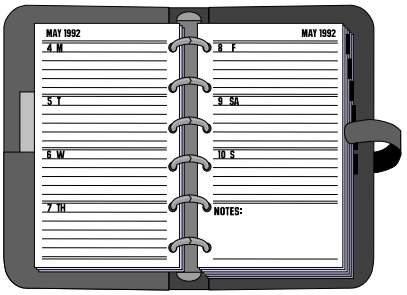
Trường Đại học hàng hải VN
Khoa công nghệ thông tin



Giáo trình
Kỹ thuật lập trình hướng đối tượng

MỤC LỤC

1.2 Một số khái niệm mới của lập trình hướng đối tượng
1.3 Ngôn ngữ lập trình hướng đối tượng
1.4 Những ứng dụng của lập trình hướng đối tượng

Chương II Những mở rộng của C++
2.1 Sự tương thích kiểu dữ liệu
2.2 Khai báo hàm nguyên mẫu
2.3 Khả năng vào ra mới
2.3.1 Hiển thị dữ liệu với đối tượng cout
2.3.2 Nhập liệu với đối tượng cin
2.4 Khai báo mọi nơi
2.5 Toán tử phạm vi
2.6 Tham chiếu
2.7 Tham số ngầm định và định nghĩa chồng hàm
2.7.1 Định nghĩa chồng hàm
2.7.2 Tham số ngầm định
2.8 Toán tử quản lý bộ nhớ động
2.8.1 Toán tử cấp phát bộ nhớ động new
2.8.2 Toán tử dọn dẹp bộ nhớ động delete

Chương III Đối tượng và lớp
3.1 Đối tượng
3.2 Lớp
3.2.1 Khai báo
3.2.2 Đối tượng ẩn
3.2.3 Phép gán đối tượng
3.3 Hàm khởi tạo và hàm dọn dẹp
3.3.1 Hàm khởi tạo
3.3.2 Hàm dọn dẹp
3.3.3 Hàm khởi tạo sao chép
3.4 Thành phần tĩnh
3.4.1 Khởi tạo thành phần dữ liệu tĩnh
3.4.2 Khởi tạo hàm thành phần tĩnh
3.5 Hàm bạn và lớp bạn
3.5.1 Đặt vấn đề
3.5.2 Hàm tự do là bạn của lớp
3.5.3 Hàm bạn của nhiều lớp
3.5.4 Hàm thành phần của lớp là bạn lớp khác
3.5.5 Tất cả hàm của lớp là bạn của lớp khác

Chương IV Định nghĩa Toán tử trên lớp
4.1 Đặt vấn đề
4.2 Hàm toán tử là hàm thành phần
4.3 Hàm toán tử là hàm bạn
4.4 Chiến lược sử dụng hàm toán tử
4.5 Một số ví dụ tiêu biểu
4.5.1 Định nghĩa chồng toán tử << và >>
4.5.2 Định nghĩa chồng toán tử new và delete
4.5.3 Hàm khởi tạo dùng là hàm toán tử

Chương V Khuôn hình

5.1 Khuôn hình hàm
5.1.1 Định nghĩa khuôn hình hàm
5.1.2 Sử dụng khuôn hình hàm với dữ liệu đơn giản
5.1.3 Sử dụng khuôn hình hàm với dữ liệu lớp
5.1.4 Các tham số kiểu
5.1.5 Các tham số biểu thức
5.1.6 Định nghĩa chồng khuôn hình hàm
5.1.7 Cụ thể hóa hàm thể hiện
5.1.8 Các hạn chế của khuôn hình hàm
5.2 Khuôn hình lớp
5.2.1 Định nghĩa khuôn hình lớp
5.2.2 Sử dụng khuôn hình lớp
5.2.3 Các tham số trong khuôn hình lớp
5.2.4 Cụ thể hóa khuôn hình lớp

Chương VI THừa kế

6.1 Giới thiệu chung
6.2 Đơn thừa kế
6.2.1 Ví dụ minh họa
6.2.2 Định nghĩa lại thành phần của lớp cơ sở trong lớp dẫn xuất
6.2.3 Tính thừa kế trong lớp dẫn xuất
6.2.4 Truyền thông tin giữa các hàm khởi tạo
6.3 Hàm ảo và tính đa hình
6.3.1 Ví dụ minh họa
6.3.2 Các thuộc tính về hàm ảo
6.4 Đa thừa kế
6.4.1 Đặt vấn đề
6.4.2 Ví dụ minh họa
6.4.3 Lớp cơ sở ảo

Tài liệu tham khảo:

C++ Kỹ thuật và ứng dụng, Scott Ladd - 1995
Kỹ thuật lập trình hướng đối tượng, Đại học Bách Khoa Hà Nội - 1999

Chương I Giới Thiệu Môn Học Lập Trình Hướng Đối Tượng

1.1 Lịch sử phát triển những phương pháp lập trình

1.1.1 Lập trình tuyến tính

Những phương pháp lập trình hình thành và phát triển song song với các thế hệ máy tính. Khi mà khả năng và tốc độ tính toán của máy tính còn thấp thì những phương pháp lập trình cũng bị hạn chế. Máy tính đầu tiên được lập trình với mã nhị phân, sử dụng công tắc cơ khí để nạp một chương trình. Cùng với sự xuất hiện của các thiết bị lưu trữ và bộ nhớ trong có dung lượng tăng lên thì những ngôn ngữ mới đưa vào sử dụng. Thay vì tính toán trên các bit, byte thì người dùng có thể viết một loạt lệnh gần ngôn ngữ tự nhiên như tiếng Anh để rồi trình dịch biến thành mã máy. Những ngôn ngữ này ban đầu được triển khai các chương trình làm những công việc đơn giản, chủ yếu liên quan tới tính toán trong công tác khoa học và không đòi hỏi gì nhiều ở ngôn ngữ lập trình. Hơn nữa phần lớn các chương trình rất ngắn, ít hơn vài trăm dòng lệnh.

Khi khả năng của máy tính cao hơn, có thể tiến hành những công việc phức tạp. Ngôn ngữ lập trình tuyến tính không còn thích hợp với các công việc đòi hỏi khả năng cao hơn này. Các tiện ích tối thiểu như sử dụng lại các đoạn mã đã viết hầu như không có tức là phải chép lại mỗi khi chúng được dùng trong chương trình. Chương trình rất dài, làm cho logic trở nên khó hiểu, nó chứa nhiều lệnh nhảy tới nhiều chỗ trong chương trình dài dòng này. Xấu hơn nữa là ngôn ngữ không thể kiểm soát phạm vi nhìn thấy của dữ liệu vì dữ liệu chỉ là toàn cục, nếu như sửa đổi biến một vị trí nào thì rất khó kiểm soát.

1.1.2 Lập trình cấu trúc

Rõ ràng là những ngôn ngữ mới với những tính năng mới cần phải được phát triển để tạo ra những ứng dụng tinh vi hơn. Vào những năm 1960 và 1970 đã có giới thiệu đầu tiên về kỹ thuật lập trình cấu trúc. Bản chất của phương pháp này là nhằm chia nhỏ một bài toán lớn thành nhiều bài toán nhỏ hơn căn cứ theo nhiệm vụ chúng phải thực hiện. Mỗi bài toán nhỏ là một chương trình con, chương trình con như là thủ tục hay hàm xử lý công việc rời rạc trong một công việc chung. Các thủ tục được giữ càng độc lập càng tốt, mỗi cái có lý luận và dữ liệu riêng. Những biến chỉ có nghĩa trong phạm vi mỗi thủ tục là biến địa phương tồn tại song song với biến toàn cục và đã có thể hạn chế tối đa các lệnh nhảy gây mất cấu trúc. Thông tin được chuyển giao giữa các thủ tục thông qua đối số và liên kết các thủ tục xây dựng một ứng dụng.

Mục tiêu là làm sao cho việc triển khai các phần mềm trở nên dễ dàng hơn đối với người dùng mà vẫn cải thiện được tính tin cậy và dễ bảo dưỡng. Một chương trình có cấu trúc đã hình thành, việc cô lập từng quá trình xử lý trong mỗi hàm khiến ta có thể viết những chương trình sáng sủa dễ đọc. Một khái niệm lớn đã đưa ra trong lập trình cấu trúc là sự trừu tượng hóa. Trừu tượng hóa là khả năng quan sát sự việc mà không cần hiểu chi tiết bên trong. Trong một chương trình có cấu trúc, ta chỉ cần biết một thủ tục được tin cậy có thể làm được công việc cụ thể gì là đủ còn làm thế nào thì không quan trọng và điều này chính là sự trừu tượng hóa chức năng. Ngày nay, kỹ thuật thiết kế và lập trình cấu trúc có thể được tìm thấy mọi nơi. Hầu như các ngôn ngữ lập trình đều có những công cụ của lập trình cấu trúc, ngay cả những ngôn ngữ cổ điển cũng khai thác những ưu điểm của kiểu lập trình này.

Sự nâng cấp cho các kiểu dữ liệu trong những ứng dụng mà người dùng cũng tiếp tục nhiều nên. Khi độ phức tạp của chương trình tăng lên thì sự phụ thuộc vào dữ liệu mà nó xử lý cũng tăng. Dễ thấy rằng cấu trúc dữ liệu trong chương trình cũng quan trọng như những tính toán thực hiện trên đó. Điều này càng hiển nhiên nếu số lệnh chương trình nhiều. Các kiểu dữ liệu nếu có thay đổi cần phải hiệu chỉnh lại tất cả các điểm có làm việc với các kiểu dữ liệu này, mặt khác một yếu điểm của phương pháp này là khi một chương trình được nhiều người đồng thời cùng tham gia, mỗi người được phân công viết một tập hợp hàm riêng biệt. Vì chương trình vẫn có dữ liệu dùng chung nên sự thay đổi của một phần tử dữ liệu kéo theo thay đổi công việc của toàn nhóm và hậu quả là mất thời gian sửa chữa chương trình.

1.1.3 Lập trình hướng đối tượng

Sự trừu tượng hoá dữ liệu tác động trên dữ liệu cũng tương tự trừu tượng hóa chức năng trên các hàm. Khi có trừu tượng hóa dữ liệu, cấu trúc dữ liệu và các phần tử có thể được sử dụng mà không cần để ý chi tiết đã xây dựng. Ví dụ như kiểu dữ liệu số thực đã được trừu tượng hóa trong mọi ngôn ngữ lập trình, ta có thể sử dụng nó mà không cần biết cách lưu trữ nhị phân trong máy tính. Bởi vì nếu người dùng luôn phải hiểu biết về mọi khía cạnh của vấn đề, ở mọi lúc và mọi hàm trong chương trình thì khó mà viết được một chương trình hiệu quả. Tuy nhiên trừu tượng hóa dữ liệu đã tồn tại sẵn trong những ngôn ngữ lập trình cho phép người dùng sử dụng dễ dàng cũng như có thể định nghĩa dữ liệu của riêng mình.

Khái niệm lập trình hướng đối tượng được xây dựng trên nền tảng của khái niệm lập trình cấu trúc và sự trừu tượng hóa dữ liệu. Sự thay đổi căn bản là một chương trình hướng đối tượng được thiết kế xoay quanh các dữ liệu hơn là theo chức năng. Điều này là hoàn toàn tự nhiên vì mục tiêu của chương trình là xử lý dữ liệu. Lập trình hướng đối tượng liên kết cấu trúc dữ liệu với các thao tác theo cách tự nhiên.

1.2 Một số khái niệm mới của lập trình hướng đối tượng

Lập trình hướng đối tượng cho phép tổ chức dữ liệu trong chương trình theo cách tự nhiên. Bất kỳ sự vật tự nhiên nào cũng được định nghĩa như một lớp. Lớp (class) là bản mẫu mô tả các thông tin cấu trúc dữ liệu, lẫn công việc cụ thể của các phần tử. Khi một phần tử được khai báo là biến của lớp thì nó gọi là đối tượng (object). Các hàm được định nghĩa trong lớp gọi là phương thức hay hàm thành phần (method) và chúng là các hàm duy nhất có thể xử lý dữ liệu của đối tượng lớp đó. Đây chính là tính chất đóng gói dữ liệu (encapsulation).

Mỗi đối tượng còn gọi là biến thể hiện (instance variable), các phương thức có thể gọi bởi các đối tượng của lớp đó. Điều này được gọi là gửi thông điệp cho đối tượng, các thông điệp phụ thuộc vào đối tượng, chỉ đối tượng nào nhận thông điệp mới phải làm việc theo thông điệp trong khi các đối tượng khác hoàn toàn độc lập.

Không giống dữ liệu cơ bản dựng sẵn, có thể sử dụng các lớp có trước như các viên gạch để xây dựng. Các lớp mới có thể tạo ra từ những lớp cũ thông qua sự kế thừa (inheritance). Một lớp mới gọi là lớp dẫn xuất (derived class) có thể thừa hưởng hàm thành phần và dữ liệu của lớp cơ sở (base class). Đương nhiên lớp mới cũng có những dữ liệu và phương thức mới và ta có thể tạo ra số lượng bất kỳ các lớp dẫn xuất. Qua cơ cấu này thì dạng hình cây của lớp đã hình thành, lớp cơ sở là lớp cha còn lớp dẫn xuất là lớp con cháu.

Kỹ thuật lập trình hướng đối tượng còn cung cấp cho người dùng một khả năng

mới, đó là tính đa hình. Tính đa hình (polymorphism) còn gọi là tính tương ứng bội là một trong những tính chất quan trọng được thiết lập trên cơ sở thừa kế mà tại đó đối tượng có thể có biểu hiện cụ thể dựa vào tình huống cụ thể. Tại vì mỗi lớp thừa kế có thể có phương thức riêng cùng với phương thức chung của lớp cơ sở, vấn đề làm thế nào phân biệt đối tượng cụ thể trong cơ cấu lớp trên để tìm đúng phương thức tương ứng của nó.

1.3 Ngôn ngữ lập trình hướng đối tượng

Đa số các ngôn ngữ bậc cao đều hỗ trợ khả năng lập trình hướng đối tượng, có một số ngôn ngữ chỉ chấp nhận cách viết lập trình hướng đối tượng thuần khiết như Java, còn một số ngôn ngữ khác chấp nhận viết song song giữa lập trình cấu trúc và hướng đối tượng như Pascal, C++... Điều này là phù hợp đối với việc bắt đầu học một ngôn ngữ mới, giáo trình này cũng sẽ sử dụng ngôn ngữ C++ để giới thiệu môn học lập trình hướng đối tượng.

C++ phát triển dựa trên trình dịch của C và có thêm hai đặc điểm mới: thứ nhất là khả năng mở rộng như tham chiếu, định nghĩa chồng, ngầm định... cũng có một số mở rộng của C++ nhưng không liên quan đến kỹ thuật lập trình hướng đối tượng mà chỉ có mục đích đơn thuần là tăng sức mạnh của ngôn ngữ, còn đặc điểm thứ hai chính là khả năng lập trình hướng đối tượng. Hiện nay C++ chưa phải là một ngôn ngữ hoàn toàn ổn định, kể từ khi ra đời tới nay đã có nhiều phiên bản của C++, nhưng đa số chúng đều tương thích lẫn nhau.

1.4 Những ứng dụng của lập trình hướng đối tượng

Hiện nay lập trình hướng đối tượng được coi như phương pháp phổ biến để thiết kế giao diện người sử dụng, kiểu như Windows. Các hệ thông tin trong thực tế rất phức tạp, chứa nhiều thuộc tính và chức năng thì lập trình hướng đối tượng tỏ ra có hiệu quả. Các lĩnh vực ứng dụng phù hợp với kỹ thuật lập trình hướng đối tượng có thể liệt kê như dưới đây:

- Các hệ thống làm việc thời gian thực.
- Các hệ mô hình hoá, mô phỏng tiến trình.
- Các hệ cơ sở dữ liệu hướng đối tượng.
- Các hệ siêu văn bản, đa phương tiện.
- Các hệ thống trí tuệ nhân tạo, chuyên gia.
- Các hệ thống song song, mạng nơ-ron...

Chương II Những mở rộng của C++

2.1 Sự tương thích kiểu dữ liệu

Những dữ liệu vô hướng của C được chấp nhận trong C++, tuy nhiên có một số khác biệt nhỏ. Nếu trong C cho phép chuyển đổi khá tuỳ ý các kiểu dữ liệu cơ bản còn trong C++ thì ngay cả dữ liệu cùng kích thước như unsigned char và char cùng một byte, int và unsigned cùng 2 byte... cũng là không tương thích. Dữ liệu con trỏ kiểu void* là tương thích với những con trỏ khác còn việc chuyển đổi ngược lại là không được, phải có thêm lệnh chuyển kiểu:

Ví dụ 1:
```
int a, unsigned b = 5;
float x = 2.0;
a = (int)b;
b = (unsigned )x;
```
Ví dụ 2:
```
void *gen;
int *adj;
gen = adj;
adj = (int*)gen;
```

2.2 Khai báo hàm nguyên mẫu

Đối với một chương trình viết bằng C có thể gọi hàm trước khi định nghĩa hàm đó, yếu tố này rất linh hoạt nhưng có nhược điểm là cú pháp không chặt chẽ, trình dịch đôi khi không phát hiện được hết lỗi khi biên dịch để thực thi, C++ đòi hỏi người sử dụng phải khai báo hàm nguyên mẫu (prototype function) có đầy đủ tên hàm, trị trả về, và danh sách số lượng kiểu dữ liệu đối số của hàm trước khi sử dụng. Mỗi khi trình dịch gặp lời gọi hàm, nó tìm ra hàm nguyên mẫu phù hợp, trường hợp có sự khác nhau có thể thực hiện một số chuyển kiểu có thể được.

C++ cảnh báo bất kể hàm nào cũng nên trả về một giá trị, trường hợp không có trị trả về thì ngầm định sẽ là kiểu int và như thế trong thân hàm bắt buộc phải có lệnh **return**. Mà điều này là không bắt buộc trong C.

2.3 Khả năng vào ra mới

Các hàm hoặc macro vào ra trong C đều sử dụng hiệu quả với C++, nhưng có một số hàm có nhiều tham số hoặc yêu cầu chỉ định dạng kiểu dữ liệu, người mới học ban đầu có thể sẽ gặp khó khăn. C++ cung cấp thêm khả năng vào ra mới dựa trên hai toán tử nhập và xuất với các đặc tính sau: đơn giản dễ sử dụng, có tính năng mở rộng đối với các dữ liệu mới. Muốn sử dụng những hàm này chỉ cần khai báo tệp tiêu đề chứa hàm nguyên mẫu là *"iostream.h"*.

Thiết bị chuẩn ra được sử dụng cùng với "<<" tương ứng với đối tượng **cout**, thiết bị chuẩn vào được sử dụng cùng với ">>" tương ứng với đối tượng **cin**.

2.3.1 Hiển thị dữ liệu với đối tượng cout

Một cách tổng quát có thể sử dụng toán tử "<<" cùng với **cout** để đưa ra màn hình các dữ liệu sau đây:

- Kiểu dữ liệu cơ sở: char, int, float, double.
- Xâu ký tự kiểu char[ ].

- Con trỏ.

"<<" là toán tử hai ngôi, toán hạng trái mô tả nơi kết xuất thông tin (trường hợp này là **cout**), toán hạng phải là một biểu thức nào đó.

Ví dụ mô tả hiển thị một số kiểu dữ liệu cơ bản với **cout:**

```
#include<conio.h>
#include<iostream>
main()
{
clrscr();
int n = 25;
long p = 250000;
unsigned q = 63000;
char c = 'a';
float x = 12.345;
double y = 12.345e6;
char *ch =  "Welcome to C++ !";
int *adr = &n;
cout<<"Value of n: "<<n<<endl;
cout<<"Value of p: "<<p<<endl;
cout<<"Value of q: "<<q<<endl;
cout<<"Value of c: "<<c<<endl;
cout<<"Value of x: "<<x<<endl;
cout<<"Value of y: "<<y<<endl;
cout<<"Value of ch: "<<ch<<endl;
cout<<"Addresse of n: "<<adr<<endl;
cout<<"Addresse of ch: "<<(void*)ch<<endl;  //Hiển thị địa chỉ biến xâu ký tự ch
getch();
}
```

2.3.2 Nhập liệu với đối tượng cin

Nếu như **cout** được sử dụng cho thiết bị ra chuẩn thì **cin** dùng để chỉ thiết bị vào chuẩn. Một cách tương tự, ">>" là toán tử hai ngôi, toán hạng trái là đối tượng **cin**, toán hạng phải là một biến có kiểu là: char, int, float, char*. Cũng có thể nhập một hay nhiều biến cùng hoặc khác kiểu dữ liệu và thực hiện liên tiếp với **cin**, tuy nhiên trong một số trường hợp thì cách làm trên không hoàn toàn chính xác.

Giống như hàm scanf(), **cin** nhập liệu dưới dạng mã ký tự ASCII, sau đó mới biến đổi về kiểu dữ liệu đã khai báo. Do đó có một số những đặc tính sau: các giá trị số hoặc ký tự được phân cách bởi SPACE, TAB, ENTER, khi nhập một xâu ký tự với **cin**, xâu này không thể chứa ký tự phân cách SPACE. C++ cảnh báo không nên sử dụng tuỳ tiện giữa **printf()**, **scanf()** của C với **cout**, **cin** của C++, chỉ nên sử dụng thống nhất một loại thôi.

Ví dụ mô tả nhập dữ liệu cơ bản với **cin:**

```
#include<iostream>
main()
{
int n;
```

```
float x;
char t[25];
do
{
cout<<"Nhập vào một số nguyên: "; cin>>n;
cout<<"Nhập vào một xâu: "; cin>>t;
cout<<"Nhập vào một số thực: "; cin>>x;
cout<<"Đã nhập: "<<n<<" , "<<t<<"  và "<<x<<endl;
}
while(n);
}
```

2.4 Khai báo mọi nơi

Những ngôn ngữ lập trình có cấu trúc yêu cầu người sử dụng phải định nghĩa dữ liệu và khai báo biến trước mã lệnh, tuy nhiên đối với chương trình dài và phức tạp thì bao giờ trong quá trình viết chắc chắn sẽ phải bổ sung thêm biến mới hoặc ít ra là sửa đổi tên biến, kiểu dữ liệu... như thế thì người sử dụng khó kiểm soát được các biến trong trường hợp chương trình có lỗi. C++ chấp nhận các khai báo xen kẽ mã lệnh tại bất kể vị trí nào trong chương trình, ví dụ như sau:

```
void fct()
{
int n = 10;
// ...giả sử có 100 lệnh ở đây
int *p = &n;
cout<<p<<endl;
//...
}
```

2.5 Toán tử phạm vi

Toán tử phạm vi (còn gọi là toán tử định trị) ký hiệu **::** được sử dụng trong nhiều trường hợp, đơn giản nhất là để phân biệt biến địa phương và biến tổng thể cùng tên trong một hàm. Chẳng hạn:

```
#include<iostream>
int x;
main()
{
int x = 10; // biến x địa phương
::x = x*2;   //  biến x tổng thể
cout<<"biến x địa phương: "<<x<<endl;
cout<<"biến x tổng thể: "<<::x<<endl;
}
```

2.6 Tham chiếu

Khái niệm mới "reference" tạm dịch là "tham chiếu", có thể hiểu rằng tham chiếu là "bí danh" của một vùng nhớ được cấp phát cho một biến nào đó. Giống nhau giữa tham chiếu

và tham trỏ là chúng đều chỉ tới các đối tượng có địa chỉ, cùng được cấp phát địa chỉ khi khai báo. Khác nhau ở cách sử dụng, biến tham chiếu khi khởi tạo phải gắn với một biến được tham chiếu nào đó và không thể thay đổi, còn tham trỏ thì rất linh hoạt có thể thay đổi được.

Ví dụ mô tả cách khai báo tham chiếu cho một biến:

```
#include<iostream>
#include<conio.h>
main()
{
clrscr();
int n=3, m=5;
int *p;
p = &n;      // p là tham trỏ lưu địa chỉ biến n
cout<<"p = "<<*p<<endl;
p = &m;     // p lưu địa chỉ biến m
cout<<"p = "<<*p<<endl;
int &q = n; // khai báo tham chiếu q chỉ đến biến n
q = 4;       // gán cho n giá trị 4
cout<<"n = "<<n<<endl;
q = m;       // gán giá trị biến m cho biến n
cout<<"n = "<<n<<endl;
getch();
}
```

Không thể gắn tham chiếu với một hằng số trừ phi có từ khoá **const** đứng trước khai báo tham chiếu:

```
int &p = 3;          // sai
const int &p = 3; // đúng
```

Truyền tham biến trong hàm của C phải sử dụng con trỏ, còn đối với Pascal ta thêm từ khóa **var** trước khai báo biến. Trước khi đánh giá tác dụng của tham chiếu trong việc truyền tham biến hãy xét ví dụ sau:

```
#include<iostream>
#include<conio.h>
void swap(int &x, int &y)
{
 int tmp = x; x = y; y = tmp;
}
main()
{
int a = 3, b = 4;
clrscr();
cout<<"Trước khi gọi swap(): "<<endl;
cout<<"a = "<<a<<" b = "<<b<<endl;
swap(a,b);
cout<<"Sau khi gọi swap(): "<<endl;
```

```
cout<<"a = "<<a<<" b = "<<b<<endl;
getch();
}
```

Hai tham số hình thức trong swap() là những tham chiếu đến các giá trị thực của a và b nên sau khi gọi hàm giá trị của chúng được hoán đổi. Cách sử dụng tham chiếu đơn giản hơn so với tham trỏ, mặt khác kích thước dữ liệu các tham số trong hàm có thể thay đổi thì với việc sử dụng tham chiếu, kích thước biến địa chỉ tham số được truyền chỉ là 2 byte nên tiết kiệm và chạy nhanh hơn.

2.7 Tham số ngầm định và định nghĩa chồng hàm

2.7.1 Định nghĩa chồng hàm

Nếu các hàm có ý nghĩa giống nhau nhưng có tên hàm khác nhau thì điều này không hợp lý, C có ba hàm cùng xác định trị tuyệt đối của một giá trị như sau: int abs(int), long labs(long), và float fabs(float). C++ cho phép sử dụng một tên cho nhiều hàm khác nhau, gọi đây là định nghĩa chồng hàm. Sau đây là ví dụ:

```
#include<iostream>
int abs(int x)
{
 if(x<0) return -x;
  else return x;
}
long abs(long x)
{
 if(x<0) return -x;
  else return x;
}
float abs(float x)
{
 if(x<(float)0) return -x;
  else return x;
}
main()
{
 int a;
 long b;
 float c;
 cout<<"a = "; cin>>a;
 cout<<"b = "; cin>>b;
 cout<<"c = "; cin>>c;
 cout<<"abs(a)= "<<abs(a)<<", abs(b)= "<<abs(b)<<", abs(c)= "<<abs(c)<<endl;
}
```

Các hàm được định nghĩa chồng trùng tên bắt buộc phải khác nhau về số lượng hoặc kiểu dữ liệu tham số hình thức. ở đây kiểu giá trị trả về không được coi là phân biệt giữa các hàm. Việc xác định hàm nào được gọi do trình dịch đảm nhiệm và tuân theo các nguyên tắc

sau: trình dịch tìm kiếm "sự tương ứng nhiều nhất" có thể được; có các mức độ tương ứng như sau:

- Phù hợp số lượng và kiểu tham số.
- Tương ứng dữ liệu số, chuẩn nhưng có sự chuyển đổi kiểu dữ liệu tự động (có cảnh báo nhưng không tạo lỗi cú pháp).
- Các chuyển đổi tự định nghĩa của người sử dụng.

Nếu tìm được hàm duy nhất phù hợp với hàm gọi trong khi biên dịch thì là đúng, ngược lại nếu như có nhiều hơn một hàm có thể chọn hoặc không có hàm nào thì trình dịch đưa ra thông báo lỗi.

2.7.2 Tham số ngầm định

Tham số ngầm định được sử dụng trong những hàm tiện ích cho người sử dụng, danh sách tham số dài khó nhớ nên người dùng chỉ phải truyền một số tối thiểu các tham số, mặt khác tham số ngầm định khác nhau đặt trong các hàm được định nghĩa chồng sử dụng cho nhiều mục đích khác nhau. Quy ước sử dụng tham số ngầm định như sau:

- Tham số ngầm định không nhất thiết là hằng mà có thể là một biểu thức có giá trị được tính tại thời điểm khai báo:

  ```
  float x = 1.5;
  int n = 1;
  void fct(float = x*2 + n);
  ```

- Danh sách tham số ngầm định phải được đặt ở cuối danh sách, liên tiếp nhau để tránh nhầm lẫn giá trị.
- Nếu muốn khai báo giá trị ngầm định cho tham trỏ phải viết * và = cách xa ít nhất một dấu cách.

Ví dụ 1:

```
#include<iostream>
void fct(int = 0);                          // sử dụng một tham số ngầm định
void fct(int = 0, long = 1);                // sử dụng hai tham số ngầm định
void fct(long = 0, float);                  // sai
void fct(int, long = 0, float = 1.0);       // sử dụng hai tham số ngầm định cuối
```

Ví dụ 2:

```
#include<iostream>
main()
{
int n = 10, p = 20;
void fct(int = 0, int = 12);    // khai báo hàm với hai tham số ngầm định
fct(n, p);                      // lời gọi hàm thông thường
fct(n);                  //  tham số thứ hai ngầm định là 12
fct();                          // sử dụng hai tham số ngầm định
}
void fct(int a, int b)
{
```

```
cout<<"tham số a: "<<a<<endl;
cout<<"tham số b: "<<b<<endl;
}
```

Ví dụ 3:

```
#include<iostream>
void fct(int, int = 12);
void fct(int);
main()
{
int n = 10, p = 20;
fct(n, p);        // đúng
fct(n);           // sai
}
```

Trình dịch cho rằng cả hai hàm định nghĩa chồng đều có thể chấp nhận được trong lời gọi hàm fct(n) nên tạo lỗi cú pháp.

2.8 Toán tử quản lý bộ nhớ động

Sử dụng cấp phát bộ nhớ động với **new** và **delete** phải khai báo tệp tiêu đề *"new.h"* đầu chương trình.

2.8.1 Toán tử cấp phát bô nhớ đông **new**

Khai báo cấp phát bộ nhớ động với toán tử **new** đơn giản hơn so với sử dụng hàm của C++, có hai cách sử dụng **new** như sau:
Dạng một:
<kiểu dữ liệu>*<biến> = **new** <kiểu dữ liệu>;
Nếu giá trị trả về là NULL thì việc cấp phát không thành công, ngược lại là biến trỏ trỏ tới vùng nhớ một phần tử theo kiểu dữ liệu đó.
Dạng hai:
<kiểu dữ liệu>*<biến> = **new** <kiểu dữ liệu> [n];
Nếu giá trị trả về là NULL thì việc cấp phát không thành công, ngược lại là biến trỏ trỏ tới vùng nhớ n phần tử theo kiểu dữ liệu đó.
vd:

```
char *st = new char[100];    // con trỏ st trỏ tới vùng nhớ 100 ký tự
int *p = new int;    // con trỏ st trỏ tới vùng nhớ số nguyên
```

2.8.2 Toán tử dọn dẹp bô nhớ đông **delete**

Sau khi đã cấp phát vùng nhớ động với **new**, nên sử dụng **delete** để dọn dẹp kết thúc công việc. Không nên sử dụng lẫn lộn giữa các hàm của C và C++:
Sau đây là một ví dụ cấp phát mảng hai chiều và dọn dẹp khi kết thúc:

```
#include<iostream>
#include<conio.h>
#include<new.h>
#define n 3

void input(int **mat)
```

```
{
for(int i=0; i<n; i++)
 for(int j=0; j<n; j++)
 {
  cout<<"Thành phần thứ ["<<i<<"]["<<j<<"]= ";
  cin>>mat[i][j];
 }
}

void display(int **mat)
{
for(int i=0; i<n; i++)

 {
 for(int j =0; j<n; j++) cout<<mat[i][j]<<" ";
 cout<<endl;
 }
}


main()
{
int **mat;
int i;
clrscr();
mat = new int*[n];
for(i=0; i<n; i++) mat[i] = new int[n];
cout<<"nhập số liệu cho mảng 3*3:"<<endl;
input(mat);
cout<<"hiển thị ma trận:"<<endl;
display(mat);
for(i=0; i<n; i++) delete mat[i];
delete mat;
getch();
}
```

Ví dụ dưới đây thực hiện nhập liệu ngẫu nhiên cho mảng một chiều các phần tử nguyên, rồi sắp xếp dãy số theo thứ tự tăng dần:

```
#include <iostream>
#include <stdlib.h>
#include <new.h>

void swap(int &x, int &y)
{
 int tmp = x; x = y; y = tmp;
}
```

```
int* input(int n)
{
 int *tmp = new int[n];
 for(int i=0; i<n; i++) tmp[i] = rand();
 return tmp;
}

void sort(int *p, int n)
{
 for(int i=0; i<n-1; i++)
  for(int j=i+1; j<n; j++)
   if(p[i]>p[j]) swap(p[i], p[j]);
}

void display(int *p, int n)
{
 for(int i=0; i<n; i++) cout<<p[i]<<" ";
 cout<<endl;
}

main()
{
 int n,*p;
 cout<<"số phần tử của mảng: "; cin>>n;
 p = input(n);
 cout<<"dãy số trước khi sắp xếp:"<<endl;
 display(p, n);
 cout<<"dãy số đã sắp xếp:"<<endl;
 sort(p, n);
 display(p, n);
 delete p;
 }
```

Chương III Đối tượng và lớp

3.1 Đối tượng

Đối tượng là một khái niệm mới trong lập trình hướng đối tượng, nó mô tả sự liên kết giữa dữ liệu và phương thức xử lý dữ liệu:

Đối tượng = Dữ liệu + Phương thức

Một chương trình hướng đối tượng hoàn chỉnh không đơn thuần chỉ được tổ chức phân rã các chức năng con mà còn đóng gói dữ liệu và các chức năng xử lý dữ liệu đó vào trong từng đối tượng, nói cách khác là phải hạn chế tối đa những khai báo kiểu, biến toàn cục, các hàm không được đóng gói trong một đối tượng nào. Như vậy cơ chế đóng gói khiến chương trình đơn giản hơn và tránh được sử dụng sai mục đích, nó cũng góp phần nâng cao chất lượng của chương trình.

Trong lập trình hướng đối tượng, người dùng nếu muốn tác động lên dữ liệu của đối tượng thì phải gửi thông điệp tới đối tượng. Với những ngôn ngữ lập trình trực quan thì có thể thấy rõ điều này: người dùng tác động lên đối tượng thông qua một sự kiện, hệ quả của sự kiện này là những thông điệp tới đối tượng. ở đây các phương thức đóng vai trò là giao diện bắt buộc giữa đối tượng và người sử dụng. Các thông điệp gửi tới đối tượng nào thì chỉ có nghĩa tới đối tượng đó chứ không liên quan tới các đối tượng cùng kiểu khác.

So với lập trình hướng đối tượng thuần khiết, C++ linh hoạt chấp nhận khả năng truy xuất tới những thành phần riêng của đối tượng. Khái niệm lớp chính là cơ sở cho các linh động này. Lớp là mô tả trừu tượng của nhóm đối tượng cùng bản chất, trong lớp là những mô tả tính chất dữ liệu, cách thức thao tác trên chúng (còn gọi là hành vi của các đối tượng). Ngược lại đối tượng là thể hiện cụ thể của lớp và việc khai báo một đối tượng ví như khai báo biến có kiểu lớp.

3.2 Lớp

3.2.1 Khai báo

Thực chất thì khái niệm lớp trong C++ là mở rộng của khái niệm cấu trúc trong C, trước khi trình bày những đặc điểm riêng của lớp ta xem xét cú pháp khai báo chuẩn của nó:

```
class <tên lớp>
{
private:
<khai báo các thành phần dữ liệu, phương thức riêng trong đối tượng>
public:
<khai báo các thành phần dữ liệu, phương thức công khai trong đối tượng>
protected:
<khai báo các thành phần dữ liệu, phương thức bảo vệ trong đối tượng>
};

// định nghĩa hàm thành phần trong lớp
<tên kiểu giá trị trả về><tên lớp>::<tên hàm>(<danh sách tham số hình thức>)
{
// nội dung định nghĩa hàm
}
```

Với cách khai báo trên, lớp là tổ chức cho hàm và dữ liệu trong khi cấu trúc chỉ chứa dữ liệu, tính chất của lớp cũng phức tạp hơn cấu trúc chỉ đơn thuần như một định nghĩa dữ liệu mới của người sử dụng. Những thành phần gán nhãn từ khóa **private** nếu không có chỉ dẫn thêm sẽ chỉ thuộc phạm vi truy xuất bên trong lớp, trường hợp không chỉ rõ nhãn gì được ngầm định là thành phần riêng tư. Những thành phần gán nhãn từ khóa **public** được sử dụng tại mọi nơi trong chương trình giống như dữ liệu trong cấu trúc **struct**. Còn những thành phần gán nhãn từ khóa **protected** là mức trung gian giữa **private** và **public**. Những thành phần loại này là không thể truy xuất từ bên ngoài trừ phi lớp có sự kế thừa. Trong lớp có thể khai báo một hoặc nhiều từ khoá trên.

Định nghĩa hàm thành phần của một lớp cần được khai báo tên lớp, toán tử phạm vi **::** trước tên hàm. Cũng có thể định nghĩa trực tiếp hàm bên trong lớp song điều đó không chỉ khiến chương trình khó đọc mà còn nảy sinh vấn đề là những hàm này sẽ là hàm **inline**, do đó chỉ những hàm có nội dung đơn giản mới đặt trực tiếp bên trong lớp mà chủ yếu là những hàm khởi tạo, dọn dẹp.

Ví dụ 1:

```
#include<iostream>
#include<conio.h>
class point
{
 private:
 int x, y;
 public:
 void init(int ox, int oy);
/*
định nghĩa trực tiếp hàm init() như là hàm inline:
void init(int ox, int oy)
{
cout<<"hàm thành phần khởi tạo"<<endl;
x = ox; y = oy;
}
*/
 void move(int dx, int dy);
 void display();
};
void point::init(int ox, int oy)
{
cout<<"hàm thành phần khởi tạo"<<endl;
x = ox; y = oy;
}
void point::move(int dx, int dy)
{
cout<<"hàm thành phần cập nhật"<<endl;
x += dx; y += dy;
}
void point::display()
{
cout<<"hàm thành phần hiển thị"<<endl;
cout<<"tọa độ: "<<x<<","<<y<<endl;
}
main()
{
clrscr();
point p;
```

```
p.init(2, -4);
p.display();
p.move(1, 3);
p.display();
getch();
}
```

Sau khi đã định nghĩa lớp và các phương thức của nó, đối tượng có thể xác lập thông qua một biến / hằng có kiểu lớp, ở đây đối tượng p kiểu lớp point được khai báo trong hàm main(). Ba hàm thành phần gọi thông qua đối tượng p trong hàm main() được chấp nhận vì chúng có thuộc tính công khai, còn hai biến p.x và p.y chỉ có thể truy xuất trong nội bộ lớp.

Ví dụ 2: Lớp có các hàm nhập liệu, kiểm tra kiểu, tính diện tích tam giác

```
#include<iostream>
#include<math.h>
#include<conio.h>
class tamgiac
{
private:
float a, b, c;
public:
void nhap();            // nhập liệu độ dài 3 cạnh
void in();              // in ra các thông báo
private:
int loaitg();           // in ra kiểu của tam giác: 1-d, 2-vc, 3-c, 4-v, 5-t
float dientich();       // tính diện tích của tam giác
};
void tamgiac::nhap()
{
 do
 {
 cout<<"cạnh a: "; cin>>a;
 cout<<"cạnh b: "; cin>>b;
 cout<<"cạnh c: "; cin>>c;
 }
 while(a+b<=c || b+c<=a || c+a<=b);
}
void tamgiac::in()
{
 cout<<"độ dài ba cạnh: "<<a<<" , "<<b<<" , "<<c<<" , "<<endl;
 switch( loaitg() )
 {
 case 1: cout<<"tam giác đều"<<endl; break;
 case 2: cout<<"tam giác vuông cân"<<endl; break;
 case 3: cout<<"tam giác cân"<<endl; break;
 case 4: cout<<"tam giác vuông"<<endl; break;
```

```
 default: cout<<"tam giác thường"<<endl; break;
 }
cout<<"diện tích tam giác: "<<dientich()<<endl;
}

float tamgiac::dientich()
{
 return (0.25*sqrt((a+b+c)*(a+b-c)*(b+c-a)*(c+a-b)));
}
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==c*c+a*a || c*c==a*a+b*b) return 2;
    else return 3;
 else if(a*a== b*b+c*c || b*b==c*c+a*a || c*c==a*a+b*b) return 4;
  else return 5;
}
main()
{
 clrscr();
 tamgiac tg;
 tg.nhap();
 tg.in();
 getch();
}
```

3.2.2 Đối tượng ẩn

Một hàm thành phần lớp trong khi định nghĩa có thể truy xuất tới những thành phần khác hay chính nó, chẳng hạn ở ví dụ 2, hàm in() gọi hàm loaitg(), dientich() mà không cần biết những hàm này thuộc đối tượng cụ thể nào. Đối tượng ngầm định cho hàm là đối tượng ẩn, để xác định tường minh địa chỉ đối tượng là dùng con trỏ **this**. Chúng khác nhau về trình bày nhưng ý nghĩa thực hiện lệnh hoàn toàn thống nhất:

Mỗi đối tượng trong C++ có sự truy cập tới vị trí riêng của nó thông qua một con trỏ quan trọng gọi là con trỏ this. **Con trỏ this trong C++ là một từ khóa đề cập đến thể hiện hiện tại của lớp**, là một tham số ẩn với tất cả hàm thành viên. Vì thế, bên trong một hàm thành viên, con trỏ this có thể tham chiếu tới đối tượng đang gọi.

Các hàm friend không có con trỏ this, bởi vì friend không phải là các thành viên của một lớp. Chỉ có các hàm thành viên trong C++ là có con trỏ this.

```
void tamgiac::in()
{
 cout<<"độ dài ba cạnh: "<<a<<" , "<<b<<" , "<<c<<" , "<<endl;
 switch(this->loaitg())
```

{
case 1: cout<<"tam giác đều"<<endl; break;
case 2: cout<<"tam giác vuông cân"<<endl; break;
case 3: cout<<"tam giác cân"<<endl; break;
case 4: cout<<"tam giác vuông"<<endl; break;
default: cout<<"tam giác thường"<<endl; break;
}
cout<<"diện tích tam giác: "<<this->dientich()<<endl;
}

```
#include <iostream>
using namespace std;
class NhanVien {
    int msnv;
    string ten;
    int tuoi;
    public:
       NhanVien(int msnv, string ten, int tuoi) {
            cout << "Trong ham xay dung: " << endl;
            cout << "   msnv: " << msnv << endl;
            cout << "   ten: " << ten << endl;
            cout << "   Tuoi: " << tuoi << endl;
            msnv = msnv;    //sử dụng this-> msnv = msnv;   mới đúng
            ten = ten;
            tuoi = tuoi;
       }
       void HienThi() {
            cout << "Ham in thong tin cua doi tuong nhan vien: " << endl;
            cout << ten << endl;
            cout << "   Ma so nhan vien: " << msnv << endl;
            cout << "   Tuoi: " << tuoi << endl;
       }
};

int main() {
    NhanVien n1 =  NhanVien(111231, "Nguyen Van A", 25);
    n1.HienThi();
    return 0;
}
```

https://freetuts.net/con-tro-this-trong-c++-1794.html

3.2.3 Phép gán đối tượng

Có thể khai báo phép gán giữa hai đối tượng cùng kiểu, chẳng hạn như ở ví dụ 1, nếu có hai đối tượng a và b cùng kiểu point mà biến a đã xác định giá trị các thành phần dữ liệu x và y:

point a,b;
a.init(-5, 2);
b = a;
Phép gán b từ a là sao chép giá trị các thành phần x và y tương ứng đôi một không kể đó là thành phần private hoặc public.

3.3 Hàm khởi tạo và hàm dọn dẹp

3.3.1 Hàm khởi tạo

Hàm khởi tạo là một hàm thành phần luôn có trong lớp, nó được tự động gọi khi một đối tượng được khai báo, nhiệm vụ của hàm khởi tạo như tên gọi của nó, dùng để khởi tạo tham số ban đầu. Những thuộc tính của hàm khởi tạo là:

- Hàm khởi tạo có cùng tên với tên lớp.
- Hàm có thuộc tính **public**.
- Hàm không trả về giá trị, kể cả kiểu void.
- Có thể có một hay nhiều hàm khởi tạo, khác nhau về tham số hình thức.

Khi khai báo mảng các đối tượng, phải khai báo hàm khởi tạo không có tham số, như ví dụ sau đây:

```
#include<iostream>
#include<conio.h>
class point
{
int x, y;
public:
point()                    // định nghĩa hàm khởi tạo không tham số
{
 x = 0; y = 0;
}
point(int ox, int oy)    // định nghĩa hàm khởi tạo có hai tham số
{
 x = ox; y = oy;
}
void display();
};
void point::display()
{
 cout<<"tọa độ: "<<x<<","<<y<<endl;
}
main()
{
clrscr();
point a(5, 2);
a.display();
point b[10];
getch();
}
```

Giải pháp thứ hai là dùng hàm khởi tạo có tham số ngầm định, không cần khai báo hàm khởi tạo không tham số:

```
class point
{
int x, y;
public:
point(int ox = 0, int oy = 0)    // định nghĩa hàm khởi tạo có hai tham số ngầm định
{
 x = ox; y = oy;
}
```

};

Đối với những lớp không định nghĩa hàm khởi tạo thì trình dịch tự động sinh ra hàm khởi tạo ngầm định, dĩ nhiên hàm này không thực hiện bất cứ nhiệm vụ gì ngoài việc "lấp chỗ trống".

3.3.2 Hàm dọn dẹp

Hàm dọn dẹp có chức năng ngược lại so với hàm khởi tạo, sau khi đối tượng không còn ý nghĩa sử dụng đối tượng xoá khỏi bộ nhớ thì hàm dọn dẹp tự động được gọi và những tham số đã khởi tạo cho đối tượng cần được xoá đi. Những thuộc tính của hàm khởi tạo là:

- Hàm có thuộc tính **public**.
- Tên hàm huỷ bỏ bắt đầu bằng dấu ~, theo sau là tên của lớp tương ứng.
- Hàm huỷ bỏ không có tham số, do đó một lớp chỉ có tối đa một hàm dọn dẹp.
- Hàm không trả về giá trị, kể cả kiểu void.

Trên thực tế với các lớp không có các thành phần dữ liệu động thì chỉ cần sử dụng hàm khởi tạo và dọn dẹp ngầm định nhưng đối với những chương trình phức tạp chứa các khối dữ liệu lớn thì nên có các hàm khởi tạo và dọn dẹp riêng.

Ví dụ 1: Định nghĩa lớp có hàm khởi tạo và dọn dẹp.

```
#include<iostream>
#include<conio.h>
int line = 1;
void fct(int p)
{
test x(2*p);
}
class test
{
 public:
 int num;
 test(int);
 ~test();
};
test::test(int n)
{
num = n;
cout<<line++<<".";
cout<<"++ Gọi hàm khởi tạo với num = "<<num<<endl;
}
test::~test()
{
cout<<line++<<".";
cout<<"— Gọi hàm dọn dẹp với num = "<<num<<endl;
}
main()
{
```

```
clrscr();
test a(1);
for(int i=1; i<=2; i++) fct(i);
}
```

3.3.3 Hàm khởi tạo sao chép

Như đã trình bày ở phần 3.2.3, trình dịch cho phép thực hiện phép gán đối tượng cùng kiểu lớp, tuy vậy người sử dụng có thể tự định nghĩa một hàm khởi tạo sao chép riêng. Dạng khai báo của hàm này là gắn tên lớp với tham chiếu như một tham số hình thức của hàm, ở đây từ khóa **const** cho biết đó là một đối tượng hằng.

```
point (point &);
point (const point &);
```

Ví dụ sau đây thêm hàm khởi tạo sao chép vào lớp point đã cho:

```
class point
{
int x, y;
public:
point(int ox = 0, int oy = 0)
{
 x = ox; y = oy;
}
point(point &p)
{
cout<<"dùng hàm khởi tạo sao chép"<<endl;
x = p.x; y = p.y;
}
void display();
};
void point::display()
{
 cout<<"tọa độ: "<<x<<","<<y<<endl;
}
point fct(point a)
{
point b = a;
return b;
}
main()
{
clrscr();
point a(5, 2);
a.display();
point b = fct(a);
b.display();
```

```
getch();
}
```

3.4 Thành phần tĩnh

Ngôn ngữ lập trình cấu trúc có định nghĩa kiểu thành phần tĩnh, nếu như một biến tĩnh được khai báo thì nó sẽ tồn tại cho tới lúc kết thúc chương trình. Còn trong lập trình hướng đối tượng, nhiều đối tượng cùng chia sẻ một thành phần tĩnh. Chỉ cần đặt từ khóa **static** trước thành phần tương ứng thì thành phần đó sẽ là thành phần tĩnh. Cách chỉ định như vậy cho biết chỉ có một thể hiện duy nhất của thành phần tĩnh và nó được dùng chung cho những thành phần khác. Ta có nhận xét rằng một thành phần tĩnh giống như một biến toàn cục trong phạm vi lớp.

3.4.1 Khởi tao thành phần dữ liệu tĩnh

Cú pháp khởi tạo một thành phần dữ liệu tĩnh phải đặt bên ngoài lớp:
<kiểu dữ liệu><tên lớp>::<tên biến> = <giá trị>;

C++ không cho rằng khai báo trên vi phạm tính riêng tư nếu biến dữ liệu có nhãn **private**, còn cách khởi tạo biến tĩnh thông qua đối tượng cụ thể cũng không hợp lý vì thành phần này vẫn tồn tại ngay khi không khai báo một đối tượng nào cả.

ví dụ:

```
#include<iostream>
#include<conio.h>
class counter
{
static int count;
public:
counter();
~counter();
};
int counter::count=0;
counter::counter()
{
cout<<"++ tạo: bây giờ có "<<++count<<" đối tượng"<<endl;
}
counter::~counter()
{
cout<<"++ xóa: bây giờ còn "<<--count<<" đối tượng"<<endl;
}
void fct()
{
 counter u, v;
}
main()
{
 clrscr();
 counter a;
 fct();
```

```
 counter b;
}
```

3.4.2 Khởi tạo hàm thành phần tĩnh

Cũng giống như thành phần dữ liệu tĩnh thì một hàm thành phần tĩnh được khai báo bắt đầu với từ khóa **static** và hàm này có đặc điểm là độc lập với bất kỳ đối tượng cụ thể nào của lớp, nói cách khác không thể sử dụng con trỏ **this** trong định nghĩa các hàm thành phần tĩnh.

Cú pháp khởi tạo một hàm thành phần tĩnh:
<trị trả về><tên lớp>::<tên hàm>(<danh sách tham số hình thức>) ;

Một số phiên bản cho phép gọi hàm thành phần tĩnh thông qua đối tượng nhưng một số khác thì không, tuy nhiên cách gọi thông qua tên lớp trực quan hơn cả vì nó phản ánh được bản chất của hàm thành phần tĩnh. Sau đây ta trở lại ví dụ trước nhưng thêm vào một hàm thành phần tĩnh:

```
#include<iostream>
#include<conio.h>
class counter
{
static int count;
public:
counter();
~counter();
static void counter_display();
};
int counter::count=0;
void counter::counter_display()
{
cout<<"hiện đang có "<<count<<" đối tượng"<<endl;
}
counter::counter()
{
cout<<"++ tạo: bây giờ có "<<++count<<" đối tượng"<<endl;
}
counter::~counter()
{
cout<<"++ xóa: bây giờ còn "<<--count<<" đối tượng"<<endl;
}
void fct()
{
 counter u;
 counter::counter_display();
 counter v;
 counter::counter_display();
}
main()
{
```

```
 clrscr();
 counter a;
 fct();
 counter::counter_display();
 counter b;
}
```

3.5 Hàm bạn và lớp bạn

3.5.1 Đặt vấn đề

Thuộc tính truy xuất thành phần của lớp trong lập trình hướng đối tượng là rất linh hoạt, một mặt vừa đảm bảo nguyên tắc đóng gói dữ liệu nhưng mặt khác vẫn có đặc tính mở. C++ cho phép hàm bạn hay lớp bạn truy xuất các thành phần riêng tư của một lớp nào đó, sự vi phạm này hoàn toàn chấp nhận được và tỏ ra hiệu quả trong những trường hợp nhất định. Ta sẽ thấy được điều đó trong bài toán này: giả sử đã định nghĩa hai lớp vector và lớp matrix, nếu muốn tạo hàm nhân giữa thành phần vector với thành phần matrix cùng có nhãn riêng tư thì khó thực hiện được điều này vì hàm này không thể thuộc một lớp nào, càng không thể là hàm tự do. Lúc đó khai báo hàm nhân là hàm bạn của hai lớp trên là giải pháp tối ưu.

Có nhiều kiểu bạn bè:

- Hàm tự do là bạn của lớp.
- Hàm thành phần của một lớp là bạn của một lớp khác.
- Hàm bạn của nhiều lớp.
- Tất cả các hàm thành phần của một lớp là bạn của một lớp khác.

3.5.2 Hàm tự do là bạn của lớp

Ta sẽ định nghĩa một hàm có tên là kiemtra() kiểm tra hai phân số p và q là một hay không. Hàm kiemtra() bắt đầu bằng từ khóa **friend** trước tên hàm chỉ định rằng nó là hàm bạn của lớp phanso.

```
#include<iostream>
#include<conio.h>
class phanso
{
int x, y;
public:
phanso(int tuso, int mauso)
{
 x = tuso; y = mauso;
}
friend int kiemtra(phanso, phanso);     // khai báo hàm bạn tên là kiemtra()
};
int kiemtra(phanso p, phanso q)
```

```
{
 if(p.x*q.y == p.y*q.x) return 1;
 else return 0;
}
main()
{
 clrscr();
 int x,y;
 cout<<"nhập liệu tử số của phân số p: "; cin>>x;
 cout<<"nhập liệu mẫu số của phân số p: "; cin>>y;
 phanso p(x, y);
 cout<<"nhập liệu tử số của phân số q: "; cin>>x;
 cout<<"nhập liệu mẫu số của phân số q: "; cin>>y;
 phanso q(x, y);
 if(kiemtra(p, q)) cout<<"phân số p đồng nhất phân số q"<<endl;
  else cout<<" phân số p khác phân số q"<<endl;
 getch();
}
```

3.5.3 Hàm bạn của nhiều lớp

Hàm bạn của nhiều lớp hoàn toàn tương tự như hàm bạn của một lớp, định nghĩa một hàm f là hàm bạn của hai lớp A và B như sau:

```
class B;
class A
{
…
friend void f(A , B);
…
};
class B
{
…
friend void f(A, B);
…
};
void f(A a, B b)
{
 // truy nhập vào các thành phần riêng của hai lớp bất kỳ A và B
}
```

3.5.4 Hàm thành phần của lớp là bạn lớp khác

Giả sử rằng hàm f() là một hàm thành phần của lớp B, muốn định nghĩa f() là hàm bạn của lớp A thì cú pháp khai báo như sau:

```
class A;
class B
```

```
{
…
int f(char, A);
…
};
class A
{
…
friend int B::f(char, A);
…
};
int B::f(char c, A a)
{
// định nghĩa hàm f
};
```

Dưới đây là ví dụ giải quyết bài toán xây dựng một hàm có tên là proc() thuộc lớp matrix và là bạn lớp vect:

```
#include<iostream>
#include<conio.h>
class vect;
class matrix
{
 double mat[3][3];
 public:
 matrix(double t[3][3])
 {
  for(int i=0; i<3; i++)
   for(int j=0; j<3; j++) mat[i][j] = t[i][j];
 }
 vect prod(vect);                  // khai báo hàm thành phần prod()
};
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);  // prod() là bạn của lớp vect
 void display()
 {
  for(int i=0; i<3; i++) cout<<v[i]<<" ";
  cout<<endl;
 }
};
```

```
vect matrix::prod(vect x)
{
 int i, j;
 double sum;
 vect res;
 for(i=0; i<3; i++)
 {
  sum = 0;
  for(j=0; j<3; j++) sum += mat[i][j]*x.v[j];
  res.v[i] = sum;
 }
 return res;
}
main()
{
 clrscr();
 vect w(1, 2, 3);
 vect res;
 double t[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
 matrix a = t;
 res = a.prod(w);
 res.display();
 getch();
}
```

3.5.5 Tất cả hàm của lớp là bạn của lớp khác

Đây thực chất là chỉ định một lớp A là bạn của lớp B, cách định nghĩa đơn giản là đặt trong lớp A chỉ thị:

```
class B;
class A
{
…
friend class B;
…
};
```

CHƯƠNG IV ĐỊNH NGHĨA TOÁN TỬ TRÊN LỚP

4.1 Đặt vấn đề

Vấn đề định nghĩa chồng toán tử đã từng có trong những ngôn ngữ lập trình có cấu trúc, ta dễ dàng sử dụng các toán tử +, -, *, / để thực hiện các phép tính giữa hai số nguyên, thực, số nguyên và trỏ… Nhưng đối với những biểu thức chứa những phép toán cho những kiểu dữ liệu cơ bản hoặc dữ liệu do người sử dụng tự định nghĩa mà trình dịch không hỗ trợ, chẳng hạn như tạo hàm liên hợp tính toán một dãy các phép tính số học cho các số phức hay các phân số khác nhau thì sẽ nảy sinh vấn đề. Nếu như mỗi hàm được biểu thị bởi những cụm từ tự nhiên thì hàm liên hợp sẽ rất dài, phức tạp và cách trình bày này là không súc tích.

C++ cho phép định nghĩa chồng cho đa số các phép toán (một ngôi hoặc hai ngôi), khả năng này cho phép xây dựng trên các lớp và chương trình trở nên ngắn gọn có ý nghĩa. Để làm được điều này, ta định nghĩa chồng toán tử bằng cách định nghĩa hoạt động của từng phép toán giống như định nghĩa một hàm, hàm toán tử bắt đầu bởi từ khóa **operator** sau đó ký hiệu phép toán tương ứng. Hàm định nghĩa chồng có thể là hàm thành phần hoặc hàm bạn tự do của lớp đó tùy theo đặc điểm của ký hiệu trong phép toán.

4.2 Hàm toán tử là hàm thành phần

Ví dụ dưới đây định nghĩa chồng một số phép toán !=, + , - , * , / giữa hai đối tượng số phức của lớp complex:

```
#include<iostream>
#include<conio.h>
#include<math.h>
class complex
{
float real, img;
public:
complex(float r = 0, float i = 0)
{
 real = r; img = i;
}
void display();
int operator!=(complex b);          // hàm so sánh hai số phức
complex operator+(complex b);  // hàm toán tử cộng hai số phức
complex operator-();                  // hàm toán tử lấy đảo dấu một số phức
complex operator-(complex b);  //  hàm toán tử trừ hai số phức
complex operator*(complex b);  //  hàm toán tử nhân hai số phức
complex operator/(complex b);  //  hàm toán tử chia hai số phức
};
void complex::display()
{
 cout<<real<<(img>=0 ? '+':'-')<<"j*"<<fabs(img)<<endl;
}
int complex::operator!=(complex b)
{
 if(real!=b.real || img!=b.img) return 1;
  else return 0;
}
complex complex::operator+(complex b)
{
 complex c;
 c.real = real + b.real;
 c.img = img + b.img;
 return c;
}
complex complex::operator-()
```

```
{
 complex tmp;
 tmp.real = -real;
 tmp.img = -img;
 return tmp;
}
complex complex::operator-(complex b)
{
 complex c;
 c.real = real - b.real;
 c.img = img - b.img;
 return c;
}
complex complex::operator*(complex b)
{
 complex c;
 c.real = real*b.real – img*b.img;
 c.img = real*b.img + img*b.real;
 return c;
}
complex complex::operator/(complex b)
{
 complex c;
 c.real = (real*b.real + img*b.img)/(b.real*b.real + b.img*b.img);
 c.img = (-real*b.img + img*b.real)/(b.real*b.real + b.img*b.img);
 return c;
}
main()
{
clrscr();
complex a(-2, 5);
complex b(3, 4);
cout<<”hai số phức:”<<endl;
a.display();
b.display();
if(a!=b) cout<<”hai số phức khác nhau”<<endl;
 else cout<<”hai số phức đồng nhất”<<endl;
cout<<”đảo dấu số phức a:”<<endl;
complex c = -a;
c.display();
cout<<”đảo dấu số phức b:”<<endl;
c = -b;
c.display();
cout<<”tổng hai số phức:”<<endl;
c = a + b;
c.display();
```

```
cout<<"hiệu hai số phức:"<<endl;
c = a - b;
c.display();
cout<<"tích hai số phức:"<<endl;
c = a * b;
c.display();
cout<<"thương hai số phức:"<<endl;
c = a / b;
c.display();
getch();
}
```

Chỉ thị c = a + b trong ví dụ trên được trình dịch hiểu là c = a.operator+ (b), các phép toán !=,-, *, / cũng tương tự như trên. Ta có nhận xét rằng a đóng vai trò tham số ngầm định (đối tượng ẩn) của hàm thành phần và như vậy thì tham số thứ nhất không thể là một đối tượng có kiểu nào khác ngoài kiểu complex, còn b là tham số tường minh thứ hai kiểu bất kỳ. Còn cách viết với ký hiệu toán tử chỉ là quy ước của trình dịch cho phép người dùng viết gọn lại, nhờ đó thấy tự nhiên hơn.

4.3 Hàm toán tử là hàm bạn

Hàm toán tử là hàm bạn thì số tham số hình thức trong trường hợp này bằng với số ngôi của ký hiệu toán tử, cách định nghĩa hàm toán tử tự do khiến cho người dùng thấy dễ hình dung hơn. Ví dụ trên được thay đổi ở khai báo một số tên tiêu đề hàm bạn còn nội dung định nghĩa hầu như không thay đổi.

```
class complex
{
float real, img;
public:
complex(float r = 0, float i = 0)
{
 real = r; img = i;
}
void display();
complex operator-();
// thêm từ khóa friend trước tên hàm toán tử hai ngôi, có đủ hai đối số a và b
friend int operator!=(complex a, complex b);
friend complex operator+(complex a, complex b);
friend complex operator-( complex a, complex b);
friend complex operator*( complex a, complex b);
friend complex operator/( complex a, complex b);
};
int operator!=(complex a, complex b)
{
if(a.real!=b.real || a.img!=b.img) return 1;
  else return 0;
}
complex operator+( complex a, complex b)
```

```
{
 complex c;
 c.real = a.real + b.real;
 c.img = a.img + b.img;
 return c;
}
complex operator-( complex a, complex b)
{
 complex c;
 c.real = a.real - b.real;
 c.img = a.img - b.img;
 return c;
}
complex operator*( complex a , complex b)
{
 complex c;
 c.real = a.real*b.real – a.img*b.img;
 c.img = a.real*b.img + a.img*b.real;
 return c;
}
complex operator/( complex a, complex b)
{
 complex c;
 c.real = (a.real*b.real + a.img*b.img)/(b.real*b.real + b.img*b.img);
 c.img = (-a.real*b.img + a.img*b.real)/(b.real*b.real + b.img*b.img);
 return c;
}
```

Mặt khác trong hàm bạn có các tham số có thể nhận kiểu khá tuỳ ý chứ không nhất thiết là kiểu lớp nào đó. Trong ví dụ này, hai đối số sẽ là một giá trị thực hoặc phức. Những phép tính định nghĩa chồng tự động chuyển kiểu về lớp complex: phần thực bằng giá trị thực còn phần ảo bằng không.

```
main()
{
clrscr();
complex a(-2, 5);
complex b(3, 4);
complex c;
cout<<”tổng số thực với số phức:”<<endl;
c = 3 + a*b ;
c.display();
cout<<”hiệu hai số thực:”<<endl;
c = 3 – 5;
c.display();
getch();
}
```

4.4 Chiến lược sử dụng hàm toán tử

Về nguyên tắc việc định nghĩa chồng hiệu quả và khá đơn giản, nhưng nếu lạm dụng hoặc sử dụng không đúng mục đích thì chương trình sẽ khó hiểu, dưới đây là một số khả năng và giới hạn của nó:

- Toán tử hai ngôi: +, - , * , / , = , == , != , < , > , << , >> , || , &&… Hai toán hạng tham gia phép toán trong hàm toán tử hai ngôi không nhất thiết cùng kiểu mà chỉ cần có ít nhất một trong hai đối số tương ứng là đối tượng là đủ. Hai toán tử **<<** và **>>** dùng với **cout**, **cin** phải được định nghĩa như hàm bạn.
- Toán tử một ngôi: *, & , ~ , ! , ++, --… Các hàm toán tử một ngôi chỉ có một đối số và phải trả về giá trị cùng kiểu với toán hạng.
- Toán tử gán: = , += , -= , *= , /= , |=… Các hàm định nghĩa chồng toán tử gán luôn được định nghĩa dưới dạng hàm thành phần của lớp, nên chỉ có một đối số tường minh và không ràng buộc gì về kiểu đối số cũng như trị trả về của phép gán.

Phần lớn các toán tử đều được định nghĩa chồng, nhưng vẫn có một số ràng buộc nhất định:

- Toán tử ".", "::" , "? :"… không được định nghĩa chồng, đương nhiên càng không được định nghĩa chồng 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.

Mặt khác định nghĩa chồng toán tử phải bảo toàn số ngôi của chính toán tử đó theo cách hiểu thông thường. Ví dụ như toán tử – có thể phù hợp cho phép toán đảo dấu một ngôi hoặc phép trừ số học hai ngôi nhưng không thể định nghĩa chồng toán tử gán "=" như là phép toán một ngôi hoặc toán tử tăng "++" như là phép toán hai ngôi.

4.5 Một số ví dụ tiêu biểu

4.5.1 Định nghĩa chồng toán tử << và >>

Chương II đã đề cập tới việc dùng toán tử vào ra **<<**, **>>** gắn liền với đối tượng **cout**, **cin** để nhập liệu dữ liệu cơ bản. Không chỉ vậy, C++ còn cung cấp khả năng định nghĩa chồng toán tử vào ra với những kiểu dữ liệu của người dùng. Chương trình sau đưa ra một cách định nghĩa chồng hai toán tử này cho nhập liệu và hiển thị một số phức:

```
#include<iostream>
#include<conio.h>
#include<math.h>
class complex
{
float real, img;
friend ostream & operator <<(ostream &os, complex &b);
friend istream & operator >>(istream &is, complex &b);
};
ostream & operator <<(ostream &os, complex &b)
{
 os<<b.real<<(b.img>=0 ? '+':'-')<<"j*"<<fabs(b.img)<<endl;
 return os;
```

```
}
istream & operator >>(istream &is, complex &b)
{
 cout<<"phần thực: "; is>>b.real;
 cout<<"phần ảo: "; is>>b.img;
 return is;
}
main()
{
 clrscr();
 cout<<"nhập số phức a: "<<endl;
 complex a; cin>>a;
 cout<<"nhập số phức b: "<<endl;
 complex b; cin>>b;
 cout<<"hiển thị các số phức: "<<endl;
 cout<<"a= "<<a;
 cout<<"b= "<<b;
 getch();
}
```

4.5.2 Định nghĩa chồng toán tử new và delete

Các toán tử **new** và **delete** được định nghĩa cho lớp nào thì chúng chỉ có ý nghĩa đối với lớp đó thôi. Với những thành phần khác vẫn dùng toán tử **new** và **delete** theo cách thông thường.

Định nghĩa chồng toán tử **new** phải sử dụng hàm thành phần và đáp ứng những quy tắc sau:

- 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 tham số này chỉ là đối số giả vì nó không được mô tả khi gọi tới toán tử **new**, mà do trình dịch tự động tính toán dựa trên kích thước của đối tượng liên đới.
- Trả về kiểu trỏ void* ứng với địa chỉ vùng nhớ động đã cấp phát.

Định nghĩa chồng toán tử **delete** cũng phải dùng hàm thành phần và đáp ứng những quy tắc sau:

- 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.
- Trả về kiểu void.

Sau đây là ví dụ định nghĩa chồng toán tử **new** và **delete** của người dùng trên lớp point. Trong các hàm này cũng có lời gọi tới các toán tử **new** và **delete** truyền thống.

```
#include<iostream>
#include<stddef.h>
#include<conio.h>
class point
{
 static int npt;
```

```
 static int npt_dyn;
 int x,y;
 public:
 point(int ox = 0, int oy = 0)
 {
  x = ox; y = oy;
  npt++;
  cout<<"++ tổng số điểm: "<<npt<<endl;
 }
 ~point()
 {
  npt--;
  cout<<"-- tổng số điểm: "<<npt<<endl;
 }
 void* operator new(size_t  sz)              // định nghĩa chồng hàm toán tử new
 {
  npt_dyn++;
  cout<<"có "<<npt_dyn<< " điểm động"<<endl;
  return (void*)::new char[sz];
 }
 void operator delete(void* dp)              // định nghĩa chồng hàm toán tử delete
 {
  npt_dyn--;
  cout<<"có "<<npt_dyn<<" điểm động"<<endl;
  ::delete dp;
 }
};
int point::npt = 0;
int point::npt_dyn = 0;
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;
}
```

Ta có nhận xét là các toán tử chuẩn **new** và **delete** được gọi (ngay cả khi chúng đã được định nghĩa chồng) thông qua toán tử phạm vi "**::**". Trong lớp có định nghĩa các hàm khởi tạo và dọn dẹp nên mỗi khi khai báo các đối tượng tĩnh hoặc động thì các hàm này tự động hiển thị các thông báo.

4.5.3 Hàm khởi tạo dùng là hàm toán tử

Chương trình sau đây minh họa khả năng dùng hàm khởi tạo complex(point) để biến một đối tượng kiểu point thành đối tượng kiểu complex:

```
#include<iostream>
#include<conio.h>
#include<math.h>
class point;
class complex
{
float real, img;
public:
complex(float r = 0, float i = 0)
{
real = r; img = i;
}
complex(point);
void display()
{
 cout<<real<<(img>=0 ? '+':'-')<<"j*"<<fabs(img)<<endl;
}
};
class point
{
int x, y;
public:
point(int ox = 0, int oy = 0)
{
x = ox; y = oy;
}
friend complex::complex(point);          // hàm bạn khởi tạo dùng làm toán tử
};
complex::complex(point p)
{
real = (float)p.x; img = (float)p.y;
}
main()
{
clrscr();
point a(3, -5);
complex c = a;
c.display();
getch();
}
```

Chương V Khuôn hình

5.1 Khuôn hình hàm

5.1.1 Định nghĩa khuôn hình hàm

Định nghĩa chồng hàm cho phép sử dụng một tên duy nhất cho nhiều hàm cùng thực hiện những công việc tương tự nhau. Mặc dù vậy những quy ước bắt buộc của định nghĩa chồng khiến cho nó vẫn có hạn chế, ví dụ như ta cần phải biết rõ kiểu dữ liệu cụ thể trước khi định nghĩa và đối với những hàm khác nhau cũng phải viết lại nội dung mặc dù chúng có thể giống hệt nhau. Chương trình dịch của C++ từ phiên bản 3.0 hoặc cao hơn đã cho phép giải quyết đơn giản vấn đề trên bằng cách định nghĩa một khuôn hình duy nhất theo cách sau đây:

```
template <class T> [trị trả về] [tên hàm]([danh sách tham số])
{
 // định nghĩa nội dung hàm
}
```

template <class T> xác định rằng đó là một khuôn hình hàm với tham số kiểu T trừu tượng nào đó, tiếp theo là [trị trả về] [tên hàm] ([danh sách tham số]) thể hiện khai báo trị trả về, tên hàm và danh sách tham số, cuối cùng là định nghĩa nội dung khuôn hình hàm. Cách định nghĩa này như là sự mở rộng của định nghĩa chồng hàm thông thường.

5.1.2 Sử dụng khuôn hình hàm với dữ liệu đơn giản

Ta sẽ đưa ra một ví dụ cách sử dụng khuôn hình hàm, thứ nhất là cho dữ liệu cơ sở:

```
#include<iostream>
#include<conio.h>
template<class T> T min(T a, T b)
{
 if(a<b) return a;
  else return b;
}
main()
{
 clrscr();
 int n = 4, p = -12;
 float x = 2.5, y = 3.25;
 cout<<"min(n, p) =  "<<min(n,p)<<endl;
 cout<<"min(x, y) =  "<<min(x,y)<<endl;
 getch();
}
```

Gọi hàm min() có hai tham số với cùng kiểu dữ liệu, với lời gọi đầu tiên hàm min(n, p) thì trình dịch tự động gọi hàm min() với hai biến kiểu int, còn lời gọi thứ hai là hai biến kiểu float và cứ thế. Dưới đây là khuôn hình hàm min cho kiểu char*, chỉ phải thay đổi chút ít trong hàm chính:

```
main()
{
 clrscr();
 char *adr1 = "DHHH";
```

```
 char *adr2 = “DHGT”;
 cout<<”min(adr1, adr2) = ”<<min(adr1, adr2)<<endl;
 getch();
}
```

Thực chất thì trình dịch sẽ sinh hàm thể hiện so sánh địa chỉ các biến trỏ ký tự chứ không so sánh nội dung hai xâu.

5.1.3 Sử dụng khuôn hình hàm với dữ liệu lớp

Để áp dụng khuôn hình hàm min() cho kiểu dữ liệu lớp vector, cần phải định nghĩa chồng toán tử “<” so sánh đối tượng của lớp. hàm này sẽ là hàm bạn có tên là *operator*<. Nếu như gọi hàm mà chưa định nghĩa toán tử “<” thì trình dịch sẽ thông báo lỗi hàm chưa được định nghĩa trước, sau đây là ví dụ minh họa:

```
#include<iostream>
#include<conio.h>
template<class T> T min(T a, T b)
{
 if(a<b) return a;
  else return b;
}
class vect
{
 int x, y;
 public:
 vect(int abs = 0, int ord = 0)
 {
 x = abs; y = ord;
 }
 void display();
 friend int operator<(vect, vect);
};
void display()
{
 cout<<x<<” “<<y<<endl;
}
friend int operator<(vect, vect)
{
 return a.x*a.x + a.y*a.y < b.x*b.x + b.y*b.y;
}
main()
{
 clrscr();
 vect u(3, 2), v(4, -1);
 cout<<”min(u, v) =  “;
```

```
 min(u, v).display();
 getch();
}
```

5.1.4 Các tham số kiểu

Một cách tổng quát, khuôn hình hàm có thể có một hoặc nhiều tham số kiểu, với mỗi tham số có từ khóa **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 đặt ở bất kỳ đâu trong định nghĩa khuôn hình hàm, nghĩa là có thể đặt tại dòng tiêu đề, hay tại vị trí khai báo biến cục bộ hoặc trong các chỉ thị thực hiện. Trong mọi trường hợp tham số kiểu phải xuất hiện ít nhất một lần trong khai báo danh sách các tham số hình thức, nhờ đó mà trình dịch có thể sinh hàm thể hiện duy nhất. Ta xem chương trình sau đây:

```
#include<iostream>
#include<conio.h>
template<class T, class U> T sum(T x, U y, T z)
{
 return x+y+z;
}
main()
{
 clrscr();
 int n = 1, p = 2, q = 3;
 float x = 2.5, y = 5.0;
 cout<<sum(n, x, p)<< endl;
 cout<<sum(x, n, y)<< endl;
 cout<<sum(n, p, q)<< endl;
 // cout<<sum(n, p, x)<< endl;  Sai
 getch();
}
```

Những lỗi như trên là vì C++ đòi hỏi tuân theo những nguyên tắc của nó: phải có sự tương ứng chính xác giữa kiểu của tham số hình thức và kiểu tham số thực sự được truyền cho hàm, khai báo hàm kiểu T và U là khác nhau trong khi n và p cùng kiểu nên gây ra lỗi cú pháp. Thậm chí C++ không đồng ý chuyển kiểu thông thường dạng T sang const T hay T[ ] sang T*, những trường hợp hoàn toàn có thể trong định nghĩa chồng hàm.

5.1.5 Các tham số biểu thức

Ngoài những tham số kiểu thì khuôn hình hàm cũng cho phép khai báo tham số hình thức kiểu xác định. Ta gọi chúng là những tham số biểu thức, chương trình sau đây định nghĩa một khuôn hình hàm cho phép đếm số lượng các phần tử 0 trong mảng tĩnh:

```
#include<iostream>
#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;
}
main()
{
 clrscr();
 int t[5] = {5, 2, 0, 2, 0};
 char c[6] = {0, 1, 2, 0, 0, 0};
 cout<<"compte(t) =  "<<compte(t, 5)<<endl;
 cout<<"compte(c) =  "<<compte(c, 6)<<endl;
 getch();
}
```

5.1.6 Định nghĩa chồng khuôn hình hàm

Giống như việc định nghĩa chồng các hàm thông thường, C++ cho phép định nghĩa các khuôn hình hàm có cùng tên nhưng tham số khác nhau, từ đó dẫn đến có nhiều họ các hàm (mỗi khuôn hình tương đương một họ các hàm). Ví dụ có ba họ hàm cùng tên min(), họ thứ nhất bao gồm các hàm tìm giá trị nhỏ nhất trong hai giá trị, họ thứ hai tìm giá trị nhỏ nhất trong ba giá trị, còn họ thứ ba tìm giá trị nhỏ nhất trong mảng:

```
#include<iostream>
#include<conio.h>
template<class T> T min(T a, T b)
{
 if(a<b) return a;
  else return b;
}
template<class T> T min(T a, T b, T c)
{
 return min(min(a, b), c);
}
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;
}
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, 2, 1};
 char c[4] = {'w', 'q', 'a', 'Q'};
```

```
 cout<<"min(n, p) = "<<min(n, p)<<endl;
 cout<<"min(n, p, q) = "<<min(n, p, q)<<endl;
 cout<<"min(x, y) = "<<min(x, y)<<endl;
 cout<<"min(x, y, z) = "<<min(x, y, z)<<endl;
 cout<<"min(t, 6) = "<<min(t, 6)<<endl;
 cout<<"min(c, 4) = "<<min(c, 4)<<endl;
 getch();
}
```

5.1.7 Cụ thể hóa hàm thể hiện

Các khuôn hình hàm dựa trên một định nghĩa chung suy ra chúng thực hiện theo cùng thực hiện một giải thuật. Đôi khi sự tổng quát đó không linh hoạt như trong ví dụ đầu chương, bằng cách cụ thể hóa dữ liệu cho tham số và thay đổi nội dung hàm ta có thể khắc phục được nhược điểm này. Chương trình con sau đây vẫn giữ định nghĩa khuôn hình so sánh hai giá trị nhưng thêm vào một hàm cùng tên tìm giá trị nhỏ nhất giữa hai xâu ký tự. Hình thức này giống như sự pha trộn giữa định nghĩa chồng hàm thông thường và khuôn hình hàm:

```
#include<iostream>
#include<conio.h>
#include<string.h>
template<class T> T min(T a, T b)
{
 if(a<b) return a;
  else return b;
}
char * min(char *cha, char *chb)
{
 if(strcmp(cha, chb) < 0) return cha;
  else return chb;
}
main()
{
 clrscr();
 int n = 12, p = 15;
 char *adr1 = "DHHH", *adr2 = "DHGT";
 cout<<"min(n, p) = "<<min(n, p)<<endl;
 cout<<"min(adr1, adr2) = "<<min(adr1, adr2)<<endl;
 getch();
}
```

5.1.8 Các hạn chế của khuôn hình hàm

Về nguyên tắc khi định nghĩa một khuôn hình hàm, một tham số kiểu có thể tương đương với bất kỳ dữ liệu nào, cho dù đó là dữ liệu chuẩn hay của người dùng tự định nghĩa. Do vậy nếu một khuôn hình có dòng đầu tiên là: template <class T> void fct(T) {...} thì hàm fct() được gọi với tham số là int, float, char, int*, int**... tuy nhiên định nghĩa bên trong khuôn

hình hàm chứa một số yếu tố dẫn đến quá trình sinh hàm thể hiện không chính xác. Có những trường hợp nảy sinh vì số kiểu dữ liệu có thể rất nhiều.

Ví dụ 1: Với dòng tiêu đề: template <class T> void fct(T*)
Ta cho rằng tham số T* tương đương một con trỏ bất kỳ nhưng fct() chỉ ứng với một con trỏ kiểu: int*, int**, t*, t**.

Ví dụ 2: Với định nghĩa khuôn hình hàm

```
template<class T> T min(T a, T b)
{
 if(a<b) return a;
  else return b;
}
```

So sánh hai biến a và b bằng toán tử "<" chỉ đúng nếu chúng là dữ liệu cơ sở còn đối với dữ liệu phức tạp khác như lớp thì người dùng phải tự định nghĩa thêm.

Ví dụ 3: Với các định nghĩa chồng khuôn hình hàm có thể gây ra sự không rõ ràng để sinh hàm thể hiện.

```
template<class T> T min(T, T)   {...}
template<class T> T min(T*, T)  {...}
template<class T> T min(T, T*)  {...}
template<class T> T min(T*, T*) {...}
```

Xét những câu lệnh sau:

```
int x, y;
```

lời gọi fct(&x, &y) đúng với cả hai khuôn hình đầu tiên hoặc cuối cùng, suy ra trình dịch bắt lỗi cú pháp, do vậy phải sửa đổi thêm trong phần tham số hình thức để phân biệt.

5.2 Khuôn hình lớp

5.2.1 Định nghĩa khuôn hình lớp

Bên cạnh khái niệm khuôn hình lớp, 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 áp dụng với nhiều lớp thể hiện khác nhau.

Lớp point trong các ví dụ trước có thành phần tọa độ x, y kiểu int. Nếu muốn tọa độ điểm có kiểu khác thì phải định nghĩa lớp khác. Để tránh lặp lại thì C++ định nghĩa một khuôn hình lớp ứng với nhiều lớp thể hiện cho các kiểu dữ liệu khác nhau:

```
template <class T> class point
{
 T x; T y;
 public:
 point( T abs = 0, T ord = 0);
 ...
};
```

Cũng như khuôn hình hàm, C++ sử dụng từ khóa **class** mô tả T đại diện cho một kiểu dữ liệu lớp. Mặt khác nếu hàm thành phần định nghĩa như các hàm **inline** thì không có thay đổi nhưng cách trình bày các hàm thành phần của khuôn hình lớp có khác biệt nếu định nghĩa hàm bên ngoài. Dòng tiêu đề đầy đủ cho hàm thành phần của khuôn hình lớp lúc này là:

template <class T> [trị trả về] [tên lớp]<T>::[tên hàm]([danh sách tham số])

Lớp point được viết lại thành:

```
#include<iostream>
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<<"tọa độ: "<<x<<","<<y<<endl;
}
```

5.2.2 Sử dụng khuôn hình lớp

Khuôn hình lớp point đã khai báo ta có thể sử dụng chúng, theo như cách chỉ định tổng quát thì các lệnh dạng:

```
point<int> ai;
point<double> ad;
```

ai và ad là đối tượng point có thành phần int hay double nhận giá trị ngầm định. Trong trường hợp truyền giá trị cho tham số cũng đơn giản, ví dụ:

```
point<int> ai(3, 5);
point<double> ad(2.5, -4.5);
```

Lớp point được sửa lại thành:

```
#include<iostream>
#include<conio.h>
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<<"tọa độ: "<<x<<","<<y<<endl;
}
```

```
main()
{
 clrscr();
 point<int> ai(3, 5);
 ai.display();
 point<char> ac('d', 'b');
 ac.display();
 point<double> ad(2.5, -4.5);
 ad.display();
 getch();
}
```

5.2.3 Các tham số trong khuôn hình lớp

Hoàn toàn giống như khuôn hình hàm, các khuôn hình lớp cũng có các tham số kiểu và tham số biểu thức. Nhìn chung có nhiều điểm giống nhau giữa khuôn hình hàm và khuôn hình lớp nhưng các ràng buộc về kiểu tham số có khác.

- Số lượng tham số kiểu có thể tuỳ ý trong khuôn hình lớp:

```
template <class T, class U, class V>  // danh sách ba tham số kiểu
class try
{
 T x;
 U t[5];
 V test(int, U);
 ...
};
```

- 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 với số lượng bằng số các tham số trong danh sách template<...>, tất nhiên các kiểu tham số này phải xác định trước. Cụ thể là với ba tham số kiểu T, U, V thì lớp thể hiện của try cũng phải có ba tham số:

```
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*, point<float>, int>;  // lớp thể hiện với ba tham số char*, point<float>, int
```

Vấn đề tương đương chính xác của khuôn hình hàm không còn hiệu lực trong khuôn hình lớp, nguyên nhân vì khuôn hình hàm còn căn cứ theo danh sách các tham số hình thức trong tiêu đề của hàm.

- Tham số biểu thức phải là hằng số:

Một khuôn hình lớp có thể chứa các tham số biểu thức nhưng chúng phải là hằng số. Giả sử có lớp có tên là table để thao tác trên các bảng chứa đối tượng bất kỳ, một cách tự nhiên ta nên có hai tham số, tham số kiểu thứ nhất gắn với từ khóa **class** chỉ đối tượng bất kỳ còn tham số thứ hai kiểu int là tham số biểu thức. Định nghĩa của lớp table như sau:

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

```
template <class T, int n>
class  table
{
 T tab[n];
 public:
 table()
 {
  cout<<"tạo bảng"<<endl;
 }
 T & operator[] (int i)
 {
  return tab[i];
 }
};
class point
{
 int x, y;
 public:
 point(int abs = 1, int ord = 1)
 {
  cout<<"hàm thành phần khởi tạo"<<endl;
  x = abs; y = ord;
 }
 void display()
 {
  cout<<"tọa độ: "<<x<<","<<y<<endl;
 }
};
main()
{
 clrscr();
 table<int, 4> ti;
 for(int i=0; i<4; i++)  ti[i] = i*i;
 cout<<"ti: ";
 for(i=0; i<4; i++)  cout<<ti[i]<<" ";
 cout<<endl;
 table<point, 3> tp;
 for(i=0; i<3; i++) tp[i].display();
 getch();
}
```

5.2.4 Cụ thể hóa khuôn hình lớp

Khả năng cụ thể hóa khuôn hình lớp có đôi chút sai khác 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. Các hàm thành phần cùng tên sẽ được thực hiện theo cùng một giải thuật. Nếu ta muốn một hàm thành phần thích ứng cho tình

huống cụ thể nào đó thì phải viết riêng định nghĩa khác cho nó. Ta thêm vào khuôn hình lớp point một cụ thể hóa hàm display() cho kiểu dữ liệu char. Vì khai báo cho đối tượng ac('d', 'b') nên lời gọi hàm ac.display() sẽ tương ứng với hàm cụ thể hóa chứ không là hàm thành phần khuôn hình lớp.

```
#include<iostream>
#include<conio.h>
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<<"tọa độ: "<<x<<","<<y<<endl;
}
void point<char>::display()
{
 cout<<"tọa độ: "<<(int)x<<","<<(int)y<<endl;
}
main()
{
 clrscr();
 point<int> ai(3, 5);
 ai.display();
 point<char> ac('d', 'b');
 ac.display();
 point<double> ad(2.5, -4.5);
 ad.display();
 getch();
}
```

Có thể cụ thể hóa giá trị các tham số theo quy tắc trong phần 5.2.3 và hàm thành phần hay một lớp. Trong ví dụ trên hàm thành phần được cụ thể hóa bằng cách đưa thêm định nghĩa vào khuôn hình lớp point, tuy nhiên cũng có thể định nghĩa chi tiết cho thể hiện point<char>. Ta khai báo như sau:

```
class point<char>
{
 // định nghĩa cho hàm display() như trên
};
```

Chương VI THừa kế

6.1 Giới thiệu chung

Thừa kế là một trong những nguyên tắc nền tảng của kỹ thuật lập trình hướng đối tượng. Kỹ thuật này là cơ sở cho việc nâng cao khả năng sử dụng lại các bộ phận của chương trình, từ một lớp ban đầu (lớp cơ sở) ta định nghĩa lớp mới (lớp dẫn xuất), lớp mới có một vài hoặc toàn bộ các thành phần lớp cũ nhưng thêm vào những thành phần mới. Nói cách khác, những thành phần của lớp mới loại bỏ những việc không còn phù hợp của lớp cơ sở, hoàn thiện những việc làm chưa tốt, cũng như bổ sung công việc mới.

Về khía cạnh cài đặt, các thành phần chương trình lớp cơ sở không phải biên dịch lại nên nâng cao tốc độ cho chương trình. Ngôn ngữ lập trình Borland Pascal, C có thư viện Turbo Vision, nơi cung cấp đối tượng lớp cơ sở, những lớp này thuộc các tệp tin *.obj hoặc *.lib, để xây dựng giao diện ứng dụng thân thiện và người dùng sử dụng chúng dễ dàng mà không cần biết xử lý chi tiết bên trong.

Thừa kế cũng cho phép một lớp có thể dẫn xuất từ cùng một lớp hoặc nhiều lớp khác nhau sẵn có, rồi đến lượt lớp này lại là cơ sở cho các lớp tiếp theo và cứ thế. Tính đa hình (còn gọi là tính tương ứng bội) là một trong những tính chất quan trọng được thiết lập trên cơ sở thừa kế mà tại đó đối tượng có thể có biểu hiện cụ thể dựa vào tình huống cụ thể. Ví dụ, các đối tượng hình học như tam giác vuông, cân, đều kế thừa từ lớp tam giác, chúng có những tính chất riêng tư như có phương thức tính diện tích độc lập nhưng đều có phương thức diện tích chung của lớp tam giác. Làm thế nào phân biệt đối tượng cụ thể để gọi đúng phương thức tương ứng của nó.

Các ngôn ngữ lập trình hướng đối tượng đều cho phép đa thừa kế, vì kỹ thuật kế thừa gồm nhiều mức dẫn đến một lớp có thể là được kế thừa lặp lại trong một lớp khác, ta gọi đó là sự xung đột thừa kế. Mỗi ngôn ngữ có giải pháp của riêng mình, C++ đưa ra khái niệm thừa kế ảo.

6.2 Đơn thừa kế

Mục đích của đơn thừa kế nhằm đáp ứng các yêu cầu sau:

- Sử dụng các thành phần cơ sở từ lớp dẫn xuất.
- Định nghĩa lại các thành phần cùng tên của lớp cơ sở trong lớp dẫn xuất.
- Truyền thông tin các hàm khởi tạo.

Giả sử đã có lớp cơ sở point, dạng khai báo một lớp dẫn xuất colorpoint dựa trên lớp cơ sở như sau:

```
class point
{
 //  khai báo, định nghĩa
};
class colorpoint : <nhãn thuộc tính truy xuất> point
{
 //  khai báo, định nghĩa
};
```

Nhãn thuộc tính truy xuất là **public**, **private**, **protected** quy định kiểu dẫn xuất, trong thực tế chỉ dùng dẫn xuất **public** và **protected** là phổ biến:

- Đối với dẫn xuất **public**, các thành phần, hàm bạn và các đối tượng lớp dẫn xuất không thể truy nhập tới các thành phần gán nhãn **private** của lớp cơ sở. Các thành phần gán nhãn khác của lớp cơ sở có nhãn không đổi và được truy xuất trong lớp dẫn xuất.
- Loại dẫn xuất **private** ít được sử dụng vì nó không cho phép truy nhập các thành phần từ lớp cơ sở, nhãn truy xuất thuộc tính các thành phần trong lớp cơ sở trở thành nhãn **private** trong lớp dẫn xuất.
- Cuối cùng là loại dẫn xuất **protected**, như đã nói bên trên rằng các thành phần có nhãn **protected** không truy xuất được bên ngoài lớp nếu không có chỉ dẫn gì thêm. Hơn thế nữa, các thành phần gán nhãn **public** và **protected** trong lớp cơ sở sẽ có nhãn **protected** và được truy xuất trong lớp dẫn xuất, trái lại các thành phần gán nhãn **private** sẽ không thay đổi nhãn nên không được truy xuất trong lớp dẫn xuất.

6.2.1 Ví dụ minh họa

Chương trình đơn giản dưới đây cho thấy cách làm việc với đơn thừa kế, lớp cơ sở xử lý dữ liệu điểm còn lớp dẫn xuất public bổ sung thêm dữ liệu màu của điểm đó. Hai lớp có các hàm khởi tạo và hàm thiết lập sao chép và có sự truyền thông tin từ lớp dẫn xuất cho lớp cơ sở, đồng thời lớp dẫn xuất có hàm thành phần display() sử dụng được các thành phần của lớp cơ sở.

```
#include<iostream>
#include<conio.h>
class point
{
int x, y;
public:
point(int ox = 0, int oy = 0)
{
 x = ox; y = oy;
}
point(point &p)
{
 x = p.x; y = p.y;
}
void move(int dx, int dy);
void display();
};
void point::move(int dx, int dy)
{
 x += dx; y += dy;
}
void point::display()
{
 cout<<”gọi hàm point::display()”<<endl;
 cout<<”tọa độ: “<<x<<”,”<<y<<endl;
}
class colorpoint : public point
{
```

```
 unsigned  color;
 public:
 colorpoint(int ox = 0, int oy = 0, unsigned c = 0) : point(ox, oy)
 {
  color = c;
 }
 colorpoint(colorpoint &p) : point((point &)p)
 {
  color = p.color;
 }
 void display();
};
void colorpoint::display()
{
 cout<<"gọi hàm colorpoint::display()"<<endl;
 point::display();
 cout<<"màu: "<<color<<endl;
}
main()
{
clrscr();
colorpoint n(2, 3, 6);
cout<<"tọa độ của điểm n:"<<endl;
n.point::display();
colorpoint p = n;
p.point::move(1, -5);
cout<<"điểm p:"<<endl;
p.display();
cout<<"chỉ hiển thị tọa độ của điểm p:"<<endl;
p.point::display();
getch();
}
```

6.2.2 Định nghĩa lại thành phần của lớp cơ sở trong lớp dẫn xuất

Định nghĩa lại hàm thành phần khác với định nghĩa chồng hàm thành phần, khái niệm này chỉ có nghĩa với kỹ thuật thừa kế. Hàm định nghĩa lại và bị định nghĩa lại ở đây là hàm display(), chúng có khai báo dòng tiêu đề như nhau nhưng có vị trí khác nhau. Định nghĩa lại hàm thành phần còn là cơ sở cho cho cài đặt tính đa hình.

Như vậy lớp colorpoint có hai phiên bản khác nhau của display() cùng tồn tại, việc truy nhập tới hàm display() của lớp dẫn xuất thông qua đối tượng colorpoint, trong khi hàm display() của lớp cơ sở muốn được gọi phải chỉ định tên của nó là point::display(). Trong định nghĩa hàm colorpoint, nếu viết tắt display() sẽ dẫn đến lời gọi đệ quy lặp vô hạn. Tương tự như thế, C++ cũng cho phép khai báo bên trong lớp dẫn xuất các thành phần cùng tên với các thành phần dữ liệu đã có trong lớp cơ sở, chấp nhận cùng hoặc khác kiểu.

6.2.3 Tính thừa kế trong lớp dẫn xuất

- Tương thích giữa đối tượng lớp dẫn xuất với đối tượng lớp cơ sở: một đối tượng lớp dẫn xuất có thể "thay thế" đối tượng lớp cơ sở, vì tất cả các thành phần lớp cơ sở đều có thể tìm thấy trong lớp dẫn xuất. Như ví dụ trên, đối tượng p xử lý thành phần tọa độ x, y thông qua các hàm point::display(), point::move(). Sự tương thích còn thể hiện ở chỗ có sự chuyển kiểu ngầm định từ đối tượng lớp dẫn xuất sang lớp cơ sở trong khi chiều ngược lại không đúng.

- Tương thích giữa con trỏ lớp dẫn xuất với con trỏ lớp cơ sở: Căn cứ theo thuộc tính trên, con trỏ đối tượng lớp cơ sở xác định được địa chỉ đối tượng lớp dẫn xuất. Có sửa đổi chút ít trong hàm chính như sau:

```
main()
{
 clrscr();
 point *adp;
 colorpoint p(2, 3, 6);
 cout<<"điểm p:"<<endl;
 p.display();
 cout<<"adp = &p"<<endl;
 adp = &p;
 adp->move(1, -5);
 adp->display();
 getch();
}
```

Kết quả thực hiện lệnh adp->display():

```
gọi hàm point::display()
tọa độ: 3,-2
```

Nói cách khác, mặc dù biến trỏ adp chứa địa chỉ của đối tượng p nhưng lệnh trên chỉ thi hành chỉ thị point::display(). Nguyên nhân là do các hàm trong point khai báo thông thường, con trỏ adp đã được liên kết sớm với lớp point mà không phụ thuộc vào đối tượng lớp cụ thể trong tiến trình biên dịch. Liên kết sớm được sử dụng rất hiệu quả nhưng trong thực tế liên kết muộn được sử dụng phổ biến vì nó linh hoạt hơn. Phần sau ta sẽ đề cập tới vấn đề này.

- Tương thích giữa tham chiếu lớp dẫn xuất với tham chiếu lớp cơ sở: Hoàn toàn giống như tương thích đối tượng.

6.2.4 Truyền thông tin giữa các hàm khởi tạo

Tiến trình truyền thông tin luôn gắn với các hàm khởi tạo, ngay cả các hàm khởi tạo sao chép, vì việc gọi hàm khởi tạo lớp dẫn xuất kéo theo khởi tạo hàm lớp cơ sở. Như vậy thứ tự thực hiện phải là đi từ cấp cơ sở tới cấp kế thừa, các thông tin ban đầu truyền vào cho lớp cơ sở, lớp dẫn xuất sẽ nhận thông tin còn lại. Cơ chế trên thực hiện một cách tự động mà người dùng không cần phải nhìn thấy. Giải pháp trên không chỉ C++ mà những đa số các ngôn ngữ khác đều chấp nhận là trong hàm khởi tạo lớp dẫn xuất ta mô tả lời gọi hàm khởi tạo lớp cơ sở, đương nhiên các tham số được truyền vào cho hàm khởi tạo lớp cơ sở được thay đổi tuỳ ý:

```
colorpoint(int ox, int oy, unsigned c) : point((ox + oy)/2, (ox - oy)/2)
{
```

```
 color = c;
}
```

6.3 Hàm ảo và tính đa hình

6.3.1 Ví dụ minh họa

Ta mới chỉ biết rằng một tham trỏ của đối tượng có thể nhận địa chỉ bất kỳ đối tượng lớp thừa kế. Tuy nhiên ưu điểm này có nhược điểm: hàm thành phần gọi từ con trỏ đối tượng được xác định ngay từ khi khai báo và ta gọi đó là liên kết sớm ("gán kiểu tĩnh"). Muốn gọi được đúng phương thức tương đương với đối tượng được trỏ đến, cần phải xác định kiểu của đối tượng được xem xét tại thời điểm thực hiện chương trình và ta gọi nó là liên kết muộn ("gán kiểu động"). Khái niệm hàm ảo được C++ đưa ra nhằm đáp ứng yêu cầu này. Ví dụ minh họa có sự thay đổi trong định nghĩa lớp point, dòng tiêu đề khai báo hàm thành phần point::display() có từ khóa **virtual** đặt trước tên hàm để chỉ định nó là hàm ảo.

```
class point
{
int x, y;
public:
point(int ox = 0, int oy = 0)
{
 x = ox; y = oy;
}
point(point &p)
{
 x = p.x; y = p.y;
}
void move(int dx, int dy);
virtual void display();
};
```

Kết quả thực hiện lệnh adp->display() bây giờ sẽ là:

```
gọi hàm colorpoint::display()
gọi hàm point::display()
tọa độ: 3,-2
màu: 6
```

Tính đa hình còn thể hiện nếu hàm thành phần trong lớp cơ sở được gọi từ một đối tượng lớp dẫn xuất, còn hàm lại gọi một hàm ảo, hàm ảo được định nghĩa đồng thời trong hai lớp cơ sở và dẫn xuất như đã biết. Ví dụ trên thì hàm ảo display() được tham trỏ đối tượng gọi trực tiếp nhưng ở đây hàm ảo identifier() được gọi thông qua hàm thành phần display():

```
#include<iostream>
#include<conio.h>
class point
{
int x, y;
public:
point(int ox = 0, int oy = 0)
{
```

```
 x = ox; y = oy;
}
point(point &p)
{
 x = p.x; y = p.y;
}
void move(int dx, int dy);
virtual void identifier();
void display();
};
void point::move(int dx, int dy)
{
 x += dx; y += dy;
}
void point::identifier()
{
 cout<<"điểm không màu"<<endl;
}
void point::display()
{
 cout<<"tọa độ: "<<x<<","<<y<<endl;
 identifier();
}
class colorpoint : public point
{
 unsigned  color;
 public:
 colorpoint(int ox = 0, int oy = 0, unsigned c = 0) : point(ox, oy)
 {
  color = c;
 }
 colorpoint(colorpoint &p) : point((point &)p)
 {
  color = p.color;
 }
 void identifier();
 void display();
};
void colorpoint::identifier()
{
 cout<<"màu: "<<color<<endl;
}
void colorpoint::display()
{
 cout<<"gọi hàm colorpoint::display()"<<endl;
 point::display();
```

```
 identifier();
}
```

Để nắm vững tính đa hình rõ nét hơn ta hãy xem xét một bài toán khác như sau: Giả sử rằng có một cái chuồng 20 ngăn nuôi các con vật là chó và mèo, ta phải xây dựng một chương trình C++ hợp lý để quản lý chuồng. Ban đầu ta định nghĩa một lớp cơ sở có tên là Animal, trên cơ sở đó các lớp Dog và Cat sẽ được dẫn xuất. Hơn nữa cả hai lớp dẫn xuất đều có một phương thức gọi là WhoAmI để hiển thị thông báo loại và tên của đối tượng cụ thể. Cuối cùng là lớp Kennel mô tả cái chuồng chứa hai loài vật. Vấn đề là đưa ra một cách tối ưu nào đó để phát hiện mỗi đối tượng cụ thể vì làm theo cách thông thường thì thời gian thực hiện tính toán cho việc vét cạn ngăn xếp sẽ dài gấp đôi:

```
#include<iostream>
#include<conio.h>
#include<string.h>
class Animal
{
 protected:
 char *name;
 public:
 Animal(char *str)
 {
  name = strdup(str);
 }
 ~Animal()
 {
  delete name;
 }
 virtual void WhoAmI()
 { }
};
class Cat : public Animal
{
 public:
 Cat(char *str) : Animal(str)
 { }
 void WhoAmI()
 {
  cout<<”tôi là con mèo, tên là ”<<name<<endl;
 }
};
class Dog : public Animal
{
 public:
 Dog(char *str) : Animal(str)
 { }
 void WhoAmI()
 {
```

```
 cout<<"tôi là con chó, tên là "<<name<<endl;
 }
};
class Kennel
{
private:
unsigned MaxAnimals;
unsigned NumAnimals;
Animal **Residents;
public:
Kennel(unsigned max);
~Kennel();
unsigned Accept(Animal *d);
Animal* Release(unsigned pen);
void ListAnimals();
};
Kennel::Kennel(unsigned max)
{
 MaxAnimals = max;
 NumAnimals = 0;
 Residents = new Animal* [MaxAnimals];
 for(unsigned i=0; i<MaxAnimals; i++) Residents[i] = NULL;
}
Kennel::~Kennel()
{
 delete Residents;
}
unsigned Kennel::Accept(Animal *d)
{
 if(NumAnimals == MaxAnimals) return 0;
 ++NumAnimals;
 int i = 0;
 while(Residents[i] != NULL) ++i;
 Residents[i] = d;
 return i+1;
}
Animal* Kennel::Release(unsigned pen)
{
 if(pen > MaxAnimals) return NULL;
 --pen;
 if(Residents[pen] != NULL)
 {
  Animal *tmp = Residents[pen];
  Residents[pen] = NULL;
  --NumAnimals;
 return tmp;
```

```
 }
 else return NULL;
}
void Kennel::ListAnimals()
{
 if(NumAnimals > 0)
 for(unsigned i=0; i< MaxAnimals; i++)
  if(Residents[i] != NULL) Residents[i]->WhoAmI();
}
main()
{
 clrscr();
 Dog d1("Rover");
 Dog d2("Spot");
 Dog d3("Chip");
 Dog d4("Buddy");
 Dog d5("Butch");
 Cat c1("Tinkerbell");
 Cat c2("Inky");
 Cat c3("Fluffy");
 Cat c4("Princess");
 Cat c5("Sylvester");
 Kennel K(20);
 K.Accept(&d1);
 unsigned c2pen = K.Accept(&c2);
 K.Accept(&d3);
 K.Accept(&c1);
 unsigned d4pen = K.Accept(&d4);
 K.Accept(&d5);
 K.Accept(&c5);
 K.ListAnimals();
 K.Release(c2pen);
 K.Release(d4pen);
 K.ListAnimals();
 getch();
}
```

6.3.2 Các thuộc tính về hàm ảo

- Không nhất thiết phải định nghĩa lại hàm ảo: Trong trường hợp tổng quát ta luôn định nghĩa lại đầy đủ trong lớp dẫn xuất các hàm đã khai báo **virtual** trong lớp cơ sở. Tuy nhiên trong một số trường hợp không nhất thiết phải thực hiện máy móc như thế, chẳng hạn nếu như hàm ảo display() đã có định nghĩa hoàn chỉnh trong lớp point thì không phải định nghĩa lại nó trong lớp colorpoint.

- Định nghĩa chồng hàm ảo: Có thể định nghĩa chồng hàm ảo nhưng nảy sinh vấn đề sau đây: hàm ảo đã định nghĩa trong lớp cơ sở và ta định nghĩa chồng nó trong lớp dẫn xuất với các tham số khác thì hàm này độc lập với hàm ảo hiện thời. Muốn có định nghĩa chồng hàm

ảo thì tất cả các hàm định nghĩa chồng của nó phải khai báo **virtual** trước chúng để trình dịch không bắt lỗi cú pháp.

- Hàm hủy bỏ ảo: Hàm khởi tạo không thể là hàm ảo, nhưng hàm dọn dẹp lại có thể. Chỉ cần khai báo virtual trước tên hàm huỷ bỏ trong lớp cơ sở, các hàm huỷ bỏ của lớp dẫn xuất cũng sẽ là ảo mà không cần cùng tên.

- Hàm ảo thuần tuý và lớp trừu tượng: Hàm ảo thuần tuý là hàm không có phần định nghĩa. Lớp có ít nhất một hàm ảo thuần tuý là lớp trừu tượng. Lớp này sử dụng để mô tả các lớp đối tượng chung nhất và các lớp kế thừa sẽ được chi tiết dần. Ta chỉ được khai báo biến con trỏ có kiểu lớp trừu tượng chứ không được khai báo biến đối tượng lớp. Lớp dẫn xuất phải chứa định nghĩa lại hàm hoặc khai báo lại hàm ảo này như một lớp trừu tượng.

Ví dụ khai báo lớp trừu tượng:

```
class obj                    // lớp trừu tượng
{
 public:
 virtual display() = 0; // hàm ảo thuần tuý không khai báo định nghĩa
};
```

6.4 Đa thừa kế

6.4.1 Đặt 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 hình thừa kế cũng đúng và được tổng quát cho trường hợp đa thừa kế nhưng có một số mục tiêu riêng phải giải đáp như sau:

- Làm thế nào biểu thị được tính độc lập của các thành phần cùng tên bên trong một lớp dẫn xuất.

- Các hàm khởi tạo và hủy bỏ được gọi như thế nào: thứ tự, truyền thông tin...

- Giải quyết thừa kế xung đột vì một lớp được kế thừa lặp lại.

6.4.2 Ví dụ minh họa

Giả sử rằng có một tình huống như sau: lớp colorpoint kế thừa hai lớp point và color, giữa các lớp có sự truyền thông tin cho nhau từ các hàm khởi tạo, những quy tắc đã trình bày trong phần 6.2.4 sẽ không thay đổi:

```
#include<iostream>
#include<conio.h>
class point
{
 int x,y;
 public:
 point(int ox, int oy)
 {
  cout<<"++ constr.point"<<endl;
  x = ox; y = oy;
 }
 ~point()
```

```
 {
  cout<<"-- destr.point"<<endl;
 }
 void display()
 {
  cout<<"tọa độ: "<<x<<","<<y<<endl;
 }
};
class col
{
 unsigned color;
 public:
 col(unsigned c)
 {
  cout<<"++ constr.col"<<endl;
  color = c;
 }
 ~col()
 {
  cout<<"-- destr.col"<<endl;
 }
 void display()
 {
  cout<<"màu: "<<color<<endl;
 }
};
class colorpoint : public point, public col
{
 public:
 colorpoint(int ox, int oy, unsigned c) : point(ox, oy), col(c)
 {
  cout<<"++ constr.colorpoint"<<endl;
 }
 ~colorpoint()
 {
  cout<<"-- destr.colorpoint"<<endl;
 }
 void display()
 {
  point::display();
  col::display();
 }
};
main()
{
 clrscr();
```

```
 colorpoint p(3, 9, 2);
 p.display();
 p.point::display();
 p.col::display();
 getch();
}
```

Thứ tự gọi hàm khởi tạo tuần tự của lớp: bắt đầu là hàm khởi tạo của lớp point rồi đến lớp col, theo sau mới là lớp dẫn xuất nhưng các hàm dọn dẹp theo thứ tự ngược lại. Khi có nhiều hàm cùng tên trong các lớp, cách duy nhất để phân biệt chúng là dùng toán tử định trị "**::**" đặt giữa tên lớp và tên hàm.

6.4.3 Lớp cơ sở ảo

Xét tình huống sau đây: lớp A là lớp cơ sở ban đầu, lớp B và C đều kế thừa từ lớp A, nhưng lớp D đa kế thừa hai lớp B và C. Trong quá trình biên dịch, nếu đối tượng lớp D gọi một hàm thành phần x của lớp A thì trình dịch sẽ bắt lỗi vì nó cho rằng không rõ hàm này được kế thừa thông qua lớp B hay C. Muốn phân biệt phải chỉ rõ A::B::x hoặc A::C::x.

Còn có giải pháp hay hơn là với từ khóa virtual đặt trước tên lớp A tại đầu dòng định nghĩa lớp B và C thì nhược điểm này mất hiệu lực. Việc chỉ định A là lớp ảo tức là A chỉ xuất hiện một lần trong lớp các con cháu của chúng:

```
#include<iostream>
#include<conio.h>
class A
{
 int x;
 public:
 void init(int ox)
 {
  x = ox;
 }
 int getx()
 {
  return x;
 }
};
class B: virtual public A
{ };
class C: virtual public A
{ };
class D: public B, public C
{ };
main()
{
 clrscr();
 D d;
 d.init(3);
 cout<<"d.B::getx() = "<<endl; cout<< d.B::getx()<<endl;
```

```
cout<<"d.C::getx() = "<<endl; cout<< d.C::getx()<<endl;
getch();
}
```