giữa:

- (i) chuyển đổi **float-->complex** bởi hàm thiết lập và phép gán **complex->complex**.
- (ii) sử dụng trực tiếp toán tử gán **float-->complex**.

Để giải quyết vấn đề này ta tuân theo quy tắc: "Các chuyển đổi do người sử dụng định nghĩa chỉ được thực hiện khi cần thiết", nghĩa là với chỉ thị gán:

a = 12;

nếu như trong lớp complex có định nghĩa toán tử gán, nó sẽ được ưu tiên thực hiện. Chương trình sau đây minh họa nhận xét này:

Ví dụ 4.15

```
/*complex10.cpp*/
#include <iostream.h>
#include <conio.h>
class complex {
    float real, image;
public:
    complex(float r) {
        cout<<"Ham thiet lap dong vai tro cua ham toan tu chuyen kieu ngam
dinh\n";
        real = r; image = 0;
    }
    complex(float r, float i ) {
        cout<<"Ham thiet lap \n";
        real = r; image = i;
    }
    complex & operator=(complex &p) {
        real = p.real;image = p.image;
        cout<<"gan complex -->complex tu "<<&p<<" sang "<<this<<endl;
        return *this;
    }
    complex & operator=(float n) {
        real = n;image = 0;
        cout<<"gan float -->complex "<<endl;
        return *this;
    }
};
void main() {
    clrscr();
    complex a(3,4);
```

```
a = 12;
getch();
}
```

Hàm thiết lập
gan float -->complex

6.2.3 Sử dụng hàm thiết lập để mở rộng ý nghĩa một phép toán

Ta xét lớp `complex` và hàm thiết lập một tham số của lớp được bổ sung thêm một hàm toán tử dưới dạng hàm bạn (trường hợp này không nên sử dụng hàm toán tử thành phần). Với các điều kiện này, khi `a` là một đối tượng kiểu `complex`, biểu thức kiểu như:

`a + 3`

sẽ có ý nghĩa. Thực vậy trong trường hợp này chương trình dịch sẽ thực hiện các thao tác sau:

- chuyển đổi từ số thực 3 sang số phức `complex`.
- cộng giữa đối tượng nhận được với `a` bằng cách gọi hàm toán tử `operator+`.

Kết quả sẽ là một đối tượng kiểu `complex`. Nói cách khác, biểu thức `a + 3` tương đương với

`operator+(a, point(3))`.

Tương tự, biểu thức

`5 + a`

sẽ tương đương với:

`operator+(point(5), a)`.

Tuy nhiên trường hợp sau sẽ không còn đúng khi `operator+` là hàm toán tử thành phần. Sau đây là một chương trình minh họa các khả năng mà chúng ta vừa đề cập.

Ví dụ 4.16

```
/*complex11.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
```

```

    float real, image;
public:
    complex(float r) {
        cout<<"Ham thiet lap dong vai tro cua ham toan tu chuyen kieu ngam
dinh\n";
        real = r; image = 0;
    }
    complex(float r, float i ) {
        cout<<"Ham thiet lap 2 tham so\n";
        real = r; image = i;
    }
    void display() {
        cout<<real<<(image>=0?"+j*":"-j*")<<fabs(image)<<endl;
    }
    friend complex operator+(complex , complex);
}
complex operator+(complex a, complex b) {
    complex c(0,0);
    c.real = a.real + b.real;
    c.image = a.image + b.image;
    return c;
}
void main() {
    clrscr();
    complex a(3,4),b(9,4);
    a = b + 5;a.display();
    a = 2 + b;a.display();
    getch();
}

```

Ham thiet lap 2 tham so

Ham thiet lap 2 tham so

```

Ham thiet lap dong vai tro cua ham toan tu chuyen kieu ngam
dinh
Ham thiet lap 2 tham so
14+j*4
Ham thiet lap dong vai tro cua ham toan tu chuyen kieu ngam
dinh
Ham thiet lap 2 tham so
11+j*4

```

Nhận xét

Hàm thiết lập làm nhiệm vụ của hàm toán tử chuyển đổi kiểu cơ sở sang kiểu lớp không nhất thiết chỉ có một tham số hình thức. Trong trường hợp hàm thiết lập có nhiều tham số hơn, các tham số tính từ tham số thứ hai phải có giá trị ngầm định.

1.1 Chuyển đổi kiểu từ lớp này sang một lớp khác

Khả năng chuyển đổi qua lại giữa kiểu cơ sở và một kiểu lớp có thể được mở rộng cho hai kiểu lớp khác nhau:

- (i) Trong lớp A, ta có thể định nghĩa một hàm toán tử để thực hiện chuyển đổi từ kiểu A sang kiểu B (cast operator).
- (ii) Hàm thiết lập của lớp A chỉ với một tham số kiểu B sẽ thực hiện chuyển đổi kiểu từ B sang A.

6.3.1. Hàm toán tử chuyển kiểu bắt buộc

Ví dụ sau đây minh họa khả năng dùng hàm toán tử `complex` của lớp `point` cho phép thực hiện chuyển đổi một đối tượng kiểu `point` thành một đối tượng kiểu `complex`.

Ví dụ 4.17

```

/*pointcomplex1.cpp*/
#include <iostream.h>
#include <conio.h>

```

```
#include <math.h>
class complex; //khai báo trước lớp complex
class point {
    int x, y;
public:
    point(int ox = 0, int oy = 0) {x = ox; y = oy;}
    operator complex(); //chuyển đổi point-->complex
};

class complex {
    float real, image;
public:
    complex(float r=0, float i=0) {
        real = r; image = i;
    }
    friend point::operator complex();
    void display() {
        cout<<real<<(image>=0?"+j*":"-j*")<<fabs(image)<<endl;
    }
};

point::operator complex() {
    complex r(x,y);
    cout<<"Chuyen doi "<<x<<" "<<y
        <<" thanh "<<r.real<<" + "<< r.image<<endl;
    return r;
}

void main() {
    clrscr();
    point a(2,5); complex c;
    c = (complex) a; c.display();
```

```

c = (complex) a; c.display();
point b(9,12);
c = b; c.display();
getch();
}

```

```

Chuyển đổi 2 5 thành 2 + 5
2+j*5
Chuyển đổi 9 12 thành 9 + 12
9+j*12

```

6.3.2 Hàm thiết lập dùng làm hàm toán tử

Chương trình sau đây minh họa khả năng dùng hàm thiết lập complex(point) biếu để thực hiện chuyển đổi một đối tượng kiểu point thành một đối tượng kiểu complex.

Ví dụ 4.18

```

/*pointcomplex2.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>

class point;//khai báo trước lớp complex

class complex {
    float real, image;
public:
    complex(float r=0, float i=0 ) {
        real = r; image = i;
    }
    complex(point);
    void display() {
        cout<<real<<(image>=0?"+"j*":"-j*")<<fabs (image)<<endl;
    }
};

class point {

```

```

int x, y;
public:
    point(int ox = 0, int oy = 0) {x = ox; y = oy;}
    friend complex::complex(point);
};

int main() {
    clrscr();
    point a(3,5);
    complex c = a; c.display();
    getch();
}

```

3+4*5

7. TÓM TẮT

7.1 Ghi chú

Toán tử được định nghĩa chung bằng cách định nghĩa một hàm toán tử. Tên hàm toán tử bao gồm từ khóa **operator** theo sau là ký hiệu của toán tử được định nghĩa chung.

Hầu hết các toán tử của C++ đều có thể định nghĩa chung. Không thể tạo ra các ký hiệu phép toán mới.

Phải đảm bảo các đặc tính nguyên thuỷ của toán tử được định nghĩa ch้อง, chẳng hạn: độ ưu tiên, trật tự kết hợp, số ngôi.

Không sử dụng tham số có giá trị ngầm định để định nghĩa ch่อง toán tử.

Các toán tử (\cdot) , $[\cdot]$, \rightarrow , $=$ yêu cầu hàm toán tử phải là **hàm thành phần** của lớp.

Hàm toán tử có thể là hàm thành phần hoặc hàm ban của lớp.

Khi hàm toán tử là hàm thành phần, toán hạng bên trái luôn là đối tượng thuộc lớp.

Nếu toán hạng bên trái là đối tượng của lớp khác thì hàm toán tử tương ứng phải là hàm ban.

Chương trình đích không tự biết cách chuyển kiểu giữa kiểu dữ liệu chuẩn và kiểu dữ liệu tự định nghĩa. Vì vậy người lập trình cần phải mô tả tường minh các chuyển đổi này dưới dạng hàm thiết lập chuyển kiểu hay hàm toán tử chuyển kiểu.

Một hàm toán tử chuyển kiểu thực hiện chuyển đổi từ một đối tượng thuộc lớp sang đối tượng thuộc lớp khác hoặc một đối tượng có kiểu được định nghĩa trước.

Hàm thiết lập chuyển kiểu có một tham số và thực hiện chuyển đổi từ một giá trị sang đối tượng kiểu lớp.

Toán tử gán là toán tử hay được định nghĩa chung nhất, đặc biệt khi lớp có các thành phần dữ liệu động.

Để định nghĩa chung toán tử tăng, giảm một ngõi, phải phân biệt hai hàm toán tử tương ứng cho dạng tiền tố và dạng hậu tố.

7.2 Các lỗi thường gặp

Tạo một toán tử mới.

Thay đổi định nghĩa của các toán tử trên các kiểu được định nghĩa trước.

Cho rằng việc định nghĩa chung một toán tử sẽ tự động kéo theo định nghĩa chung của các toán tử liên quan.

Quên định nghĩa chung toán tử gán và hàm thiết lập sao chép cho các lớp có các thành phần dữ liệu động.

7.3 Một số thói quen lập trình tốt

Sử dụng toán tử định nghĩa chung khi điều đó làm cho chương trình trong sáng hơn.

Tránh lạm dụng định nghĩa chung toán tử vì điều đó dẫn đến khó kiểm soát chương trình.

Chú ý đến các tính chất nguyên thuỷ của toán tử được định nghĩa chung.

Hàm thiết lập, toán tử gán, hàm thiết lập sao chép của một lớp thường đi cùng nhau.

8. BÀI TẬP

Bài tập 4.1.

Bổ sung thêm một số toán tử trên lớp số phức complex.

Định nghĩa hai phép toán vào ra trên lớp vector.

Bài tập 4.2.

Một ma trận được hiểu là một vector mà mỗi thành phần của nó lại là một vector. Theo tinh thần đó hãy định nghĩa lớp matrix dựa trên vector. Tìm cách để cho chương trình dịch hiểu được phép truy nhập

$m[i][j]$ trong đó m là một đối tượng thuộc lớp matrix.

Bài tập 4.3.

Định nghĩa các phép nhân, cộng, trừ trên các ma trận vuông.

Bài tập 4.4.

Dựa trên định nghĩa của lớp complex , lớp vector và lớp matrix để xây dựng chương trình mô phỏng các thao tác trên ma trận và vector phức.

Bài tập 4.5.

Thay thế hàm thành phần tick () trong lớp date_time bài tập 3.10 bởi hàm toán tử tương ứng với phép toán ++.

Bài tập 4.6.

Tạo lớp phân số PS với các khả năng sau:

- (i) Tạo một hàm thiết lập với mẫu số dương, ở dạng tối giản hoặc rút gọn về dạng tối giản.
- (ii) Định nghĩa chia các toán tử cộng, trừ, nhân, chia cho lớp này.
- (iii) Định nghĩa chia các toán tử quan hệ trên lớp số phức.

KỸ THUẬT THÙA KẾ (Inheritance)

Mục đích chương này:

1. Cài đặt sự thừa kế.
2. Sử dụng các thành phần của lớp cơ sở.
3. Định nghĩa lại các hàm thành phần.
4. Truyền thông tin giữa các hàm thiết lập của lớp dẫn xuất và lớp cơ sở.
5. Các loại dẫn xuất khác nhau và sự thay đổi trạng thái của các thành phần lớp cơ sở.
6. Sự tương thích giữa các đối tượng của lớp dẫn xuất và lớp cơ sở.
7. Toán tử gán và thừa kế.
8. Hàm ảo và tính đa hình.
9. Hàm ảo thuần tuý và lớp cơ sở trừu tượng.
10. Đa thừa kế và các vấn đề liên quan.

1. GIỚI THIỆU CHUNG

Thừa kế là một trong bốn nguyên tắc cơ sở của phương pháp lập trình hướng đối tượng. Đặc biệt đây là cơ sở cho việc nâng cao khả năng sử dụng lại các bộ phận của chương trình. Thừa kế cho phép ta định nghĩa một lớp mới, gọi là lớp dẫn xuất, từ một lớp đã có, gọi là lớp cơ sở. Lớp dẫn xuất sẽ thừa kế các thành phần (đữ liệu, hàm) của lớp cơ sở, đồng thời thêm vào các thành phần mới, bao hàm cả việc làm “tốt hơn” hoặc làm lại những công việc mà trong lớp cơ sở chưa làm tốt hoặc không còn phù hợp với lớp dẫn xuất. Chẳng hạn có thể định nghĩa lớp “mặt hàng nhập khẩu” dựa trên lớp “mặt hàng”, bằng cách bổ sung thêm thuộc tính “thuế”. Khi đó cách tính chênh lệch giá bán, mua cũ trong lớp “mặt hàng” sẽ không phù hợp nữa nên cần phải sửa lại cho phù hợp. Lớp điểm có màu được định nghĩa dựa trên lớp điểm không màu bằng cách bổ sung thêm thuộc tính màu, hàm `display()` lúc này ngoài việc hiển thị hai thành phần тоạ độ còn phải cho biết màu của đối tượng điểm. Trong cả hai ví dụ đưa ra, trong lớp dẫn xuất đều có sự bổ sung và thay đổi thích hợp với tính hình mới.

Thừa kế cho phép không cần phải biên dịch lại các thành phần chương trình vốn đã có trong các lớp cơ sở và hơn thế nữa không cần phải có chương trình nguồn tương ứng. Kỹ thuật này cho phép chúng ta phát triển các công cụ mới dựa trên

những gì đã có được. Người sử dụng Borland C hay Turbo Pascal 6.0/7.0 rất thích sử dụng Turbo Vision - một thư viện cung cấp các lớp, đối tượng là cơ sở để xây dựng các giao diện ứng dụng hết sức thân thiện đối với người sử dụng. Tất cả các lớp này đều được cung cấp dưới dạng các tập tin *.obj, *.lib nghĩa là người sử dụng hoàn toàn không thể và không cần phải biết rõ phần chương trình nguồn tương ứng. Tuy nhiên điều đó không quan trọng khi người lập trình được phép thừa kế các lớp định nghĩa trước đó.

Thừa kế cũng cho phép nhiều lớp có thể dẫn xuất từ cùng một lớp cơ sở, nhưng không chỉ giới hạn ở một mức: một lớp dẫn xuất có thể là lớp cơ sở cho các lớp dẫn xuất khác. Ở đây ta thấy rằng khái niệm thừa kế giống như công cụ cho phép mô tả cụ thể hóa các khái niệm theo nghĩa: lớp dẫn xuất là một cụ thể hóa hơn nữa của lớp cơ sở và nếu bỏ đi các đặc biệt trong các lớp dẫn xuất sẽ chỉ còn các đặc điểm chung nằm trong lớp cơ sở.

Hình 5.1 mô tả một sơ đồ thừa kế của các lớp, có cung đi từ lớp này sang lớp kia nếu chúng có quan hệ thừa kế. Ta gọi đó là đồ thị thừa kế. Sau đây là một số mô tả cho các lớp xuất hiện trong đồ thị thừa kế ở trên.

1. Lớp mặt hàng

các thuộc tính

tên

số lượng trong kho

giá mua

giá bán

các phương thức

hàm chênh lệch giá bán mua

{giá bán - giá mua}

thủ tục mua(q)

{Thêm vào trong kho q đơn vị mặt hàng}

thủ tục bán(q)

{Bớt đi q đơn vị mặt hàng có trong kho}

2. Lớp mặt hàng nhập khẩu thừa kế từ mặt hàng

các thuộc tính

thuế nhập khẩu

các phương thức

hàm chênh lệch giá bán - mua

{giá bán - giá mua* thuế nhập khẩu}

3. Lớp xe gắn máy thừa kế từ mặt hàng nhập khẩu các thuộc tính

dung tích xy lanh

các phương thức

4. Lớp hàng điện tử dân dụng thừa kế từ mặt hàng các thuộc tính

điện áp

thời hạn bảo hành

các phương thức

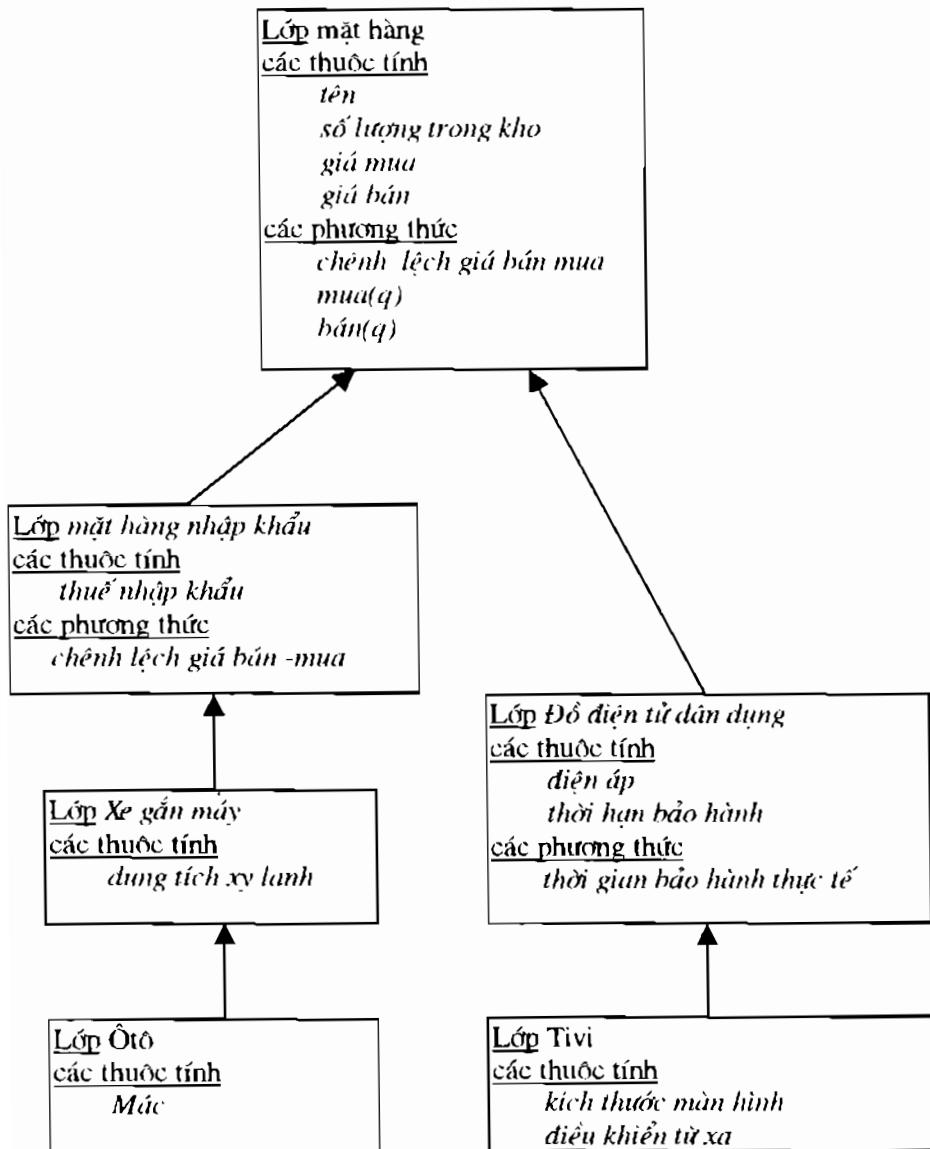
hàm thời gian bảo hành thực tế

...

Tính đa hình cũng là một trong các điểm lý thú trong lập trình hướng đối tượng, được thiết lập trên cơ sở thừa kế trong đó đối tượng có thể có biểu hiện khác nhau tuỳ thuộc vào tình huống cụ thể. Tính đa hình ấy có thể xảy ra ở một hành vi của đối tượng hay trong toàn bộ đối tượng. Ví dụ trực quan thể hiện tính đa hình là một tì vi có thể vừa là đối tượng của mặt hàng vừa là đối tượng của lớp mặt hàng điện tử dân dụng. Các đối tượng hình học như hình vuông, hình tròn, hình chữ nhật đều có cùng cách vẽ như nhau: xác định hai điểm đầu và cuối, nối hai điểm này. Do vậy thuật toán tuy giống nhau đối với tất cả các đối tượng hình, nhưng cách vẽ thì phụ thuộc vào từng lớp đối tượng cụ thể. Ta nói phương thức nối điểm của các đối tượng hình học có tính đa hình. Tính đa hình còn được thể hiện trong cách thức hiển thị thông tin trong các đối tượng điểm màu/không màu.

Các ngôn ngữ lập trình hướng đối tượng đều cho phép đa thừa kế, theo đó một lớp có thể là dẫn xuất của nhiều lớp khác. Do vậy dẫn tới khả năng một lớp cơ sở có thể được thừa kế nhiều lần trong một lớp dẫn xuất khác, ta gọi đó là sự xung đột thừa kế. Điều này hoàn toàn không hay, cần phải tránh. Từng ngôn ngữ sẽ có những giải pháp của riêng mình, C++ đưa ra khái niệm thừa kế ảo.

Trong chương này ta sẽ đề cập tới các khả năng của C++ để thể hiện nguyên tắc thừa kế khi viết chương trình.



Hình 5.1 Ví dụ về sơ đồ thừa kế.

2. ĐƠN THÙA KẾ

2.1 Ví dụ minh họa

Chương trình inheri1.cpp sau đây là một ví dụ thể hiện tính thừa kế đơn của lớp coloredpoint, mô tả các điểm màu trên mặt phẳng, từ lớp các điểm không màu nối chung point. Chương trình này đề cập đến khá nhiều khía cạnh, liên quan đến kỹ thuật cài đặt tính thừa kế trong C++, đó là:

- (i) Truy nhập các thành phần lớp cơ sở từ lớp dẫn xuất
- (ii) Định nghĩa lại (đè) các thành phần lớp cơ sở trong lớp dẫn xuất
- (iii) Truyền thông tin giữa các hàm thiết lập.

Ví dụ 5.1

```
/*inheri1.cpp*/
#include <iostream.h>
#include <conio.h>

class point {
    float x,y;
public:
    point() {x = 0; y = 0;}
    point(float ox, float oy) {x = ox; y = oy; }
    point(point &p) {x = p.x; y = p.y; }
    void display() {
        cout<<"Goi ham point::display()\n";
        cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
    void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
};

/*lop coloredpoint thua ke tu point*/
class coloredpoint : public point {
    unsigned int color;
public:
```

```

coloredpoint():point() {
    color =0;
}
coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
    color = c;
}
coloredpoint(coloredpoint &b):point((point &)b) {
    color = b.color;
}
void display() {
    cout<<"Ham coloredpoint::display()\n";
    point::display();/*gọi tới hàm cùng tên trong lớp cơ sở*/
    cout<<"Mau "<<color<<endl;
}
};

void main() {
    clrscr();
    coloredpoint m;
    cout<<"Diem m \n";
    m.display();
    cout<<"Chi hien thi toa do cua m\n";
    m.point::display();/*gọi phương thức display trong lớp point*/
    coloredpoint n(2,3,6);
    cout<<"Diem n \n";
    n.display();
    cout<<"Chi hien thi toa do cua n\n";
    n.point::display();
    coloredpoint p =n;
    cout<<"Diem p \n";
    p.display();
    cout<<"Chi hien thi toa do cua p\n";
    p.point::display();
}

```

```

getch(); // Nhập phím bất kỳ để thoát khỏi chương trình

} // Kết thúc khung của lớp coloredpoint, kết thúc khung của lớp point
// và kết thúc khung của lớp derivedpoint

Diem m

Ham coloredpoint::display()
Goi ham point::display()

Toa do :0 0
Mau 0
Chi hien thi toa do cua m

Goi ham point::display()

Toa do :0 0

Diem n

Ham coloredpoint::display()
Goi ham point::display()

Toa do :2 3
Mau 6
Chi hien thi toa do cua n

Goi ham point::display()

Toa do :2 3

Diem p

Ham coloredpoint::display()
Goi ham point::display()

Toa do :2 3
Mau 6
Chi hien thi toa do cua p

Goi ham point::display()

Toa do :2 3

```

2.2 Truy nhập các thành phần của lớp cơ sở từ lớp dẫn xuất

Các thành phần **private** trong lớp cơ sở không thể truy nhập được từ các lớp dẫn xuất. Chẳng hạn các thành phần **private** x và y trong lớp cơ sở point không được dùng trong định nghĩa các hàm thành phần của lớp dẫn xuất coloredpoint. Tức là “*Phạm vi lớp*” chỉ mở rộng cho bạn bè, mà không được mở rộng đến các lớp con cháu.

Trong khi đó, trong định nghĩa của các hàm thành phần trong lớp dẫn xuất được phép truy nhập đến các thành phần **protected** và **public** trong lớp dẫn xuất, chẳng hạn có thể gọi tới hàm thành phần `point::display()` của lớp cơ sở bên trong định nghĩa hàm thành phần `coloredpoint::display()` trong lớp dẫn xuất.

2.3 Định nghĩa lại các thành phần của lớp cơ sở trong lớp dẫn xuất

Chương trình `inher1.cpp` cũng có một ví dụ minh họa cho điều này: hàm `display()` tuy đã có trong lớp `point`, được định nghĩa lại trong lớp `coloredpoint`. Lúc này, thực tế là có hai phiên bản khác nhau của `display()` cùng tồn tại trong lớp `coloredpoint`, một được định nghĩa trong lớp `point`, được xác định bởi `point::display()` và một được định nghĩa trong lớp `coloredpoint` và được xác định bởi `coloredpoint::display()`. Trong phạm vi của lớp dẫn xuất hàm thứ hai “che lấp” hàm thứ nhất, nghĩa là tên gọi `display()` trong định nghĩa của hàm thành phần của lớp `coloredpoint` hoặc trong lời gọi hàm thành phần `display()` từ một đối tượng lớp `coloredpoint` phải tham chiếu đến `coloredpoint::display()`. Nếu muốn gọi tới hàm `display()` của lớp `point` ta phải viết đầy đủ tên của nó, nghĩa là `point::display()`. Trong định nghĩa hàm `coloredpoint::display()` nếu ta thay thế `point::display()` bởi `display()` thì sẽ tạo ra lời gọi đệ quy vô hạn lần.

Cần chú ý rằng việc định nghĩa lại một hàm thành phần khác với định nghĩa chung hàm thành phần, bởi vì khái niệm định nghĩa lại chỉ được xét tới khi ta nói đến sự thừa kế. Hàm định nghĩa lại và hàm bị định nghĩa lại giống hệt nhau về tên, tham số và giá trị trả về, chúng chỉ khác nhau ở vị trí, một hàm đặt trong lớp dẫn xuất và hàm kia thì có mặt trong lớp cơ sở. Trong khi đó, các hàm chung chỉ có cùng tên, thường khác nhau về danh sách tham số và tất cả chúng thuộc về cùng một lớp. Định nghĩa lại hàm thành phần chính là cơ sở cho việc cài đặt tính đa hình của các phương thức của đối tượng.

C++ còn cho phép khai báo bên trong lớp dẫn xuất các thành phần dữ liệu cùng tên với các thành phần dữ liệu đã có trong lớp cơ sở. Hai thành phần cùng tên này có thể cùng kiểu hay khác kiểu. Lúc này bên trong một đối tượng của lớp dẫn xuất có tới hai thành phần khác nhau có cùng tên, nhưng trong phạm vi lớp dẫn xuất tên chung đó nhằm chỉ định thành phần được khai báo lại trong lớp dẫn xuất. Khi muốn chỉ định thành phần cùng tên trong lớp cơ sở, phải sử dụng tên lớp cơ sở và toán tử phạm vi “`::`” đặt trước tên thành phần đó.

2.4 Tính thừa kế trong lớp dẫn xuất

2.4.1 Sự tương thích của đối tượng thuộc lớp dẫn xuất với đối tượng thuộc lớp cơ sở

Một cách tổng quát, trong lập trình hướng đối tượng, một đối tượng của lớp dẫn xuất có thể “thay thế” một đối tượng của lớp cơ sở. Nghĩa là: tất cả những thành phần dữ liệu có trong lớp cơ sở đều tìm thấy trong lớp dẫn xuất; tất cả các

hành động thực hiện được trên lớp cơ sở luôn luôn có thể làm được trên các lớp dẫn xuất. Trong chương trình inher1.cpp ta thấy rằng: đối tượng m của lớp coloredpoint có đồng thời hai thành phần toạ độ x, y và thêm thành phần dữ liệu bổ sung color; ngoài ra có thể gọi các hàm thành phần point::display() và point::move(...) thông qua đối tượng m.

Tính tương thích giữa một đối tượng của lớp dẫn xuất và đối tượng lớp cơ sở được thể hiện ở chỗ có thể chuyển kiểu ngầm định từ một đối tượng của lớp dẫn xuất sang một đối tượng của lớp cơ sở. Xét các chỉ thị sau:

```
point p;
p.display();
coloredpoint col pc(2,3,5);
```

câu lệnh

```
p=pc;
p.display();
pc.display();
```

cho kết quả

```
Điểm p không màu
Gọi ham point::display()
Tọa độ :0 0
Điểm pc có màu
Hàm coloredpoint::display()
Gọi ham point::display()
Tọa độ :2 3
Màu 5
p =pc
Gọi ham point::display()
Tọa độ :2 3
```

Tuy nhiên nhận xét trên đây không hoàn toàn đúng xét theo chiều ngược lại. Câu lệnh sau đây không đúng nếu không có định nghĩa chông toán tử gán giữa hai đối tượng với các kiểu dữ liệu coloredpoint và point.

```
pc=p;
```

Chú ý

Từ pc chỉ có thể gọi đến các thành phần hàm **public** trong lớp point (xem thêm phần sau để hiểu rõ hơn).

2.4.2 Tương thích giữa con trỏ lớp dẫn xuất và con trỏ lớp cơ sở

Tương tự như vấn đề trình bày trong phần trên một con trỏ đối tượng lớp cơ sở có thể chỉ đến một đối tượng lớp dẫn xuất, còn một con trỏ lớp dẫn xuất không thể nhận địa chỉ của đối tượng lớp cơ sở, trừ trường hợp ép kiểu. Ta xét chương trình ví dụ sau:

Ví dụ 5.2

```
/*inheri2.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    float x,y;
public:
    point() {x = 0; y = 0;}
    point(float ox, float oy) {x = ox; y = oy; }
    point(point &p) {x = p.x; y = p.y;}
    void display() {
        cout<<"Goi ham point::display()\n";
        cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
    void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
};

/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
    unsigned int color;
public:
    coloredpoint():point() {
```

```

color =0;
}

coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
    color = c;
}

coloredpoint(coloredpoint &b):point((point &)b) {
    color = b.color;
}

void display() {
    cout<<"Ham coloredpoint::display()\n";
    point::display();/*gọi tới hàm cùng tên trong lớp cơ sở*/
    cout<<"Mau "<<color<<endl;
}

};

void main() {
    clrscr();
    point *adp;
    coloredpoint pc(2,3,5);
    pc.display();
    cout<<"adp = &pc \n";
    adp=&pc;
    adp->move(2,3);
    cout<<"adp->move (2,3)\n";
    adp->display();
    getch();
}

```

Hàm coloredpoint::display()

Gọi hàm point::display()

Tọa độ :2 3

Mau 5

adp = &pc

```

adp->move(2, 3)
Ham coloredpoint::display()
Goi ham point::display()
Toa do :4 6
Mau 5
Goi ham point::display()
Toa do :4 6

```

Nhận xét

Chú ý kết quả thực hiện chỉ thị :

```
adp->display();
```

```
Goi ham point::display()
```

```
Toa do :4 6
```

Như vậy, mặc dù adp chứa địa chỉ của đối tượng coloredpoint là pc, nhưng adp->display()

vẫn là

```
point::display()
```

chứ không phải coloredpoint::display().

Hiện tượng này được giải thích là do adp được khai báo là con trỏ kiểu point và vì các hàm trong point đều được khai báo như bình thường nên adp chỉ có thể gọi được các hàm thành phần có trong point chứ không phụ thuộc vào đối tượng mà adp chứa địa chỉ. Muốn có được tính đa hình cho display() nghĩa là lời gọi tới display() phụ thuộc vào kiểu đối tượng có địa chỉ chứa trong adp, ta phải khai báo hàm thành phần display() trong point như là một hàm ảo. Phần sau chúng ta sẽ đề cập vấn đề này.

2.4.3 Tương thích giữa tham chiếu lớp dẫn xuất và tham chiếu lớp cơ sở

Vấn đề được xét trong phần 2.4.2 cũng hoàn toàn tương tự đối với tham chiếu. Ta xét chương trình sau:

Ví dụ 5.3

```

/*inheri3.cpp*/
#include <iostream.h>
#include <conio.h>

```

```
class point {
    float x,y;
public:
    point() {x = 0; y = 0;}
    point(float ox, float oy) {x = ox; y = oy; }
    point(point &p) {x = p.x; y = p.y;}
    void display() {
        cout<<"Goi ham point::display() \n";
        cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
    void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
};

/* lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
    unsigned int color;
public:
    coloredpoint():point() {
        color = 0;
    }
    coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
        color = c;
    }
    coloredpoint(coloredpoint &b):point((point &)b) {
        color = b.color;
    }
    void display() {
        cout<<"Ham coloredpoint::display()\n";
        point::display();
        cout<<"Mau "<<color<<endl;
    }
}
```

```

    }

};

void main() {
    clrscr();
    coloredpoint pc(2,3,5); "Làm sao để ta có thể tạo ra một đối tượng
pc.display();          có thuộc tính riêng mà không cần phải khai báo nó riêng?
cout<<"point &rp = pc \n";
    point &rp=pc;
    rp.move(2,3);
    cout<<"rp.move(2,3)\n";
    pc.display();
    rp.display();
    getch();
}

```

Hàm coloredpoint::display()

Gọi hàm point::display()

Tọa độ :2 3

Màu 5

point &rp = pc

rp.move(2,3)

Hàm coloredpoint::display()

Gọi hàm point::display()

Tọa độ :4 6

Màu 5

Gọi hàm point::display()

Tọa độ :4 6

2.5 Hàm thiết lập trong lớp dẫn xuất

2.5.1 Hàm thiết lập trong lớp

Nếu lớp có khai báo tường minh ít nhất một hàm thiết lập thì khi tạo ra một đối tượng sẽ có một lời gọi đến một trong các hàm thiết lập được định nghĩa. Việc chọn lựa hàm thiết lập dựa theo các tham số được cung cấp kèm theo. Trường hợp không

có hàm thiết lập nào phù hợp sẽ sinh ra một lỗi biên dịch. Như vậy không thể tạo ra một đối tượng nếu không dùng đến một trong các hàm thiết lập đã được định nghĩa.

Trong trường hợp thật sự không có hàm thiết lập tường minh ta không thể mô tả tường tận các dữ liệu của đối tượng liên quan. Xét chương trình sau:

Ví dụ 5.4

```
#include <iostream.h>
#include <conio.h>

/*khai báo lớp point mà không có hàm thiết lập tường minh*/
class point {
public:
    void display();
    void move(float dx, float dy);
};

void main()
{
    point p;
    p.display();
    getch();
}
```

Điều này sẽ gây lỗi:

```
Diem p
Goi ham point::display()
Toa do :8.187236e-34 2.637535e-11
```

2.5.2 Phản cấp lời gọi

Một đối tượng lớp dẫn xuất về thực chất có thể coi là một đối tượng của lớp cơ sở, vì vậy việc gọi hàm thiết lập lớp dẫn xuất để tạo đối tượng lớp dẫn xuất sẽ kéo theo việc gọi đến một hàm thiết lập trong lớp cơ sở.

Về nguyên tắc, những phần của đối tượng lớp dẫn xuất thuộc về lớp cơ sở sẽ được tạo ra trước khi các thông tin mới được xác lập. Như vậy, thứ tự thực hiện của các hàm thiết lập sẽ là: hàm thiết lập cho lớp cơ sở, rồi đến hàm thiết lập cho lớp dẫn xuất nhằm bổ sung những thông tin còn thiếu.

Cơ chế trên đây được thực hiện một cách ngầm định, không cần phải gọi tường minh hàm thiết lập lớp cơ sở trong hàm thiết lập lớp dẫn xuất (thực tế là cũng không thể thực hiện được điều đó vì không thể gọi hàm thiết lập của bất kỳ lớp nào một cách tường minh).

Giải pháp của C++ (cũng được nhiều ngôn ngữ lập trình hướng đối tượng khác chấp nhận) là: trong định nghĩa của hàm thiết lập lớp dẫn xuất, ta mô tả luôn một lời gọi tới một trong các hàm thiết lập lớp cơ sở. Hãy xem lại định nghĩa của các hàm thiết lập lớp `coloredpoint` trong chương trình `inher1.cpp`:

Một số nhận xét quan trọng

- (i) Nếu muốn sử dụng hàm thiết lập ngầm định của lớp cơ sở thì có thể không cần mô tả cùng với định nghĩa của hàm thiết lập lớp dẫn xuất, nghĩa là định nghĩa của hàm `coloredpoint::coloredpoint()` có thể viết lại như sau:

```
coloredpoint() /*để tạo phản đối tượng point sử dụng hàm point::point()*/
{
    color =0;
}
```

- (ii) Các tham số mà hàm thiết lập lớp dẫn xuất truyền cho hàm thiết lập lớp cơ sở không nhất thiết lấy nguyên si từ các tham số nó nhận được mà có thể được chế biến đi ít nhiều. Ví dụ, ta có thể viết lại định nghĩa cho `coloredpoint::coloredpoint(float, float, unsigned)` như sau:

```
coloredpoint::coloredpoint
(float ox, float oy, unsigned c) : point((ox+oy)/2, (ox-oy)/2)
{
    color = c;
}
```

2.5.3 Hàm thiết lập sao chép

Nếu trong lớp dẫn xuất không khai báo tường minh hàm thiết lập sao chép, thì công việc này được chương trình biên dịch đảm nhiệm nhờ định nghĩa hàm thiết lập sao chép ngầm định.

Về nguyên tắc, trong định nghĩa của hàm thiết lập sao chép lớp dẫn xuất ta có thể mô tả bất kỳ hàm thiết lập nào có mặt trong lớp cơ sở. Chương trình sau minh họa điều đó:

Ví dụ 5.5

```
/*inher4.cpp*/
#include <iostream.h>
#include <conio.h>

class point {
    float x,y;
public:
    float getx() {return x;}
    float gety() {return y;}
    point() {x = 0; y = 0;}
    point(float ox, float oy) {x = ox; y = oy; }
    point(point &p) {x = p.x; y = p.y;}
    void display() {
        cout<<"Goi ham point::display()\n";
        cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
    void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
};

/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
    unsigned int color;
public:
```

```
coloredpoint():point() {
    color =0;
}

coloredpoint(float ox, float oy, unsigned int c);
coloredpoint(coloredpoint &b):point(b.getx(),b.gety()) {
    cout<<"Goi ham thiet lap sao chep"">>
        <<" coloredpoint::coloredpoint(coloredpoint &) \n";
    color = b.color;
}

void display() {
    cout<<"Ham coloredpoint::display()\n";
    point::display();
    cout<<"Mau "<<color<<endl;
}

};

coloredpoint::coloredpoint
(float ox, float oy, unsigned c) : point(ox, oy)
{
    color = c;
}

void main() {
    clrscr();
    coloredpoint pc(2,3,5);
    cout<<"pc = ";
    pc.display();
    cout<<"coloredpoint qc = pc;\n";
    coloredpoint qc = pc;
    cout<<"qc= ";
    qc.display();
    getch();
}
```

```

pc = Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
coloredpoint qc = pc;
Goi      ham      thiet      lap      sao      chep
coloredpoint::coloredpoint(coloredpoint &)
qc= Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5

```

Trong thực tế, ta thường sử dụng một cách làm khác, gọi hàm thiết lập sao chép của lớp cơ sở trong định nghĩa của hàm thiết lập sao chép lớp dẫn xuất. Cách tiếp cận này yêu cầu phải tách cho được tham chiếu đến đối tượng lớp cơ sở để dùng làm tham số của hàm thiết lập cơ sở.

Ta xét định nghĩa hàm

`coloredpoint::coloredpoint(coloredpoint&)`

trong chương trình sau:

Ví dụ 5.6

`/*inheri5.cpp*/`

```

#include <iostream.h>
#include <conio.h>
class point {
    float x,y;
public:
    point() {x = 0; y = 0;}
    point(float ox, float oy) {x = ox; y = oy; }
    point(point &p) {x = p.x; y = p.y;}
    void display() {
        cout<<"Goi ham point::display()\n";
        cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
}

```

```
void move(float dx, float dy) {
    x += dx;
    y += dy;
}

/*
*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
    unsigned int color;
public:
    coloredpoint():point() {
        color =0;
    }
    coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
        color = c;
    }
    coloredpoint(coloredpoint &b):point(&point &)b) {
        color = b.color;
    }
    void display() {
        cout<<"Ham coloredpoint::display()\n";
        point::display();
        cout<<"Mau "<<color<<endl;
    }
}
void main() {
    clrscr();
    coloredpoint m(2,3,5);
    cout<<"Diem m \n";
    m.display();
    cout<<"coloredpoint p =m;\n";
    coloredpoint p =m;
    cout<<"Diem p \n";
}
```

nhập liệu từ bàn phím

```
p.display();
getch();
}
```

Diem m

```
Ham coloredpoint::display()
Goi ham point::display()
```

Toa do :2 3

Mau 5

coloredpoint p =m;

Diem p

Ham coloredpoint::display()

Goi ham point::display()

Toa do :2 3

Mau 5

Nếu bạn đọc cùn nhớ phần 2.4.3 của chương này thì sẽ thấy rằng định nghĩa của `coloredpoint::coloredpoint(coloredpoint &)` có thể cải tiến thêm một chút như sau:

```
coloredpoint (coloredpoint &b) : point(b) {
```

color = b.color;

Đơn vị

Đơn vị

Đơn vị

đơn vị

Thực tế ở đây có sự chuyển kiểu ngầm định từ `coloredpoint &` sang `point &`.

2.6 Các kiểu dẫn xuất khác nhau

Tuỳ thuộc vào từ khoá đứng trước lớp cơ sở trong khai báo lớp dẫn xuất, người ta phân biệt ba loại dẫn xuất như sau:

Từ khoá	Kiểu dẫn xuất
<code>public</code>	dẫn xuất public
<code>private</code>	dẫn xuất private
<code>protected</code>	dẫn xuất protected

2.6.1 Dẫn xuất public

Trong dẫn xuất **public**, các thành phần, các hàm bạn và các đối tượng của lớp dẫn xuất không thể truy nhập đến các thành phần **private** của lớp cơ sở.

Các thành phần **protected** trong lớp cơ sở trở thành các thành phần **private** trong lớp dẫn xuất.

Các thành phần **public** của lớp cơ sở vẫn là **public** trong lớp dẫn xuất.

2.6.2 Dẫn xuất private

Trong trường hợp này, các thành phần **public** trong lớp cơ sở không thể truy nhập được từ các đối tượng của lớp dẫn xuất, nghĩa là chúng trở thành các thành phần **private** trong lớp dẫn xuất.

Các thành phần **protected** trong lớp cơ sở có thể truy nhập được từ các hàm thành phần và các hàm bạn của lớp dẫn xuất.

Dẫn xuất **private** được sử dụng trong một số tình huống đặc biệt khi lớp dẫn xuất không khai báo thêm các thành phần hàm mới mà chỉ định nghĩa lại các phương thức đã có trong lớp cơ sở.

2.6.3 Dẫn xuất protected

Trong dẫn xuất loại này, các thành phần **public**, **protected** trong lớp cơ sở trở thành các thành phần **protected** trong lớp dẫn xuất.

Bảng tổng kết các kiểu dẫn xuất

Lớp cơ sở			Dẫn xuất public		Dẫn xuất protected		Dẫn xuất private	
TTĐ FMA	TN NSD	TN NSD	TTM	TN NSD	TTM	TN NSD	TTM	TN NSD
pub	C	C	pub	C	pro	K	pri	K
pro	C	K	pro	K	pro	K	pri	K
pri	C	K	pri	K	pri	K	pri	K

Ghi chú

Từ viết tắt	Điễn giải
TTĐ	Trạng thái đầu
TTM	Trạng thái mới
TN FMA	Truy nhập bởi các hàm thành phần hoặc hàm bạn.
TN NSD	Truy nhập bởi người sử dụng

pro	protected
pub	public
pri	private
C	Có
K	Không

3. HÀM ẢO VÀ TÍNH ĐA HÌNH

3.1 Đặt vấn đề

Cho đến nay, ta chỉ mới biết rằng một tham trỏ tới một đối tượng có thể nhận địa chỉ của bất cứ đối tượng con cháu nào (các đối tượng của các lớp thừa kế). Tuy vậy ưu điểm này phải trả giá: lời gọi đến một phương thức của một đối tượng được trả trỏ luôn được coi như lời gọi đến phương thức tương ứng với kiểu con trỏ chứ không phải tương ứng với đối tượng đang được trả trỏ. Ta đã có dịp minh chứng điều này trong ví dụ ở phần 2.4.2 của chương này. Ta gọi đó là “gán kiểu tĩnh-static typing” hay “gán kiểu sớm-early binding” (tức là hàm thành phần gọi từ con trỏ đối tượng được xác định ngay khi khai báo).

Thực tế có nhiều vấn đề khi xác định phương thức hay hành động cụ thể tùy thuộc vào thời điểm tham chiếu đối tượng, chẳng hạn khi vẽ các đối tượng hình chữ nhật, hình vuông, hình tròn, tất cả đều gọi tới phương thức vẽ được thừa kế từ lớp tổng quát hơn (lớp hình vẽ). Trong định nghĩa của phương thức đó có lời gọi đến một phương thức khác là nỗi niềm chỉ được xác định một cách tương minh trong từng đối tượng cụ thể là hình chữ nhật, hình vuông hay hình tròn.

Để gọi được phương thức tương ứng với đối tượng được trả trỏ, cần phải xác định được kiểu của đối tượng được xem xét tại thời điểm thực hiện chương trình bởi lẽ kiểu của đối tượng được chỉ định bởi cùng một con trỏ có thể thay đổi trong quá trình thực hiện của chương trình. Ta gọi đó là “gán kiểu động - dynamic typing” hay “gán kiểu muộn - late binding”, nghĩa là xác định hàm thành phần nào tương ứng với một lời gọi hàm thành phần từ con trỏ đối tượng phụ thuộc cụ thể vào đối tượng mà con trỏ đang chứa địa chỉ. Khái niệm *hàm ảo* được C++ đưa ra nhằm đáp ứng nhu cầu này. Chương trình polymorphism1.cpp sau đây đưa ra một ví dụ về hàm ảo:

Ví dụ 5.7

```
/*polymorphism1.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    float x,y;
public:
```

```

point() {
    cout<<"point::point()\n";
    x = 0;
    y = 0;
}

point(float ox, float oy) {
    cout<<"point::point(float, float)\n";
    x = ox;
    y = oy;
}

point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
}

virtual void display() {
    cout<<"Goi ham point::display() \n";
    cout<<"Toa do :"<<x<<" "<<y<<endl;
}

void move(float dx, float dy) {
    x += dx;
    y += dy;
}

};

/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
    unsigned int color;
public:
    coloredpoint():point() {
        cout<<"coloredpoint::coloredpoint()\n";
        color =0;
    }
}

```

```

coloredpoint(float ox, float oy, unsigned int c);
coloredpoint(coloredpoint &b):point((point &)b) {
    cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
    color = b.color;
}

void display() {
    cout<<"Ham coloredpoint::display()\n";
    point::display();
    cout<<"Mau "<<color<<endl;
}
};

coloredpoint::coloredpoint(float ox, float oy, unsigned c) : point(ox,
oy) {
    cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
    color = c;
}

void main() {
    clrscr();
    cout<<"coloredpoint pc(2,3,5);\n";
    coloredpoint pc(2,3,5);
    cout<<"pc.display();\n";
    pc.display();
    cout<<"point *ptr=&pc;\n";
    point *ptr=&pc;
    cout<<"ptr->display();\n";
    ptr->display();
    cout<<"point p(10,20);\n";
    point p(10,20);
    cout<<"ptr = &p\n";
    ptr = &p;
    cout<<"p.display()\n";
    p.display();
}

```

```

cout<<"ptr->display()\n"; //In ra kết quả: Toa do(2,3,5)
ptr->display();           //Làm điều gì đây? (đó là điểm của lớp Point)
getch();                  //Nhập phím bất kỳ để thoát khỏi cửa sổ
}

```

```

coloredpoint pc(2,3,5);
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)
pc.display();

Hàm coloredpoint::display()
Gọi hàm point::display()

Toa do :2 3
Mau 5

point *ptr=&pc;
ptr->display();

Hàm coloredpoint::display()
Gọi hàm point::display()

Toa do :2 3
Mau 5

point p(10,20);
point::point(float, float)
ptr = &p
p.display();

Gọi hàm point::display()
Toa do :10 20
ptr->display();
Gọi hàm point::display()
Toa do :10 20

```

Nhận xét

- Trong định nghĩa của lớp point, ở dòng tiêu đề khai báo hàm thành phần point::display() có từ khoá **virtual**, từ khoá này có thể đặt trước hay sau tên kiểu dữ liệu nhưng phải trước tên hàm để chỉ định rằng hàm point::display() là một hàm ảo.

2. **Hàm thành phần point::display()** được định nghĩa lại trong lớp dẫn xuất tuy rằng trong trường hợp tổng quát điều này không bắt buộc nếu như bản thân hàm đó đã được định nghĩa. Khi đó tùy thuộc vào kiểu của đối tượng có địa chỉ chứa trong con trỏ lớp dẫn xuất ptr mà lời gọi hàm `ptr->display()` sẽ gọi đến `point::display()` hay `coloredpoint::display()`:

Tính đa hình còn thể hiện khi một hàm thành phần trong lớp cơ sở được gọi từ một đối tượng lớp dẫn xuất, còn bản thân hàm đó thì gọi tới hàm thành phần được định nghĩa đồng thời trong cả lớp cơ sở (khai báo **virtual** có mặt ở đây) và trong các lớp dẫn xuất. Mời bạn đọc xem chương trình polymorphism2.cpp sau đây:

Ví dụ 5.8

```
/*polymorphism2.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    float x,y;
public:
    point() {
        cout<<"point::point()\n";
        x = 0;
        y = 0;
    }
    point(float ox, float oy) {
        cout<<"point::point(float, float)\n";
        x = ox;
        y = oy;
    }
    point(point &p) {
        cout<<"point::point(point &)\n";
        x = p.x;
        y = p.y;
    }
    void display() ;
    void move(float dx, float dy) {

```

```

x += dx;
y += dy;
}

virtual void Identifier() {
    cout<<"Diem khong mau \n";
}
};

void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<endl;
    Identifier();
}

/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
    unsigned int color;
public:
    coloredpoint():point() {
        cout<<"coloredpoint::coloredpoint()\n";
        color =0;
    }

    coloredpoint(float ox, float oy, unsigned int c);
    coloredpoint(coloredpoint &b):point((point &)b) {
        cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
        color = b.color;
    }

    void Identifier() {
        cout<<"Mau : "<<color<<endl;
    }
};

coloredpoint::coloredpoint(float ox, float oy, unsigned c)
    : point(ox, oy)

{
    cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
}

```

```

    color = c;
}

void main() {
clrscr();
cout<<"coloredpoint pc(2,3,5);\\n";
coloredpoint pc(2,3,5);
cout<<"pc.display()\\n";
pc.display();
cout<<"point p(10,20);\\n";
point p(10,20);
cout<<"p.display()\\n";
p.display();
getch();
}

```

```

coloredpoint pc{2,3,5};
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)
pc.display()
Toa do : 2 3
Mau : 5
point p(10,20);
point::point(float, float)
p.display()
Toa do : 10 20
Diem khong mau

```

Trong chương trình trên, lớp coloredpoint thừa kế từ lớp point hàm thành phần display(). Hàm này gọi tới hàm thành phần ảo Identifier được định nghĩa lại trong coloredpoint. Tùy thuộc vào đối tượng gọi hàm display() là của point hay của coloredpoint chương trình dịch sẽ phát lời gọi đến point::Identifier() hay coloredpoint::Identifier(). Nói cách khác trong trường hợp này Identifier cũng biểu hiện tính đa hình.

3.2 Tổng quát về hàm ảo

3.2.1 Phạm vi của khai báo virtual

Khi một hàm f() được khai báo **virtual** ở trong một lớp A, nó được xem như thể hiện của sự ghép kiểu động trong lớp A và trong tất cả các lớp dẫn xuất từ A hoặc từ con cháu của A. Xét lời gọi tới hàm Identifier() trong hàm display() được gọi từ các đối tượng khác nhau trong chương trình sau:

Ví dụ 5.9

```
#include <iostream.h>
#include <conio.h>
class point {
    float x,y;
public:
    point{} {
        cout<<"point::point()\n";
        x = 0;
        y = 0;
    }
    point(float ox, float oy) {
        cout<<"point::point(float, float)\n";
        x = ox;
        y = oy;
    }
    point(point &p) {
        cout<<"point::point(point &)\n";
        x = p.x;
        y = p.y;
    }
    void display() ;
    void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
}
```

```

virtual void Identifier() {
    cout<<"Diem khong mau \n";
}
};

void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<endl;
    Identifier();
}

class coloredpoint : public point {
    unsigned int color;
public:
    coloredpoint():point() {
        cout<<"coloredpoint::coloredpoint()\n";
        color =0;
    }
    coloredpoint(float ox, float oy, unsigned int c);
    coloredpoint(coloredpoint &b):point((point &)b) {
        cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
        color = b.color;
    }
    void Identifier() {
        cout<<"Mau : "<<color<<endl;
    }
};

class threedimpoint : public point {
    float z;
public:
    threedimpoint() {
        z = 0;
    }
    threedimpoint(float ox, float oy, float oz):point (ox, oy) {
        z = oz;
    }
};

```

```

    }

threedimpoint(threedimpoint &p) :point(p) {
    z = p.z;
}

void Identifier() {
    cout<<"Toa do z : "<<z<<endl;
}

};

class coloredthreedimpoint : public threedimpoint {
    unsigned color;
public:
    coloredthreedimpoint() {
        color = 0;
    }

    coloredthreedimpoint(float ox, float oy, float oz,unsigned c):
        threedimpoint (ox, oy, oz)
    {
        color = c;
    }

    coloredthreedimpoint(coloredthreedimpoint &p) :threedimpoint(p) {
        color = p.color;
    }

    void Identifier() {
        cout<<"Diem mau : "<<color<<endl;
    }

};

coloredpoint::coloredpoint(float ox, float oy, unsigned c) :
    point(ox, oy)
{
    cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
    color = c;
}

```

```

}

void main() {
    clrscr();
    cout<<"coloredpoint pc(2,3,5);\n";
    coloredpoint pc(2,3,5);
    cout<<"pc.display()\n";
    pc.display();/*gọi tới coloredpoint::Identifier()*/
    cout<<"point p(10,20);\n";
    point p(10,20);
    cout<<"p.display()\n";
    p.display();/*gọi tới point::Identifier()*/
    cout<<"threedimpoint p3d(2,3,4);\n";
    threedimpoint p3d(2,3,4);
    cout<<"p3d.display()\n";
    p3d.display();/*gọi tới threedimpoint::Identifier()*/
    cout<<"coloredthreedimpoint p3dc(2,3,4,10);\n";
    coloredthreedimpoint p3dc(2,3,4,10);
    cout<<"p3dc.display()\n";
    p3dc.display();/*gọi tới coloredthreedimpoint::Identifier()*/
    getch();
}

```

```

coloredpoint pc(2,3,5);
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)
pc.display()
Toa do : 2 3
Mau : 5
point p(10,20);
point::point(float, float)
p.display()
Toa do : 10 20
Diem khong mau

```

```

threeedimpoint p3d(2,3,4);
point::point(float, float)
p3d.display();
Toa do : 2 3
Toa do z : 4
coloredthreeedimpoint p3dc(2,3,4,10);
point::point(float, float)
p3dc.display();
Toa do : 2 3
Diem mau : 10

```

3.2.2 Không nhất thiết phải định nghĩa lại hàm virtual

Trong trường hợp tổng quát, ta luôn phải định nghĩa lại ở trong các lớp dẫn xuất các phương thức đã được khai báo là **virtual** trong lớp cơ sở. Trong một số trường hợp không nhất thiết buộc phải làm như vậy. Khi mà hàm `display()` đã có một định nghĩa hoàn chỉnh trong `point`, ta không cần định nghĩa lại nó trong lớp `coloredpoint`. Chương trình `polymorphism4.cpp` sau đây minh chứng cho nhận định này.

Ví dụ 5.10

```

#include <iostream.h>
#include <conio.h>

class point {
    float x,y;
public:
    point() {
        cout<<"point::point()\n";
        x = 0;
        y = 0;
    }
    point(float ox, float oy) {
        cout<<"point::point(float, float)\n";
        x = ox;
        y = oy;
    }
}

```

```

point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
}

virtual void display() {
    cout<<"Gọi hàm point::display() \n";
    cout<<"Toa độ :"<<x<< " "<<y<<endl;
}

void move(float dx, float dy) {
    x += dx;
    y += dy;
}

};

class coloredpoint : public point {
    unsigned int color;
public:
    coloredpoint():point() {
        cout<<"coloredpoint::coloredpoint()\n";
        color =0;
    }

    coloredpoint(float ox, float oy, unsigned int c);
    coloredpoint(coloredpoint &b):point((point &)b) {
        cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
        color = b.color;
    }

    coloredpoint::coloredpoint(float ox, float oy, unsigned c) :
        point(ox, oy)
    {
        cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
        color = c;
    }
}

```

```

}

void main() {
    clrscr();
    cout<<"coloredpoint pc(2,3,5);\\n";
    coloredpoint pc(2,3,5);
    cout<<"pc.display();\\n";
    pc.display();
    cout<<"point *ptr=&pc;\\n";
    point *ptr=&pc;
    cout<<"ptr->display();\\n";
    ptr->display();
    cout<<"point p(10,20);\\n";
    point p(10,20);
    cout<<"ptr = &p\\n";
    ptr = &p;
    cout<<"p.display();\\n";
    p.display();
    cout<<"ptr->display();\\n";
    ptr->display();
    getch();
}

```

```

coloredpoint pc(2,3,5);
point::point(float, float);
coloredpoint::coloredpoint(float, float, unsigned)
pc.display();
Goi ham point::display()
Toa do :2 3
point *ptr=&pc;
ptr->display();
Goi ham point::display()
Toa do :2 3
point p(10,20);

```

```
point::point(float, float)
ptr = &p
p.display()
Gọi ham point::display()
Toa do :10 20
ptr->display();
Gọi ham point::display()
Toa do :10 20
```

3.2.3 Định nghĩa chồng hàm ảo

Có thể định nghĩa chồng một hàm ảo và các hàm định nghĩa chồng như thế có thể không còn là hàm ảo nữa.

Hơn nữa, nếu ta định nghĩa một hàm ảo trong một lớp và lại định nghĩa chồng nó trong một lớp dẫn xuất với các tham số khác thì có thể xem hàm định nghĩa chồng đó là một hàm hoàn toàn khác không liên quan gì đến hàm ảo hiện tại, nghĩa là nếu nó không được khai báo **virtual** thì nó có tính chất “gán kiểu tĩnh-static typing”. Nói chung, nếu định nghĩa chồng hàm ảo thì tất cả các hàm định nghĩa chồng của nó nên được khai báo là **virtual** để việc xác định lời gọi hàm đơn giản hơn.

3.2.4 Khai báo hàm ảo ở một lớp bất kỳ trong sơ đồ thừa kế

Trong chương trình polymorphism5.cpp, hàm `point::Identifier()` không là **virtual** trong khi đó khai báo **virtual** lại được áp dụng cho hàm `threedimpoint::Identifier()`. Chương trình dịch sẽ xem `point::Identifier()` và `threedimpoint::Identifier()` có tính chất khác nhau: `point::Identifier()` có tính chất “gán kiểu tĩnh-static typing”, trong khi đó `threedimpoint::Identifier()` lại có tính chất “gán kiểu động-dynamic typing”.

Ví dụ 5.11

```
/*polymorphism5.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
    float x,y;
public:
    point() {
```

```
cout<<"point::point()\n";
x = 0;
y = 0;
}

point(float ox, float oy) {
    cout<<"point::point(float, float)\n";
    x = ox;
    y = oy;
}

point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
}

void display() ;

void move(float dx, float dy) {
    x += dx;
    y += dy;
}

void Identifier() {
    cout<<"Diem khong mau \n";
}

};

void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<endl;
    Identifier();
}

class threedimpoint : public point {
    float z;
public:
    threedimpoint() {
        z = 0;
```

```

    }
threeedimpoint(float ox, float oy, float oz):point (ox, oy) {
    z = oz;
}
threeedimpoint(threeedimpoint &p) :point(p) {
    z = p.z;
}
virtual void Identifier() {
    cout<<"Toa do z : "<<z<<endl;
}
};

class coloredthreeedimpoint : public threeedimpoint {
    unsigned color;
public:
    coloredthreeedimpoint() {
        color = 0;
    }
    coloredthreeedimpoint(float ox, float oy, float oz,unsigned c):
        threeedimpoint (ox, oy, oz)
    {
        color = c;
    }
    coloredthreeedimpoint(coloredthreeedimpoint &p) :threeedimpoint(p) {
        color = p.color;
    }
    void Identifier() {
        cout<<"Diem mau : "<<color<<endl;
    }
};

void main() {
    clrscr();
    cout<<"point p(10,20);\n";
}

```

```

point p(10,20);
cout<<"p.display()\n";
p.display();
cout<<"threedimpoint p3d(2,3,4);\n";
threedimpoint p3d(2,3,4);
cout<<"p3d.display()\n";
p3d.display();
cout<<"p3d.Identifier()\n";
p3d.Identifier();
cout<<"coloredthreedimpoint p3dc(2,3,4,10);\n";
coloredthreedimpoint p3dc(2,3,4,10);
cout<<"p3dc.display()\n";
p3dc.display();
cout<<"p3dc.Identifier()\n";
p3dc.Identifier();
getch();
}

```

```

point p(10,20);
point::point(float, float)
p.display()
Toa do : 10 20
Diem khong mau
threedimpoint p3d(2,3,4);
point::point(float, float)
p3d.display();
Toa do : 2 3
Diem khong mau
p3d.Identifier();
Toa do z : 4
coloredthreedimpoint p3dc(2,3,4,10);
point::point(float, float)
p3dc.display();

```

```
Toa do : 2 3
Diem khong mau
p3dc.Identifier();
Diem mau : 10
```

3.2.5 Hàm huỷ bỏ ảo

Hàm thiết lập không thể là hàm ảo, trong khi đó hàm huỷ bỏ lại có thể. Ta quan sát sự cố xảy ra khi sử dụng tính đa hình để xử lý các đối tượng của các lớp trong sơ đồ thừa kế được cấp phát động. Nếu mỗi đối tượng được định dẹp tường minh nhờ sử dụng toán tử **delete** cho con trỏ lớp cơ sở chỉ đến đối tượng thì hàm huỷ bỏ của lớp cơ sở sẽ được gọi mà không cần biết kiểu của đối tượng đang được xử lý, cũng như tên hàm huỷ bỏ của lớp tương ứng với đối tượng (tuy có thể khác với hàm huỷ bỏ của lớp cơ sở).

Một giải pháp đơn giản cho vấn đề này là khai báo hàm huỷ bỏ của lớp cơ sở là hàm ảo, làm cho các hàm huỷ bỏ của các lớp dẫn xuất là ảo mà không yêu cầu chúng phải có cùng tên. Chương trình polymorphism6.cpp sau đây minh họa tính đa hình của hàm ảo:

Ví dụ 5.12

```
#include <iostream.h>
#include <conio.h>

class point {
    float x,y;
public:
    point() {
        cout<<"point::point()\n";
        x = 0;
        y = 0;
    }
    point(float ox, float oy) {
        cout<<"point::point(float, float)\n";
        x = ox;
        y = oy;
    }
    point(point &p) {
        cout<<"point::point(point &)\n";
```

```

x = p.x;
y = p.y;
}

virtual ~point() {
    cout<<"point::~point() \n";
}

void display();
void move(float dx, float dy) {
    x += dx;
    y += dy;
}

virtual void Identifier() {
    cout<<"Diem khong mau \n";
}

};

void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<endl;
    Identifier();
}

class threedimpoint : public point {
    float z;
public:
    threedimpoint() {
        z = 0;
    }

    threedimpoint(float ox, float oy, float oz):point (ox, oy) {
        cout<<"threedimpoint::threedimpoint(float, float, float)\n";
        z = oz;
    }

    threedimpoint(threedimpoint &p) :point(p) {
        z = p.z;
    }
}

```

```

~threedimpoint() {
    cout<<"threedimpoint::~threedimpoint()";
}

void Identifier() {
    cout<<"Toa do z : "<<z<<endl;
}

};

class coloredthreedimpoint : public threedimpoint {
    unsigned color;

public:
    coloredthreedimpoint() {
        color = 0;
    }

    coloredthreedimpoint(float ox, float oy, float oz,unsigned c):
        threedimpoint(ox, oy, oz)
    {

        cout<<"coloredthreedimpoint::coloredthreedimpoint(float,
float,float,unsigned)\n";
        color = c;
    }

    coloredthreedimpoint(coloredthreedimpoint &p) :threedimpoint(p) {
        color = p.color;
    }

~coloredthreedimpoint() {
    cout<<"coloredthreedimpoint::~coloredthreedimpoint()\n";
}

void Identifier() {
    cout<<"Diem mau : "<<color<<endl;
}

};

void main() {
    clrscr();
}

```

```

point *p0 = new point(2,10);
point *p1 = new threedimpoint(2,3,5);
point *p2 = new coloredthreedimpoint(2,3,4,10);
delete p0;
delete p1;
delete p2;
getch();
}

```

```

point::point(float, float)
point::point(float, float)
threedimpoint::threedimpoint(float, float, float)
point::point(float, float)
threedimpoint::threedimpoint(float, float, float)
coloredthreedimpoint::coloredthreedimpoint(float,
float, float, unsigned)
point::~point()
threedimpoint::~threedimpoint() point::~point()
coloredthreedimpoint::~coloredthreedimpoint()
threedimpoint::~threedimpoint() point::~point()

```

3.3 Lớp trừu tượng và hàm ảo thuần tuý

Hàm ảo thuần tuý là hàm không có phân định nghĩa. Định nghĩa một hàm ảo như thế được viết như sau:

```

class mere {
public:
    virtual display() = 0; //hàm ảo thuần tuý
};

```

Lớp có ít nhất một hàm thành phần ảo thuần tuý được gọi là lớp trừu tượng. Phải tuân theo một số quy tắc sau đây:

- (i) Không thể sử dụng lớp trừu tượng khi khai báo các biến, mà để mô tả lớp các đối tượng một cách chung nhất. Sau đó các lớp thừa kế sẽ được chi tiết dần dần. Ở đây, lớp trừu tượng là tổng quát hoá của các lớp thừa kế nó, và ngược lại các lớp dẫn xuất là sự cụ thể hoá của lớp trừu tượng.

- (ii) Được phép khai báo con trỏ có kiểu là một lớp trùu tượng.
- (iii) Một hàm ảo thuần tuý khai báo trong một lớp trùu tượng cơ sở phải được định nghĩa lại trong một lớp dẫn xuất hoặc nếu không thì phải được tiếp tục khai báo ảo trong lớp dẫn xuất. Trong trường hợp này lớp dẫn xuất mới này lại là một lớp trùu tượng. Trong phần 4.5 sẽ có một ví dụ về việc sử dụng lớp trùu tượng để xây dựng một danh sách các đối tượng không đồng nhất.

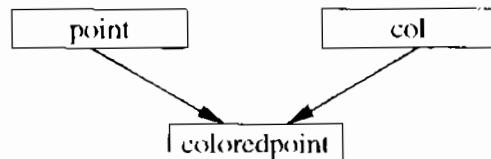
4. ĐA THÙA KẾ

4.1 Đa vấn đề

Đa thừa kế cho phép một lớp có thể là dẫn xuất của nhiều lớp cơ sở, do vậy những gì đã đề cập trong phần đơn thừa kế được tổng quát hoá cho trường hợp đa thừa kế. Tuy vậy cần phải giải quyết các vấn đề sau:

1. Làm thế nào biểu thị được tính độc lập của các thành phần cùng tên bên trong một lớp dẫn xuất?
2. Các hàm thiết lập và huỷ bỏ được gọi như thế nào: thứ tự, truyền thông tin v.v.?
3. Làm thế nào giải quyết tình trạng thừa kế xung đột trong đó, lớp D dẫn xuất từ B và C, và cả hai cũng là dẫn xuất của A.

Chúng ta xem xét một tình huống đơn giản : một lớp có tên là coloredpoint thừa kế từ hai lớp point và col.



Giả sử các lớp point và col được trình bày như sau:

<pre> class point { float x,y; public: point(...){...} ~point(...) void display(){...} }; </pre>	<pre> class col { unsigned color; public: col (...) {...} ~color() {...} void display() {...} }; </pre>
--	---

Chúng ta có thể định nghĩa lớp coloredpoint thừa kế hai lớp này như sau:

```
class coloredpoint:public point,public col
{...};
```

Quan sát câu lệnh khai báo trên ta thấy tiếp theo tên lớp dẫn xuất là một danh sách các lớp cơ sở cùng với các từ khoá xác định kiểu dẫn xuất.

Bên trong lớp `coloredpoint` ta có thể định nghĩa các thành phần mới (ở đây giới hạn với hàm thiết lập, hàm huỷ bỏ và hàm hiển thị).

Trong trường hợp đơn thừa kế, hàm thiết lập lớp dẫn xuất phải truyền một số thông tin cho hàm thiết lập lớp cơ sở. Điều này vẫn cần trong trường hợp đa thừa kế, chỉ khác là phải truyền thông tin cho hai hàm thiết lập tương ứng với hai lớp cơ sở. Dòng tiêu đề trong phần định nghĩa hàm thiết lập `coloredpoint` có dạng:

<code>coloredpoint(.....)</code>	<code>point(.....)</code>	<code>, col(.....)</code>
các tham số cho coloredpoint	các tham số cho point	các tham số cho col

Thứ tự gọi các hàm thiết lập như sau: các hàm thiết lập của các lớp cơ sở theo thứ tự khai báo của các lớp cơ sở trong lớp dẫn xuất (ở đây là `point` và `col`), và cuối cùng là hàm thiết lập của lớp dẫn xuất (ở đây là `coloredpoint`).

Còn các hàm huỷ bỏ lại được gọi theo thứ tự ngược lại khi đối tượng `coloredpoint` bị xoá.

Giống như trong đơn thừa kế, trong hàm thành phần của lớp dẫn xuất có thể sử dụng tất cả các hàm thành phần **public** (hoặc **protected**) của lớp cơ sở.

Khi có nhiều hàm thành phần cùng tên trong các lớp khác nhau (định nghĩa lại một hàm), ta có thể loại bỏ sự nhập nhằng bằng cách sử dụng toán tử phạm vi `::`. Ngoài ra để tạo thói quen lập trình tốt, khi thực hiện các lời gọi hàm thành phần nên tuân theo thứ tự khai báo của các lớp cơ sở. Như vậy bên trong hàm `coloredpoint::display()` có các câu lệnh sau:

```
point::display();
col::display();
```

Tất nhiên khi hàm thành phần trong lớp cơ sở không được định nghĩa lại trong lớp dẫn xuất thì sẽ không có sự nhập nhằng và do đó không cần dùng đến toán tử `::`.

Việc sử dụng lớp `coloredpoint` không có gì đặc biệt, mọi đối tượng kiểu `coloredpoint` thực hiện lời gọi đến các hàm thành phần của `coloredpoint`

cũng như là các hàm thành phần **public** của point và col. Nếu có trùng tên hàm, phải sử dụng toán tử phạm vi “::”. Chẳng hạn với khai báo:

```
coloredpoint p(3,9,2);
```

câu lệnh

```
p.display()
```

sẽ gọi tới

```
coloredpoint::display(),
```

còn câu lệnh

```
p.point::display()
```

gọi tới

```
point::display().
```

Nếu một trong hai lớp point và col lại là dẫn xuất của một lớp cơ sở khác, thì có thể sử dụng được các thành phần bên trong lớp cơ sở mới này. Sau đây là một ví dụ hoàn chỉnh minh họa việc định nghĩa và sử dụng coloredpoint.

Ví dụ 5.13

```
/*mulinher1.cpp*/
#include <iostream.h>
#include <conio.h>
class point{
    float x,y;
public:
    point (float ox,float oy)
    {
        cout<<"++Constr. point\n";
        x=ox;
        y=oy;
    }
    ~point() {cout<<"--Destr. point\n";}
    void display()
    {
        cout<<"Toa do : "<<x<<" "<<y<<"\n";
    }
}
```

```
    }

};

class col {
    unsigned color;
public:
    col(unsigned c)
    {
        cout<<"++Constr. col \n";
        color=c;
    }
    ~col() {cout<<"--Destr. col\n";}
    void display() {cout<<"Mau : "<<color<<"\n";}
};

class coloredpoint :public point,public col {
public:
    coloredpoint(float ox,float oy, unsigned c) : point(ox,oy),col(c)
    {
        cout<<"++Constr. coloredpoint\n";
    }
    ~coloredpoint() {
        cout<<"--Destr. coloredpoint\n";
    }
    void display() {
        point::display();
        col::display();
    }
};

void main()
{
    clrscr();
```

```
coloredpoint p(3,9,2);
cout<<-----\n";
p.display();
cout<<-----\n";
p.point::display();
cout<<-----\n";
p.col::display();
cout<<-----\n";
getch();
}
```

```
++Constr. point
++Constr. col
++Constr. coloredpoint
```

```
Toa do : 3 9
```

```
Mau : 2
```

```
-----
```

```
Toa do : 3 9
```

```
-----
```

```
Mau : 2
```

```
-----
```

```
--Destr. coloredpoint
--Destr. col
--Destr. point
```

Nhận xét

Trong trường hợp các thành phần dữ liệu của các lớp cơ sở trùng tên, ta phân biệt chúng bằng cách sử dụng tên lớp tương ứng đi kèm với toán tử phạm vi “::”.

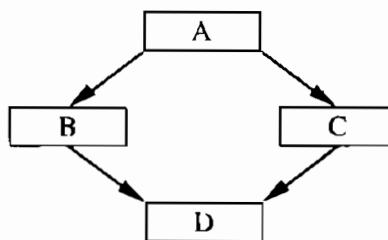
Xét ví dụ:

```
class A { ...  
public:  
    int x;  
...  
};  
  
class B { ...  
public:  
    int x;  
...  
};  
  
class C:public A,public B  
{...};
```

Lớp C có hai thành phần dữ liệu cùng tên là X, một thừa kế từ A, một thừa kế từ B. Bên trong các thành phần hàm của C, chúng ta phân biệt chúng nhờ toán tử phạm vi “::”; đó là A::x và B::x

4.2 Lớp cơ sở áó

Xét tình huống như sau:



tương ứng với các khai báo sau:

```
class A {  
...  
int x,y;  
};  
  
class B:public A{...};  
  
class C:public A{...};  
  
class D:public B,public C  
{...};
```

Theo một nghĩa nào đó, có thể nói rằng D “thừa kế” A hai lần. Trong các tình huống như vậy, các thành phần của A (hàm hoặc dữ liệu) sẽ xuất hiện trong D hai lần. Đối với các hàm thành phần thì điều này không quan trọng bởi chỉ có duy nhất một hàm cho một lớp cơ sở, các hàm thành phần là chung cho mọi đối tượng của lớp. Tuy nhiên, các thành phần dữ liệu lại được lặp lại trong các đối tượng khác nhau (thành phần dữ liệu của mỗi đối tượng là độc lập).

Như vậy, phải chăng có sự dư thừa dữ liệu? Câu trả lời phụ thuộc vào từng tình huống cụ thể. Nếu chúng ta muốn D có hai bản sao dữ liệu của A, ta phải phân biệt chúng nhau:

A::B::x và A::C::x

Thông thường, chúng ta không muốn dữ liệu bị lặp lại và giải quyết bằng cách chọn một trong hai bản sao dữ liệu để thao tác. Tuy nhiên điều đó thật chán ngắt và không an toàn.

Ngôn ngữ C++ cho phép chỉ tổ hợp một lần duy nhất các thành phần của lớp A trong lớp D nhờ khai báo trong các lớp B và C (chứ không phải trong D!) rằng lớp A là ảo (từ khóa **virtual**):

```
class B:public virtual A{....};
class C:public virtual A{....};
class D:public B,public C{....};
```

Việc chỉ thị A là ảo trong khai báo của B và C nghĩa là A sẽ chỉ xuất hiện một lần trong các con cháu của chúng. Nói cách khác, khai báo này không ảnh hưởng đến các lớp B và C. Chú ý rằng từ khóa **virtual** có thể được đặt trước hoặc sau từ khóa **public** (hoặc **private**, **protected**). Ta xét chương trình ví dụ sau đây:

Ví dụ 5.14

```
/*mulinher2.cpp
Solution to multiple inheritance*/
#include <iostream.h>
#include <conio.h>
class A {
    float x,y;
public:
    void set(float ox, float oy) {
        x = ox; y = oy;
    }
    float getx() {
```

```
    return x;
}

float gety() {
    return y;
}

};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

void main() {
    clrscr();
    D d;
    cout<<"d.B::set(2,3);\n";
    d.B::set(2,3);
    cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
    cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
    cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
    cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;

    cout<<"d.C::set(10,20);\n";
    d.B::set(2,3);
    cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
    cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
    cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
    cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;
    getch();
}
```

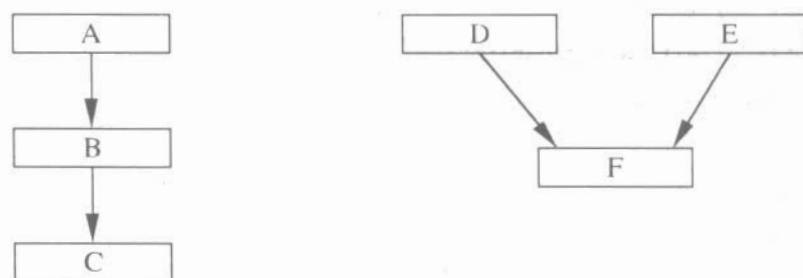
```

d.B::set(2,3);
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3
d.C::set(10,20);
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3

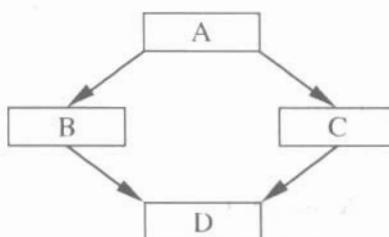
```

4.3 Hàm thiết lập và huỷ bỏ - với lớp ảo

Thông thường khi khởi tạo các đối tượng dẫn xuất các hàm thiết lập được gọi theo thứ tự xuất hiện trong danh sách các lớp cơ sở được khai báo, rồi đến hàm thiết lập của lớp dẫn xuất. Thông tin được chuyển từ hàm thiết lập của lớp dẫn xuất sang hàm thiết lập của các lớp cơ sở.



Trong tình huống có lớp cơ sở **virtual**, vấn đề có khác đôi chút. Xem hình vẽ sau:



Trong trường hợp này, ta chỉ xây dựng một đối tượng duy nhất có kiểu A. Các tham số được truyền cho hàm thiết lập A được mô tả ngay khi định nghĩa hàm thiết

lập D. Và như vậy, không cần mô tả các thông tin truyền cho A ở mức B và C. Cần phải tuân theo quy tắc, quy định sau đây:

Thứ tự gọi các hàm thiết lập: hàm thiết lập của một lớp ảo luôn luôn được gọi trước các hàm thiết lập khác.

Với sơ đồ thừa kế có trên hình vẽ, thứ tự gọi hàm thiết lập sẽ là: A,B,C và cuối cùng là D. Chương trình sau minh chứng cho nhận xét này:

Ví dụ 5.15

```
/*mulinher3.cpp
Solution to multiple inheritance*/
#include <iostream.h>
#include <conio.h>
class A {
    float x,y;
public:
    A() {x = 0; y =0;}
    A(float ox, float oy) {
        cout<<"A::A(float, float)\n";
        x = ox; y = oy;
    }
    float getx() {
        return x;
    }
    float gety() {
        return y;
    }
};
class B : public virtual A {
public:
    B(float ox, float oy) : A(ox,oy) {
        cout<<"B::B(float, float)\n";
    }
};
```

```
class C : public virtual A {
public:
    C(float ox, float oy) :A(ox,oy) {
        cout<<"C::C(float, float)\n";
    }
};

class D : public B, public C {
public:
    D(float ox, float oy) : A(ox,oy),B(10,4),C(1,1) {
        cout<<"D::D(float, float);\n";
    }
};

void main() {
    clrscr();
    D d(2,3);
    cout<<"D d(2,3)\n";
    cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
    cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
    cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
    cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;

    cout<<"D d1(10,20);\n";
    D d1(10,20);
    cout<<"d1.C::getx() = "; cout<<d1.C::getx()<<endl;
    cout<<"d1.B::getx() = "; cout<<d1.B::getx()<<endl;
    cout<<"d1.C::gety() = "; cout<<d1.C::gety()<<endl;
    cout<<"d1.B::gety() = "; cout<<d1.B::gety()<<endl;
    getch();
}
```

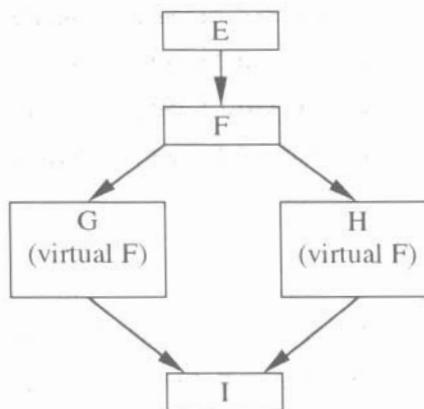
```
A::A(float, float)
B::B(float, float)
C::C(float, float)
```

```

D::D(float,float);
D d(2,3)
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3
D d1(10,20);
A::A(float, float)
B::B(float, float)
C::C(float, float)
D::D(float,float);
d1.C::getx() = 10
d1.B::getx() = 10
d1.C::gety() = 20
d1.B::gety() = 20

```

Ở tình huống khác:



thứ tự thực hiện các hàm thiết lập là: E, F, G, H, I. Ta xét chương trình sau:

Ví dụ 5.16

```

/*mulinher4.cpp
Solution to multiple inheritance*/
#include <iostream.h>
#include <conio.h>

```

```
class O {
    float o;
public:
    O(float oo) {
        cout<<"O::O(float)\n";
        o = oo;
    }
};

class A : public O{
    float x,y;
public:
    A(float oo, float ox, float oy):O(oo) {
        cout<<"A::A(float, float, float)\n";
        x = ox; y = oy;
    }

    float getx() {
        return x;
    }

    float gety() {
        return y;
    }
};

class B : public virtual A {
public:
    B(float oo, float ox, float oy) : A(oo,ox,oy) {
        cout<<"B::B(float, float, float)\n";
    }
};

class C : public virtual A {
public:
    C(float oo, float ox, float oy):A(oo,ox,oy) {
        cout<<"C::C(float, float, float)\n";
    }
};
```

```

    }

};

class D : public B, public C {
public:
    D(float oo, float ox, float oy) :
        A(oo,ox,oy),B(oo,10,4),C(oo,1,1) {
        cout<<"D::D(float, float, float);\n";
    }
};

void main() {
    clrscr();
    D d(2,3,5);
    cout<<"D d(2,3,5)\n";
    cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
    cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
    cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
    cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;
    cout<<"D d1(10,20,30);\n";
    D d1(10,20,30);
    cout<<"d1.C::getx() = "; cout<<d1.C::getx()<<endl;
    cout<<"d1.B::getx() = "; cout<<d1.B::getx()<<endl;
    cout<<"d1.C::gety() = "; cout<<d1.C::gety()<<endl;
    cout<<"d1.B::gety() = "; cout<<d1.B::gety()<<endl;
    getch();
}

```

```

O::O(float)
A::A(float, float, float)
B::B(float, float, float)
C::C(float, float, float)
D::D(float, float, float);
D d(2,3,5)
d.C::getx() = 3

```

```

d.B::getx() = 3
d.C::gety() = 5
d.B::gety() = 5
D d1{10,20,30};

O::O(float)

A::A(float, float, float)
B::B(float, float, float)
C::C(float, float, float)
D::D(float, float, float);

d1.C::getx() = 20
d1.B::getx() = 20
d1.C::gety() = 30
d1.B::gety() = 30

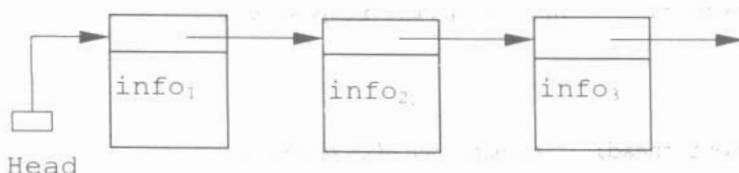
```

4.4 Danh sách mốc nối các đối tượng

Trong phần này, chúng ta đưa ra cách xây dựng một lớp, cho phép quản lý một danh sách mốc nối các đối tượng kiểu `point`. Các công việc cần phải làm là thiết kế một lớp đặc biệt để quản lý danh sách mốc nối độc lập với kiểu đối tượng liên quan. Một lớp như vậy hoàn toàn mang ý nghĩa trùu tượng, cùng với `point` sẽ là lớp thừa kế cho một lớp dẫn xuất chứa danh sách mốc nối.

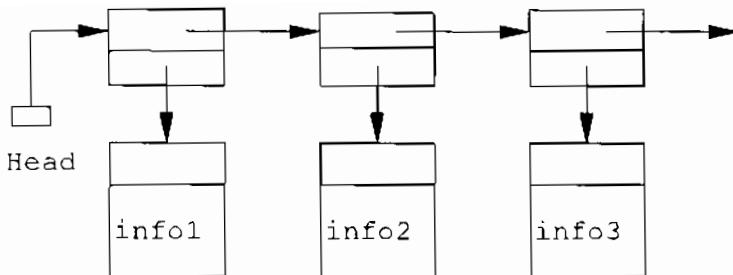
Xây dựng lớp trùu tượng

Ta giới hạn ở xây dựng một danh sách mốc nối đơn giản, với các phân tử có hai thành phần: thành phần dữ liệu và thành phần con trỏ, chỉ tới phân tử tiếp sau. Sơ đồ của danh sách như vậy có dạng như sau:



ở đây `Head` là con trỏ đối tượng, chỉ đến phân tử đầu tiên của danh sách mốc nối.

Tuy nhiên, có thể thông tin tương ứng với mỗi thành phần trong danh sách chưa được biết nên để biểu thị mỗi `infoi` ta dùng một con trỏ chỉ tới một đối tượng dữ liệu nào đó (`void *`). Ta có sơ đồ như sau:



Ở đây, mỗi phần tử của danh sách chứa:

- (i) Một con trỏ đến phần tử tiếp theo,
- (ii) Một con trỏ đến đối tượng chứa thông tin liên quan.

Còn lớp `list` để quản lý danh sách sẽ có ít nhất:

- (i) một thành phần dữ liệu: con trỏ đến phần tử đầu tiên (`Head`),
- (ii) một hàm thành phần có chức năng chèn vào danh sách một đối tượng tại một địa chỉ nào đó. Chú ý rằng địa chỉ này phải có kiểu `void *` để đảm bảo sự tương thích với các kiểu dữ liệu khác nhau.

Có thể hình dung sơ bộ lớp `list` được khai báo như sau:

```

struct element //cấu trúc một phần tử của danh sách list
{
    element *next; //con trỏ đến thành phần đi sau
    void *content; //con trỏ đến một đối tượng tùy ý
};

class list
{
    element *head; //con trỏ đến phần tử đầu tiên
public:
    list(); //hàm thiết lập
    ~list(); //hàm huỷ bỏ
    void add(void *); //thêm một phần tử vào đầu danh sách
    ...
}

```

Để quản lý danh sách ta có thể thêm các chức năng xử lý khác như:

- (i) Hiển thị các đối tượng được chỉ bởi danh sách
- (ii) Tìm kiếm một phần tử
- (iii) Xoá một phần tử

v.v...

Các thao tác này có thể được thực hiện trực tiếp trên list nhưng nhằm mục đích che dấu và do đặc điểm của các thông tin trong mỗi phần tử ta đều không biết, nên tốt nhất là để cho các hàm thành phần thực hiện.

Các thao tác trên danh sách được nêu ra ở trên đều gắn với cơ chế duyệt danh sách. Việc duyệt này cần phải được kiểm soát từ bên ngoài danh sách nhưng có thể dựa trên các thành phần hàm cơ sở là:

- (i) Khởi tạo việc duyệt
- (ii) Chuyển sang phần tử tiếp theo.

Hai hàm thành phần này đều dùng tham số là con trỏ đến phần tử hiện tại, ta đặt tên cho nó là current. Ngoài ra, hai hàm thành phần trên sẽ trả về các thông tin liên quan phần tử hiện tại, có thể lựa chọn:

- (i) Địa chỉ của phần tử hiện tại, kiểu là element *
- (ii) Địa chỉ của đối tượng hiện tại, có kiểu content *
- (iii) Nội dung dữ liệu của phần tử hiện tại.

Giải pháp đầu bao hàm giải pháp thứ hai bởi lẽ nếu ptr là địa chỉ của phần tử hiện tại trong danh sách, ptr->content sẽ cho địa chỉ của đối tượng dữ liệu tương ứng. Nhưng thực tế người lập trình không được quyền truy nhập đến các phần tử của danh sách list. Còn giải pháp thứ 3 làm mất khả năng kiểm tra các phần tử của danh sách và thay đổi chúng bởi lẽ chỉ với nội dung dữ liệu của phần tử hiện tại, ta không thể truy nhập tới các phần tử tiếp theo.

Ở đây, ta chọn giải pháp thứ 2. Trong trường hợp này, người ta vẫn chưa thể phát hiện phần tử cuối danh sách. Do vậy ta sẽ đưa vào một hàm thành phần bổ sung cho phép kiểm tra liệu đã gặp cuối danh sách hay chưa.

Theo các phân tích trên, ta sẽ dùng thêm ba hàm thành phần:

```
void *first();
void *prochain();
int last();
```

Sau đây là định nghĩa của lớp tròn tượng list:

Ví dụ 5.17

```
#include <stddef.h> //để định nghĩa NULL
//class list

struct element
{
    element *next;
    void *content;
};

class list
{
public:
    list()
    {
        head=NULL; current=head;
    }
    ~list();
    void add(void *)
    void first()
    {
        current=head;
    }
    void *prochain()
    {
        void *adel=NULL;
        if (current !=NULL) {
            adel =current->content;
            current=current->next;
        }
        return adel;
    }
    int last() {return (current==NULL);}
};

list::~list() {
```

```
element *suiv;
current=head;
while(current!=NULL)
    (suiv=current->next;delete current;current=suiv;}
}
void list::add(void *choses) {
    element *adel= new element;
    adel->next =head;
    adel->content=choses;
    head=adel;
}
```

Giả sử lớp point được định nghĩa như sau:

```
#include <iostream.h>
class point
{
    int x,y;
public:
    point(int abs=0,int ord=0) {x=abs;y=ord;}
    void display() {cout <<" Toa do : "<<x<<, " "<<y<<"\n";}
};
```

Ta định nghĩa một lớp mới list_points thừa kế từ list và point, cho phép quản lý danh sách mốc nối các đối tượng point như sau:

```
class list_points :public list,public point
{
public:
    list_points() {}
    void display();
};
void list_points::display()
{
    first();
    while (!last()) {
```

```
point *ptr=(point *)prochain();ptr->display();  
}
```

Sau đây là chương trình hoàn chỉnh:

Ví dụ 5.18

```
/*list hom.cpp  
homogenous list*/  
  
#include <iostream.h>  
  
#include <stddef.h> //de dinh nghia NULL  
  
#include <conio.h>  
  
//class list  
  
struct element {  
    element *next;  
    void *content;  
};  
  
class list {  
    element *head;  
    element *current;  
  
public:  
    list() {  
        head=NULL;current=head;  
    }  
    ~list();  
    void add(void *);  
    void first() {  
        current=head;}  
    void *nextelement() {  
        void *adel=NULL;  
        if (current !=NULL) {  
            adel =current->content;  
            current=current->next;  
        }  
        return adel;
```

```
    }
    int last() {return (current==NULL);}
};

list::~list() {
    element *suiv;
    current=head;
    while(current!=NULL) {
        suiv=current->next;
        delete current;
        current=suiv;
    }
}

void list::add(void *choses) {
    element *adel= new element;
    adel->next =head;
    adel->content=choses;
    head=adel;
}

class point
{
    int x,y;
public:
    point (int abs=0,int ord=0){
        x=abs;
        y=ord;
    }
    void display(){
        cout<<" Toa do : "<<x<<" "<<y<<"\n";
    }
};

class list_points :public list,public point {
public:
    list_points() {}
```

```

void display();
};

void list_points::display() {
    first();
    while (!last()) {
        point *ptr=(point *)nextelement();
        ptr->display();
    }
}

void main() {
    clrscr();
    list_points l;
    point a(2,3),b(5,9),c(0,8);
    l.add(&a); l.display();cout<<"-----\n";
    l.add(&b); l.display();cout<<"-----\n";
    l.add(&c); l.display();cout<<"-----\n";
    getch();
}

```

Toa do : 2 3

Toa do : 5 9

Toa do : 2 3

Toa do : 0 8

Toa do : 5 9

Toa do : 2 3

4.5 Tạo danh sách mốc nối không đồng nhất

Phản trước đã xét ví dụ về danh sách mốc nối đồng nhất. Về nguyên tắc, các phần tử trong danh sách có thể chứa các thông tin bất kỳ-danh sách không đồng nhất. Vấn đề chỉ phức tạp khi ta hiển thị các thông tin dữ liệu liên quan, vì khi đó ta hoàn toàn không biết được thông tin về kiểu đối tượng khi khai báo tham trả (có thể xem lại hàm `display_list()` để thấy rõ hơn).

Khái niệm lớp trừu tượng và hàm ảo cho phép ta xây dựng thêm một lớp dùng làm cơ sở cho các lớp có đối tượng tham gia danh sách mốc nối. Trong khai báo của lớp cơ sở này, có một hàm thành phần sẽ được định nghĩa lại trong các lớp dẫn xuất. Các hàm này có chức năng (trong các lớp cơ sở) hiển thị các thông tin liên quan đến đối tượng tương ứng của một lớp.

Đến đây, nhờ khai báo một tham trả đến lớp cơ sở này, mọi vấn đề truy nhập đến các hàm hiển thị thông tin về từng thành phần sẽ không còn khó khăn nữa. Sau đây là một chương trình minh họa cho nhận xét trên. Trong chương trình này, lớp cơ sở chung cho các lớp dùng làm kiểu dữ liệu cho các thành phần của danh sách chỉ được dùng để khai báo con trả để gọi tới các thành phần hàm hiển thị, nên tốt nhất ta là khai báo hàm đó như là hàm ảo thuận tuý và lớp tương ứng sẽ là lớp trừu tượng.

Ví dụ 5.19

```
/*list.hete.cpp
heterogenous list*/
#include <iostream.h>
#include <stddef.h> //chứa giá trị NULL.
#include <conio.h>
//lớp trừu tượng
class mere {
public:
    virtual void display() = 0 ; //hàm ảo thuận tuý
};

//lớp quản lý danh sách mốc nối
struct element //cấu trúc một thành phần của danh sách
{
    element *next;//chỉ đến thành phần sau
    void *content; //chỉ đến một đối tượng bất kỳ
};
```

```

class list
{
    element *head;
    element *current;
public:
    list(){
        head = NULL;
        current=head;
    }
~list();
void add(void *);//thêm một thành phần vào đầu danh sách
void first(){
    current=head;
}

void *nextelement(){
    void *adnext=NULL;
    if (current !=NULL)
    {
        adnext=current->content;
        current=current->next;
    }
    return adnext;
}

void display_list();
int last() {return (current==NULL);}
};

list::~list() {
    element *suiv;
    current = head;
    while(current !=NULL) {

```

```

        suiv =current->next;
        delete current;
        current=suiv;
    }
}

void list::add(void * chose) {
    element *adel = new element;
    adel->next = head;
    adel->content = chose;
    head = adel;
}

void list::display_list() {
    mere *ptr //chú ý phải là mere* chư khong phai void *
    first(); //in từ đầu danh sách
    while (!last()) {
        ptr =(mere *) nextelement();
        ptr->display();
    }
}

//lớp điểm
class point : public mere {
    int x,y;
public:
    point(int abs =0, int ord =0) {x=abs;y=ord;}
    void display()
        (cout << "Toa do : "<<x<<" "<<y<<"\n");
};

//lớp số phức
class complexe :public mere {
    double reel,imag;
public:
    complexe(double r=0,double i=0) {reel =r; imag=i;}
}

```

```

void display() {
    cout<<" So phuc : "<<reel<<" + "<<imag<<"i\n";
}
};

//chương trình thử
void main() {
    clrscr();
    list l;
    point a(2,3),b(5,9);
    complexe x(4.5,2.7), y(2.35,4.86);
    l.add(&a);
    l.add(&x);
    l.display_list();
    cout<<"-----\n";
    l.add(&y);
    l.add(&b);
    l.display_list();
    getch();
}

```

```

So phuc : 4.5 + 2.7i
Toa do : 2 3
-----
Toa do : 5 9
So phuc : 2.35 + 4.86i
So phuc : 4.5 + 2.7i
Toa do : 2 3

```

5. TÓM TẮT

5.1 Ghi nhớ

Thừa kế nâng cao khả năng sử dụng lại của các đoạn mã chương trình.

Người lập trình có thể khai báo lớp mới thừa kế dữ liệu và hàm thành phần từ một lớp cơ sở đã được định nghĩa trước đó. Ta gọi lớp mới là lớp dẫn xuất.

Trong đơn thừa kế, một lớp chỉ có thể có một lớp cơ sở. Trong đa thừa kế cho phép một lớp là dẫn xuất của nhiều lớp.

Lớp dẫn xuất thường bổ sung thêm các thành phần dữ liệu và các hàm thành phần trong định nghĩa, ta nói lớp dẫn xuất cụ thể hơn so với lớp cơ sở và vì vậy thường mô tả một lớp các đối tượng có phạm vi hẹp hơn lớp cơ sở.

Lớp dẫn xuất không có quyền truy nhập đến các thành phần **private** của lớp cơ sở. Tuy nhiên lớp cơ sở có quyền truy xuất đến các thành phần công cộng và được bảo vệ (**protected**).

Hàm thiết lập của lớp dẫn xuất thường tự động gọi các hàm thiết lập của các lớp cơ sở để khởi tạo giá trị cho các thành phần trong lớp cơ sở.

Hàm huỷ bỏ được gọi theo thứ tự ngược lại.

Thuộc tính truy nhập **protected** là mức trung gian giữa thuộc tính **public** và **private**. Chỉ có các hàm thành phần và hàm bạn của lớp cơ sở và lớp dẫn xuất có quyền truy xuất đến các thành phần **protected** của lớp cơ sở.

Có thể định nghĩa lại các thành phần của lớp cơ sở trong lớp dẫn xuất khi thành phần đó không còn phù hợp trong lớp dẫn xuất.

Có thể gán nội dung đối tượng lớp dẫn xuất cho một đối tượng lớp cơ sở. Một con trỏ lớp dẫn xuất có thể chuyển đổi thành con trỏ lớp cơ sở.

Hàm ảo được khai báo với từ khóa **virtual** trong lớp cơ sở.

Các lớp dẫn xuất có thể đưa ra các cài đặt lại cho các hàm ảo của lớp cơ sở nếu muốn, trái lại chúng có thể sử dụng định nghĩa đã nêu trong lớp cơ sở.

Nếu hàm ảo được gọi bằng cách tham chiếu qua tên một đối tượng thì tham chiếu đó được xác định dựa trên lớp của đối tượng tương ứng.

Một lớp có hàm ảo không có định nghĩa (hàm ảo thuần túy) được gọi là lớp trừu tượng. Các lớp trừu tượng không thể dùng để khai báo các đối tượng nhưng có thể khai báo con trỏ có kiểu lớp trừu tượng.

Tính đa hình là khả năng các đối tượng của các lớp khác nhau có quan hệ thừa kế phản ứng khác nhau đối với cùng một lời gọi hàm thành phần.

Tính đa hình được cài đặt dựa trên hàm ảo.

Khi một yêu cầu được đưa ra thông qua con trỏ lớp cơ sở để tham chiếu đến hàm ảo, C++ lựa chọn hàm thích hợp trong lớp dẫn xuất thích hợp gần với đối tượng đang được trả tới.

Thông qua việc sử dụng hàm ảo và tính đa hình, một lời gọi hàm thành phần có thể có những hành động khác nhau dựa vào kiểu của đối tượng nhận được lời gọi.

5.2 Các lời thường gặp

Cho con trỏ lớp dẫn xuất chỉ đến đối tượng lớp cơ sở và gọi tới các hàm thành phần không có trong lớp cơ sở.

Định nghĩa lại trong lớp dẫn xuất một hàm ảo trong lớp cơ sở mà không đảm bảo chắc chắn rằng phiên bản mới của hàm trong lớp dẫn xuất cũng trả về cùng giá trị như phiên bản cũ của hàm.

Khai báo đối tượng của lớp trừu tượng.

Khai báo hàm thiết lập là hàm ảo.

5.3 Một số thói quen lập trình tốt

Khi thừa kế các khả năng không cần thiết trong lớp dẫn xuất, tối thiểu nên định nghĩa lại chúng.

6. BÀI TẬP

Bài 5.1.

Mô phỏng các thao tác trên một cây nhị phân tìm kiếm với nội dung tại mỗi nút là một số nguyên.

Bài 5.2.

Mô phỏng hoạt động của ngăn xếp, hàng đợi cây nhị phân dưới dạng danh sách mốc nối sử dụng mô hình lớp.

KHUÔN HÌNH

(Template)

Mục đích chương này:

1. Hiểu được lợi ích của việc sử dụng khuôn hình hàm và khuôn hình lớp để viết chương trình.
2. Biết cách tạo và sử dụng một khuôn hình hàm và khuôn hình lớp.
3. Khái niệm các tham số kiểu và các tham số biểu thức trong khuôn hình hàm, khuôn hình lớp.
4. Định nghĩa chồng khuôn hình hàm.
5. Cụ thể hóa một khuôn hình hàm, một hàm thành phần của khuôn hình lớp.
6. Thuật toán sản sinh một thể hiện hàm (hàm thể hiện) của một khuôn hình hàm.
7. Các vấn đề khác của lập trình hướng đối tượng liên quan đến khuôn hình lớp.

1. KUÔN HÌNH HÀM

1.1 Khuôn hình hàm là gì?

Ta đã biết định nghĩa chồng hàm cho phép dùng một tên duy nhất cho nhiều hàm thực hiện các công việc khác nhau. Khái niệm khuôn hình hàm cũng cho phép sử dụng cùng một tên duy nhất để thực hiện các công việc khác nhau, tuy nhiên so với định nghĩa chồng hàm, nó có phần mạnh hơn và chặt chẽ hơn; mạnh hơn vì chỉ cần viết định nghĩa khuôn hình hàm một lần, rồi sau đó chương trình biên dịch làm cho nó thích ứng với các kiểu dữ liệu khác nhau; chặt chẽ hơn bởi vì dựa theo khuôn hình hàm, tất cả các hàm thể hiện được sinh ra bởi trình biên dịch sẽ tương ứng với cùng một định nghĩa và như vậy sẽ có cùng một giải thuật.

1.2 Tạo một khuôn hình hàm

Giả thiết rằng chúng ta cần viết một hàm `min` đưa ra giá trị nhỏ nhất trong hai giá trị có cùng kiểu. Ta có thể viết một định nghĩa như thế đối với kiểu `int` như sau:

```
int min (int a, int b) {  
    if (a < b) return a;  
    else return b;
```

```
}
```

Giả sử, ta lại phải viết định nghĩa hàm `min()` cho kiểu `double, float, char, char*`...

```
float min(float a, float b) {  
    if (a < b) return a;  
    else b; }
```

Nếu tiếp tục như vậy, sẽ có khuynh hướng phải viết rất nhiều định nghĩa hàm hoàn toàn tương tự nhau; chỉ có kiểu dữ liệu các tham số là thay đổi. Các chương trình biên dịch C++ hiện có cho phép giải quyết đơn giản vấn đề trên bằng cách định nghĩa một khuôn hình hàm duy nhất theo cách như sau:

```
#include <iostream.h>  
//tạo một khuôn hình hàm  
template <class T> T min(T a, T b) {  
    if (a < b) return a;  
    else return b;  
}
```

So sánh với định nghĩa hàm thông thường, ta thấy chỉ có dòng đầu tiên bị thay đổi:

```
template <class T> T min (T a, T b)
```

lòng đố

```
template<class T>
```

xác định ràng đó là một khuôn hình với một tham số kiểu T;

Phản còng lại

```
T min(T a, T b)
```

nói rằng, `min()` là một hàm với hai tham số hình thức kiểu T và có giá trị trả về cũng là kiểu T.

1.3 Sử dụng khuôn hình hàm

1.3.1 Khuôn hình hàm cho kiểu dữ liệu cơ sở

Để sử dụng khuôn hình hàm `min()` vừa tạo ra, chỉ cần sử dụng hàm `min()` trong những điều kiện phù hợp (ở đây có nghĩa là hai tham số của hàm có cùng kiểu dữ liệu). Như vậy, nếu trong một chương trình có hai tham số nguyên n và p,

với lời gọi `min(n, p)` chương trình biên dịch sẽ tự động sản sinh ra hàm `min()` (ta gọi là một hàm thể hiện) tương ứng với hai tham số kiểu nguyên `int`. Nếu chúng ta gọi `min()` với hai tham số kiểu `float`, chương trình biên dịch cũng sẽ tự động sản sinh một hàm thể hiện `min` khác tương ứng với các tham số kiểu `float` và cứ thế. Sau đây là một ví dụ hoàn chỉnh:

Ví dụ 6.1

```
/*template1.cpp*/
#include <iostream.h>
#include <conio.h>

//tạo một khuôn hình hàm min
template <class T> T min(T a, T b) {
    if (a < b) return a;
    else return b;
}

//ví dụ sử dụng khuôn hình hàm min
void main() {
    clrscr();
    int n = 4, p = 12;
    float x = 2.5, y = 3.25;
    cout << "min (n, p) = " << min (n, p) << "\n"; //int min(int, int)
    cout << "min (x, y) = " << min (x, y) << "\n"; //float min(float, float)
    getch();
}
```

```
min (n, p) = 4
min (x, y) = 2.5
```

1.3.2 Khuôn hình hàm min cho kiểu `char*`

```
/*template2.cpp*/
#include <iostream.h>
#include <conio.h>

template <class T> T min (T a, T b) {
    if (a < b) return a;
```

```

    else return b;
}
void main() {
    clrscr();
    char * adr1 = "DHBK";
    char * adr2 = "CDSD";
    cout << "min (adr1, adr2) =" << min (adr1, adr2);
    getch();
}

```

min (adr1, adr2) = DHBK

Kết quả khá thú vị vì ta hy vọng hàm `min()` trả về xâu "CDSD". Thực tế, với biểu thức `min(adr1, adr2)`, chương trình biên dịch đã sinh ra hàm thể hiện sau đây:

```

char * min(char * a, char * b) {
    if (a < b) return a;
    else return b;
}

```

Việc so sánh $a < b$ thực hiện trên các giá trị biến trả (ở đây trong các khuôn hình máy PC ta luôn luôn có $a < b$). Ngược lại việc hiển thị thực hiện bởi toán tử `<<` sẽ đưa ra xâu ký tự trả bởi con trả ký tự.

1.3.3 Khuôn hình hàm `min` với kiểu dữ liệu lớp

Để áp dụng khuôn hình hàm `min()` ở trên với kiểu lớp, cần phải định nghĩa lớp sao cho có thể áp dụng phép toán so sánh "`<`" với các đối tượng của lớp này, nghĩa là ta phải định nghĩa một hàm toán tử `operator <` cho lớp. Sau đây là một ví dụ minh họa:

Ví dụ 6.2

```

/*template3.cpp*/
#include <iostream.h>
#include <conio.h>
//khuôn hình hàm min
template <class T> T min( T a, T b ) {
    if (a < b) return a;
}

```

```

        else return b;
    }

//lớp vect
class vect {
    int x, y;

public:
    vect(int abs = 0, int ord = 0) { x= abs, y= ord; }
    void display() { cout <<x<<" "<<y<<"\n"; }
    friend int operator < (vect , vect);
};

int operator < (vect a, vect b) {
    return a.x*a.x + a.y*a.y < b.x*b.x + b.y*b.y;
}

void main() {
    clrscr();
    vect u(3,2),v(4,1);
    cout<<"min (u, v) = ";
    min(u,v).display();
    getch();
}

```

min (u, v) = 3 2

Nếu ta áp dụng khuôn hình hàm `min()` đối với một lớp mà chưa định nghĩa toán tử “<”, chương trình biên dịch sẽ đưa ra một thông báo lỗi tương tự như việc định nghĩa một hàm `min()` cho kiểu lớp đó.

1.4 Các tham số kiểu của khuôn hình hàm

Phần này trình bày cách đưa vào các tham số kiểu trong một khuôn hình hàm, để chương trình biên dịch sản sinh một hàm thể hiện.

1.4.1 Các tham số kiểu trong định nghĩa khuôn hình hàm

Một cách tổng quát, khuôn hình hàm có thể có một hay nhiều tham số kiểu, với mỗi tham số này có từ khoá **class** đi liền trước, chẳng hạn như:

```
template <class T, class U> int fct (T a, T *b, U c) {...}
```

Các tham số này có thể để ở bất kỳ đâu trong định nghĩa của khuôn hình hàm, nghĩa là:

- (i) Trong dòng tiêu đề (như đã chỉ ra trong ví dụ trên)
- (ii) Trong các khai báo các biến cục bộ
- (iii) Trong các chỉ thị thực hiện.

Chẳng hạn:

```
template <class T, class U> int fct (T a, T *b, U c) {
    T x; //biến cục bộ x kiểu T
    U *adr; //biến cục bộ adr kiểu U *
    ...
    adr = new T [10]; //cấp phát một mảng 10 thành phần kiểu T ... n = sizeof(T);
}
```

Ta xem chương trình sau:

Ví dụ 6.3

```
/*templat4.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T, class U> T fct(T x, U y, T z) {
    return x + y + z;
}
void main() {
    clrscr();
    int n= 1, p = 2, q = 3;
    float x =2.5, y = 5.0;
    cout <<fct( n, x, p)<<"\n";// (int)5
    cout <<fct(x, n, y)<<"\n"; // (float)8.5
    cout <<fct(n, p, q)<<"\n"; // (int)6
    //cout <<fct(n, p, x)<<"\n"; // lỗi
    getch();
}
```

Trong mọi trường hợp, mỗi tham số kiểu phải xuất hiện ít nhất một lần trong khai báo danh sách các tham số hình thức của khuôn hình hàm. Điều đó hoàn toàn logic bởi vì nhờ các tham số này, chương trình dịch mới có thể sản sinh ra hàm thể hiện cần thiết. Điều gì sẽ xảy ra nếu trong danh sách các tham số của khuôn hình hàm không có đủ các tham số kiểu? Hiển nhiên khi đó chương trình dịch không thể xác định các tham số kiểu dữ liệu thực ứng với các tham số kiểu hình thức trong template<...>.

Khuôn hình hàm sau đây thực hiện trao đổi nội dung của hai biến.

Ví dụ 6.4

```
/*templat5.cpp*/
#include <iostream.h>
#include <conio.h>
//định nghĩa khuôn hình hàm đổi chỗ nội dung hai biến với kiểu bất kỳ
template <class X> void swap(X &a, X &b) {
    X temp;
    temp=a;
    a=b;
    b=temp;
}
void main() {
    clrscr();
    int i=10, j=20;
    float x=10.1, y=23.1;
    cout<<"I J ban dau: "<<i<<" "<<j<<endl;
    cout<<"X Y ban dau: "<<x<<" "<<y<<endl;
    swap(i,j); //đổi chỗ hai số nguyên
    swap(x,y); //đổi chỗ hai số nguyên
    cout<<"I J sau khi doi cho: "<<i<<" "<<j<<endl;
    cout<<"X Y sau khi doi cho: "<<x<<" "<<y<<endl;
    getch();
}
```

I J ban dau: 10 20

X Y ban dau: 10.1 23.1

I J sau khi đổi chỗ: 20 10

X Y sau khi đổi chỗ: 23.1 10.1

1.5 Giải thuật sản sinh một hàm thể hiện

Để tạo ra một hàm thể hiện, ta cần xác định các tham số và trả về giá trị. Khi đó, ta có thể sử dụng lệnh `return` để trả về giá trị.

Trở lại khuôn hình hàm `min()`:

```
template <class T> T min(T a, T b) {
    if (a < b) return a;
    else return b;
}
```

Với các khai báo:

```
int n;
char c;
```

câu hỏi đặt ra là: chương trình dịch sẽ làm gì khi gặp lời gọi kiểu như là `min(n, c)`? Câu trả lời dựa trên hai nguyên tắc sau đây:

- C++ quy định phải có một sự tương ứng chính xác giữa kiểu của tham số hình thức và kiểu tham số thực sự được truyền cho hàm, tức là ta chỉ có thể sử dụng khuôn hình hàm `min()` trong các lời gọi với hai tham số có cùng kiểu. Lời gọi `min(n, c)` không được chấp nhận và sẽ gây ra lỗi biên dịch.
- C++ thậm chí còn không cho phép các chuyển kiểu thông thường như là: T thành `const T` hay `T[]` thành `T *`, những trường hợp hoàn toàn được phép trong định nghĩa chồng hàm.

Ta tham khảo đoạn chương trình sau đây:

```
int n;
char c;
unsigned int q;
const int r = 10;
int t[10];
int *adi;
...
min (n, c) //lỗi
min (n, q) //lỗi
min (n, r) //lỗi
```

```
min (t, adi) //lỗi
```

1.6 Khởi tạo các biến có kiểu dữ liệu chuẩn

Trong khuôn hình hàm, tham số kiểu có thể tương ứng khi thì một kiểu dữ liệu chuẩn, khi thì một kiểu dữ liệu lớp. Sẽ làm gì khi ta cần phải khai báo bên trong khuôn hình hàm một đối tượng và truyền một hay nhiều tham số cho hàm thiết lập của lớp. Xem ví dụ sau đây:

```
template <class T> fct(T a) {
    T x(3); //x là một đối tượng cục bộ kiểu T mà chúng ta xây dựng bằng cách
    //truyền giá trị 3 cho hàm thiết lập ...
}
```

Khi sử dụng hàm `fct()` cho một kiểu dữ liệu lớp, mọi việc đều tốt đẹp. Ngược lại, nếu chúng ta cố gắng áp dụng cho một kiểu dữ liệu chuẩn, chẳng hạn như `int`, khi đó chương trình dịch sản sinh ra hàm sau đây:

```
fct( int a) { int x(3); ... }
```

Để cho chỉ thi

```
int x(3) ;
```

không gây ra lỗi, C++ đã ngầm hiểu câu lệnh đó như là phép khởi tạo biến `x` với giá trị 3, nghĩa là:

```
int x = 3;
```

Một cách tương tự:

```
double x(3.5); //thay vì double x = 3.5;
char c('e'); //thay vì char c = 'e';
```

1.7 Các hạn chế của khuôn hình hàm

Về nguyên tắc, khi định nghĩa một khuôn hình hàm, một tham số kiểu có thể tương ứng với bất kỳ kiểu dữ liệu nào, cho dù đó là một kiểu chuẩn hay một kiểu lớp do người dùng định nghĩa. Do vậy không thể hạn chế việc thể hiện đối với một số kiểu dữ liệu cụ thể nào đó. Chẳng hạn, nếu một khuôn hình hàm có dòng đầu tiên:

```
template <class T> void fct(T)
```

chúng ta có thể gọi `fct()` với một tham số với kiểu bất kỳ: `int`, `float`, `int *`, `int **`, `T` (*`T` là một kiểu dữ liệu nào đấy)

Tuy nhiên, chính định nghĩa bên trong khuôn hình hàm lại chứa một số yếu tố có thể làm cho việc sản sinh hàm thể hiện không đúng như mong muốn. Ta gọi đó là các hạn chế của các khuôn hình hàm.

Đầu tiên, chúng ta có thể cho rằng một tham số kiểu có thể tương ứng với một con trỏ. Do đó, với dòng tiêu đề:

```
template <class T> void fct(T *)
```

ta chỉ có thể gọi fct() với một con trỏ đến một kiểu nào đó: `int*`, `int **`, `t *`, `t **`.

Trong các trường hợp khác, sẽ gây ra các lỗi biên dịch. Ngoài ra, trong định nghĩa của một khuôn hình hàm, có thể có các chỉ thị không thích hợp đối với một số kiểu dữ liệu nhất định. Chẳng hạn, khuôn hình hàm:

```
template <class T> T min(T a, T b) {
    if (a < b) return a;
    else return b;
}
```

không thể dùng được nếu T tương ứng với một kiểu lớp trong đó phép toán “`<`” không được định nghĩa chồng. Một cách tương tự với một khuôn hình hàm kiểu:

```
template <class T> void fct(T) {
    ... T x(2, 5); /*đối tượng cục bộ được khởi tạo bằng một hàm thiết lập với
                    hai tham số*/
}
```

không thể áp dụng cho các kiểu dữ liệu lớp không có hàm thiết lập với hai tham số.

Tóm lại, mặc dù không tồn tại một cơ chế hình thức để hạn chế khả năng áp dụng của các khuôn hình hàm, nhưng bên trong mỗi một khuôn hình hàm đều có chứa những nhân tố để người ta có thể biết được khuôn hình hàm đó có thể được áp dụng đến mức nào.

1.8 Các tham số biểu thức của một khuôn hình hàm

Trong định nghĩa của một khuôn hình hàm có thể khai báo các tham số hình thức với kiểu xác định. Ta gọi chúng là các tham số biểu thức. Chương trình `templat6.cpp` sau đây định nghĩa một khuôn hình hàm cho phép đếm số lượng các phần tử nul (0 đối với các giá trị số hoặc NULL nếu là con trỏ) trong một bảng với kiểu bất kỳ và kích thước nào đó:

Ví dụ 6.5

```
/*templat6.cpp*/
```

```
#include <iostream.h>
#include <conio.h>
template <class T> int compte(T * tab, int n) {
    int i, nz = 0;
    for (i=0; i<n; i++)
        if (!tab[i])
            nz++;
    return nz;
}
void main() {
    clrscr();
    int t[5] = {5, 2, 0, 2, 0};
    char c[6] = { 0, 12, 0, 0, 0, 0};
    cout<<" compte (t) = "<<compte(t, 5)<<"\n";
    cout<<" compte (c) = "<<compte(c,6)<<"\n";
    getch();
}
```

```
compte (t) = 2
compte (c) = 4
```

Ta có thể nói rằng khuôn hình hàm `compte` định nghĩa một họ các hàm `compte` trong đó kiểu của tham số đầu tiên là tùy ý (được xác định bởi lời gọi); còn kiểu của tham số thứ hai đã xác định (kiểu `int`).

1.9 Định nghĩa chồng các khuôn hình hàm

Giống như việc định nghĩa chồng các hàm thông thường, C++ cho phép định nghĩa chồng các khuôn hình hàm, tức là có thể định nghĩa một hay nhiều khuôn hình hàm có cùng tên nhưng với các tham số khác nhau. Điều đó sẽ tạo ra nhiều họ các hàm (mỗi khuôn hình hàm tương ứng với một họ các hàm). Ví dụ có ba họ hàm `min`:

- (i) Họ thứ nhất bao gồm các hàm tìm giá trị nhỏ nhất trong hai giá trị,
- (ii) Họ thứ hai tìm số nhỏ nhất trong ba số,
- (iii) Họ thứ ba tìm số nhỏ nhất trong một mảng.

Ví dụ 6.6

```
/*templat7.cpp*/
#include <iostream.h>
#include <conio.h>
//khuôn hình 1
template <class T> T min(T a, T b) {
    if (a < b) return a;
    else return b;
}
//khuôn hình 2
template <class T> T min(T a, T b, T c) {
    return min (min (a, b), c);
}
//khuôn hình 3
template <class T> T min (T *t, int n) {
    T res = t[0];
    for(int i = 1; i < n; i++)
        if (res > t[i])
            res = t[i];
    return res;
}
void main() {
    clrscr();
    int n = 12, p = 15, q = 2;
    float x = 3.5, y = 4.25, z = 0.25;
    int t[6] = {2, 3, 4,-1, 21};
```

```

char c[4] = {'w', 'q', 'a', 'Q'};
cout<<"min(n,p) = "<<min(n,p)<<"\n"; // khuôn hình 1 int min(int,int)
cout<<"min(n,p,q) = "<<min(n,p,q)<<"\n"; // khuôn hình 2 int min(int,int,int)
cout<<"min(x,y) = "<<min(x,y)<<"\n"; // khuôn hình 1 float min(float,float)
cout<<"min(x,y,z) = "<<min(x,y,z)<<"\n"; // khuôn hình 2 float min(float,float,
float)
cout<<"min(t,6) = "<<min(t,6)<<"\n"; // khuôn hình 3 int min(int *,int)
cout<<"min(c,4) = "<<min(c,4)<<"\n"; // khuôn hình 3 char min(char *,int)
getch();
}

```

```

min(n,p) = 12
min(n,p,q) = 2
min(x,y) = 3.5
min(x,y,z) = 0.25
min(t,6) = -1
min(c,4) = Q

```

Nhận xét

Cũng giống như định nghĩa chồng các hàm, việc định nghĩa chồng các khuôn hình hàm có thể gây ra sự nh�p nh ng trong việc sản sinh các hàm th  hi n. Chẳng hạn với b n họ hàm sau đây:

```

template <class T> T fct(T, T) {...}
template <class T> T fct(T *, T) {...}
template <class T> T fct(T, T*) {...}
template <class T> T fct(T *, T*) {...}

```

X t các câu lệnh sau đây:

```

int x;
int y;

```

Lời gợi

```
fct(&x, &y)
```

có thể tương ứng với khuôn hình hàm 1 hay khuôn hình hàm 4.

1.10 Cụ thể hóa các hàm thể hiện

Một khuôn hình hàm định nghĩa một họ các hàm dựa trên một định nghĩa chung, nói cách khác chúng thực hiện theo cùng một giải thuật. Trong một số trường hợp, sự tổng quát này có thể chịu “rủi ro”, chẳng hạn như trong trường hợp áp dụng khuôn hình hàm min cho kiểu **char*** như đã nói ở trên. Khái niệm cụ thể hóa, đưa ra một giải pháp khắc phục các “rủi ro” kiểu như trên. C++ cho phép ta cung cấp, ngoài định nghĩa của một khuôn hình hàm, định nghĩa của một số các hàm cho một số kiểu dữ liệu của tham số. Ta xét chương trình ví dụ sau đây:

Ví dụ 6.7

```
/*template8.cpp*/
#include <iostream.h>
#include <string.h>
#include <conio.h>
//khuôn hình hàm min
template <class T> T min (T a, T b) {
    if (a < b) return a;
    else return b;
}
//hàm min cho kiểu xâu ký tự
char * min (char *cha, char *chb) {
    if (strcmp(cha, chb) <0) return cha;
    else return chb;
}
void main() {
    clrscr();
    int n= 12, p = 15;
    char *adr1= "DHBK", *adr2 ="CD2D";
    cout<<"min(n, p) = "<<min(n, p)<<"\n"; //khuôn hình hàm
    cout<<"min(adr1,adr2) = "<<min(adr1,adr2)<<endl; //hàm char * min (char *
                                                //char *)
```

```

getch();  

}  
} đưa ra kết quả là 12 và 12 là giá trị trả về của min(n, p) và min(adr1, adr2).
min(n, p) = 12  
min(adr1, adr2) = CD2D

```

Như vậy, bản chất của cụ thể hoá khuôn hình hàm là định nghĩa các hàm thông thường có cùng tên với khuôn hình hàm để giải quyết một số trường hợp rủi ro khi ta áp dụng khuôn hình hàm cho một số kiểu dữ liệu đặc biệt nào đó.

1.11 Tổng kết về các khuôn hình hàm

Một cách tổng quát, ta có thể định nghĩa một hay nhiều khuôn hình cùng tên, mỗi khuôn hình có các tham số kiểu cũng như là các tham số biểu thức riêng. Hơn nữa, có thể cung cấp các hàm thông thường với cùng tên với một khuôn hình hàm; trong trường hợp này ta nói đó là sự cụ thể hoá một hàm thể hiện.

Trong trường hợp tổng quát khi có đồng thời cả hàm định nghĩa chung và khuôn hình hàm, chương trình dịch lựa chọn hàm tương ứng với một lời gọi hàm dựa trên các nguyên tắc sau đây:

- (i) Đầu tiên, kiểm tra tất cả các hàm thông thường cùng tên và chú ý đến sự tương ứng chính xác; nếu chỉ có một hàm phù hợp, hàm đó được chọn; còn nếu có nhiều hàm cùng thỏa mãn (có sự nhập nhằng) sẽ tạo ra một lỗi biên dịch và quá trình tìm kiếm bị gián đoạn.
- (ii) Nếu không có hàm thông thường nào tương ứng chính xác với lời gọi, khi đó ta kiểm tra tất cả các khuôn hình hàm có cùng tên với lời gọi; nếu chỉ có một tương ứng chính xác được tìm thấy, hàm thể hiện tương ứng được sản sinh và vấn đề được giải quyết; còn nếu có nhiều hơn một khuôn hình hàm (có sự nhập nhằng) điều đó sẽ gây ra lỗi biên dịch và quá trình tìm kiếm bị ngắt.
- (iii) Cuối cùng, nếu không có khuôn hình hàm phù hợp, ta kiểm tra một lần nữa tất cả các hàm thông thường cùng tên với lời gọi. Trong trường hợp này chúng ta phải tìm kiếm sự tương ứng dựa vào cả các chuyển kiểu cho phép trong C/C++.

2. KHUÔN HÌNH LỚP

2.1 Khuôn hình lớp là gì?

Bên cạnh khái niệm khuôn hình hàm, C++ còn cho phép định nghĩa khuôn hình lớp. Cũng giống như khuôn hình hàm, ở đây ta chỉ cần viết định nghĩa các khuôn hình lớp một lần rồi sau đó có thể áp dụng chúng với các kiểu dữ liệu khác nhau để được các lớp thể hiện khác nhau.

2.2 Tạo một khuôn hình lớp

Ta thường tạo ra lớp point theo kiểu (ở đây ta bỏ qua định nghĩa của các hàm thành phần):

```
class point {
    int x, y;
public:
    point (int abs =0, int ord =0);
    void display();
    //...
};
```

Trong ví dụ này, ta định nghĩa một lớp các điểm có tọa độ nguyên. Nếu muốn tọa độ điểm có kiểu dữ liệu khác (**float**, **double**, **long**, **unsigned int**) ta phải định nghĩa một lớp khác bằng cách thay thế, trong định nghĩa lớp point, từ khoá **int** bằng từ khoá tương ứng với kiểu dữ liệu mong muốn.

Để tránh sự trùng lặp trong các tình huống như trên, chương trình dịch C++ cho phép định nghĩa một khuôn hình lớp và sau đó, áp dụng khuôn hình lớp này với các kiểu dữ liệu khác nhau để thu được các lớp thể hiện như mong muốn:

```
template <class T> class point {
    T x; T y;
public:
    point (T abs=0, T ord=0);
    void display();
};
```

Cũng giống như các khuôn hình hàm, tập hợp template<class T> xác định rằng đó là một khuôn hình trong đó có một tham số kiểu T; Cũng cần phải nhắc lại rằng, C++ sử dụng từ khoá **class** chỉ để nói rằng T đại diện cho một kiểu dữ liệu nào đó. Tiếp theo đây ta bàn đến việc định nghĩa các hàm thành phần của khuôn hình lớp. Người ta phân biệt hai trường hợp: (i) Khi hàm thành phần được định nghĩa bên trong định nghĩa lớp trường hợp này không có gì thay đổi. Xét định nghĩa hàm thiết lập sau đây:

```
template <class T> class point {
    T x; T y;
public:
    point(T abs=0, T ord=0) {
```

```

    x = abs; y = ord;
}

...
};

}

```

(ii) Ngược lại, khi định nghĩa của hàm thành phần nằm ngoài định nghĩa lớp, khi đó cần phải “nhắc lại” cho chương trình dịch biết: các tham số kiểu của khuôn hình lớp, có nghĩa là phải nhắc lại: template <class T> trước định nghĩa hàm, còn tên của khuôn hình lớp được viết như là point<T>

Tóm lại, dòng tiêu đề đầy đủ cho hàm thành phần display() của khuôn hình lớp point như sau:

```
template <class T> void point<T>::display()
```

Sau đây là định nghĩa đầy đủ của khuôn hình lớp point:

```
#include <iostream.h>

//tạo khuôn hình hàm

template <class T> class point { T x, y;

public:
    // định nghĩa hàm thành phần ở bên trong khuôn hình lớp
    point(T abs = 0, T ord = 0) {
        x = abs; y = ord;
    }

    void display();
};

// định nghĩa hàm thành phần ở bên ngoài khuôn hình lớp
template <class T> void point<T>::display() {
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
}
```

2.3 Sử dụng khuôn hình lớp

Một khi khuôn hình lớp point đã được định nghĩa, một khai báo như :

```
point<int> ai;
```

khai báo một đối tượng ai có hai thành phần tọa độ là kiểu nguyên (`int`). Điều đó có nghĩa là `point<int>` có vai trò như một kiểu dữ liệu lớp; người ta gọi nó là một lớp thể hiện của khuôn hình lớp `point`. Một cách tổng quát, khi áp dụng một kiểu dữ liệu nào đó với khuôn hình lớp `point` ta sẽ có được một lớp thể hiện tương ứng với kiểu dữ liệu. Như vậy:

```
point<double> ad;
```

định nghĩa một đối tượng `ad` có các tọa độ là số thực; còn với `point<double>` đóng vai trò một lớp và được gọi là một lớp thể hiện của khuôn hình lớp `point`.

Trong trường hợp cần phải truyền các tham số cho các hàm thiết lập, ta làm bình thường. Ví dụ:

```
point<int> ai(3, 5);
point<double> ad(2.5, 4.4);
```

2.4 Ví dụ sử dụng khuôn hình lớp

Ta xét ví dụ sau:

Ví dụ 6.8

```
/*template9.cpp*/
#include <iostream.h>
#include <conio.h>
//tạo một khuôn hình lớp
template <class T> class point {
    T x, y;
public:
    point(T abs = 0, T ord = 0) {
        x = abs; y = ord;
    }
    void display() {
        cout<<"Toa do: "<<x<<" "<<y<<"\n";
    }
}
void main() {
    clrscr();
}
```

```

point<int> ai(3,5); ai.display();
point<char> ac('d','y'); ac.display();
point<double> ad(3.5, 2.3); ad.display();
getch();
}

```

Toa do: 3 5

Toa do: d y

Toa do: 3.5 2.3

2.5 Các tham số trong khuôn hình lớp

Hoàn toàn giống như khuôn hình hàm, các khuôn hình lớp có thể có các tham số kiểu và tham số biểu thức. Trong phần này ta bàn về các tham số kiểu; còn các tham số biểu thức sẽ được nói trong phần sau. Tuy có nhiều điểm giống nhau giữa khuôn hình hàm và khuôn hình lớp, nhưng các ràng buộc đối với các kiểu tham số lại không như nhau.

2.5.1 Số lượng các tham số kiểu trong một khuôn hình lớp

Xét ví dụ khai báo sau:

```
template <class T, class U, class V> //danh sách ba tham số kiểu
```

```
class try {
```

```

    T x;
    U t[5];
    ...
    V fml (int, U);
    ...
}
```

2.5.2 Sản sinh một lớp thể hiện

Một lớp thể hiện được khai báo bằng cách liệt kê đằng sau tên khuôn hình lớp các tham số thực (là tên các kiểu dữ liệu) với số lượng bằng với số các tham số trong danh sách (`template<...>`) của khuôn hình lớp. Sau đây đưa ra một số ví dụ về lớp thể hiện của khuôn hình lớp `try`:

```

try <int, float, int> // lớp thể hiện với ba tham số int, float, int
try <int,int *,double> // lớp thể hiện với ba tham số int, int *, double

```

```
try <char *, int, obj> // lớp thể hiện với ba tham số char *, int, obj
```

Trong dòng cuối ta cuối giả định obj là một kiểu dữ liệu đã được định nghĩa trước đó. Thậm chí có thể sử dụng các lớp thể hiện để làm tham số thực cho các lớp thể hiện khác, chẳng hạn:

```
try <float, point<int>, double>
try <point<int>, point<float>, char *>
```

Cần chú ý rằng, vấn đề tương ứng chính xác được nói tới trong các khuôn hình hàm không còn hiệu lực với các khuôn hình lớp. Với các khuôn hình hàm, việc sản sinh một thể hiện không chỉ dựa vào danh sách các tham số có trong template<...> mà còn dựa vào danh sách các tham số hình thức trong tiêu đề của hàm.

Một tham số hình thức của một khuôn hình hàm có thể có kiểu, là một lớp thể hiện nào đó, chẳng hạn:

```
template <class T> void fct(point<T>) { ... }
```

Việc khởi tạo mới các kiểu dữ liệu mới vẫn áp dụng được trong các khuôn hình lớp. Một khuôn hình lớp có thể có các thành phần (dữ liệu hoặc hàm) **static**. Trong trường hợp này, cần phải biết rằng, mỗi thể hiện của lớp có một tập hợp các thành phần **static** của riêng mình:

2.6 Các tham số biểu thức trong khuôn hình lớp

Một khuôn hình lớp có thể chứa các tham số biểu thức. So với khuôn hình hàm, khái niệm tham số biểu thức trong khuôn hình lớp có một số điểm khác biệt: tham số thực tế tương ứng với tham số biểu thức phải là một hằng số.

Giả sử rằng ta muốn định nghĩa một lớp table để thao tác trên các bảng chứa các đối tượng có kiểu bất kỳ. Một cách tự nhiên ta nghĩ ngay đến việc tạo một khuôn hình lớp với một tham số kiểu. Đồng thời còn có thể dùng một tham số thứ hai để xác định số thành phần của mảng. Trong trường hợp này, định nghĩa của khuôn hình lớp có dạng như sau:

```
template <class T, int n> class table {
    T tab[n];
public:
    ...
};
```

Danh sách các tham số (template<...>) chứa hai tham số với đặc điểm khác nhau hoàn toàn: một tham số kiểu được xác định bởi từ khoá **class**, một tham số

biểu thức kiểu `int >`. Chúng ta sẽ phải chỉ rõ giá trị của chúng trong khai báo các lớp thể hiện. Chẳng hạn, lớp thể hiện:

`table <int , 4>`

tương ứng với khai báo như sau:

```
class table<int,4> {
    int tab[4];
public:
    ...
};
```

Sau đây là một ví dụ hoàn chỉnh:

Ví dụ 6.9

```
/*templat10.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T, int n> class table {
    T tab[n];
public:
    table() { cout<<"Tao bang\n"; }
    T & operator[](int i)
    { return tab[i]; }
};

class point {
    int x, y;
public:
    point (int abs = 1, int ord = 1) {
        x = abs; y = ord;
        cout<<"Tao diem "<<x<<" "<<y<<"\n";
    }
    void display() {
        cout<<"Toa do: "<<x<<" "<<y<<"\n";
    }
}
```

```

};

void main() {
    clrscr();
    table<int, 4> ti;
    for(int i = 0; i < 4; i++)
        ti[i] = i;
    cout<<"ti: ";
    for(i = 0; i < 4; i++)
        cout <<ti[i]<< " ";
    cout<<"\n";
    table <point, 3> tp;
    for(i = 0; i < 3; i++)
        tp[i].display();
    getch();
}

```

```

Tao bang
ti: 0 1 2 3
Tao diem 1 1
Tao diem 1 1
Tao diem 1 1
Tao bang
Toa do: 1 1
Toa do: 1 1
Toa do: 1 1

```

2.7 Tổng quát về khuôn hình lớp

Ta có thể khai báo một số tuỳ ý các tham số biểu thức trong danh sách các tham số của khuôn hình hàm. Các tham số này có thể xuất hiện ở bất kỳ nơi nào trong định nghĩa của khuôn hình lớp. Khi sản sinh một lớp có các tham số biểu thức, các tham số thực tế tương ứng phải là các biểu thức hằng phù hợp với kiểu dữ liệu đã khai báo trong danh sách các tham số hình thức của khuôn hình lớp.

2.8 Cụ thể hóa khuôn hình lớp

Khi nồng cụ thể hóa khuôn hình lớp có đôi chút khác biệt so với khuôn hình hàm.

Khuôn hình lớp định nghĩa họ các lớp trong đó mỗi lớp chứa đồng thời định nghĩa của chính nó và các hàm thành phần. Như vậy, tất cả các hàm thành phần cùng tên sẽ được thực hiện theo cùng một giải thuật. Nếu ta muốn cho một hàm thành phần thích ứng với một tinh huống cụ thể cụ thể nào đó, có thể viết một định nghĩa khác cho nó. Sau đây là một ví dụ cải tiến khuôn hình lớp point. Ở đây chúng ta đã cụ thể hóa hàm hiển thị `display()` cho trường hợp kiểu dữ liệu `char`.

Ví dụ 6.10

```
/*templat11.cpp*/
#include <iostream.h>
#include <conio.h>
//tạo một khuôn hình lớp
template <class T> class point {
    T x, y;
public:
    point(T abs = 0, T ord = 0) {
        x = abs; y = ord;
    }
    void display();
};

template <class T> void point<T>::display() {
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
}

//Thêm một hàm display cụ thể hóa trong trường hợp các ký tự
void point<char>::display() {
    cout<<"Toa do: "<<(int)x<<" "<<(int)y<<"\n";
}

void main() {
    clrscr();
    point <int> ai(3,5);
    ai.display();
}
```

```

point <char> ac('d','y');
ac.display();
point <double> ad(3.5, 2.3);
ad.display();
getch();
}

```

```

Toa do: 3 5
Toa do: 100 121
Toa do: 3.5 2.3

```

Ta chú ý dòng tiêu đề trong khai báo một thành phần được cụ thể hoá:

```
void point<char>::display()
```

Khai báo này nhắc chương trình dịch sử dụng hàm này thay thế hàm `display()` của khuôn hình lớp `point` (trong trường hợp giá trị thực tế cho tham số kiểu là `char`).

Nhận xét

- (i) Có thể cụ thể hoá giá trị của tất cả các tham số. Xét khuôn hình lớp sau đây:

```

template <class T, int n> class table {
    T tab[n];
public:
    table() {cout<<" Tao bang\n"; }
    ...
};

```

Khi đó, chúng ta có thể viết một định nghĩa cụ thể hoá cho hàm thiết lập cho các bảng 10 phần tử kiểu `point` như sau:

```
table<point,10>::table(...) {...}
```

- (ii) Có thể cụ thể hoá một hàm thành phần hay một lớp. Trong ví dụ 6.10 chương trình `template11.cpp` đã cụ thể hoá một hàm thành phần của khuôn hình lớp. Nói chung có thể: cụ thể hoá một hay nhiều hàm thành phần, hoặc không cần thay đổi định nghĩa của bản thân lớp (thực tế cần phải làm như vậy) mà cụ thể hoá bản thân lớp bằng cách đưa thêm định nghĩa. Khả năng thứ hai này có dẫn tới việc phải cụ thể hoá một số hàm

thành phần. Chẳng hạn, sau khi định nghĩa khuôn hình template<class T> class point, ta có thể định nghĩa một phiên bản cụ thể cho kiểu dữ liệu point thích hợp với thể hiện point<char>. Ta làm như sau:

```
class point<char> {  
//định nghĩa mới };
```

2.9 Sự giống nhau của các lớp thể hiện

Xét câu lệnh gán giữa hai đối tượng. Như chúng ta đã biết, chỉ có thể thực hiện được phép gán khi hai đối tượng có cùng kiểu (với trường hợp thừa kế vẫn đề đặc biệt hơn một chút nhưng thực chất chỉ là chuyển kiểu ngầm định đổi với biểu thức về phái của lệnh gán). Trước đây, ta biết rằng hai đối tượng có cùng kiểu nếu chúng được khai báo với cùng một tên lớp. Trong trường hợp khuôn hình lớp, nên hiểu sự cùng kiểu ở đây như thế nào? Thực tế, hai lớp thể hiện tương ứng với cùng một kiểu nếu các tham số kiểu tương ứng nhau một cách chính xác và các tham số biểu thức có cùng giá trị. Như vậy (giả thiết rằng chúng ta đã định nghĩa khuôn hình lớp table trong mục 2.6) với các khai báo:

```
table <int, 12> t1;  
table <float, 12> t2;
```

ta không có quyền viết:

```
t2 = t1; //không tương thích giữa hai tham số đầu
```

Cũng vậy, với các khai báo:

```
table <int, 12> ta;  
table <int, 20> tb;
```

cũng không được quyền viết

```
ta = tb; //giá trị thực của hai tham số sau khác nhau.
```

Những qui tắc chặt chẽ trên nhằm làm cho phép gán ngầm định được thực hiện chính xác. Trường hợp có các định nghĩa chồng toán tử gán, có thể không nhất thiết phải tuân theo qui tắc đã nêu trên. Chẳng hạn hoàn toàn có thể thực hiện được phép gán

```
t2 = t1;
```

nếu ta có định nghĩa hai lớp thể hiện table<int, m> và table<float, n> và trong lớp thứ hai có định nghĩa chồng phép gán bằng.

2.10 Các lớp thể hiện và các khai báo bạn bè

Các khuôn hình lớp cũng cho phép khai báo bạn bè. Bên trong một khuôn hình lớp, ta có thể thực hiện ba kiểu khai báo bạn bè như sau.

2.10.1 Khai báo các lớp bạn hoặc các hàm bạn thông thường

Giả sử A là một lớp thông thường và fct() là một hàm thông thường. Xét khai báo sau đây trong đó khai báo A là lớp bạn và fct() là hàm bạn của tất cả các lớp thể hiện của khuôn hình lớp:

```
template <class T> class try {
    int x; public:
    friend class A;
    friend int fct(float);
    ...
};
```

2.10.2 Khai báo bạn bè của một thể hiện của khuôn hình hàm, khuôn hình lớp

Xét hai ví dụ khai báo sau đây. Giả sử chúng ta có khuôn hình lớp và khuôn hình hàm sau:

```
template <class T> class point {...};
template <class T> int fct (T) {...};
```

Ta định nghĩa hai khuôn hình lớp như sau:

```
template <class T, class U> class try1 {
    int x; public:
    friend class point<int>;
    friend int fct(double);
    ...
};
```

Khai báo này xác định hai thể hiện rất cụ thể của khuôn hình hàm fct và khuôn hình lớp point là bạn của khuôn hình lớp try1.

```
template <class T, class U> class try2 {
    int x; public:
    friend class point<T>;
```

```
friend int fct(U);
...
};
```

So với `try1`, trong `try2` người ta không xác định rõ các thể hiện của `fct()` và `point` là bạn của một thể hiện của `try2`. Các thể hiện này sẽ được cụ thể tại thời điểm chúng ta tạo ra một lớp thể hiện của `try2`. Ví dụ, với lớp thể hiện `try2<double, long>` ta có lớp thể hiện bạn là `point<double>` và hàm thể hiện bạn là `fct<long>`.

2.10.3 Khai báo bạn bè của khuôn hình hàm, khuôn hình lớp

Xét ví dụ sau đây:

```
template <class T, class U> class try3 {
int x; public:
template <class X> friend class point<X>;
template <class X> friend int fct(X);
...
};
```

Lần này, tất cả các thể hiện của khuôn hình lớp `point` đều là bạn của các thể hiện nào của khuôn hình lớp `try3`. Tương tự như vậy tất cả các thể hiện của khuôn hình hàm `fct()` đều là bạn của các thể hiện của khuôn hình lớp `try3`.

2.11 Ví dụ về lớp bảng có hai chỉ số

Trước đây, để định nghĩa một lớp `table` có hai chỉ số ta phải sử dụng một lớp `vector` trung gian. Ở đây, ta xét một cách làm khác nhờ sử dụng khuôn hình lớp.

Với định nghĩa sau:

```
template <class T, int n> class table {
T tab[n];
public:
T &operator [](int i) {
    return tab[i];
}
};
```

ta có thể khai báo:

```
table <table<int,2>,3> t2d;
```

Trong trường hợp này, thực chất ta có một bảng ba phân tử, mỗi phân tử có kiểu `table<int,2>`. Nói cách khác, mỗi phân tử lại là một bảng chứa hai số nguyên. Ký hiệu `t2d[1][2]` biểu thị một tham chiếu đến thành phần thứ ba của `t2d[1]`, bảng `t2d[1]` là phân tử thứ hai của bảng hai chiều các số nguyên `t2d`. Sau đây là một chương trình hoàn chỉnh:

Ví dụ 6.10

```
#include <iostream.h>
#include <conio.h>
template <class T, int n> class table {
    T tab[n];
public:
    table() //hàm thiết lập
    {
        cout<<"Tao bang co "<<n<<"phan tu\n";
    }
    T & operator[] (int i) //hàm toán tử []
    {
        return tab[i];
    }
};

void main() {
    clrscr();
    table <table<int,2>,3> t2d;
    t2d[1][2] = 15;
    cout <<"t2d [1] [2] ="<<t2d[1][2]<<"\n";
    int i, j;
    for (i = 0; i < 2; i++)
        for(j = 0; j < 3; j++)
            t2d[i][j] = i*3+j;
    for(i = 0; i < 2; i++)
        for(j = 0; j < 3; j++)
```

```

    cout <<t2d[i][j]<< " ";
    cout<<"\n";
}
}

```

```

Tao bang co 2 phan tu
Tao bang co 2 phan tu
Tao bang co 2 phan tu
Tao bang co 3 phan tu
t2d [1] [2] = 15
0 1 2 3 4 5 6 7 8

```

Chú ý

Chương trình ví dụ trên đây còn có một số điểm yếu: chưa quản lý vấn đề tràn chỉ số bán, không khởi tạo giá trị cho các thành phần của bảng. Để giải quyết cần có một hàm thiết lập với một tham số có kiểu **int**, và một hàm thiết lập có nhiệm vụ chuyển kiểu **int** thành kiểu **T** bất kỳ. Các yêu cầu này coi như là bài tập.

Ví dụ 6.11

```

/*templa13.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T, int n> class table {
    T tab[n];
    int limit;
public:
    table (int init = 0);
    T & operator[] (int i) {
        if (i < 0 || i >limit)
            cout<<"Tran chi so "<<i<<"\n";
        else return tab[i];
    }
};
template <class T, int n> table<T,n>::table(int init = 0) {
    int i;

```

```

for (i = 0; i < n; i++) tab[i] = init;
    limit = n - 1;
cout<<"Tao bang kich thuoc "<<n<<" init = "<<init<<"\n";
}
void main() {
    clrscr();
    table <table<int,3>,2>ti; //khởi tạo ngầm định
    table <table<float,4>,2> td(10); //khởi tạo bảng 10
    ti[1][6] = 15;
    ti[8][-1] = 20;
    cout<<ti[1][2]<<"\n"; cout<<td[1][0]<<"\n";
    getch();
}

```

```

Tao bang kich thuoc 3 init = 0
Tao bang kich thuoc 2 init = 0
Tao bang kich thuoc 4 init = 0
Tao bang kich thuoc 4 init = 0
Tao bang kich thuoc 4 init = 10
Tao bang kich thuoc 4 init = 10
Tao bang kich thuoc 2 init = 10
Tran chi so 6
Tran chi so 8
Tran chi so -1 0 10

```

Chú ý

Nếu để ý các thông báo tạo các bảng ta thấy rằng ta thu được nhiều hơn hai lần so với dự kiến với các bảng có hai chỉ số. Lời giải thích nằm ở trong câu lệnh gán `tab[i] = init` của hàm thiết lập `table`. Khi khai báo mảng các đối tượng T chương trình dịch gọi tới các hàm thiết lập ngầm định với các tham số bằng 0. Để thực hiện lệnh gán `tab[i] = init` chương trình dịch sẽ thực hiện chuyển đổi từ kiểu số nguyên

sang một đối tượng tạm thời kiểu T. Điều này cũng sẽ gọi tới hàm thiết lập table(int). Chẳng hạn trong khai báo của bảng td, ta thấy, chương trình dịch tạo ra hai bảng một chiều các số thực với kích thước là 4 và init =0, đồng thời cũng có hai lần tạo các bảng trung gian với kích thước bằng 4 và init=10.

3. TÓM TẮT

3.1 Ghi nhớ

Khuôn hình lớp/hàm là phương tiện mô tả ý nghĩa của một lớp/hàm tổng quát còn lớp/hàm thể hiện là một bản sao của khuôn hình tổng quát với các kiểu dữ liệu cụ thể.

Các khuôn hình lớp/hàm thường được tham số hoá, tuy nhiên vẫn có thể sử dụng các kiểu cụ thể trong các khuôn hình lớp/hàm nếu cần.

4. BÀI TẬP

Bài 6.1. Viết chương trình khai báo khuôn hình lớp để mô phỏng hoạt động của hàng đợi hoặc ngăn xếp trên các kiểu đối tượng khác nhau.

CÁC KÊNH XUẤT NHẬP

1. GIỚI THIỆU CHUNG

1.1 Khái niệm về kinh

Trong các chương trước, chúng ta thường sử dụng các chỉ thị viết ra thiết bị ra chuẩn như :

```
cout << n;
```

Chỉ thị này gọi đến toán tử “<<” và cung cấp cho nó hai toán hạng, một tương ứng với “kênh xuất - output stream”(ở đây là **cout**), toán hạng thứ hai là biểu thức mà chúng ta muốn viết giá trị của nó (ở đây là n).

Tương tự, các chỉ thị đọc từ thiết bị vào chuẩn kiểu như:

```
cin >> x;
```

gọi tới toán tử “>>” và cung cấp cho nó hai toán hạng, một là “kênh nhập - input stream”(ở đây là **cin**), còn toán hạng thứ hai là một biến mà ta muốn nhập giá trị cho nó.

Một cách tổng quát, một kinh(stream) được hiểu như một kinh truyền:

- (i) nhận thông tin, trong trường hợp ta nói đến dòng xuất
- (ii) cung cấp thông tin, trong trường hợp ta nói đến dòng nhập.

Các toán tử “<<” và “>>” ở đây đóng vai trò chuyen giao thông tin, cùng với khuôn dạng của chúng.

Một kinh có thể được nối với một thiết bị ngoại vi hoặc một tập tin. Kênh **cout** được định nghĩa nối đến thiết bị ra chuẩn (tương đương stdout). Cũng vậy, kênh **cin** được định nghĩa trước để nối đến thiết bị vào chuẩn (stdin). Thông thường **cout** tương ứng với màn hình, còn **cin** thì đại diện cho bàn phím. Tuy nhiên trong trường hợp cần thiết thì có thể đổi hướng các vào ra chuẩn này đến một tập tin.

Ngoài các kinh chuẩn **cin** và **cout**, người sử dụng có thể định nghĩa cho mình các kinh xuất nhập khác để kết nối với các tập tin.

1.2 Thư viện các lớp vào ra

C++ cung cấp một thư viện các lớp phục vụ cho công việc vào ra. Lớp

`streambuf` là cơ sở cho tất cả các thao tác vào ra bằng toán tử; nó định nghĩa các đặc trưng cơ bản của các vùng đệm lưu trữ các ký tự để xuất hay nhập. Lớp `ios` là lớp dẫn xuất từ `streambuf`, `ios` định nghĩa các dạng cơ bản và khả năng kiểm tra lỗi dùng cho `streambuf`. `ios` là một lớp cơ sở ảo cho các lớp `istream` và `ostream`. Mỗi lớp này có định nghĩa chung toán tử “`<<`” và “`>>`” cho các kiểu dữ liệu cơ sở khác nhau. Để sử dụng các khả năng này phải dùng chỉ thị `#include` đối với tập tin tiêu đề `iostream.h`. Cơ chế lớp của C++ cho phép tạo ra hệ thống giao tiếp có khả năng mở rộng và nhất quán. Trong chương 4 đã đưa ra hai định nghĩa chung cho các toán tử vào/ra trong C++.

Phụ lục này tập trung trình bày các khả năng vào ra do C++ cung cấp, bao gồm các nội dung sau:

- (i) khả năng của `ostream`, `istream`,
- (ii) kiểm soát lỗi vào ra.

2. LỚP OSTREAM

2.1 Định nghĩa chung toán tử `<<` trong lớp `ostream`

Trong lớp `ostream`, toán tử “`<<`” được định nghĩa cho tất cả các kiểu dữ liệu cơ sở dưới dạng một hàm toán tử thành phần:

```
ostream &operator<< (expression)
```

trong đó `expression` có kiểu cơ sở bất kỳ. Vai trò của hàm toán tử này là chuyển giá trị của biểu thức tới kênh liên quan, đồng thời định dạng giá trị đó một cách thích hợp. Xét chỉ thị:

```
cout<<n;
```

Nếu `n` có giá trị 1234, toán tử “`<<`” sẽ chuyển đổi giá trị nhị phân của `n` sang hệ thập phân và chuyển đến `cout` các ký tự tương ứng với các chữ số của số thập phân nhận được (ở đây là 1, 2, 3, 4).

Ngoài ra, toán tử này trả về giá trị là tham chiếu đến kênh xuất đã gọi nó, sau khi thông tin được viết ra. Do vậy, cho phép viết liên tiếp nhiều giá trị lên cùng một kênh:

```
cout<<"Gia tri : <<n<<"\n";
```

2.2 Hàm `put`

Hàm thành phần `put` trong lớp `ostream` dùng để đưa ra kênh xuất tham số ký tự. Chỉ thị

```
cout.put(c);
```

sẽ đưa ra kenh xuất **cout** ký tự c.

Giá trị trả về của put là tham chiếu đến kenh xuất đang sử dụng. Có thể ghi liên tiếp các ký tự trên cùng kenh xuất như sau:

```
cout.put(c1).put(c2).put(c3);
```

Chỉ trỏ trên tương đương với ba chỉ thị riêng biệt:

```
cout.put(c1);  
cout.put(c2);  
cout.put(c3);
```

2.3 Hàm write

Hàm thành phần write cho phép ghi ra kenh xuất một chuỗi các ký tự có chiều dài đã cho. Ví dụ, với:

```
char t[] = "hello";
```

chỉ thị

```
cout.write(t, 4);
```

sẽ gửi đến cout bốn ký tự đầu tiên của xâu t là h e l l.

Giống như put, hàm write trả về giá trị là tham chiếu đến chính kenh xuất vừa nhận thông tin. Tương tự, có thể gọi liên tiếp các hàm write như đối với hàm put:

```
//in ra ba ký tự đầu tiên của xâu t.
```

```
cout.write(t,1).write(t+1,1).write(t+2,1);
```

2.4 Khả năng định dạng

2.4.1 Chọn cơ số thể hiện

Khi viết một giá trị số nguyên, cơ số ngầm định để biểu diễn giá trị là hệ đếm thập phân. Tuy nhiên ta có thể lựa chọn các cơ số khác nhau để hiển thị giá trị: 10 (decimal), 16 (hexa decimal), 8 (octal). Chương trình *io1.cpp* sau đây đưa ra một ví dụ minh họa:

Ví dụ

```
/*io1.cpp*/
```

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

```
#include <conio.h>
void main() {
    clrscr();
    int n = 12000;
    cout<<"Ngam dinh " <<n<<endl;
    cout<<"Duoit he 16 "<<hex<<n<<endl;
    cout<<"Duoit he 10 "<<dec<<n<<endl;
    cout<<"Duoit he 8 " <<oct<<n<<endl;
    cout<<"va ..." " <<n<<endl;
    getch();
}
```

```
Ngam dinh 12000
Duoit he 16 2ee0
Duoit he 10 12000
Duoit he 8 27340
va ... 27340
```

Các ký hiệu hex, dec, oct được gọi là toán tử định dạng. Đố chính là các toán tử đã được định nghĩa trước trong ostream. Các toán tử này chỉ có một toán hạng là đối tượng ostream và trả về chính đối tượng đó sau khi nó thực hiện một số thao tác nhất định. Trong ví dụ này, thao tác được thực hiện là thay đổi cơ số hiển thị giá trị và thông tin về cơ số sẽ được ghi lại trong ostream cho các lần thực hiện tiếp sau.

2.4.2 Đặt độ rộng

Lớp ostream cung cấp cho người sử dụng các phương thức hoặc các toán tử để kiểm soát cách thức máy tính định dạng xuất và nhập các giá trị. Để xác định độ rộng của trường để hiện thị thông tin ta sử dụng phương thức width. Xét các chỉ thị sau:

```
int x = 10;
cout.width(5);
cout<<x;
```

Giá trị của x sẽ được hiển thị sát lề phải trong trường với độ rộng 5 ký tự. Nếu kích thước của x lớn hơn độ rộng đã đặt thì giá trị đã định của độ rộng sẽ bị bỏ qua và toàn bộ giá trị của x sẽ được hiển thị. Giá trị ngầm định của độ rộng cho một kênh xuất nhập là 0, nghĩa là dữ liệu được xuất ra theo kích thước thực tế mà

không độn thêm ký tự gì. Sau mỗi lần xuất, độ rộng sẽ được đặt lại giá trị là 0. Đoạn chương trình sau:

```
int x = 1, y =2;
cout.width(5);
cout<<x<<' '<<y;
```

sẽ xuất ra giá trị của x trong một trường có 5 ký tự, sau đó là một dấu trắng và giá trị của y với kích thước thực tế của nó. width cũng không phải là phương thức duy nhất được dùng để thay đổi đặc tính của các kênh xuất/nhập. Xét các chỉ thị sau:

```
float pi=3.1415927;
int orig_prec = cout.precision(2);
cout<<pi;
cout.precision(orig_prec);
```

Trong đoạn chương trình trên, phương thức precision dùng để xác định lại số chữ số sẽ được in ra sau dấu chấm thập phân cho các giá trị thực. Phương thức precision có thể có tham số và sẽ trả về số chữ số thập phân thực có đứng sau dấu chấm. Giá trị ngầm định cho precision là 6.

Trong trường hợp giá trị đặt cho độ rộng lớn hơn chiều dài của giá trị được xuất ra thì dùng các ký tự độn để lấp khoảng trống. Ký tự độn ngầm định là dấu cách. Tuy vậy có thể sử dụng phương thức fill để sử dụng một ký tự khác dấu trắng. Đoạn chương trình sau:

```
int x = 10;
cout.fill('0');
cout.width(5);
cout<<x;
```

cho kết quả:

00010

Có thể thay thế phương thức width trong **cout**, **cin** bằng toán tử setw như sau:

cout.setw(5); //đặt độ rộng là 5

thay vì sử dụng

```
cout.width(5);
cout<<x;
```

ta dùng

```
cout<<setw(5)<<x;
```

3. LỚP ISTREAM

3.1 Định nghĩa chồng toán tử ">>" trong lớp istream

Trong lớp istream, toán tử ">>" được định nghĩa chồng để có thể làm việc với tất cả các kiểu dữ liệu cơ sở (bao gồm cả **char ***) dưới dạng hàm thành phần:

```
istream & operator >>(&base_type)
```

Theo khai báo này toán tử ">>" có hai toán hạng, toán hạng đứng bên trái sẽ là đối tượng kiểu istream, đối tượng này sẽ là tham số ngầm định cho hàm toán tử. Toán hạng đứng bên phải ">>" là tham chiếu đến biến kiểu cơ sở sẽ nhập giá trị. Thực hiện ">>" sẽ cho kết quả là một tham chiếu đến đối tượng có kiểu istream. Thông thường, đó chính là đối tượng kênh nhập dữ liệu. Cũng như đối với **cout** và "<<", toán tử ">>" cũng cho phép nhập liên tiếp các biến khác nhau.

Các dấu phân cách bao gồm: ' ' '\t' '\v' '\n' '\r' '\f' sẽ không được xem xét khi đọc; chẳng hạn, xét vòng lặp thực hiện chỉ thị (trong đó c có kiểu ký tự char):

```
cin >>c;
```

Với đầu vào có dạng

x i

n c

h a o

thì chỉ có các ký tự x, i, n, c, h, a, o được đọc.

Để đọc được các ký tự trắng, phải sử dụng hàm thành phần **get** trong istream.

Mặt khác khi đọc một xâu ký tự không thể đọc các dấu trắng trong xâu. Chẳng hạn, với nội dung của dòng nhập là:

“Xin chào”

thì chỉ lấy được phần đầu “Xin” trong xâu này để làm nội dung.

Để có thể đọc được các xâu có chứa dấu phân cách sử dụng hàm thành phần `getline` định nghĩa trong lớp `istream`.

3.2 Hàm thành phần get

Hàm thành phần

```
istream & get( char & );
```

cho phép đọc một ký tự từ kênh nhập và gán nó cho biến có kiểu ký tự (là tham số của hàm). Hàm này trả về giá trị là một tham chiếu đến kênh nhập, nên có thể gọi `get` liên tiếp để đọc nhiều ký tự.

Khác với toán tử “`>>`”, hàm `get` có thể đọc tất cả các ký tự kể cả là các dấu phân cách. Bạn đọc có thể kiểm tra số ký tự đọc được nhờ sử dụng `get` đối với dòng nhập có nội dung:

x i

n c

hao

Khi gặp EOF (hết dòng nhập) hàm `get` trả về giá trị 0. Xét đoạn chương trình sau:

```
char c;
...
while (cin.get(c)) //chép lại dòng nhập cin
    cout.put(c);      //lên dòng xuất cout
    // công việc sẽ dừng khi eof vì khi đó (cin)=0
```

Còn một hàm thành phần `get` khác của lớp `istream`:

```
int get()
```

Khi gặp dấu kết thúc tập tin, hàm trả về giá trị EOF còn bình thường hàm đưa lại ký tự đọc được.

3.3 Các hàm thành phần `getline` và `gcount`

Hai hàm này sử dụng để đọc các xâu ký tự.

Khai báo của `getline` có dạng:

```
istream & getline(char * ch, int size, char delim='\\n')
```

Hàm `getline` đọc các ký tự trên kênh nhập gọi nó và đặt vào vùng nhớ có địa chỉ xác định bởi `ch`. Hàm bị ngắt khi:

- ký tự phân cách `delim` xuất hiện trong dòng nhập
- hoặc đã đọc đủ `size-1` ký tự.

Trong cả hai trường hợp, hàm này bổ sung thêm một ký tự kết thúc xâu ngay sau các ký tự đọc được (xem lại hàm `gets()` trong `stdio.h`).

Ký tự phân cách `delim` có giá trị ngầm định là '`\n`' khi đọc các dòng văn bản.

Hàm `gcount` cho biết số ký tự được đọc trong chỉ thị gọi hàm `getline` gần nhất, ở đây không tính tới ký tự phân cách cũng như ký tự cuối xâu được thêm vào tự động.

Xem các chỉ thị sau:

```
const LG_LIG = 120; // chiều dài cực đại của một dòng
...
char ch[LG_LIG+1]; // khai báo 1 dòng
int lg;
...
while(cin.getline(ch,LG_LIG)) {
    lg = cin.gcount();
    // xử lý một dòng có lg ký tự
}
```

3.4 Hàm thành phần `read`

Hàm `read` cho phép đọc từ kênh nhập một dãy ký tự có chiều dài xác định. Chẳng hạn, với:

```
char t[10];
```

chỉ thị

```
cin.read(t, 5);
```

sẽ đọc từ thiết bị vào 5 ký tự và đưa vào 5 ký tự đầu tiên của mảng các ký tự t.

Hàm `read` không phân biệt dấu trắng với các ký tự khác trên kenh nhập.

1.0 Một số hàm khác

Hàm `putback(char c)` cho phép trả lại kenh nhập một ký tự c (tham số của hàm).

Hàm `peek()` đưa ra ký tự tiếp theo trong dòng nhập nhưng không lấy ký tự đó ra khỏi dòng nhập.

4. TRẠNG THÁI LỖI CỦA KÊNH NHẬP

Mỗi kenh nhập hay xuất đều có một số cờ xác định trạng thái lỗi của kenh hiện tại. Trong mục này trước hết ta sẽ xem xét ý nghĩa của các cờ này, sau đó sẽ tìm hiểu cách để lấy giá trị của chúng và thay đổi các giá trị của các cờ theo mục đích của chúng ta. Cuối cùng ta xem xét định nghĩa chung các phép toán () và ! nhằm đơn giản hoá cách sử dụng một kenh dữ liệu.

4.1 Các cờ lỗi

Các cờ lỗi được định nghĩa như là các hàng trong lớp `ios` dẫn xuất từ `ostream` và `istream`. Đó là:

eofbit	Kết thúc tập tin; cờ này được kích hoạt nếu gặp dấu kết thúc tập tin. Nói cách khác khi kenh nhập không còn ký tự để đọc tiếp nữa.
failbit	Bit này được bật khi thao tác vào ra tiếp theo không thể tiến hành được.
badbit	Bit này được bật khi kenh ở trạng thái không thể khôi phục được.

`failbit` và `badbit` chỉ khác nhau đối với các kenh nhập. Khi `failbit` được kích hoạt, các thông tin trước đó trong kenh nhập không bị mất; trong khi đó điều này không còn đúng đối với `badbit`.

Ngoài ra, còn có cờ `goodbit` tương ứng với trạng thái không có lỗi.

Có thể nói rằng một thao tác vào ra thành công khi `goodbit` hay `eofbit` được bật. Tương tự, thao tác vào ra tiếp theo chỉ được tiến hành nếu `goodbit` được bật.

Khi một dòng ở trạng thái lỗi, mọi thao tác tiếp theo phải chờ cho đến khi:

- trạng thái lỗi được sửa chữa,

- các cờ lỗi được tắt.

Ta sẽ xem xét các hàm thực hiện các công việc này trong các mục dưới đây.

4.2 Các thao tác trên các bit lỗi

Có hai loại hàm thành phần thực hiện các thao tác này:

- (i) Các hàm thành phần cho phép giá trị các cờ lỗi,
- (ii) Các hàm thành phần cho phép bật/tắt các cờ lỗi đó.

4.2.1 Đọc giá trị

Trong lớp `ios` có định nghĩa năm hàm thành phần sau đây:

<code>eof()</code>	trả về 1 nếu gặp dấu kết thúc file, có nghĩa là <code>eofbit</code> được kích hoạt.
<code>bad()</code>	trả về 1 nếu <code>badbit</code> được bật.
<code>fail()</code>	trả về 1 nếu <code>failbit</code> được bật.
<code>good()</code>	trả về 1 nếu ba hàm trên cho giá trị 0
<code>rdstate()</code>	trả về một số nguyên tương ứng với tất cả các cờ lỗi.

4.2.2 Thay đổi trạng thái lỗi

Trong `istream/ostream` có hàm thành phần

```
void clear(int i = 0)
```

để bật các bit lỗi tương ứng với giá trị được sử dụng làm tham số. Thông thường, ta xác định giá trị đó dựa trên các hàng số của các cờ lỗi. Chẳng hạn, nếu `f1` biểu thị một kenh, chỉ thị:

```
f1.clear(ios::badbit);
```

sẽ bật cờ lỗi `badbit` và tắt tất cả các cờ còn lại.

Nếu ta muốn bật cờ này đồng thời không muốn thay đổi giá trị các cờ khác, sử dụng chỉ thị sau:

```
f1.clear(ios::badbit|f1.rdstate());
```

4.3 Định nghĩa các toán tử () và !

Có thể kiểm tra một kenh bằng cách xem nó như một giá trị logic. Điều này được thực hiện nhờ việc định nghĩa chung trong lớp `ios` các toán tử () và !.

Chi tiết hơn, toán tử () được định nghĩa chung dưới dạng (trong đó `f1` biểu thị

một dòng):

(fl)

- trả về một giá trị khác 0 nếu các cờ lỗi được tắt, có nghĩa là hàm good() có giá trị bằng 1.
- trả về giá trị 0 trong trường hợp ngược lại, có nghĩa là khi good() có giá trị 0.

Như vậy:

if (fl) ...

có thể thay thế cho (hoặc được thay thế bởi)

if (fl.good()) ...

Cũng vậy, toán tử ! được định nghĩa như là

!fl

- trả về giá trị không nếu có ít nhất một cờ lỗi được bật lên
- trả về giá trị khác không trong trường hợp ngược lại.

Như vậy:

if (!fl) ...

có thể thay thế (hoặc được thay thế bởi)

if (!fl.good()) ...

5. QUẢN LÝ ĐỊNH DẠNG

Các kenh xuất/nhập dùng giá trị cờ để điều khiển dạng nhập và xuất. Một ưu điểm của phương pháp quản lý định dạng sử dụng trong C++ là cho phép người lập trình bỏ qua tất cả các khía cạnh định dạng trong các chỉ thị đưa ra, nếu sử dụng các cờ ngầm định. Bên cạnh đó, khi có nhu cầu, người lập trình có thể đưa ra các định dạng thích hợp (một lần cho tất cả các chỉ thị vào/ra) với các loại dữ liệu.

5.1 Trạng thái định dạng của một dòng

Trạng thái định dạng của một dòng chứa:

- một từ trạng thái, trong đó mỗi bit có một ý nghĩa xác định
- các giá trị số mô tả giá trị của các hàng sau:

Độ rộng	Số ký tự để đưa thông tin ra. Giá trị này là tham số của setw, một toán tử định nghĩa trong ostream/istream. Khi giá trị này quá nhỏ, nó sẽ không có tác dụng nữa, các dòng sẽ hiển thị thông tin theo độ rộng bằng kích thước mà dữ liệu có.
Độ chính xác	Số chữ số được hiển thị sau dấu chấm thập phân trong dạng dấu chấm cố định và cũng là số ký tự có ý nghĩa trong ký pháp khoa học.
Ký tự thay thế	Nghĩa là các ký tự được sử dụng để điền thêm vào phần còn trống khi giá trị đưa ra không đủ để điền hết độ rộng. Ký tự thay thế ngầm định là dấu cách.

5.2 Từ trạng thái định dạng

Từ trạng thái định dạng được mô tả như một số nguyên trong đó mỗi bit (cờ) tương ứng với một hàng số định nghĩa trong lớp ios. Mỗi cờ định dạng được bật khi bit tương ứng có giá trị 1, trái lại ta nói cờ bị tắt. Giá trị của các cờ có thể sử dụng để:

- nhận diện bit tương ứng trong từ định dạng
- để tạo nên một từ trạng thái.

Các trường bit (ít nhất 3) được thay đổi giá trị mà không cần cung cấp tham số cho các hàm thành phần, là do chúng được định nghĩa ngay bên trong từ trạng thái, là đối tượng gọi hàm thành phần có tác dụng thay đổi nội dung các cờ.

Sau đây là danh sách các cờ kèm theo tên của trường bit tương ứng.

Tên trường	Tên bit	Ý nghĩa
	ios::skipws	Bỏ qua các dấu phân cách (khi nhập)
ios::adjustfield	ios::left	Căn lề bên trái (xuất)
	ios::right	Căn lề bên phải (xuất)
	ios::internal	Các ký tự độn được điền giữa dấu và giá trị
ios::basefield	ios::dec	Cơ số hiển thị là cơ số 10
	ios::hex	Cơ số hiển thị là cơ số 16
	ios::oct	Cơ số hiển thị là cơ số 8
	ios::showbase	

ios::showpoint	Hiển thị các chữ số 0 sau các số thập phân ngay cả khi chúng không có. Ngầm định cờ này không được bật.
ios::uppercase	Tất cả các chữ số hiển thị sẽ được chuyển đổi sang chữ in.
ios::showpos	Dấu + sẽ được xuất ra trước bất kỳ số nguyên nào. Ngầm định cờ này không được bật.
ios::scientific	Khi được bật, các giá trị dấu phẩy động sẽ được xuất ra theo dạng khoa học. Số chỉ có một con số đứng trước dấu chấm thập phân và các con số thập phân có nghĩa sẽ đi sau nó, sau đó là chữ "e" ở dạng chữ hoa hay thường (tùy thuộc cờ uppercase), theo sau là giá trị số mũ.
ios::fixed	Khi được bật, giá trị được xuất ra theo dạng số thập phân, có các chữ số thập phân theo sau dấu chấm thập phân. Nếu không bật cả hai cờ thì dạng biểu diễn khoa học sẽ dùng khi số mũ nhỏ hơn -4 hoặc lớn hơn giá trị được mô tả bởi precision.
ios::unibuf	Khi được bật, kênh xuất nhập được thiết lập lại sau mỗi lần xuất ra. Cờ này không bật theo ngầm định.
ios::stdio	Cờ này được sử dụng để dọn dẹp các thiết bị xuất stdout và stderr.

5.3 Thao tác trên trạng thái định dạng

Để tác động lên các trạng thái định dạng, có thể sử dụng các toán tử thao tác định dạng hoặc sử dụng các hàm thành phần của các lớp istream và ostream.

Tùy theo từng trường hợp, các thao tác này có thể tác động lên toàn bộ từ trạng thái hay chỉ các giá trị: độ rộng, độ chính xác, ký tự đệm. Bên cạnh đó còn có các hàm thành phần cho phép chúng ta lưu giữ giá trị các trạng thái định dạng để

khôi phục lại về sau.

5.3.1 Các toán tử thao tác định dạng không tham số (TTĐDKTS)

Đây là các toán tử định dạng được sử dụng ở dạng sau (trong đó `f1` đóng vai trò một dòng nhập/xuất, `manipulator` là toán tử định dạng):

`f1<<manipulator`

hay

`f1>>manipulator`

Kết quả thực hiện cho ta tham chiếu đến kênh hiện tại, do vậy đó cho phép xử lý chúng như cách thức chuyên thông tin. Đặc biệt nó còn cho phép áp dụng nhiều lần liên tiếp các toán tử “<<” và “>>”.

Sau đây là danh sách các toán tử định dạng không tham số:

TTĐDKTS	Sử dụng trong các kệnh	Hoạt động
dec	vào/ra	Kích hoạt cờ cơ số biểu diễn hệ 10
hex	vào/ra	Kích hoạt cờ cơ số biểu diễn hệ 16
oct	vào/ra	Kích hoạt cờ cơ số biểu diễn hệ 8
ws	vào	Kích hoạt cờ skipws
endl	ra	Thêm dấu xuống dòng
ends	ra	Thêm ký tự kết thúc xâu
flush	ra	Làm rỗng bộ đệm

5.3.2 Các toán tử định dạng có tham số (TTĐDCCTS)

Các toán tử này được khai báo trong các lớp `ostream`, `istream` dưới dạng hàm thành phần:

`istream &manipulator(argument)`

hoặc

`ostream &manipulator(argument)`

Các toán tử này được sử dụng giống như các toán tử định dạng không có tham số. Tuy nhiên, muốn sử dụng chúng phải tham chiếu tập tin tiêu đề `iomanip.h` bằng chỉ thị:

```
#include <iomanip.h>
```

Sau đây là danh sách các toán tử định dạng có tham số:

TTĐĐCTS	Sử dụng cho các dòng	Vai trò
setbase (int)	vào/ra	Định nghĩa cơ sở hiển thị
resetiosflags (long)	vào/ra	Đặt lại 0 tất cả các bit có mặt trong tham số
setiosflags (long)	vào/ra	Kích hoạt các bit có trong tham số
setfill (int)	vào/ra	định nghĩa lại ký tự đệm
setprecision (int)	vào/ra	Định nghĩa độ chính xác cho các số thực
setw (int)	vào/ra	Định nghĩa độ rộng

5.3.3 Các hàm thành phần

Trong hai lớp istream và ostream có bốn hàm thành phần: setf, fill, precision, và width được mô tả như sau:

Hàm setf

Hàm này cho phép thay đổi từ trạng thái định dạng. Hàm này có hai phiên bản khác nhau:

```
long setf(long)
```

Lời gọi tới phiên bản này kích hoạt các cờ được mô tả trong tham số. Giá trị trả về của hàm là trạng thái cũ của từ trạng thái định dạng. Lưu ý rằng hàm này không tác động đến các cờ không được mô tả. Như vậy, với f1 biểu thị một kênh, chỉ thi:

```
f1.setf(ios::oct);
```

sẽ kích hoạt cờ oct. Tuy nhiên, rất có thể các cờ khác như dec hay hex vẫn còn tác dụng. Dạng thứ hai của hàm setf hay được sử dụng trong thực tế là:

```
long setf(long, long)
```

Lời gọi tới phiên bản này kích hoạt các cờ mô tả trong tham số thứ nhất ở trong tham số thứ hai. Chẳng hạn, nếu f1 là một kênh, chỉ thi sau:

```
f1.setf(ios::oct, ios::basedfield);
```

sẽ kích hoạt cờ ios::oct và tắt các cờ khác trong ios::basefield.

Giá trị trả về của lời gọi này là giá trị cũ của tham số thứ hai.

Hàm fill

Hàm này cho phép xác định và xác lập lại ký tự độn. Cũng có hai phiên bản khác nhau cho hàm này:

```
char fill()
```

Phiên bản này trả về ký tự độn hiện đang được sử dụng, trong khi đó

```
char fill(char)
```

được sử dụng để thay đổi ký tự độn.

Hàm precision

Hàm này cho phép xác định hoặc xác lập lại độ chính xác biểu diễn số thực. Hai phiên bản khác nhau cho hàm là:

```
int precision()
```

sẽ trả về giá trị mô tả độ chính xác hiện thời, còn

```
int precision(int)
```

đặt lại độ chính xác mới, đồng thời trả về giá trị cũ.

Hàm width

Hàm này cho phép xác định hay xác lập lại độ rộng của trường hiển thị thông tin. Cũng có hai phiên bản khác nhau:

```
int width()
```

sẽ trả về độ rộng đang được sử dụng hiện tại, còn

```
int width(int)
```

sẽ trả về độ rộng hiện thời đồng thời xác lập độ rộng mới là tham số được mô tả trong lời gọi hàm.

6. LIÊN KẾT KÊNH XUẤT/NHẬP VỚI MỘT TẬP TIN

Mục này trình bày cách để chuyển hướng vào ra tới một tập tin, đồng thời cũng giới thiệu các khả năng truy nhập trực tiếp vào các tập tin.

6.1 Liên kết xuất với một tập tin

Để liên kết một kênh xuất với một tập tin, ta chỉ cần tạo một đối tượng kiểu lớp `ofstream`, một lớp kế thừa từ `ostream`. Việc sử dụng lớp này cần tới tập tin tiêu đề `fstream.h`.

Hàm thiết lập của lớp `ofstream` có **hai tham số**:

- tên của tập tin liên quan (dưới dạng một xâu ký tự)

- chế độ mở tập tin được xác định bởi một số nguyên.

Lớp ios có định nghĩa một số giá trị mô tả các chế độ mở tập tin khác nhau.

Chỉ thị sau đây là một ví dụ minh họa:

```
ofstream output("abc.txt", ios::out);
```

Khi đó, đối tượng output sẽ được liên kết với tập tin tên là abc.txt, tập tin này được mở ở chế độ ghi. Sau khi đã tạo được một đối tượng ofstream, việc ghi ra tập tin được thực hiện giống như kết xuất ra một kênh xuất, chẳng hạn:

```
output<<12<<"abc"<<endl;
```

Có thể kiểm tra trạng thái lỗi của dòng xuất tương ứng với tập tin giống như cách ta đã dùng đối với các kênh xuất chuẩn:

```
if(output) ...
```

Chương trình ví dụ sau mô tả cách thức ghi một số số nguyên vào một tập tin.

```
/*io2.cpp*/
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <conio.h>
const int LGMAX = 20;
void main() {
    clrscr();
    char filename[LGMAX+1];
    int n;
    cout<<"Ten tap tin : ";
    cin>>setw(LGMAX)>>filename;
    ofstream output(filename,ios::out);
    if (!output) {
        cout<<"Khong the tao duoc tap tin\n";
        exit(1);
    }
    do {
        cin >>n;
        output<<n<<endl;
    } while (n != -1);
}
```

```

if (n>0) output<<n<<' ';
}while(n>0 && (output));
output<<endl;
output.close();
}

```

6.2 Liên kết kênh nhập với một tập tin

Một đối tượng của lớp ifstream sẽ được sử dụng để liên kết với một tập tin chứa thông tin cần nhập. Giống như ofstream, lớp ifstream cũng được định nghĩa trong tập tiêu đề fstream.h. Lớp ifstream có hàm thiết lập với hai tham số giống như ofstream. Chỉ thị sau đây sẽ liên kết một đối tượng ifstream với tập tin abc.txt:

```
ifstream input("abc.txt", ios::in);
```

Việc sử dụng input để đọc nội dung abc.txt giống hệt như việc sử dụng **cin** để đọc dữ liệu từ bàn phím. Ta xét ví dụ sau:

```

/*ia3.cpp*/
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <conio.h>
const int LGMAX = 20;
void main() {
    clrscr();
    char filename[LGMAX+1];
    int n;
    cout<<"Ten tap tin : ";
    cin>>setw(LGMAX)>>filename;
    ifstream input(filename,ios::in);
    if (!input) {
        cout<<"Khong the mo duoc tap tin\n";
        exit(1);
    }
}

```

```

while(input) {
    input>>n;
    cout<<n<<endl;
}
input.close();
}

```

Nhận xét

Lớp `fstream` (thừa kế từ hai lớp `ifstream` và `ofstream`) dùng để định nghĩa các kênh dữ liệu thực hiện đồng thời cả hai chức năng nhập và xuất trên một tập tin. Việc khai báo một đối tượng kiểu `fstream` cũng giống như khai báo đối tượng `ofstream` và `ifstream`. Chỉ thị:

```
fstream file("abc.txt", ios::in|ios::out);
```

sẽ gắn đối tượng `file` với tập tin `abc.txt`, được mở để đọc và ghi đồng thời.

6.3 Các khả năng truy nhập trực tiếp

Việc truy nhập (đọc/ghi) đến tập tin dựa trên một phần tử là con trỏ tập tin. Tại mỗi thời điểm, con trỏ tập tin xác định một vị trí tại đó thực hiện thao tác truy nhập. Có thể xem con trỏ này như cách đếm số phim trong máy ảnh. Sau mỗi một thao tác truy nhập, con trỏ tập tin tự động chuyển sang vị trí tiếp theo giống như việc lèn phim mỗi khi bấm máy ảnh. Ta gọi cách truy nhập tập tin kiểu này là truy nhập tuần tự. Các chương trình `io2.cpp`, `io3.cpp` sử dụng cách truy nhập này để đọc và ghi thông tin trên các tập tin. Nhược điểm của cách truy nhập tuần tự là phải di từ đầu tập tin qua các tất cả các phần tử có trong tập tin để đến được phần tử cần thiết, do vậy tốn không ít thời gian. Cách truy nhập trực tiếp sẽ cho phép đến thẳng tới phần tử chúng ta cần nhờ sử dụng một số hàm thành phần thích hợp trong các lớp `ifstream` và `ofstream`.

Trong lớp `ifstream` có hàm `seekg` và trong lớp `ofstream` có hàm `seekp` được dùng để di chuyển con trỏ tập tin. Mỗi hàm thành phần đó có hai tham số:

- Tham số thứ nhất là số nguyên mô tả dịch chuyển (tính theo byte) con trỏ bao nhiêu vị trí so với vị trí gốc, được mô tả bởi tham số thứ hai (xem hai hàm `fseek` trong `stdio.h`).

- Tham số thứ hai lấy một trong ba giá trị sau:

<code>ios::beg</code>	vị trí gốc là đầu tập tin
<code>ios::cur</code>	vị trí gốc là vị trí hiện thời của con trỏ tập tin
<code>ios::end</code>	vị trí gốc là cuối tập tin.

Hai hàm tellg (đối với ifstream) và tellp (đối với ofstream) dùng để xác định vị trí hiện thời của các con trỏ tập tin.

Chương trình sau đây minh họa khả năng truy nhập tập tin trực tiếp.

```
/*io4.cpp*/
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <conio.h>
const int LGMAX = 20;
void main() {
    clrscr();
    char filename[LGMAX+1];
    int n,num;
    cout<<"Ten tap tin : ";
    cin>>setw(LGMAX)>>filename;
    ifstream input(filename,ios::in);
    if(!input) {
        cout<<"Khong mo duoc tap tin";
        exit(1);
    }
    do {
        cout<<"So thu tu cua so nguyen se tim : ";
        cin>>num;
        if (num >0) {
            input.seekg(sizeof(int)*(num-1),ios::beg);
            input>>n;
            if (input) cout<<"--Gia tri : "<<n<<endl;
            else {
                cout<<"--Loi\n";
                input.clear();
            }
        }
    }
```

```
} while(num);
    input.close();
}
```

Ten tap tin : abc.txt

So thu tu cua so nguyen se tim : 3

--Gia tri : 3

So thu tu cua so nguyen se tim : 2

--Gia tri : 22

So thu tu cua so nguyen se tim : 1

--Gia tri : 1

So thu tu cua so nguyen se tim : 3

--Gia tri : 3

So thu tu cua so nguyen se tim : 4

--Gia tri : 3

So thu tu cua so nguyen se tim : 5

--Gia tri : 4

So thu tu cua so nguyen se tim : 6

--Gia tri : 3

So thu tu cua so nguyen se tim : 7

--Gia tri : 2

So thu tu cua so nguyen se tim : 6

--Gia tri : 3

So thu tu cua so nguyen se tim : 100

--Loi

So thu tu cua so nguyen se tim : 0

6.4 Các chế độ mở tập tin khác nhau

Chế độ	Mô tả hành động tương ứng
ios::in	Mở một tập tin để đọc (bắt buộc đối với ifstream)
ios::out	Mở một tập tin để ghi (bắt buộc đối với ofstream)
ios::app	Mở một tập tin để gắn thêm các thông tin vào cuối.
ios::ate	Đặt con trỏ tập tin vào cuối tập tin
ios::trunc	Nếu tập tin đã có, nội dung của nó sẽ bị mất.
ios::nocreate	Tập tin bắt buộc phải tồn tại.
ios::noreplace	Tập tin chưa tồn tại
ios::binary	Tập tin được mở ở chế độ nhị phân ¹
ios::text	Tập tin được mở ở chế độ văn bản.

Để thực hiện được nhiều hành động trên cùng một tập tin phải tổ hợp các bit mô tả chế độ bằng cách sử dụng toán tử | (kết hợp).

Chẳng hạn:

```
fstream f("abc.txt", ios::in|ios::out);
```

¹ Trong các môi trường dos và windows người ta phân biệt các tập tin văn bản và tập tin nhị phân. Khi mở một tập tin phải xác định ngay là liệu chúng ta sẽ làm việc với các tập tin loại nào. Sự phân biệt này về thực chất liên quan đến việc xử lý ký tự cuối dòng.

XỬ LÝ LỖI

Bắt đầu từ phiên bản 3.0, C++ cung cấp cơ chế xử lý lỗi (exception handling) do người sử dụng điều hành. Trong chương trình có thể đưa ra quyết định rằng một phép xử lý nào đó bị lỗi bằng cách sử dụng từ khoá **throw**. Khi có lỗi, việc xử lý bị ngắt và quyền điều khiển sẽ trao trả cho đoạn xử lý lỗi mà người sử dụng bắt được.

1. BẤY VÀ BẮT LỖI

Ta lấy ví dụ viết một hàm tính giá trị của một phân số với đầu vào là tử số và mẫu số. Rắc rối sẽ nảy sinh khi người sử dụng hàm truyền vào cho mẫu số giá trị bằng 0. Để giải quyết trường hợp này, C++ sẽ tự động tạo sinh bẫy lỗi “gặp trường hợp mẫu số bằng 0”. Sau đây là chương trình ví dụ chỗ hàm tính giá trị phân số có xử lý lỗi.

```
#include <iostream.h>

// lớp kiểu lỗi mà không có thành phần nào
class Loi_Chia_0 {};

float GiaTriPS(int ts, int ms) {
    // phát lỗi nếu mẫu = 0
    if ( ms==0 ) throw( Loi_Chia_0 );
    return float(ts)/ms;
}

void main(){
    int ts, ms;
    cout << "Tinh gia tri phan so\n";
    cout << "TS = "; cin >> ts;
    cout << "MS = "; cin >> ms;
    try {      // thực hiện có bắt lỗi
        float gt = GiaTriPS(ts, ms);
        cout << "Gia tri PS la: " << gt;
    }
}
```

```

    catch ( Loi_Chia_0 ) // bắt lỗi kiểu Loi_Chia_0
    {
        cout << "Loi: Mau so bang 0";
    }
}

```

Lỗi mà ta nhận được là lỗi truy cập bộ nhớ không hợp lệ (Access Violation).

Tính giá trị phân số

TS = 1

MS = 0

Loi: Mau so bang 0

Chương trình này có hai phần bẫy lỗi và bắt lỗi. Hàm tính phân số sẽ phát sinh một bẫy lỗi bằng từ khoá **throw** khi mẫu số bằng 0. Phía sau từ khoá throw là đối tượng thuộc lớp phục vụ bắt lỗi. Ở đây ta sử dụng lớp *Loi_Chia_0* không có thành phần nào bên trong để phục vụ cho việc bắt lỗi này. Khi một lỗi được phát sinh, toàn bộ những lệnh tiếp theo trong hàm xử lý bị huỷ bỏ. Trong thân chương trình chúng ta sử dụng cấu trúc **try... catch...** để bắt lỗi. Hàm *GiaTriPS* được đặt trong **try**, do vậy khi nó phát sinh lỗi (lỗi loại *Loi_Chia_0* được phát sinh khi mẫu số truyền vào là 0) thì chương trình dừng hoạt động và trao quyền điều khiển cho đoạn mã bắt lỗi này. Ta đã dùng **catch (Loi_Chia_0)** để bắt lỗi. Như vậy, khi có một lỗi chia 0 thì chương trình sẽ in ra dòng thông báo “Loi: Mau so bang 0”. Khi trong chương trình có một lỗi phát sinh mà không có đoạn bắt lỗi tương ứng thì chương trình sẽ tự động kết thúc bất thường. Điều này sẽ gây trở ngại đáng kể cho việc gỡ rối chương trình.

Một chương trình có thể phát sinh nhiều loại lỗi khác nhau. Loại lỗi phát sinh thể hiện ở kiểu lớp lỗi được sử dụng trong các lệnh **throw**. Do vậy, ta cũng có thể sử dụng nhiều lần **catch** để bắt các loại lỗi khác nhau trong một chương trình như trong ví dụ sau.

```

#include <iostream.h>

class Loi_A {};
class Loi_B {};

void PhatLoi(int i){
    // neu i khac 0 thi phat Loi_A con khong Loi_B
    if (i) throw ( Loi_A );
    throw ( Loi_B );
}

```

```

void main(){
    int i;
    cout << "i = "; cin >> i;
    try
    {
        PhatLoi( i );
    }
    catch ( Loi_A )
    {
        cout << "Loi loai A";
    }
    catch ( Loi_B )
    {
        cout << "Loi loai B";
    }
}

```

```

i = 1
Loi loai A

i = 0
Loi loai B

```

Ta cũng có thể sử dụng `catch(...)` để bắt tất cả các loại lỗi. Lớp lỗi có thể có các thành phần giống như lớp bình thường dùng để làm các thông số xử lý lỗi khi bắt được. Chẳng hạn, trong chương trình xử lý lỗi chia 0 ở trên ta có thể thêm vào thuộc tính để lưu lại tử số của phép chia lỗi dùng in ra khi bắt lỗi.

```

#include <iostream.h>
class Loi_Chia_0{
public:
    int ts;
    Loi_Chia_0(int t): ts(t) {}
};

float GiaTriPS(int ts, int ms){

```

```

    if ( ms==0 ) throw( Loi_Chia_0(ts) );
    return float(ts)/ms;
}

void main(){
    int ts, ms;
    cout << "Tinh gia tri phan so\n";
    cout << "TS = "; cin >> ts;
    cout << "MS = "; cin >> ms;
    try {
        float gt = GiaTriPS(ts, ms);
        cout << "Gia tri PS la: " << gt;
    }
    catch ( Loi_Chia_0 loi )
    {
        cout << "Loi chia " << loi.ts << " cho 0";
    }
}

```

```

Tinh gia tri phan so
TS = 1
MS = 0
Loi chia 1 cho 0

```

2. HOẠT ĐỘNG CỦA CHƯƠNG TRÌNH KHI MỘT LỖI PHÁT SINH

Khi một lỗi trong chương trình đã bị bắt thì chương trình tiếp tục hoạt động bình thường theo mã lệnh xử lý lỗi bắt được. Trong một hàm xử lý người sử dụng có thể bắt lỗi xảy ra và sau đó tiếp tục ném nó để duy trì lỗi của chương trình bằng từ khoá **throw** mà không có kiểu loại lỗi phía sau.

```

#include <iostream.h>
class Loi {};
void PhatLoi(){
    throw ( Loi );
}

```

```

void BatLoi() {
    try {
        PhatLoi();
    }
    catch ( Loi ) {
        cout << "Loi da bi bat va duoc nem lai\n";
        throw; // ném lại lỗi bị bắt
    }
}

void main() {
    try {
        BatLoi();
    }
    catch ( Loi ) {
        cout << "Loi bi bat lai lan hai";
    }
}

```

Loi da bi bat va duoc nem lai
 Loi bi bat lai lan hai

Khi một lỗi xảy ra mà không có một bắt lỗi nào đáp ứng thì chương trình sẽ kết thúc và trước khi kết thúc nó sẽ thực hiện hàm xử lý được xác lập trong câu lệnh set_terminate.

```

#include <iostream.h>
class Loi {};

void KetThucLoi(){
    cout << "Chuong trinh bi loi va ket thuc bat thuong\n";
}

void PhatLoi(){
throw ( Loi );
}

void main(){

```

```
// xác lập hàm kết thúc bất thường
set_terminate(KetThucLoi);

cout << "Gọi ham phat loi ma khong bat\n";
PhatLoi();

cout << "Kết thúc bình thường\n";
}
```

Gọi ham phat loi ma khong bat

Chương trình bị lỗi và kết thúc bất thường

Trong một hàm xử lý người xử dụng có thể xác định tất cả những lỗi có thể xảy ra bằng cách dùng từ khoá **throw** đúng ngay sau khai báo tham số hàm và liệt kê những lớp lỗi có thể xảy ra để trong cặp dấu (). Nếu không có lớp lỗi nào được liệt kê thì hiểu rằng hàm đó sẽ không có lỗi. Còn nếu không có từ khoá **throw** thì hàm đó có thể có bất kì lỗi nào.

```
// hàm có thể có Lỗi_A hoặc Lỗi_B
void fct1() throw(Loi_A, Loi_B);

// hàm sẽ không có lỗi nào
void fct2() throw();

// hàm có thể có bất kì loại lỗi nào
void fct3();
```

Trường hợp với một hàm nào đó tuy đã được xác định một số lỗi có thể xảy ra, nhưng khi chạy lại xuất hiện một lỗi không phải là lỗi đã xác định, thì chương trình sẽ kết thúc. Trước khi kết thúc nó sẽ thực hiện phép xử lý của hàm được xác lập bằng hàm **set_unexpected**.

```
#include <iostream.h>

class Loi_A {};
class Loi_B {};

void LoiKhongCho()
{
    cout << "Chương trình kết thúc vì gặp lỗi không cho\n";
}

void PhatLoi() throw(Loi_B){
    throw (Loi_A);
```

```

}

void main() {
    // xác lập hàm kết thúc khi gặp lỗi không chờ đợi
    set_unexpected(LoiKhongCho);

    try {
        cout << "Gọi ham phat loi\n";
        PhatLoi();
    }
    catch ( ... ) {
        cout << "Loi da bi bat";
    }
}

```

Gọi ham phat loi

Chuong trinh ket thuc vi gap loi khong cho

3. XỬ LÝ LỖI TRONG LỚP ỨNG DỤNG

Trong mục này chúng ta sẽ xây dựng một lớp ứng dụng mảng động có kiểm soát lỗi khởi tạo với số phần tử nhỏ hơn hoặc bằng không và lỗi truy nhập phần tử ngoài chỉ số.

```

#include <iostream.h>
#include <stdlib.h>

class Loi{
public:
    virtual void InLoi()=0;
};

// lỗi khởi tạo
class Loi_KT : public Loi {
public:
    int spt;
    Loi_KT(int n): spt(n) {}

    void InLoi() {
        cout << "Loi khai tao voi " << spt << " phan tu\n";
    }
};

```

```

};

// loi truy cap
class Loi_TC : public Loi {
public:
    int cs;
    Loi_TC(int i): cs(i) {}
    void InLoi() {
        cout << "Loi truy cap chi so " << cs << "\n";
    }
};

class Array
{
    int spt; // so phan tu mang
    int *dl; // du lieu cua mang
public:
    Array(int n): spt(n) {
        if ( spt <= 0 ) throw( Loi_KT(spt) );
        dl = new int[spt];
    }
    ~Array() { delete dl; }
    int & operator [] (int i){
        if ( i<0 || i>=spt ) throw( Loi_TC(i) );
        return dl[i];
    }
};

void main(){
    try {
        Array a(-3);
        a[5] = 10;
    }
    // bat toan bo loi ke thua tu Loi
    catch ( Loi& l ) {

```

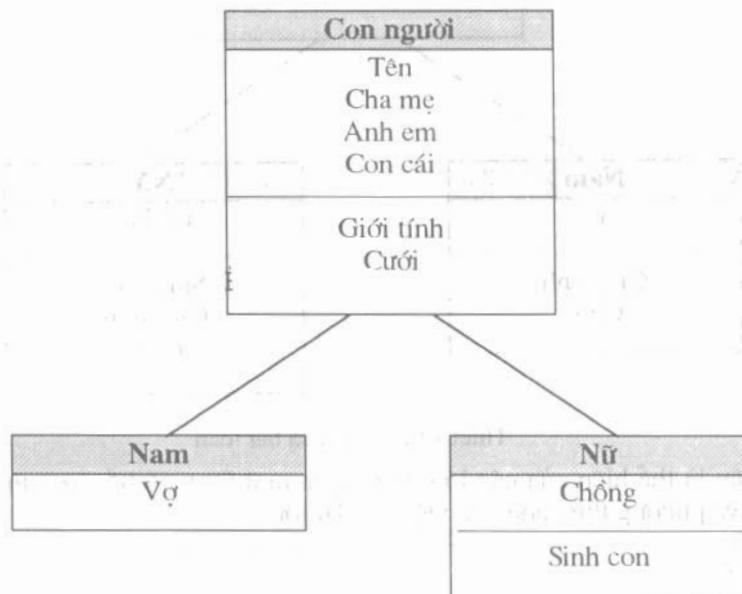
```
    l.InLoi(); // in loi tuong ung boi  
}  
}
```

Lỗi khởi tạo với ~3 phần tử

Trong chương trình trên đã dùng kỹ thuật đa hình để tạo một lớp lỗi trừu tượng cơ sở có phương thức in lỗi ảo. Các lỗi cụ thể kế thừa từ lớp này và thi hành cụ thể việc in lỗi cho nó. Khi bắt lỗi, chỉ cần bắt lỗi lớp cơ sở một cách tổng quát thì tất cả các kiểu lỗi trong lớp kế thừa đều bị bắt. Khi gọi thủ tục in lỗi bị bắt, hệ thống sẽ in đúng lỗi của lớp nó thuộc vào tương tự như tính tương ứng bội.

BÀI TOÁN QUAN HỆ GIA ĐÌNH

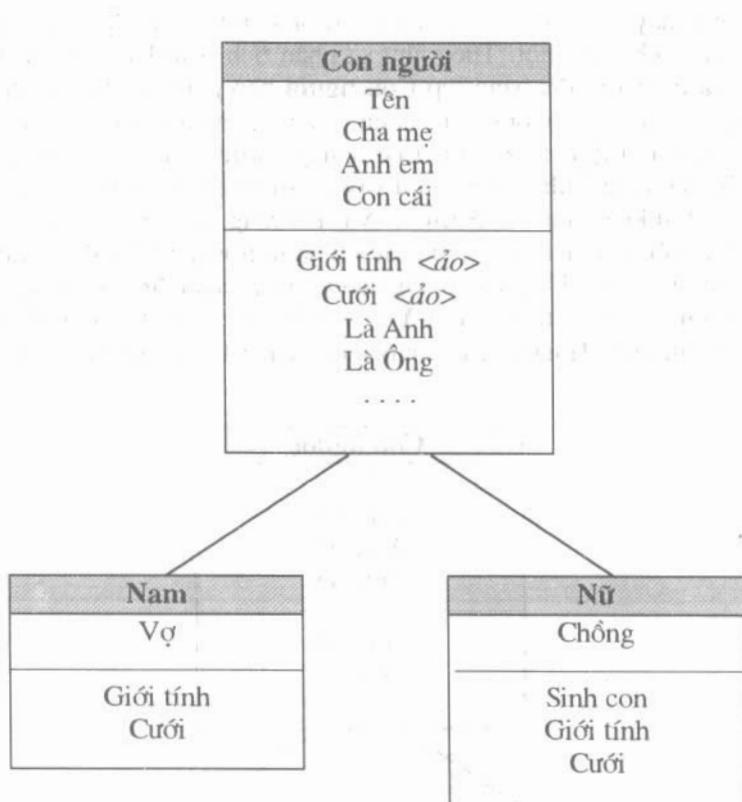
Trong mục này, ta sẽ xây dựng chương trình cho bài toán quan hệ gia đình đã được phân tích ở chương một. Theo như sự phân tích ban đầu của bài toán, ta có một tập các cá thể và mô tả bằng lớp **Con người** bao gồm các thuộc tính tên, anh em, cha mẹ, ... và các phương thức sinh, cưới, ... Nhưng ta có nhận xét rằng phương thức sinh chỉ thực hiện được trên những cá thể là nữ và phương thức cưới chỉ xảy ra cho hai cá thể khác giới. Như vậy có sự phân chia tập đối tượng của bài toán thành hai lớp khác nhau là **Nam** và **Nữ**. Rõ ràng, hai lớp này phải kế thừa từ lớp **Con người**. Lớp **Con người** sẽ chứa các thuộc tính và phương thức chung, dù cá thể là Nam hay Nữ cũng đều phải có. Ngoài cá thành phần được kế thừa từ lớp **Con người**, lớp **Nam** có thêm thuộc tính **Vợ**, lớp **Nữ** có thêm thuộc tính **Chồng** và phương thức **Sinh con**. Thiết kế các lớp ban đầu của bài toán như hình dưới đây.



Thiết kế sơ bộ các lớp của bài toán

Phương thức **Giới tính** dùng để trả lời xem một cá thể đó là Nam hay Nữ. Nếu là Nam kết quả là 1 còn là kết quả là 0. Rõ ràng tại lớp **Con người** phương thức **Giới tính** không thể trả lời được đó là Nam hay Nữ. Câu trả lời chỉ xác định tại các phương thức **Giới tính** ở lớp kế thừa **Nam** và **Nữ**. Phương thức **Giới tính** ở lớp **Nam**

trả kết quả là 1 còn ở lớp Nữ trả kết quả là 0. Để thực hiện được kĩ thuật này, ta dùng kĩ thuật hàm ảo trong LTHĐT. Lập luận tương tự cho phương thức **Cưới**, bởi vì phương thức này cần biết cưới chồng hay cưới vợ. Để trả lời cho các câu hỏi về mối quan hệ gia đình chúng ta cũng cần phải xây những phương thức để trả lời các câu hỏi như **Là Anh**, **Là Ông(X)**, v.v... Hình dưới đây thiết kế các lớp của bài toán.



Thiết kế các lớp của bài toán

Sau đây là thể hiện của các lớp dưới ngôn ngữ C++ có bổ sung thêm một số thuộc tính và phương thức phục vụ việc cài đặt lớp.

```

class Nguoi {
    friend class Nam;
    friend class Nu;
    char Ten[25];
    Nam *Bo;
}
  
```

```
Nu *Me;
Nguoi *AnhChi[10], *CacEm[10], *CacCon[10];
int SoAnhChi, SoEm, SoCon;

Nguoi(char *ten, Nam *bo, Nu *me) :
    Bo(bo), Me(me), SoAnhChi(0), SoEm(0), SoCon(0)
{
    strcpy(Ten, ten);
}

void ThemAnhChi(Nguoi* nguoi)
{
    AnhChi[SoAnhChi++] = nguoi;
}

void ThemEm(Nguoi* nguoi)
{
    CacEm[SoEm++] = nguoi;
}

void ThemCon(Nguoi* nguoi)
{
    CacCon[SoCon++] = nguoi;
}

public:
// 1 là Nam, 0 là Nữ
virtual int GioiTinh()=0;
virtual int Cuoi(Nguoi*)=0;
int LaCha(Nguoi *);
int LaMe(Nguoi *);
int LaCon(Nguoi *);
int LaAnh(Nguoi *);
int LaChi(Nguoi *);
int LaEm(Nguoi *);
```

```
int LaCo(Nguoi *);  
int LaDi(Nguoi *);  
int LaChu(Nguoi *);  
int LaCau(Nguoi *);  
int LaMo(Nguoi *);  
int LaBac(Nguoi *);  
int LaOngNoi(Nguoi *);  
int LaBaNoi(Nguoi *);  
int LaOngNgoai(Nguoi *);  
int LaBaNgoai(Nguoi *);  
int LaAnhHo(Nguoi *);  
int LaChiHo(Nguoi *);  
int LaEmHo(Nguoi *);  
virtual int LaVo(Nguoi*)=0;  
virtual int LaChong(Nguoi*)=0;  
};  
  
class Nam : public Nguoi  
{  
    Nu *Vo;  
    int LaVo(Nguoi *) { return 0; }  
public:  
    Nam(char *ten, Nam *bo=0, Nu *me=0) :  
        Nguoi(ten, bo, me), Vo(0) {}  
    int GioiTinh() { return 1; }  
    int Cuoi(Nguoi *vo);  
    int LaChong(Nguoi * nguai);  
};  
  
class Nu : public Nguoi  
{
```

```

Nam *Chong;
int LaChong(Nguoi *) { return 0; }

public:
    Nu(char *ten, Nam *bo=0, Nu *me=0):
        Nguoi(ten, bo, me), Chong(0) {}
    int GioiTinh() { return 0; }
    int Cuoi(Nguoi *chong);
    void SinhCon(char* ten, int gioitinh);
    int LaVo(Nguoi * nguoi);
};


```

Phương thức cưới chỉ thực hiện đổi với cá thể chưa lập gia đình. Trong trường hợp đã lập gia đình thì nó sẽ trả ra 0.

```

int Nam::Cuoi(Nguoi *vo)
{
    if (Vo || vo->GioiTinh()) return 0;
    Vo = (Nu*)vo;
    Vo->Cuoi(this);
    return 1;
}

int Nu::Cuoi(Nguoi *chong)
{
    if (Chong || chong->GioiTinh()==0) return 0;
    Chong = (Nam*)chong;
    Chong->Cuoi(this);
    return 1;
}

void Nu::SinhCon(char *ten, int gioitinh)
{
    Nguoi* nguoi = TaoNguoi(ten, gioitinh, Chong, this);
    ThemCon(nguoi);
    if (Chong) Chong->ThemCon(nguoi);
}

```

```

for (int i=0; i<SoCon; i++)
{
    CacCon[i]->ThemEm/nguo[i];
    nguo->ThemAnhChi (CacCon[i]);
}
}

```

Trong phương thức sinh con đã dùng hàm TaoNguoi để tạo ra một thể hiện của lớp Nam hay Nữ phụ thuộc vào giới tính. Đối tượng mới tạo ra sẽ được gán nhập vào cộng đồng và được xem xét các mối quan hệ sau này, chi tiết về hàm này chúng ta sẽ bàn luận sau. Sau khi đã có đối tượng thì các mối quan hệ gia đình cha, mẹ, con cái, anh, chị, em phải được xác lập.

Thực hiện cài đặt các phương thức trả lời câu hỏi quan hệ của lớp Con người. Đối với các mối quan hệ gần như LaAnh, LaCha thì việc kiểm tra rất đơn giản thông qua các thuộc tính Bo, Me, AnhChi,... của đối tượng. Nhưng đối với mối quan hệ xa hơn chút ít như LaOngNoi, LaCo, LaChu,... thì trở nên khó khăn hơn nhiều. Chúng ta dùng một phương pháp kiểm tra đơn giản theo mô hình toán học. Ví dụ, A là ông nội của B khi và chỉ khi trong cộng đồng tồn tại một đối tượng X mà A là cha của X và X là cha của B. Như vậy để thực hiện được kiểm tra này thì cần phải lưu được toàn bộ các đối tượng trong cộng đồng để phục vụ tìm kiếm X. Trong chương trình dùng một mảng tĩnh các con trỏ tới đối tượng Con người để quản lý cộng đồng người này. Mảng này được khai báo như một thành phần tĩnh của lớp Con người. Phương thức TaoNguoi sẽ tạo một đối tượng là Nam hoặc Nữ phụ thuộc giới tính truyền vào. Đối tượng mới tạo ra sẽ được thêm vào cộng đồng. Phương thức TimNguoi tìm đối tượng trong cộng đồng có tên nhu tên đưa vào, nếu không tìm thấy trả về NULL. Dưới đây là một số bổ sung cho lớp Con người.

```

class Nguoi
{
    ...
    static Nguoi* NhanDan[100];
    static int SoDan;
public:
    ...
    static int LaySoDan() { return SoDan; }
    static Nguoi* ThemDan(Nguoi* nguo)
    {
        ...
        return NhanDan[SoDan++] = nguo;
    }
}

```

```

    }

static Nguoi* TaoNguoi(char*, int, Nam *bo=0, Nu *me=0);
static Nguoi* TimNguoi(char* ten);
};

Nguoi* Nguoi::NhanDan[];
int     Nguoi::SoDan = 0;

Nguoi*
Nguoi::TaoNguoi(char* ten, int gioitinh, Nam *bo, Nu *me)
{
    return gioitinh ? ThemDan(new Nam(ten, bo, me))
                    : ThemDan(new Nu(ten, bo, me));
}

Nguoi* Nguoi::TimNguoi(char* ten)
{
    for (int i=0; i<SoDan; i++)
        if (strcmp(ten, NhanDan[i]->LayTen())==0)
            return NhanDan[i];
    return 0;
}

```

Sau khi đã tổ chức được dữ liệu lưu trữ con người chúng ta có thể xây dựng các hàm kiểm tra quan hệ như dưới đây. A là đối tượng gọi hàm kiểm tra, B là đối tượng truyền vào, X là đối tượng tìm kiếm NhanDan[i] (dùng vòng **for** để duyệt).

```

int Nguoi::LaOngNoi(Nguoi *nguoi)
{
    if (GioiTinh()==0) return 0;
    for (int i=0; i<SoDan; i++)
        if (LaCha(NhanDan[i]) && NhanDan[i]->LaCha(nguoi))
            return 1;
}

```

```

    return 0;
}

int Nguoi::LaBaNoi(Nguoi *nguoi)
{
    if (GioiTinh()) return 0;
    for (int i=0; i<SoDan; i++)
        if (LaMe(NhanDan[i])&&NhanDan[i]->LaCha(nguoi))
            return 1;
    return 0;
}

```

Mục đích của chương trình là sau khi đã có dữ liệu phải trả lời được câu hỏi quan hệ khi đưa tên hai người vào. Sau khi người sử dụng đưa tên vào chúng ta sẽ tìm đối tượng đó trong cộng đồng. Nếu tìm thấy thì gọi hàm đưa ra thông báo quan hệ của hai đối tượng vừa tìm thấy. Trong hàm dưới đây thực hiện bằng cách kiểm tra các quan hệ bè trên nếu tất cả không thỏa mãn thì lại đổi ngược lại đối tượng gọi phương thức để kiểm tra.

```

char qh[256];
// đưa ra thông báo về quan hệ của 2 đối tượng
char* QuanHe(Nguoi* A, Nguoi* B)
{
    for (int i=1; i<=2; i++)
    {
        strcpy(qh, A->LayTen());
        strcat(qh, " va ");
        strcat(qh, B->LayTen());
        strcat(qh, " co quan he ");
        if (A->LaOngNoi(B))
            return strcat(qh, "ong chau noi");
        if (A->LaBaNoi(B))
            return strcat(qh, "ba chau noi");
        if (A->LaOngNgoai(B))
            return strcat(qh, "ong chau ngoai");
    }
}

```

```

if (A->LaBaNgoai(B))
    return strcat(qh, "ba chau ngoai");
if (A->LaCha(B))
    return strcat(qh, "cha con");
if (A->LaMe(B))
    return strcat(qh, "me con");
if (A->LaCo(B))
    return strcat(qh, "co chau");
if (A->LaDi(B))
    return strcat(qh, "di chau");
if (A->LaChu(B))
    return strcat(qh, "chu chau");
if (A->LaBac(B))
    return strcat(qh, "bac chau");
if (A->LaAnh(B))
    return strcat(qh, "anh em");
if (A->LaChi(B))
    return strcat(qh, "chi em");
if (A->LaAnhHo(B))
    return strcat(qh, "anh em ho");
if (A->LaChiHo(B))
    return strcat(qh, "chi em ho");
if (A->LaVo(B))
    return strcat(qh, "vo chong");

Nguoi* temp = A;
A = B;
B = temp;
}

strcpy(qh, A->LayTen());
strcat(qh, " va ");

```

```

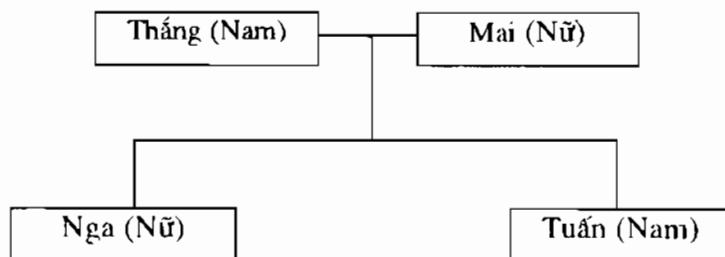
    strcat(qh, B->LayTen());
    return strcat(qh, " khong co quan he gia dinh");
}

// tìm quan hệ của hai đối tượng có tên nhập vào từ bàn phím
void TimQuanHe()
{
    clrscr();
    char ten1[25];
    cout << "Ten nguoi thu nhat: ";
    gets(ten1);
    Nguoi *A = Nguoi::TimNguoi(ten1);
    if (A==0)
    {
        cout << "Khong co nguoi ten " << ten1 << endl;
        getch();
        return;
    }
    char ten2[25];
    cout << "Ten nguoi thu hai: ";
    gets(ten2);
    Nguoi *B = Nguoi::TimNguoi(ten2);
    if (B==0)
    {
        cout << "Khong co nguoi ten " << ten2 << endl;
        getch();
        return;
    }
    cout << QuanHe(A, B) << endl;
    getch();
}

```

Để đơn giản trong việc thiết kế nhập dữ liệu cho chương trình, chúng ta dùng

phương pháp nhập dữ liệu từ tệp. Việc hình thành con người và các mối quan hệ của chúng là thông qua các sự kiện chính: Tạo một con người, Đám cưới của hai người, Sinh con của người phụ nữ. Do vậy tệp dữ liệu phải thể hiện được điều này. Dùng tệp văn bản để mô tả các sự kiện tỏ ra thích hợp trong trường hợp này. Dưới đây là một tệp văn bản mô tả cho một quan hệ gia đình đơn giản.



Tệp văn bản dữ liệu:

Tao		Tạo người tên Thang là nam
Thang		
Tao		Tạo người tên Mai là nữ
Mai		
Cuoi		
Thang		Thang cưới Mai
Mai		
Sinh		
Mai		Mai sinh con gái tên là Nga
Nga		
Sinh		
Mai		
Tuan		Mai sinh con trai tên là Tuấn
1		

Hàm nhập dữ liệu từ tệp văn bản.

```

void NhapDuLieu() {
    clrscr();
    char s[80];
    cout << "Ten tệp nhập dữ liệu: ";
    cin >> s;
}
  
```

```

ifstream input(s, ios::in|ios::nocreate);
input.seekg(0L, ios::end );
if ( input.tellg() < 0) {
    cout << "Loi mo tep ! \n";
    getch();
    return;
}
input.seekg(0L, ios::beg);
cout << "Dang nhap du lieu.....\n";
int dong = 1;
while (1) {
    input.getline(s, sizeof(s));
    if (input.gcount()==0) break;
    if (strcmp(s, "")==0) {
        dong++;
        continue;
    }
    if (strcmp(s, "Tao") == 0) {
        char ten[25];
        input.getline(ten, sizeof(ten));
        if (strcmp(ten, "") ) {
            int gt;
            input >> gt;
            cout << "Tao nguoi ten " << ten << endl;
            if (Nguoi::TimNguoi(ten))
                cout << "Da co nguoi ten la " << ten << endl;
            else
                Nguoi::TaoNguoi(ten, gt);
            dong += 2;
            continue;
        }
    }
}

```

```

}
if (strcmp(s, "Cuoi") == 0) {
    char ten1[25];
    input.getline(ten1, sizeof(ten1));
    char ten2[25];
    input.getline(ten2, sizeof(ten2));
    Nguoi* A = Nguoi::TimNguoi(ten1);
    Nguoi* B = Nguoi::TimNguoi(ten2);
    cout << "Cuoi " << ten1 << " va " << ten2 << endl;
    if (A == 0)
        cout << "Khong co nguoi ten " << ten1 << endl;
    if (B == 0)
        cout << "Khong co nguoi ten " << ten2 << endl;
    if (A && B && A->Cuoi(B) == 0)
        cout << "Khong cuoi duoc\n";
    dong += 2;
    continue;
}
if (strcmp(s, "Sinh") == 0)
{
    char ten1[25];
    input.getline(ten1, sizeof(ten1));
    char ten2[25];
    input.getline(ten2, sizeof(ten2));
    if (strcmp(ten2, "") != 0) {
        cout << ten1 << " sinh con " << ten2 << endl;
        Nguoi* A = Nguoi::TimNguoi(ten1);
        if (A == 0)
            cout << "Khong co nguoi ten " << ten1 << endl;
        int gt;
        input >> gt;
    }
}

```

```

if (Nguoi::TimNguoi(ten2))
    cout << "Da co nguoi ten la " << ten2 << endl;
else {
    if ( A )
        if ( A->GioiTinh() )
            cout << ten1 << " la nam khong sinh con duoc\n";
        else
            ((Nu*)A)->SinhCon(ten2, gt);
    }
    dong += 3;
    continue;
}
}

cout << "Loi o dong thu " << dong << endl;
break;
}
cout << "Ket thuc nhap.\n";
getch();
}
}

```

Viết thêm menu cho chương trình có dạng như sau:

Lựa chọn công việc theo số

1. Nhập dữ liệu
2. Tìm quan hệ
3. Kết thúc

```

void Menu() {
    clrscr();
    cout << "\n\n Lựa chọn công việc theo số\n\n";
    cout << " 1. Nhập dữ liệu\n";
    cout << " 2. Tìm quan hệ\n";
    cout << " 3. Kết thúc\n";
}

```

```

}

void main(){
    int i;
    Menu();
    do
    {
        i = getch();
        switch (i)
        {
            case '1':
                Nguoi::XoaDuLieu();
                NhapDuLieu();
                Menu();
                break;
            case '2':
                TimQuanHe();
                Menu();
        }
    }while (i!='3');
    Nguoi::XoaDuLieu();
}

```

Ví dụ sau khi nhập dữ liệu cho chương trình từ tệp dữ liệu như ta có ở trên và thực hiện tìm quan hệ màn hình sẽ có dạng như sau:

```

Ten nguoi thu nhat: Thang
Ten nguoi thu hai: Mai
Mai va Thang co quan he vo chong

```

MÃ CHƯƠNG TRÌNH

BÀI TOÁN QUAN HỆ GIA ĐÌNH

```
#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <conio.h>
#include <stdio.h>

class Nguoi
{
    friend class Nam;
    friend class Nu;

    static Nguoi* NhanDan[100];
    static int SoDan;

    char Ten[25];
    Nam *Bo;
    Nu *Me;
    Nguoi *AnhChi[10], *CacEm[10], *CacCon[10];
    int SoAnhChi, SoEm, SoCon;

    Nguoi(char *ten, Nam *bo, Nu *me) :
        Bo(bo), Me(me), SoAnhChi(0), SoEm(0), SoCon(0)
    {
        strcpy(Ten, ten);
    }

    void ThemAnhChi(Nguoi* nguoi)
```

```

{
    AnhChi[SoAnhChi++] = nguoi;
}

void ThemEm(Nguoi* nguoi)
{
    CacEm[SoEm++] = nguoi;
}

void ThemCon(Nguoi* nguoi)
{
    CacCon[SoCon++] = nguoi;
}

public:

static int LaySoDan() { return SoDan; }

static Nguoi* ThemDan(Nguoi* nguoi)
{
    return NhanDan[SoDan++] = nguoi;
}

static Nguoi* TaoNguoi(char* ten, int gioitinh, Nam *bo=0, Nu *me=0);
static Nguoi* TimNguoi(char* ten);
static void XoaDuLieu();

virtual int GioiTinh()=0;
virtual int Cuoi(Nguoi*)=0;

int LaCha(Nguoi *);
int LaMe(Nguoi *);
int LaCon(Nguoi *);
int LaAnh(Nguoi *);
int LaChi(Nguoi *);
int LaEm(Nguoi *);
int LaCo(Nguoi *);

```

```
int LaDi(Nguoi *);  
int LaChu(Nguoi *);  
int LaCau(Nguoi *);  
int LaMo(Nguoi *);  
int LaBac(Nguoi *);  
int LaOngNoi(Nguoi *);  
int LaBaNoi(Nguoi *);  
int LaOngNgoai(Nguoi *);  
int LaAnhHo(Nguoi *);  
int LaChiHo(Nguoi *);  
int LaEmHo(Nguoi *);  
  
virtual int LaVo(Nguoi*)=0;  
virtual int LaChong(Nguoi*)=0;  
  
char* LayTen() { return Ten; }  
};  
  
Nguoi* Nguoi::NhanDan[];  
int Nguoi::SoDan = 0;  
  
class Nam : public Nguoi  
{  
    Nu *Vo;  
  
    int LaVo(Nguoi *) { return 0; }  
  
public:  
    Nam(char *ten, Nam *bo=0, Nu *me=0) : Nguoi(ten, bo, me), Vo(0) {}  
  
    int GioiTinh() { return 1; }  
};
```

```
.int Cuoi(Nguoi *vo);

    int LaChong(Nguoi * nguoi);

};

class Nu : public Nguoi
{
    Nam *Chong;

    int LaChong(Nguoi *) { return 0; }

public:
    Nu(char *ten, Nam *bo=0, Nu *me=0) : Nguoi(ten, bo, me), Chong(0) {}

    int GioiTinh() { return 0; }

    int Cuoi(Nguoi *chong);

    void SinhCon(char* ten, int gioitinh);

    int LaVo(Nguoi * nguoi);
};

Nguoi* Nguoi::TaoNguoi(char* ten, int gioitinh, Nam *bo, Nu *me)
{
    return gioitinh ? ThemDan(new Nam(ten, bo, me)) : ThemDan(new Nu(ten,
bo, me));
}

Nguoi* Nguoi::TimNguoi(char* ten)
```

```

for (int i=0; i<SoDan; i++)
    if (strcmp(ten, NhanDan[i]->LayTen())==0) return NhanDan[i];
return 0;
}

void Nguoi::XoaDuLieu()
{
    for (int i=0; i<SoDan; i++)
        delete NhanDan[i];
    SoDan = 0;
}

inline int Nguoi::LaCha(Nguoi *nguoi)
{
    return nguoi->Bo==this;
}

inline int Nguoi::LaMe(Nguoi *nguoi)
{
    return nguoi->Me==this;
}

int Nguoi::LaCon(Nguoi *nguoi)
{
    return nguoi->LaCha(this) || nguoi->LaMe(this);
}

int Nguoi::LaAnh(Nguoi *nguoi)
{
    if (GioiTinh()==0) return 0;
    return nguoi->LaEm(this);
}

int Nguoi::LaChi(Nguoi *nguoi)
{
    if (GioiTinh()) return 0;
    return nguoi->LaEm(this);
}

```

```

}

int Nguoi::LaEm(Nguoi *nguoi)
{
    for (int i=0; i<SoAnhChi; i++)
        if (AnhChi[i]==nguoi) return 1;
    return 0;
}

int Nguoi::LaCo(Nguoi *nguoi)
{
    if (GioiTinh()) return 0;
    for (int i=0; i<SoDan; i++)
    {
        if (NhanDan[i]->LaAnh(this)&&NhanDan[i]->LaCha(nguoi)) return 1;
    }
    return 0;
}

int Nguoi::LaDi(Nguoi *nguoi)
{
    if (GioiTinh()) return 0;
    for (int i=0; i<SoDan; i++)
        if (NhanDan[i]->LaChi(this)&&NhanDan[i]->LaMe(nguoi)) return 1;
    return 0;
}

int Nguoi::LaChu(Nguoi *nguoi)
{
    if (GioiTinh()==0) return 0;
    for (int i=0; i<SoDan; i++)
        if (LaEm(NhanDan[i])&&NhanDan[i]->LaCha(nguoi)) return 1;
    return 0;
}

int LaCau(Nguoi *nguoi)
{
    if (GioiTinh()==0) return 0;

```

```
for (int i=0; i<SoDan; i++)
    if (LaEm(NhanDan[i]) && NhanDan[i]->LaMe/nguo)) return 1;
    return 0;
}

int LaMo(Nguoi *nguo)
{
    if (GioiTinh()) return 0;
    for (int i=0; i<SoDan; i++)
        if (LaVo(NhanDan[i]) && NhanDan[i]->LaCau/nguo)) return 1;
    return 0;
}

int Nguoi::LaBac(Nguoi *nguo)
{
    for (int i=0; i<SoDan; i++)
        if (NhanDan[i]->LaEm(this) && nguo->LaCon(NhanDan[i])) return 1;
    return 0;
}

int Nguoi::LaOngNoi(Nguoi *nguo)
{
    if (GioiTinh()==0) return 0;
    for (int i=0; i<SoDan; i++)
        if (LaCha(NhanDan[i]) && NhanDan[i]->LaCha/nguo)) return 1;
    return 0;
}

int Nguoi::LaBaNoi(Nguoi *nguo)
{
    if (GioiTinh()) return 0;
    for (int i=0; i<SoDan; i++)
        if (LaMe(NhanDan[i]) && NhanDan[i]->LaCha/nguo)) return 1;
    return 0;
}

int Nguoi::LaOngNgoai(Nguoi *nguo)
{
```

```

if (GioiTinh()==0) return 0;
for (int i=0; i<SoDan; i++)
    if (LaCha(NhanDan[i])&&NhanDan[i]->LaMe/nguo)) return 1;
    return 0;
}

int Nguoi::LaBaNgoai(Nguoi *nguo)
{
    if (GioiTinh()) return 0;
    for (int i=0; i<SoDan; i++)
        if (LaMe(NhanDan[i])&&NhanDan[i]->LaMe/nguo)) return 1;
    return 0;
}

int Nguoi::LaAnhHo(Nguoi *nguo)
{
    if (GioiTinh()==0) return 0;
    return nguo->LaEmHo(this);
}

int Nguoi::LaChiHo(Nguoi *nguo)
{
    if (GioiTinh()) return 0;
    return nguo->LaEmHo(this);
}

int Nguoi::LaEmHo(Nguoi *nguo)
{
    for (int i=0; i<SoDan; i++)
        if ( LaCon(NhanDan[i])&&(NhanDan[i]->LaChu/nguo))
            ||NhanDan[i]->LaDi/nguo||!NhanDan[i]->LaCo/nguo)) )
return 1;
    return 0;
}

int Nam::Cuoi(Nguoi *vo)
{

```

```

if (Vo||vo->GioiTinh()) return 0;
Vo = (Nu*)vo;
Vo->Cuoi(this);
return 1;
}

inline int Nam::LaChong(Nguoi *nguoi) { return Vo==nguoi; }

int Nu::Cuoi(Nguoi *chong)
{
    if (Chong||chong->GioiTinh()==0) return 0;
    Chong = (Nam*)chong;
    Chong->Cuoi(this);
    return 1;
}

void Nu::SinhCon(char *ten, int gioitinh)
{
    Nguoi* nguoi = TaoNguoi(ten, gioitinh, Chong, this);
    ThemCon(nguoi);
    if (Chong) Chong->ThemCon(nguoi);
    for (int i=0; i<SoCon; i++)
    {
        CacCon[i]->ThemEm(nguoi);
        nguoi->ThemAnhChi(CacCon[i]);
    }
}

inline int Nu::LaVo(Nguoi *nguoi) { return Chong==nguoi; }

char qh[256];

char* QuanHe(Nguoi* A, Nguoi* B)

```

```

{
    for (int i=1; i<=2; i++)
    {
        strcpy(qh, A->LayTen());
        strcat(qh, " va ");
        strcat(qh, B->LayTen());
        strcat(qh, " co quan he ");
        if (A->LaOngNoi(B))
            return strcat(qh, "ong chau noi");
        if (A->LaBaNoi(B))
            return strcat(qh, "ba chau noi");
        if (A->LaOngNgoai(B))
            return strcat(qh, "ong chau ngoai");
        if (A->LaBaNgoai(B))
            return strcat(qh, "ba chau ngoai");
        if (A->LaCha(B))
            return strcat(qh, "cha con");
        if (A->LaMe(B))
            return strcat(qh, "me con");
        if (A->LaCo(B))
            return strcat(qh, "co chau");
        if (A->LaDi(B))
            return strcat(qh, "di chau");
        if (A->LaChu(B))
            return strcat(qh, "chu chau");
        if (A->LaBac(B))
            return strcat(qh, "bac chau");
        if (A->LaAnh(B))
            return strcat(qh, "anh em");
        if (A->LaChi(B))
            return strcat(qh, "chi em");
        if (A->LaAnhHo(B))
            return strcat(qh, "anh em ho");
    }
}

```

```
    if (A->LaChiHo(B))
        return strcat(qh, "chi em ho");
    if (A->LaVo(B))
        return strcat(qh, "vo chong");

    Nguoi* temp = A;
    A = B;
    B = temp;
}

strcpy(qh, A->LayTen());
strcat(qh, " va ");
strcat(qh, B->LayTen());
return strcat(qh, " khong co quan he gia dinh");
}

// thủ tục nhập dữ liệu từ tệp để tạo cây gia đình
void NhapDuLieu()
{
    clrscr();
    char s[80];
    cout << "Ten tep nhap du lieu: ";
    cin >> s;

    ifstream input(s, ios::in|ios::nocreate);
    input.seekg(0L, ios::end );
    if ( input.tellg() < 0)
    {
        cout << "Loi mo tep ! \n";
        getch();
        return;
    }
    input.seekg(0L, ios::beg);
```

```

cout << "Dang nhap du lieu.....\n";
int dong = 1;
while (1)
{
    input.getline(s, sizeof(s));
    if (input.gcount() == 0) break;
    if (strcmp(s, "") == 0)
    {
        dong++;
        continue;
    }
    if (strcmp(s, "Tao") == 0)
    {
        char ten[25];
        input.getline(ten, sizeof(ten));
        if (strcmp(ten, ""))
        {
            int gt;
            input >> gt;
            cout << "Tao nguoi ten " << ten << endl;
            if (Nguoi::TimNguoi(ten))
                cout << "Da co nguoi ten la " << ten << endl;
            else
                Nguoi::TaoNguoi(ten, gt);
            dong += 2;
            continue;
        }
    }
    if (strcmp(s, "Cuoi") == 0)
    {
        char ten1[25];
        input.getline(ten1, sizeof(ten1));
        char ten2[25];
    }
}

```

```

        input.getline(ten2, sizeof(ten2));
        Nguoi* A = Nguoi::TimNguoi(ten1);
        Nguoi* B = Nguoi::TimNguoi(ten2);
        cout << "Cuoi " << ten1 << " va " << ten2 << endl;
        if (A==0) cout << "Khong co nguoi ten " << ten1 << endl;
        if (B==0) cout << "Khong co nguoi ten " << ten2 << endl;
        if (A&&B&&A->Cuoi(B)==0)
            cout << "Khong cuoi duoc\n";
        dong += 2;
        continue;
    }
    if (strcmp(s, "Sinh")==0)
    {
        char ten1[25];
        input.getline(ten1, sizeof(ten1));
        char ten2[25];
        input.getline(ten2, sizeof(ten2));
        if (strcmp(ten2, ""))
        {
            cout << ten1 << " sinh con "<< ten2 << endl;
            Nguoi* A = Nguoi::TimNguoi(ten1);
            if (A==0) cout << "Khong co nguoi ten " << ten1 << endl;
            int gt;
            input >> gt;
            if (Nguoi::TimNguoi(ten2))
                cout << "Da co nguoi ten la " << ten2 << endl;
            else
            {
                if ( A )
                    if ( A->GioiTinh() )
                        cout << ten1 << " la nam khong sinh con
duoc\n";
                else

```

```
    ( (Nu*) A) ->SinhCon(ten2, gt);
}
dong += 3;
continue;
}
}

cout << "Loi o dong thu " << dong << endl;
break;
}
cout << "Ket thuc nhap.\n";
getch();
}
```

void TimQuanHe()

```
{
clrscr();
char ten1[25];
cout << "Ten nguoi thu nhat: ";
gets(ten1);
Nguoi *A = Nguoi::TimNguoi(ten1);
if (A==0)
{
    cout << "Khong co nguoi ten " << ten1 << endl;
    getch();
    return;
}
char ten2[25];
cout << "Ten nguoi thu hai: ";
gets(ten2);
Nguoi *B = Nguoi::TimNguoi(ten2);
if (B==0)
{
    cout << "Khong co nguoi ten " << ten2 << endl;
```

```
getch();
return;
}

cout << QuanHe(A, B) << endl;
getch();
}

void Menu()
{
    clrscr();
    cout << "\n\n   Lua chon cong viec theo so\n\n";
    cout << "    1. Nhap du lieu\n";
    cout << "    2. Tim quan he\n";
    cout << "    3. Ket thuc\n";
}

void main()
{
    int i;
    Menu();
    do
    {
        i = getch();
        switch (i)
        {
            case '1':
                Nguoi::XoaDuLieu();
                NhaphDuLieu();
                Menu();
                break;
            case '2':
                TimQuanHe();
                Menu();
        }
    }
}
```

```
    }
}while (i!='3');
Nguoi::XoaDuLieu();
}
```

TÀI LIỆU THAM KHẢO

- [1]. Claude Delannoy, *Programmer en langage C++*, EYROLLES.
- [2]. Scott Robert Ladd, *Turbo C++ Techniques and Applications*, M&T Books.
- [3]. H.M. Deitel & P.J. Deitel, *C How to program*.

MỤC LỤC

Trang

Lời giới thiệu.....	
Lời nói đầu	
Chương 1. Lập trình hướng đối tượng, phương pháp giải quyết bài toán mới	
1. Phương pháp lập trình	1
2. Bài toán quan hệ gia đình.....	2
3. Lập trình hướng đối tượng.....	6
3.1 Một số khái niệm	7
3.2 Các ưu điểm của LTHĐT	8
3.3 Những ứng dụng của LTHĐT	9
4. Các ngôn ngữ lập trình hướng đối tượng.....	9
5. Ngôn ngữ lập trình C++	10
Chương 2. Các mở rộng của C++ so với C	
1. Các điểm không tương thích giữa C++ và ANSI C.....	13
1.1 Định nghĩa hàm.....	13
1.2 Khai báo hàm nguyên mẫu	13
1.3 Sự tương thích giữa con trả <i>void</i> và các con trả khác	14
2. Các khả năng vào/ra mới của C++	15
2.1 Ghi dữ liệu lên thiết bị ra chuẩn (màn hình) <i>cout</i>	15
2.2 Các khả năng viết ra trên <i>cout</i>	16
2.3 Đọc dữ liệu từ thiết bị vào chuẩn (bàn phím) <i>cin</i>	18
3. Những tiện ích cho người lập trình.....	19
3.1 Chú thích cuối dòng	19
3.2 Khai báo mọi nơi	20
3.3 Toán tử phạm vi “::”	20
4. Hàm <i>inline</i>	21
5. Tham chiếu	23
5.1 Tham chiếu tới một biến.....	23

5.2	Truyền tham số cho hàm bằng tham chiếu	25
5.3	Giá trị trả về của hàm là tham chiếu.....	28
6.	Định nghĩa chồng hàm (Overloading functions)	29
	<i>Trường hợp các hàm có một tham số</i>	31
	<i>Trường hợp các hàm có nhiều tham số.....</i>	32
7.	Tham số ngầm định trong lời gọi hàm.....	32
8.	Bổ sung thêm các toán tử quản lý bộ nhớ động: new và delete	35
8.1	Toán tử cấp phát bộ nhớ động new	35
8.2	Toán tử giải phóng vùng nhớ động delete	36
9.	Tóm tắt	38
9.1	Ghi nhớ.....	38
9.2	Các lỗi thường gặp	39
9.3	Một số thói quen lập trình tốt	39
10.	Bài tập	39

Chương 3. Đối tượng và lớp

1.	Đối tượng	40
2.	Lớp.....	42
2.1	Khai báo lớp.....	42
2.1.1	Tạo đối tượng	44
2.1.2	Các thành phần dữ liệu.....	45
2.1.3	Các hàm thành phần.....	45
2.1.4	Tham số ngầm định trong lời gọi hàm thành phần.....	49
2.1.5	Phạm vi lớp.....	50
2.1.6	Từ khoá xác định thuộc tính truy xuất.....	50
2.1.7	Gọi một hàm thành phần trong một hàm thành phần khác	54
2.2	Khả năng của các hàm thành phần	54
2.2.1	Định nghĩa chồng các hàm thành phần.....	54
2.2.2	Các tham số với giá trị ngầm định.....	56
2.2.3	Sử dụng đối tượng như tham số của hàm thành phần	57
2.2.4	Con trỏ this	59
3.	Phép gán các đối tượng	59

4.	Hàm thiết lập (constructor) và hàm huỷ bỏ (destructor)	60
4.1	Hàm thiết lập	60
4.1.1	Chức năng của hàm thiết lập	60
4.1.2	Một số đặc điểm quan trọng của hàm thiết lập	62
4.1.3	Hàm thiết lập ngầm định	63
4.1.4	Con trỏ đối tượng	67
4.1.5	Khai báo tham chiếu đối tượng	69
4.2	Hàm huỷ bỏ	70
4.2.1	Chức năng của hàm huỷ bỏ	70
4.2.2	Một số qui định đối với hàm huỷ bỏ	71
4.3	Sự cần thiết của các hàm thiết lập và huỷ bỏ lớp vector trong không gian n chiều	72
4.4	Hàm thiết lập sao chép (COPY CONSTRUCTOR)	75
4.4.1	Các tình huống sử dụng hàm thiết lập sao chép	75
4.4.2	Hàm thiết lập sao chép ngầm định	76
4.4.3	Khai báo và định nghĩa hàm thiết lập sao chép tường minh	77
4.4.4	Hàm thiết lập sao chép cho lớp vector	79
5.	Các thành phần tĩnh (static)	83
5.1	Thành phần dữ liệu static	83
5.2	Khởi tạo các thành phần dữ liệu tĩnh	85
5.3	Các hàm thành phần static	87
6.	Đối tượng hằng (CONSTANT)	89
6.1	Đối tượng hằng	89
6.2	Hàm thành phần const	89
7.	Hàm bạn và lớp bạn	89
7.1	Đặt vấn đề	89
7.2	Hàm tự do bạn của một lớp	90
7.3	Các kiểu bạn bè khác	92
7.3.1	Hàm thành phần của lớp là bạn của lớp khác	92
7.3.2	Hàm bạn của nhiều lớp	93
7.3.3	Tất cả các hàm của lớp là bạn của lớp khác	95

7.4	Bài toán nhân ma trận với vector	95
	Giải pháp thứ nhất-prod là hàm bạn tự do	95
	Giải pháp thứ hai-prod là hàm thành phần của lớp matrix và là bạn của vector...	97
8.	Ví dụ tổng hợp.....	98
9.	Tóm tắt	103
9.1	Ghi nhớ.....	103
9.2	Các lỗi thường gặp	104
9.3	Một số thói quen lập trình tốt	105
10.	Bài tập	105

Chương 4. Định nghĩa toán tử trên lớp

1.	Giới thiệu chung	109
2.	Ví dụ trên lớp số phức	110
2.1	Hàm toán tử là hàm thành phần	110
2.2	Hàm toán tử là hàm bạn	112
3.	Khả năng và giới hạn của định nghĩa chồng toán tử.....	122
	Phản lớn toán tử trong C++ đều có thể định nghĩa chồng	122
	Trường hợp các toán tử ++ và --	123
	Lựa chọn giữa hàm thành phần và hàm bạn	124
4.	Chiến lược sử dụng hàm toán tử.....	124
	Các phép toán một ngôi	124
	Các phép toán hai ngôi	124
	Các phép gán	124
	Toán tử truy nhập thành phần “->”.....	125
	Toán tử truy nhập thành phần theo chỉ số	125
	Toán tử gọi hàm	125
5.	Một số ví dụ tiêu biểu	125
5.1	Định nghĩa chồng phép gán “=”.....	125
5.2	Định nghĩa chồng phép “[]”	130
5.3	Định nghĩa chồng << và >>	133
5.4	Định nghĩa chồng các toán tử new và delete	135
5.5	Phép nhân ma trận véc tơ.....	137

6.	Chuyển đổi kiểu	142
6.1	Hàm toán tử chuyển kiểu ép buộc.....	143
6.1.1	Hàm toán tử chuyển kiểu trong lời gọi hàm.....	145
6.1.2	Hàm toán tử chuyển kiểu trong biểu thức	147
6.2	Hàm toán tử chuyển đổi kiểu cơ sở sang kiểu lớp	148
6.2.1	Hàm thiết lập trong các chuyển đổi kiểu liên tiếp.....	150
6.2.2	Lựa chọn giữa hàm thiết lập và phép toán gán	150
6.2.3	Sử dụng hàm thiết lập để mở rộng ý nghĩa một phép toán.....	152
6.3	Chuyển đổi kiểu từ lớp này sang một lớp khác.....	154
6.3.1	Hàm toán tử chuyển kiểu bắt buộc	154
6.3.2	Hàm thiết lập dùng làm hàm toán tử	156
7.	Tóm tắt	157
7.1	Ghi nhớ.....	157
7.2	Các lỗi thường gặp	158
7.3	Một số thói quen lập trình tốt	158
8.	Bài tập	158

Chương 5. Kỹ thuật thừa kế

1.	Giới thiệu chung	161
2.	Đơn thừa kế	165
2.1	Ví dụ minh họa	165
2.2	Truy nhập các thành phần của lớp cơ sở từ lớp dẫn xuất.....	167
2.3	Định nghĩa lại các thành phần của lớp cơ sở trong lớp dẫn xuất	168
2.4	Tính thừa kế trong lớp dẫn xuất	168
2.4.1	Sự tương thích của đối tượng thuộc lớp dẫn xuất với đối tượng thuộc lớp cơ sở.....	168
2.4.2	Tương thích giữa con trỏ lớp dẫn xuất và con trỏ lớp cơ sở	170
2.4.3	Tương thích giữa tham chiếu lớp dẫn xuất và tham chiếu lớp cơ sở..	172
2.5	Hàm thiết lập trong lớp dẫn xuất	174
2.5.1	Hàm thiết lập trong lớp	174
2.5.2	Phân cấp lời gọi	176
2.5.3	Hàm thiết lập sao chép	177

2.6	Các kiểu dẫn xuất khác nhau.....	181
2.6.1	Dẫn xuất public	182
2.6.2	Dẫn xuất private	182
2.6.3	Dẫn xuất protected.....	182
	Bảng tổng kết các kiểu dẫn xuất.....	182
3.	Hàm ảo và tính đa hình	183
3.1	Đặt vấn đề	183
3.2	Tổng quát về hàm ảo	190
3.2.1	Phạm vi của khai báo virtual	190
3.2.2	Không nhất thiết phải định nghĩa lại hàm virtual	194
3.2.3	Định nghĩa chống hàm ảo	197
3.2.4	Khai báo hàm ảo ở một lớp bất kỳ trong sơ đồ thừa kế.....	197
3.2.5	Hàm huỷ bỏ ảo	201
3.3	Lớp trùu tượng và hàm ảo thuần tuý	204
4.	Đa thừa kế	205
4.1	Đặt vấn đề	205
4.2	Lớp cơ sở ảo	210
4.3	Hàm thiết lập và huỷ bỏ - với lớp ảo	213
4.4	Danh sách mốc nối các đối tượng	219
	Xây dựng lớp trùu tượng	219
4.5	Tạo danh sách mốc nối không đồng nhất	227
5.	Tóm tắt	231
5.1	Ghi nhớ.....	231
5.2	Các lỗi thường gặp	232
5.3	Một số thói quen lập trình tốt	232
6.	Bài tập	232

Chương 6. Khuôn hình

1.	Khuôn hình hàm	233
1.1	Khuôn hình hàm là gì?	233
1.2	Tạo một khuôn hình hàm.....	233

1.3	Sử dụng khuôn hình hàm	234
1.3.1	Khuôn hình hàm cho kiểu dữ liệu cơ sở	234
1.3.2	Khuôn hình hàm min cho kiểu char *	235
1.3.3	Khuôn hình hàm min với kiểu dữ liệu lớp	236
1.4	Các tham số kiểu của khuôn hình hàm	237
1.4.1	Các tham số kiểu trong định nghĩa khuôn hình hàm	237
1.5	Giải thuật sản sinh một hàm thể hiện	240
1.6	Khởi tạo các biến có kiểu dữ liệu chuẩn	241
1.7	Các hạn chế của khuôn hình hàm	241
1.8	Các tham số biểu thức của một khuôn hình hàm	242
1.9	Định nghĩa chống các khuôn hình hàm	244
1.10	Cụ thể hóa các hàm thể hiện	246
1.11	Tổng kết về các khuôn hình hàm	247
2.	Khuôn hình lớp	247
2.1	Khuôn hình lớp là gì?	247
2.2	Tạo một khuôn hình lớp	248
2.3	Sử dụng khuôn hình lớp	249
2.4	Ví dụ sử dụng khuôn hình lớp	250
2.5	Các tham số trong khuôn hình lớp	251
2.5.1	Số lượng các tham số kiểu trong một khuôn hình lớp	251
2.5.2	Sản sinh một lớp thể hiện	251
2.6	Các tham số biểu thức trong khuôn hình lớp	252
2.7	Tổng quát về khuôn hình lớp	254
2.8	Cụ thể hóa khuôn hình lớp	255
2.9	Sự giống nhau của các lớp thể hiện	257
2.10	Các lớp thể hiện và các khai báo bạn bè	258
2.10.1	Khai báo các lớp bạn hoặc các hàm bạn thông thường	258
2.10.2	Khai báo bạn bè của một thể hiện của khuôn hình hàm, khuôn hình lớp	258
2.10.3	Khai báo bạn bè của khuôn hình hàm, khuôn hình lớp	259
2.11	Ví dụ về lớp bảng có hai chỉ số	259

3. Tóm tắt, Ghi nhớ.....	263
4. Bài tập	263
Phụ lục I. Các kênh xuất nhập	
1. Giới thiệu chung	265
1.1 Khái niệm về kênh	265
1.2 Thư viện các lớp vào ra.....	265
2. Lớp ostream	266
2.1 Định nghĩa chặng toán tử << trong lớp ostream	266
2.2 Hàm put	266
2.3 Hàm write	267
2.4 Khả năng định dạng	267
2.4.1 Chọn cơ sở thể hiện	267
2.4.2 Đặt độ rộng.....	268
3. Lớp istream.....	270
3.1 Định nghĩa chặng toán tử “>>” trong lớp istream	270
3.2 Hàm thành phần get	271
3.3 Các hàm thành phần getline và gcount.....	272
3.4 Hàm thành phần read	272
3.5 Một số hàm khác	273
4. Trạng thái lỗi của kênh nhập	273
4.1 Các cờ lỗi	273
4.2 Các thao tác trên các bit lỗi	274
4.2.1 Đọc giá trị	274
4.2.2 Thay đổi trạng thái lỗi	274
4.3 Định nghĩa các toán tử () và !	274
5. Quản lý định dạng	275
5.1 Trạng thái định dạng của một dòng	275
5.2 Từ trạng thái định dạng	276
5.3 Thao tác trên trạng thái định dạng	277
5.3.1 Các toán tử thao tác định dạng không tham số (TTĐDKTS)	278
5.3.2 Các toán tử định dạng có tham số (TTĐDDCTS).....	278

5.3.3 Các hàm thành phần.....	279
6. Liên kết kênh xuất/nhập với một tập tin	280
6.1 Liên kết xuất với một tập tin.....	280
6.2 Liên kết kênh nhập với một tập tin	282
6.3 Các khả năng truy nhập trực tiếp	283
6.4 Các chế độ mở tập tin khác nhau	286
<i>Phụ lục 2. Xử lý lỗi</i>	
1. Bẫy và bắt lỗi	287
2. Hoạt động của chương trình khi một lỗi phát sinh	290
3. Xử lý lỗi trong lớp ứng dụng	293
<i>Phụ lục 3. Bài toán quan hệ gia đình</i>	297
<i>Phụ lục 4. Ma nguồn bài toán quan hệ gia đình</i>	313
Tài liệu tham khảo.....	329

1111123

IP TRUYỀN HÌNH & ĐI
PHÒNG LỚP



1111123

Giá: 35000đ

Giá : 35.000đ