

**BỘ GIAO THÔNG VẬN TẢI
TRƯỜNG ĐẠI HỌC HÀNG HẢI
BỘ MÔN: KHOA HỌC MÁY TÍNH
KHOA: CÔNG NGHỆ THÔNG TIN**

BÀI GIẢNG
LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VÀ C++

**TÊN HỌC PHẦN : Lập trình hướng đối tượng và C++
MÃ HỌC PHẦN : 17209
TRÌNH ĐỘ ĐÀO TẠO : ĐẠI HỌC CHÍNH QUY
DÙNG CHO SV NGÀNH : CÔNG NGHỆ THÔNG TIN**

HẢI PHÒNG - 2008

Mô tả vắn tắt nội dung và khối lượng các học phần

Tên học phần: Lập trình hướng đối tượng .

Bộ môn phụ trách giảng dạy: Khoa học máy tính.

Mã học phần: 15508

Loại học phần : 3

Khoa phụ trách: CNTT

Tổng số TC: 3

TS tiết	Lý thuyết	Thực hành/ Xemina	Tự học	Bài tập lớn	Đồ án môn học
75	45	30	0	0	0

Điều kiện tiên quyết:

Sinh viên phải học và thi đạt các học phần sau mới được đăng ký học học phần này:

Kỹ thuật lập trình Pascal, Kỹ thuật lập trình C.

Mục tiêu của học phần:

Cung cấp kiến thức của phương pháp lập trình hướng đối tượng và rèn luyện kỹ năng lập trình .

Nội dung chủ yếu:

- Những mở rộng của lập trình hướng đối tượng.
- Đối tượng và lớp.
- Đóng gói, thừa kế, đa hình.
- Bản mẫu...

Nội dung chi tiết:

TÊN CHƯƠNG MỤC	PHÂN PHỐI SỐ TIẾT				
	TS	LT	Thực hành	BT	KT
Chương 1: Lập trình hướng đối tượng và ngôn ngữ C++	3	3			
1.1 Ưu điểm của lập trình hướng đối tượng					
1.2 Giới thiệu ngôn ngữ C++					
Chương 2: Những khái niệm mở đầu	9	6	3		
2.1 Cài đặt ngôn ngữ C++					
2.2 Cấu trúc một chương trình C++					
2.3 Kiểu dữ liệu cơ sở					
2.4 Quy tắc sử dụng từ khóa, tên chuẩn, tên từ đặt.					
2.5 Các chỉ thị gán, so sánh, điều kiện nếu thì ..					
Chương 3: Con trỏ, tham chiếu và hàm	6	3	3		
3.1 Khai báo hàm con, hàm chính.					
3.2 Quy tắc đổi kiểu dữ liệu, kiểu trỏ.					
3.3 Định nghĩa chồng hàm, tham số ngầm định..					
3.5 Tham chiếu.					
Chương 4: Các dòng vào ra trong C++	9	5	3		1
4.1 Đối tượng vào ra cout, cin					
4.2 Đẩy dòng dữ liệu lên màn hình					
4.3 Nhập dòng dữ liệu từ bàn phím					
4.4 Định dạng dòng dữ liệu hiển thị					
4.5 Vào ra với tệp					
Chương 5: Đối tượng và Lớp	18	8	9		1
5.1 Định nghĩa đối tượng					
5.2 Khai báo lớp					

5.3 Hàm thiết lập, huỷ bỏ					
5.4 Thành phần tính, hàm bạn, lớp bạn					
5.5 Định nghĩa chồng toán tử					
Chương 6: Thừa kế	9	6	3		
6.1 Lớp cơ sở, lớp dẫn xuất					
6.2 Quy tắc thừa kế					
6.3 Tương thích lớp cơ sở và lớp dẫn xuất					
6.4 Đơn thừa kế, đa thừa kế					
Chương 7 Ràng buộc động và Đa thể	6	3	3		
7.1 Hàm ảo, ràng buộc tĩnh, động					
7.2 Đa thể					
Chương 8: Bản mẫu	12	5	6		1
8.1 Hàm bản mẫu					
8.2 Ưu khuyết điểm của hàm bản mẫu					
8.3 Lớp bản mẫu					
Chương 9: Giới thiệu về phân tích thiết kế hướng đối tượng	3	3			

Nhiệm vụ của sinh viên: Lên lớp đầy đủ và chấp hành mọi quy định của Nhà trường.

Tài liệu học tập:

1. Tên tác giả. Tên sách. Nhà xuất bản. Năm xuất bản.
2. Phạm Văn Ất. Kỹ thuật lập trình hướng đối tượng. NXB KHKT. 1998
3. Một số website liên quan.

Hình thức và tiêu chuẩn đánh giá sinh viên:

- Thi viết hoặc thi thực hành.
- Sinh viên phải bảo đảm các điều kiện theo Quy chế của Nhà trường và của Bộ.

Thang điểm : Thang điểm chữ A,B,C,D,F.

Điểm đánh giá học phần: $Z=0,3X+0,7Y$.

Chương 1: Lập trình hướng đối tượng và ngôn ngữ C++

1. Sự phát triển của các kỹ thuật lập trình

Phần này trình bày về một số kỹ thuật hay phương pháp lập trình được phát triển để giải quyết các vấn đề trong Tin học kể từ khi máy tính ra đời. Sự phát triển của các kỹ thuật lập trình liên quan chặt chẽ tới sự phát triển phần cứng của máy vi tính cũng như việc ứng dụng máy tính vào giải quyết các vấn đề trong thực tế. Chúng ta có thể chia các phương pháp lập trình thành các kiểu sau:

- Lập trình không có cấu trúc
- Lập trình hướng thủ tục
- Lập trình theo kiểu module hóa
- Lập trình hướng đối tượng

Chúng ta sẽ lần lượt xem xét các kỹ thuật lập trình này.

1.1 Lập trình không có cấu trúc (hay lập trình tuyến tính)

Thông thường mọi người bắt đầu học lập trình bằng cách viết các chương trình nhỏ và đơn giản chỉ chứa một “chương trình chính”. Ở đây một chương trình chính có nghĩa là một tập các lệnh hoặc câu lệnh làm việc với các dữ liệu toàn cục trong cả chương trình (các biến dùng trong chương trình là các biến toàn cục). Chúng ta có thể minh họa bằng hình vẽ sau đây:



**Lập trình không có cấu trúc. Chương trình chính
thao tác trực tiếp trên các dữ liệu toàn cục**

Một số nhược điểm của lập trình không có cấu trúc:

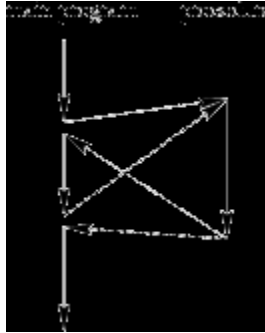
- Lập trình không có cấu trúc không có khả năng kiểm soát tính thấy được của dữ liệu. Mọi dữ liệu trong chương trình đều là biến toàn cục do đó có thể bị thay đổi bởi bất kỳ phần nào đó của chương trình.
- Việc không kiểm soát được tính thấy được của dữ liệu dẫn đến các khó khăn trong việc gỡ lỗi chương trình, đặc biệt là các chương trình lớn.
- Kỹ thuật lập trình không có cấu trúc có rất nhiều bất lợi lớn khi chương trình đủ lớn. Ví dụ nếu chúng ta cần thực hiện lại một đoạn câu lệnh trên một tập dữ liệu khác thì buộc phải copy đoạn lệnh đó tới vị trí trong chương trình mà chúng ta muốn thực hiện. Điều này làm nảy sinh ý tưởng trích ra các đoạn lệnh thường xuyên cần thực hiện đó, đặt tên cho chúng và đưa ra một kỹ thuật cho phép gọi và trả về các giá trị từ các thủ tục này.

1.2 Lập trình thủ tục hay lập trình có cấu trúc

Với lập trình thủ tục hay hướng thủ tục chúng ta có thể nhóm các câu lệnh thường xuyên thực hiện trong chương trình chính lại một chỗ và đặt tên đoạn câu lệnh đó thành một thủ tục. Một lời gọi tới thủ tục sẽ được sử dụng để thực hiện đoạn câu lệnh đó. Sau khi thủ tục thực hiện xong điều khiển trong chương trình được trả về ngay sau vị trí lời gọi tới thủ tục trong chương trình chính. Với các cơ chế truyền tham số cho

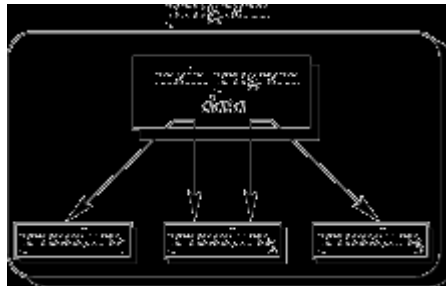
thủ tục chúng ta có các chương trình con . Một chương trình chính bao gồm nhiều chương trình con và các chương trình được viết mang tính cấu trúc cao hơn, đồng thời cũng ít lỗi hơn. Nếu một chương trình con là đúng đắn thì kết quả thực hiện trả về luôn đúng và chúng ta không cần phải quan tâm tới các chi tiết bên trong của thủ tục . Còn nếu có lỗi chúng ta có thể thu hẹp phạm vi gỡ lỗi trong các chương trình con chưa được chứng minh là đúng đắn , đây được xem như trừu tượng hàm và là nền tảng cho lập trình thủ tục.

Một chương trình chính với lập trình thủ tục có thể được xem là tập hợp các lời gọi thủ tục.



Lập trình thủ tục. Sau khi chương trình con thực hiện xong điều khiển được trả về ngay sau vị trí lời gọi tới chương trình con

Chương trình chính có nhiệm vụ truyền các dữ liệu cho các lời gọi cụ thể, dữ liệu được xử lý cục bộ trong chương trình con sau đó các kết quả thực hiện này được trả về cho chương trình chính . Như vậy luồng dữ liệu có thể được minh họa như là một đồ thị phân cấp, một cây:

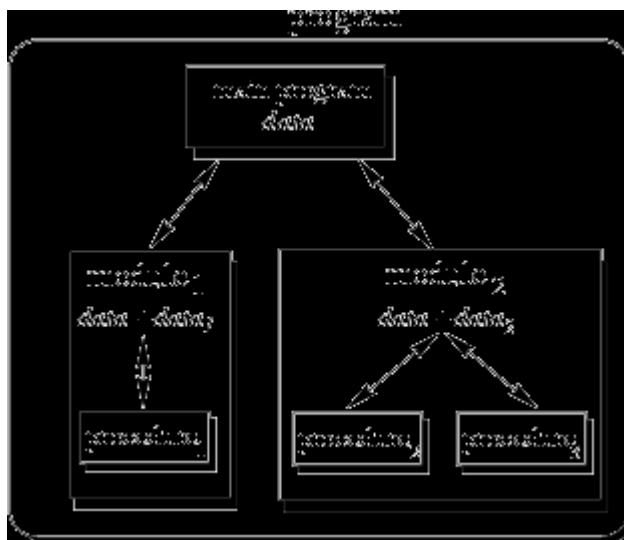


Lập trình hướng thủ tục. Chương trình chính phối hợp các lời gọi tới các thủ tục với các dữ liệu thích hợp là các tham số

Lập trình hướng thủ tục là một kỹ thuật lập trình có nhiều ưu điểm . Khái niệm chương trình con là một ý tưởng rất hay , nó cho phép một chương trình lớn có thể được chia thành nhiều chương trình con nhỏ hơn , do đó dễ viết hơn và ít lỗi hơn . Để có thể sử dụng được các thủ tục chung hoặc một nhóm các thủ tục trong các chương trình khác, người ta đã phát minh ra một kỹ thuật lập trình mới , đó là kỹ thuật lập trình theo kiểu module.

1.3 Lập trình module

Trong lập trình module các thủ tục có cùng một chức năng chung sẽ được nhóm lại với nhau tạo thành một module riêng biệt. Một chương trình sẽ không chỉ bao gồm một phần đơn lẻ. Nó được chia thành một vài phần nhỏ hơn tương tác với nhau qua các lời gọi thủ tục và tạo thành toàn bộ chương trình.



Lập trình module. Chương trình chính là sự kết hợp giữa các lời gọi tới các thủ tục trong các module riêng biệt với các dữ liệu thích hợp

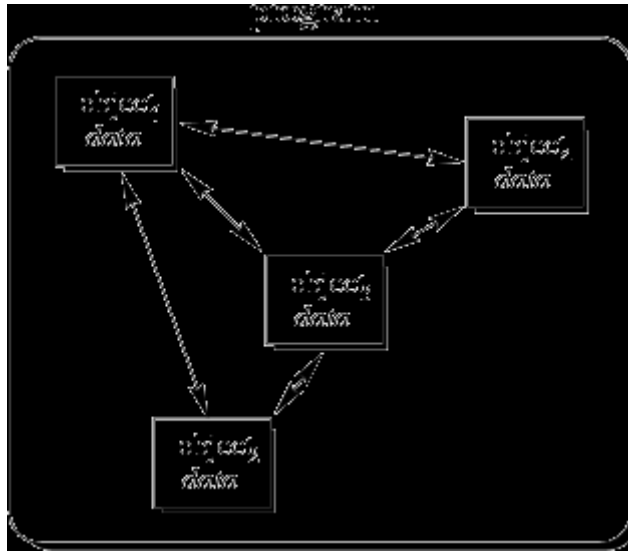
Mỗi module có dữ liệu riêng của nó. Điều này cho phép các module có thể kiểm soát các dữ liệu riêng của nó bằng các lời gọi tới các thủ tục trong module đó. Tuy nhiên mỗi module chỉ xuất hiện nhiều nhất một lần trong cả chương trình.

Yếu điểm của lập trình thủ tục và lập trình module hóa:

- Khi độ phức tạp của chương trình tăng lên sự phụ thuộc của nó vào các kiểu dữ liệu cơ bản mà nó xử lý cũng tăng theo. Vấn đề trở nên rõ ràng rằng cấu trúc dữ liệu sử dụng trong chương trình cũng quan trọng không kém các phép toán thực hiện trên chúng. Điều này càng lộ rõ khi kích thước chương trình tăng. Các kiểu dữ liệu được xử lý nhiều trong các thủ tục của một chương trình có cấu trúc. Do đó khi thay đổi cài đặt của một kiểu dữ liệu sẽ dẫn đến nhiều thay đổi trong các thủ tục sử dụng nó.
- Một nhược điểm nữa là khi cần dùng nhiều nhóm làm việc để xây dựng một chương trình chung. Trong lập trình có cấu trúc mỗi người sẽ được giao xây dựng một số thủ tục và kiểu dữ liệu. Những lập trình viên xử lý các thủ tục khác nhau nhưng lại có liên quan tới các kiểu dữ liệu dùng chung nên nếu một người thay đổi kiểu dữ liệu thì sẽ làm ảnh hưởng tới công việc của nhiều người khác, đặc biệt là khi có sai sót trong việc liên lạc giữa các thành viên của nhóm.
- Việc phát triển các phần mềm mất nhiều thời gian tập trung xây dựng lại các cấu trúc dữ liệu cơ bản. Khi xây dựng một chương trình mới trong lập trình có cấu trúc lập trình viên thường phải xây dựng lại các cấu trúc dữ liệu cơ bản cho phù hợp với bài toán và điều này đôi khi rất mất thời gian.

1.4 Lập trình hướng đối tượng

Trong lập trình hướng đối tượng trong mỗi chương trình chúng ta có một số các đối tượng (**object**) có thể tương tác với nhau , thuộc các lớp (**class**) khác nhau , mỗi đối tượng tự quản lý lấy các dữ liệu của riêng chúng.



Lập trình hướng đối tượng . Các đối tượng tương tác với nhau bằng cách gửi các thông điệp.

Chương trình chính sẽ bao gồm một số đối tượng là thể hiện (*instance*) của các lớp, các đối tượng này tương tác với nhau thực hiện các chức năng của chương trình . Các lớp trong lập trình hướng đối tượng có thể xem như là một sự trừu tượng ở mức cao hơn của các cấu trúc (struct hay record) hay kiểu dữ liệu do người dùng định nghĩa trong các ngôn ngữ lập trình có cấu trúc với sự tích hợp cả các toán tử và dữ liệu trên các kiểu đó.

Các ưu điểm của lập trình hướng đối tượng:

- Lập trình hướng đối tượng ra đời đã giải quyết được nhiều nhược điểm tồn tại trong lập trình có cấu trúc . Trong lập trình OOP có ít lỗi hơn và việc gỡ lỗi cũng đơn giản hơn, đồng thời lập trình theo nhóm có thể thực hiện rất hiệu quả . Ít lỗi là một trong các ưu điểm chính của OOP vì theo thống kê thì việc bảo trì hệ thống phần mềm sau khi giao cho người dùng chiếm tới 70% giá thành phần mềm.
- Việc thay đổi các cài đặt chi tiết bên dưới trong lập trình OOP không làm ảnh hưởng tới các phần khác của chương trình do đó việc mở rộng qui mô của một chương trình dễ dàng hơn , đồng thời là giảm thời gian cần thiết để phát triển phần mềm.
- Với khái niệm kế thừa các lập trình viên có thể xây dựng các chương trình từ các phần mềm sẵn có.
- OOP có tính khả chuyển cao . Một chương trình viết trên một hệ thống nền (chẳng hạn Windows) có thể chạy trên nhiều hệ thống nền khác nhau (chẳng hạn Linux, Unix...).
- OOP có hiệu quả cao . Thực tế cho thấy các hệ thống được xây dựng bằng OOP có hiệu năng cao.

2. Một số khái niệm cơ bản của lập trình hướng đối tượng

1.1 Kiểu dữ liệu trừu tượng ADT(Astract Data Type)

Một số người định nghĩa OOP là lập trình với các kiểu dữ liệu trừu tượng và các mối quan hệ giữa chúng. Trong phần này chúng ta sẽ xem xét các kiểu dữ liệu trừu tượng như là một khái niệm cơ bản của OOP và sử dụng một số ví dụ để minh họa.

Định nghĩa về kiểu dữ liệu trừu tượng:

Một kiểu dữ liệu trừu tượng là một mô hình toán học của các đối tượng dữ liệu tạo thành một kiểu dữ liệu và các toán tử (phép toán) thao tác trên các đối tượng đó. Chú ý là trong định nghĩa này các toán tử thao tác trên các đối tượng dữ liệu gắn liền với các đối tượng tạo thành một kiểu dữ liệu trừu tượng. Đặc tả về một kiểu dữ liệu trừu tượng không có bất kỳ một chi tiết cụ thể nào về cài đặt bên trong của kiểu dữ liệu. Việc cài đặt một kiểu dữ liệu trừu tượng đòi hỏi một quá trình chuyển đổi từ đặc tả của nó sang một cài đặt cụ thể trên một ngôn ngữ lập trình cụ thể. Điều này cho phép chúng ta phân biệt các ADT với các thuật ngữ kiểu dữ liệu (data type) và cấu trúc dữ liệu (data structure). Thuật ngữ kiểu dữ liệu đề cập tới một cài đặt cụ thể (có thể là kiểu built in hoặc do người dùng định nghĩa) của một mô hình toán học được đặc tả bởi một ADT. Cấu trúc dữ liệu đề cập tới một tập các biến có cùng kiểu được gắn kết với nhau theo một cách thức xác định nào đó.

Ví dụ về kiểu dữ liệu trừu tượng: Số nguyên.

Kiểu dữ liệu trừu tượng số nguyên: ADT Integer:

Dữ liệu: một tập các chữ số và một dấu tiền tố là + hoặc -. Chúng ta ký hiệu cả số là N.

Các toán tử:

constructor: khởi tạo một số nguyên

sub(k): trả về hiệu $N - k$.

add(k): trả về tổng $N + k$.

.....

End

1.2 Đối tượng (Objects) và lớp (Classes)

Trong một chương trình hướng đối tượng chúng ta có các đối tượng. Các đối tượng này là đại diện cho các đối tượng thực trong thực tế. Có thể coi khái niệm đối tượng trong OOP chính là các kiểu dữ liệu trong các ngôn ngữ lập trình có cấu trúc. Mỗi một đối tượng có các dữ liệu riêng của nó và được gọi là các member variable hoặc là các data member. Các toán tử thao tác trên các dữ liệu này được gọi là các member function.

Mỗi một đối tượng là thể hiện (instance) của một lớp. Như vậy lớp là đại diện cho các đối tượng có các member function giống nhau và các data member cùng kiểu. Lớp là một sự trừu tượng hóa của khái niệm đối tượng. Tuy nhiên lớp không phải là một ADT, nó là một cài đặt của một đặc tả ADT. Các đối tượng của cùng một lớp có thể chia sẻ các dữ liệu dùng chung, dữ liệu kiểu này được gọi là class variable.

1.3 Kế thừa (Inheritance)

Khái niệm kế thừa này sinh từ nhu cầu sử dụng lại các thành phần phần mềm để phát triển các phần mềm mới hoặc mở rộng chức năng của phần mềm hiện tại. Kế thừa là một cơ chế cho phép các đối tượng của một lớp có thể truy cập tới các member variable và function của một lớp đã được xây dựng trước đó mà không cần xây dựng lại các thành phần đó. Điều này cho phép chúng ta có thể tạo ra các lớp mới là một mở rộng hoặc cá biệt hóa của một lớp sẵn có. Lớp mới (gọi là derived class) kế thừa từ lớp

cũ (gọi là lớp cơ sở base class). Các ngôn ngữ lập trình hướng đối tượng có thể hỗ trợ khái niệm đa kế thừa cho phép một lớp có thể kế thừa từ nhiều lớp cơ sở.

Lớp kế thừa derived class có thể có thêm các data member mới hoặc các member function mới. Thêm vào đó lớp kế thừa có thể tiến hành định nghĩa lại một hàm của lớp cơ sở và trong trường hợp này người ta nói rằng lớp kế thừa đã overload hàm thành viên của lớp cơ sở.

1.4 Dynamic Binding (tạm dịch là ràng buộc động) và Polymorphism (đa xạ hoặc đa thể)

Chúng ta lấy một ví dụ để minh họa cho hai khái niệm này. Giả sử chúng ta có một lớp cơ sở là Shape, hai lớp kế thừa từ lớp Shape là Circle và Rectangle. Lớp Shape là một lớp trừu tượng có một member function trừu tượng là draw(). Hai lớp Circle và Rectangle thực hiện overload lại hàm draw của lớp Shape với các chi tiết cài đặt khác nhau chẳng hạn với lớp Circle hàm draw sẽ vẽ một vòng tròn còn với lớp Rectangle thì sẽ vẽ một hình chữ nhật. Và chúng ta có một đoạn chương trình chính hợp lệ như sau:

```
int main()
{
    Shape shape_list[4];
    int choose;
    int i;
    for(i=0;i<4;i++)
    {
        cout << "Ngay muon ve hinh tron(0) hay hinh chu nhac(1)";
        cin >> choose;
        if(choose==0)
        {
            shape_list[i] = new Circle();
        }else{
            shape_list[i] = new Rectangle();
        }
    }
    for(i=0;i<4;i++)
    {
        shape_list[i]->draw();
    }
}
```

Khi biên dịch chương trình này thành mã thực hiện (file .exe) trình biên dịch không thể xác định được trong mảng shape_list thì phần tử nào là Circle phần tử nào là Rectangle và do đó không thể xác định được phiên bản nào của hàm draw sẽ được gọi thực hiện. Việc gọi tới phiên bản nào của hàm draw để thực hiện sẽ được quyết định tại thời điểm thực hiện chương trình, sau khi đã biên dịch và điều này được gọi là dynamic binding hoặc late binding. Ngược lại nếu việc xác định phiên bản nào sẽ được gọi thực hiện tương ứng với dữ liệu gắn với nó được quyết định ngay trong khi biên dịch thì người ta gọi đó là static binding.

Ví dụ này cũng cung cấp cho chúng ta một minh họa về khả năng đa thể (polymorphism). Khái niệm đa thể được dùng để chỉ khả năng của một thông điệp có thể được gửi tới cho các đối tượng của nhiều lớp khác nhau tại thời điểm thực hiện

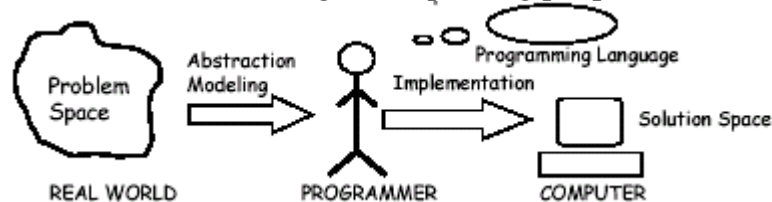
chương trình. Chúng ta thấy rõ lời gọi tới hàm draw sẽ được gửi tới cho các đối tượng của hai lớp Circle và Rectangle tại thời điểm chương trình được thực hiện.

Ngoài các khái niệm cơ bản trên OOP còn có thêm một số khái niệm khác chẳng hạn như name space và exception handling nhưng không phải là các khái niệm bản chất.

3. Ngôn ngữ lập trình C++ và OOP.

Giống như bất kỳ một ngôn ngữ nào của con người, một ngôn ngữ lập trình là phương tiện để diễn tả các khái niệm, ý tưởng. Việc phát triển các chương trình hay phần mềm là quá trình mô hình hóa các trạng thái tự nhiên của thế giới thực và xây dựng các chương trình dựa trên các mô hình đó.

Các chương trình thực hiện chức năng mô tả phương pháp cài đặt của mô hình.



Các thể hệ ngôn ngữ lập trình: Có thể phân chia các thể hệ ngôn ngữ lập trình thành 4 thể hệ:

- 1: vào năm 1954 – 1958 (Fortran I) với đặc điểm là các biểu thức toán học
- 2: vào năm 1959 – 1961 (Fortran II, Cobol) với các thủ tục
- 3: vào những năm 1962 – 1970 (Pascal, Simula) với đặc trưng là các khối, các lớp...

4: đang phát triển chưa có dẫn chứng thực tế.

Các ngôn ngữ này ngày càng cách xa ngôn ngữ máy và các trình biên dịch của chúng ngày càng phải làm việc nhiều hơn.

1.1 Sự phát triển của các ngôn ngữ lập trình hướng đối tượng

1967 Simula

1970 to 1983 Smalltalk

1979 Common LISP Object System

1980 Stroustrup starts on C++

1981 Byte Smalltalk issue

1983 Objective C

1986 C++

1987 Actor, Eiffel

1991 C++ release 3.0

1995 Java

1983 to 1989 Language books with OO concepts

1989 to 1992 Object-oriented design books

1992 to present Object-oriented methodology books

Các ngôn ngữ lập trình khác

Java

Self

Python

Perl

Prograph

Modula 3
Oberon
Smalltalk Venders
ParcPlace, Digtalk, Quasar
Prolog++
Ada 9X
Object Pascal (Delphi)
Object X, X = fortran, cobal, etc.
C#.

Như vậy là có rất nhiều ngôn ngữ lập trình hướng đối tượng đã ra đời và chiếm ưu thế trong số chúng là C++ và Java. Mỗi ngôn ngữ đều có đặc điểm riêng của nó và thích hợp với các lĩnh vực khác nhau nhưng có lẽ C++ là ngôn ngữ cài đặt nhiều đặc điểm của OOP nhất.

1.2 Ngôn ngữ lập trình C++.

C++ là một ngôn ngữ lập trình hướng đối tượng được Bjarne Stroustrup (AT & T Bell Lab) (giải thưởng ACM Grace Murray Hopper năm 1994) phát triển từ ngôn ngữ C.

C++ kế thừa cú pháp và một số đặc điểm ưu việt của C : ví dụ như xử lý con trỏ , thư viện các hàm phong phú đa dạng , tính khả chuyên cao , chương trình chạy nhanh Tuy nhiên về bản chất thì C++ khác hoàn toàn so với C, điều này là do C++ là một ngôn ngữ lập trình hướng đối tượng.

Chương 2: Những khái niệm mở đầu

1. Chương trình đầu tiên

1.1 Quá trình biên dịch một chương trình C++

Tất cả các ngôn ngữ trên máy tính đều được dịch từ một dạng nào đó mà con người có thể hiểu được một cách dễ dàng (các file mã nguồn được viết bằng một ngôn ngữ bậc cao) sang dạng có thể thực hiện được trên máy tính (các lệnh dưới dạng ngôn ngữ máy). Các chương trình thực hiện quá trình này chia thành hai dạng được gọi tên là các trình thông dịch (interpreter) và các trình biên dịch (compiler).

Trình thông dịch : Một trình thông dịch sẽ dịch mã nguồn thành các hành động (activity), các hành động này có thể bao gồm một nhóm các lệnh máy và tiến hành thực hiện ngay lập tức các hành động này . Ví dụ như BASIC là một ngôn ngữ điển hình cho các ngôn ngữ thông dịch . BASIC cổ điển thông dịch từng dòng lệnh thực hiện và sau đó quên ngay lập tức dòng lệnh vừa thông dịch . Điều này làm cho quá trình thực hiện cả một chương trình chậm vì bộ thông dịch phải tiến hành dịch lại các đoạn mã trùng lặp . BASIC ngày nay đã thêm vào quá trình biên dịch để cải thiện tốc độ của chương trình. Các bộ thông dịch hiện đại chẳng hạn như Python , tiến hành dịch toàn bộ chương trình qua một ngôn ngữ trung gian sau đó thực hiện bằng một bộ thông dịch nhanh hơn rất nhiều.

Các ngôn ngữ làm việc theo kiểu thông dịch thường có một số hạn chế nhất định khi xây dựng các dự án lớn (Có lẽ chỉ duy nhất Python là một ngoại lệ). Bộ thông dịch cần phải luôn được lưu trong bộ nhớ để thực hiện các mã chương trình , và thậm chí ngay cả bộ thông dịch có tốc độ nhanh nhất cũng không thể cải thiện được hoàn toàn các hạn chế tốc độ . Hầu hết các bộ thông dịch đều yêu cầu toàn bộ mã nguồn cần phải được thông dịch một lần duy nhất . Điều này không những dẫn đến các hạn chế về kích

thước của chương trình mà còn tạo ra các lỗi rất khó gỡ rối nếu như ngôn ngữ không cung cấp các công cụ hiệu quả để xác định hiệu ứng của các đoạn mã khác nhau.

Trình biên dịch : Một trình biên dịch dịch mã nguồn trực tiếp thành ngôn ngữ assembly hoặc các lệnh máy. Kết quả cuối cùng là một file duy nhất hoặc các file chứa các mã máy. Đây là một quá trình phức tạp và đòi hỏi một vài bước. Quá trình chuyển đổi từ mã chương trình ban đầu thành mã thực hiện là tương đối dài đối với một trình biên dịch.

Tùy thuộc vào sự nhạy cảm của người viết trình biên dịch, các chương trình sinh ra bởi một trình biên dịch có xu hướng đòi hỏi ít bộ nhớ hơn khi thực hiện, và chúng chạy nhanh hơn rất nhiều. Mặc dù kích thước và tốc độ thường là các lý do hàng đầu cho việc sử dụng một trình biên dịch, trong rất nhiều trường hợp đây không phải là các lý do quan trọng nhất. Một vài ngôn ngữ (chẳng hạn như C) được thiết kế để các phần tách biệt của một chương trình có thể được biên dịch độc lập hoàn toàn với nhau. Các phần này sau đó thậm chí có thể kết hợp thành một chương trình thực hiện cuối cùng duy nhất bởi một công cụ có tên là trình liên kết. Quá trình này gọi là *separate compilation* (biên dịch độc lập).

Biên dịch độc lập có rất nhiều điểm lợi. Một chương trình nếu dịch ngay lập tức toàn bộ sẽ vượt quá các giới hạn của trình biên dịch hay môi trường biên dịch có thể được biên dịch theo từng phần. Các chương trình có thể được xây dựng và kiểm thử từng phần một. Nếu một phần nào đó đã làm việc đúng đắn nó có thể được lưu lại như là một khối đã hoàn thành. Tập các phần đã làm việc và được kiểm thử có thể kết hợp lại với nhau tạo thành các thư viện để các lập trình viên khác có thể sử dụng. Các đặc điểm này hỗ trợ cho việc tạo ra các chương trình lớn.

Các đặc điểm gỡ lỗi của trình biên dịch đã cải tiến một cách đáng kể qua thời gian. Các trình biên dịch đầu tiên chỉ sinh ra mã máy, và lập trình viên phải chen các câu lệnh in vào để xem thực sự chương trình đang làm gì. Điều này không phải lúc nào cũng hiệu quả. Các trình biên dịch hiện đại có thể chen các thông tin về mã nguồn vào mã thực hiện của chương trình. Thông tin này sẽ được sử dụng bởi các bộ gỡ lỗi cấp độ nguồn đầy năng lực để chỉ ra chính xác điều gì đang diễn ra trong một chương trình bằng cách theo dấu (tracing) quá trình thực hiện của nó qua toàn bộ mã nguồn.

Một vài trình biên dịch giải quyết vấn đề tốc độ biên dịch bằng cách thực hiện quá trình biên dịch trong bộ nhớ (*in-memory compilation*). Các trình biên dịch theo kiểu này lưu trình biên dịch trong bộ nhớ RAM. Đối với các chương trình nhỏ, quá trình này có thể xem như là một trình thông dịch.

Quá trình biên dịch

Để lập trình bằng C và C++ chúng ta cần phải hiểu các bước và các công cụ trong quá trình biên dịch. Một vài ngôn ngữ (đặc biệt là C và C++) bắt đầu thực hiện quá trình biên dịch bằng cách chạy một bộ tiền xử lý đối với mã nguồn. Bộ tiền xử lý là một chương trình đơn giản thay thế các mẫu trong mã nguồn bằng các mẫu khác mà các lập trình viên đã định nghĩa (sử dụng các chỉ thị tiền xử lý: *preprocessor directives*). Các chỉ thị tiền xử lý được sử dụng để tiết kiệm việc gõ các đoạn chương trình thường xuyên sử dụng và tăng khả năng dễ đọc cho mã nguồn. Tuy nhiên các chỉ thị tiền xử lý này đôi khi cũng gây ra những lỗi rất tinh vi và khó phát hiện. Mã sinh ra bởi bộ tiền xử lý này thường được ghi lên một file tạm.

Các trình biên dịch thường thực hiện công việc của nó theo hai pha. Đầu tiên là phân tích mã tiền xử lý. Bộ biên dịch chia mã tiền xử lý thành các đơn vị nhỏ và tổ

chức chúng thành một cấu trúc gọi là cây. Ví dụ như trong biểu thức: “A+B” các phần tử “A”, “+”, “B” sẽ được lưu trên nút của cây phân tích.

Một bộ tối ưu hóa toàn cục (*global optimizer*) đôi khi cũng được sử dụng để tạo ra mã chương trình nhỏ hơn, nhanh hơn.

Trong pha thứ hai, bộ sinh mã duyệt qua cây phân tích và sinh ra hoặc là mã assemble hoặc mã máy cho các nút của cây. Nếu như bộ sinh mã tạo ra mã assemble, thì sau đó chương trình dịch mã assembler sẽ thực hiện công việc tiếp theo. Kết quả của hai trường hợp trên đều là một module object (một file thường có đuôi là .o hoặc .obj). Sau đó một bộ tối ưu hoá nhỏ (peep-hole) sẽ được sử dụng để loại bỏ các đoạn chứa các câu lệnh assembly thừa.

Việc sử dụng từ “object” để mô tả các đoạn mã máy là một thực tế không đúng lắm. Từ này đã được dùng trước cả khi lập trình hướng đối tượng ra đời. Từ “object” được sử dụng có ý nghĩa như là từ “goal” khi nói về việc biên dịch, trong khi đó trong lập trình hướng đối tượng nó lại có nghĩa là “a thing with boundaries”.

Trình liên kết kết hợp một danh sách các module object thành một chương trình thực hiện có thể nạp vào bộ nhớ và thực hiện bởi hệ điều hành. Khi một hàm trong một module object tạo ra một tham chiếu tới một hàm hoặc một biến trong một module object khác, trình liên kết sẽ sắp xếp lại các tham chiếu này; điều này đảm bảo rằng tất cả các hàm và dữ liệu external được sử dụng trong quá trình biên dịch là đều tồn tại. Trình liên kết cũng thêm vào các module object đặc biệt để thực hiện các hành động khởi động.

Trình liên kết có thể tìm kiếm trên các file đặc biệt gọi là các thư viện để sắp xếp lại tất cả các tham chiếu tới chúng. Mỗi thư viện chứa một tập các module object trong một file. Một thư viện được tạo ra và bảo trì bởi một lập trình viên có tên là *librarian*.

Kiểm tra kiểu tĩnh

Trình biên dịch thực hiện kiểm tra kiểu trong pha đầu tiên của quá trình biên dịch. Quá trình kiểm tra này thực hiện kiểm thử việc sử dụng các tham số của các hàm và ngăn chặn rất nhiều lỗi lập trình khác nhau. Vì quá trình kiểm tra kiểu được thực hiện trong quá trình biên dịch chứ không phải trong quá trình chương trình thực hiện nên nó được gọi là kiểm tra kiểu tĩnh.

Một vài ngôn ngữ lập trình hướng đối tượng (Java chẳng hạn) thực hiện kiểm tra kiểu tại thời điểm chương trình chạy (*dynamic type checking*). Nếu kết hợp cả việc kiểm tra kiểu tĩnh và động thì sẽ hiệu quả hơn nhưng kiểm tra kiểu động cũng làm cho chương trình thực hiện bị ảnh hưởng đôi chút.

C++ sử dụng kiểm tra kiểu tĩnh. Kiểm tra kiểu tĩnh báo cho lập trình viên về các lỗi về sử dụng sai kiểu dữ liệu trong quá trình biên dịch, và do đó tối ưu hóa tốc độ thực hiện chương trình. Khi học C++ chúng ta sẽ thấy hầu hết các quyết định thiết kế của ngôn ngữ đều tập trung vào củng cố các đặc điểm: tốc độ nhanh, hướng đối tượng, các đặc điểm mà đã làm cho ngôn ngữ C trở nên nổi tiếng.

Chúng ta có thể không dùng tùy chọn kiểm tra kiểu tĩnh của C++ hoặc cũng có thể thực hiện việc kiểm tra kiểu động - chỉ cần viết thêm mã.

Các công cụ cho việc biên dịch độc lập

Việc biên dịch độc lập rất cần thiết nhất là đối với các dự án lớn. Trong ngôn ngữ C và C++, một lập trình viên có thể tạo ra các đoạn chương trình nhỏ để quản lý và được kiểm thử độc lập. Công cụ cơ bản để chia một chương trình thành các phần nhỏ là khả năng tạo ra các thay thế được đặt tên hay là các chương trình con. Trong C và

C++ một chương trình con được gọi là một hàm , và các hàm là các đoạn mã có thể được thay thế trong các file khác nhau , cho phép thực hiện quá trình biên dịch độc lập . Nói một cách khác các hàm là các đơn vị nguyên tử của mã nguồn , vì chúng ta không thể đặt các phần khác nhau của hàm trong các file khác nhau nên nội dung của một hàm cần phải được đặt hoàn toàn trong một file (mặc dù các file có thể chứa nhiều hơn 1 hàm).

Khi chúng ta gọi đến một hàm , chúng ta thường truyền cho nó một vài tham số , đó là các giá trị mà chúng ta muốn hàm làm việc với khi nó thực hiện . Khi hàm thực hiện xong chúng ta thường nhận được một giá trị trả về , một giá trị mà hàm trả lại như là một kết quả. Cũng có thể viết các hàm không nhận các tham số và không trả về bất kỳ giá trị nào.

Để tạo ra một chương trình với nhiều file , các hàm trong một file phải truy cập tới các hàm và dữ liệu trong các file khác . Khi biên dịch một file , trình biên dịch C hoặc C++ phải biết về các hàm và dữ liệu trong các file khác đặc biệt là tên và cách dùng chúng. Trình biên dịch đảm bảo các hàm và dữ liệu được sử dụng đúng đắn . Quá trình báo cho trình biên dịch tên và nguyên mẫu của các hàm và dữ liệu bên ngoài được gọi là khai báo (*declaration*). Khi chúng ta đã khai báo một hàm hoặc biến trình biên dịch sẽ biết cách thức kiểm tra để đảm bảo các hàm và dữ liệu này được sử dụng đúng đắn .

Including các file Header

Hầu hết các thư viện đều chứa một số lượng đáng kể các hàm và biến . Để tiết kiệm công sức và đảm bảo sự nhất quán khi khai báo ngoài các phần tử này , C và C++ đã sử dụng một loại file được gọi là file header. Mỗi file header là một file chứa các khai báo ngoài cho 1 thư viện; theo qui ước các file này có phần mở rộng là .h, nhưng chúng ta cũng có thể dùng các đuôi file khác cho chúng chẳng hạn như .hpp hoặc .hxx.

Lập trình viên tạo ra các file thư viện sẽ cung cấp các header file . Để khai báo các hàm và các biến bên ngoài thư viện người dùng chỉ cần thực hiện include file header đó. Để include một file header chúng ta sử dụng chỉ thị tiền xử lý **#include**. Chỉ thị này sẽ báo cho bộ xử lý mở file header có tên tương ứng và chèn nội dung của file đó vào chỗ mà chỉ thị #include được sử dụng. Tên file sử dụng sau chỉ thị #include có thể nằm giữa hai dấu < và > hoặc giữa hai dấu “ ”.

Ví dụ: #include <header.h>

Nếu chúng ta sử dụng chỉ thị include theo cách trên thì bộ tiền xử lý sẽ tìm file header theo cách đặc thù đối với cài đặt của chúng ta , nhưng thường thì sẽ có một vài đường dẫn mà chúng ta chỉ định cụ thể trong biến môi trường của trình biên dịch hoặc trên dòng lệnh để sử dụng cho việc tìm các file header. Cơ chế thiết lập các đường dẫn này phụ thuộc vào trình biên dịch và môi trường mà chúng ta làm việc .

Ví dụ: #include “header.h”

Chỉ thị tiền xử lý như trên thường có ý nghĩa là báo cho bộ tiền xử lý tìm file tương ứng trong thư mục hiện tại trước nếu không thấy thì sẽ tìm giống như trong trường hợp tên file include được đặt giữa hai dấu < và >.

Nói chung thì đối với các file include chuẩn hoặc được sử dụng nhiều chúng ta nên đặt nó trong thư mục mặc định là include dưới thư mục cài đặt trình biên dịch và dùng chỉ thị theo kiểu <>, còn đối với các file đặc thù với ứng dụng cụ thể thì dùng kiểu tên file đặt giữa hai dấu “ ”.

Trong quá trình phát triển của C++ các nhà cung cấp các trình biên dịch có các qui ước đặt tên khác nhau và các hệ điều hành lại có các hạn chế tên khác nhau đặc biệt là độ dài của tên file . Các vấn đề này gây ra các vấn đề về tính khả chuyển của chương

trình. Để khắc phục vấn đề này người ta đã sử dụng một định dạng chuẩn cho phép các tên file header có thể dài hơn 8 ký tự và bỏ đi phần tên mở rộng.

Để phân biệt một chương trình C và C++ đôi khi người ta còn dùng cách thêm một ký tự “c” vào trước tên của các file header, chi tiết này cũng được chấp nhận đối với C và C++.

Quá trình liên kết

Trình liên kết tập hợp các module object (thường là các file có phần mở rộng là .o hoặc .obj), được sinh ra bởi trình biên dịch, thành một chương trình có thể thực hiện được và hệ điều hành có thể nạp vào bộ nhớ và chạy. Đây là pha cuối cùng trong quá trình biên dịch.

Các đặc điểm của các trình liên kết thay đổi phụ thuộc vào các hệ thống khác nhau. Nói chung chúng ta chỉ cần chỉ rõ cho trình liên kết biết tên của các module object và các thư viện mà chúng ta muốn liên kết, và tên của chương trình khả chạy cuối cùng. Một vài hệ thống đòi hỏi chúng ta cần phải tự gọi tới các trình liên kết. Tuy nhiên hầu hết các trình biên dịch hoàn chỉnh đều thực hiện hộ chúng ta công việc này.

Sử dụng các thư viện

Giờ đây chúng ta đã biết các thuật ngữ cơ bản, chúng ta có thể hiểu cách thức sử dụng một thư viện. Để sử dụng một thư viện cần phải:

- Include file header của thư viện
- Sử dụng các hàm và các biến trong thư viện
- Liên kết thư viện vào chương trình khả chạy cuối cùng

Các bước này cũng đúng với các module object không có trong các thư viện. Including một file header và liên kết các module object là các bước cơ bản để thực hiện việc biên dịch độc lập trong C và C++.

Trình liên kết làm thế nào để tìm một file thư viện

Khi chúng ta tạo ra một tham chiếu ngoài tới một hàm số hoặc một biến số trong C hoặc C++, trình liên kết, khi bắt gặp tham chiếu này, có thể thực hiện một trong hai việc sau: nếu nó chưa thấy phần định nghĩa của hàm hay biến này, nó sẽ thêm định danh vào danh sách các tham chiếu chưa được định nghĩa của nó. Nếu như trình liên kết đã bắt gặp định nghĩa của tham chiếu đó, tham chiếu sẽ được sắp xếp lại.

Nếu như trình liên kết không tìm thấy định nghĩa của tham chiếu trong danh sách các module object nó sẽ tiến hành tìm kiếm trong các thư viện. Các thư viện có một vài loại chỉ số nên trình liên kết không cần thiết phải tìm kiếm hết trong các module object của thư viện – nó chỉ cần xem xét các phần chỉ mục. Khi trình liên kết tìm thấy một định nghĩa trong một thư viện, toàn bộ module object chứ không chỉ phần định nghĩa của hàm, sẽ được liên kết vào chương trình thực hiện. Chú ý rằng toàn bộ thư viện sẽ không được liên kết, chỉ có phần định nghĩa mà chương trình tham chiếu tới. Như vậy nếu chúng ta muốn tối ưu về kích thước của chương trình chúng ta có thể cho mỗi hàm vào một file khi xây dựng các thư viện riêng của mình. Điều này đòi hỏi công sức edit nhiều hơn nhưng cũng có thể có ích.

Vì trình liên kết tìm kiếm các file theo thứ tự chúng ta có thể che đi sự tồn tại của một hàm thư viện bằng cách dùng hàm của chúng ta với phần định nghĩa và prototype y hệt như hàm thư viện. Tuy nhiên điều này cũng có thể gây ra các lỗi mà chúng ta không thể kiểm soát được.

Khi một chương trình khả chạy được viết bằng C hoặc C++ được tạo ra, một số các thành phần nhất định sẽ được liên kết với nó một cách bí mật. Một trong các thành phần này chính là module khởi động (startup), module này chứa các thủ tục khởi tạo cần phải được thực hiện bất cứ khi nào một chương trình C hay C++ bắt đầu chạy. Các thủ tục này thiết lập stack và các biến khởi tạo nhất định trong chương trình.

Trình biên dịch luôn thực hiện việc tìm kiếm trong các thư viện chuẩn để thực hiện liên kết các hàm chuẩn mà chúng ta dùng trong chương trình nên để dùng các hàm trong các thư viện chuẩn chúng ta đơn giản chỉ cần include file header của thư viện đó. Còn đối với các thư viện riêng do chúng ta tạo ra chúng ta cần chỉ rõ tên thư viện cho trình liên kết (chẳng hạn thư viện graphics không phải là một thư viện chuẩn).

1.2 Chương trình đầu tiên.

Cách tốt nhất để học lập trình là xem các chương trình của người khác viết và học tập các kỹ thuật lập trình của họ. Sau đây là chương trình HelloWorld được viết bằng C++, một chương trình mà hầu hết các sách lập trình đều lấy làm ví dụ mở đầu.

```
// Chương trình HelloWorld
// File hello.cpp
// In ra màn hình xâu "Hello, World!"

#include <iostream.h> // Khai báo luồng cout để sử dụng

int main() {
    cout << "Hello, World! I am "
    << 20 << " today." endl;
    return 0;
}
```

Điều đầu tiên chúng ta cần biết là một chương trình C hoặc C++ là một tập các hàm, biến và các lời gọi hàm. Khi chương trình thực hiện nó sẽ gọi đến một hàm đặc biệt mà bất cứ chương trình nào cũng có đó là hàm main.

Về mặt thuật toán và nội dung chương trình này không có gì đặc biệt, nó in ra màn hình một dòng chào mừng: "Hello, World!". Chúng ta sẽ lần lượt khám phá các đặc điểm của C++ qua các câu lệnh của chương trình đơn giản này.

Hai dòng đầu tiên của chương trình là hai dòng chú thích, giới thiệu về chức năng của chương trình. C++ chấp nhận kiểu viết chú thích theo kiểu của C:

```
/*
chú thích có thể gồm nhiều dòng
*/
```

Nhưng đưa ra một kiểu chú thích khác tiện lợi hơn là:

```
// chú thích, chú ý là chú thích này chỉ nằm trên một dòng
```

Một số lập trình viên thích dùng kiểu chú thích của C++ hơn vì như thế dễ dàng phân biệt một chương trình C với một chương trình C++. Mặc dù đây không phải là một qui tắc bắt buộc song chúng ta nên dùng kiểu thứ hai và nếu có dùng kiểu thứ nhất thì cần phải theo một qui luật nhất định.

Tiếp theo là một chỉ thị tiền xử lý #include. Ở đây chúng ta include file header iostream chứa các dòng vào ra chuẩn của C++. Thường khi chúng ta include một file header chúng ta nên có kèm một vài chú thích ngắn gọn về mục đích của file đó, chẳng

hạn ở đây chúng ta include file header iostream là vì cần sử dụng đối tượng cout trong thư viện iostream.

Tiếp theo là hàm main () có kiểu trả về là int và không nhận tham số nào. Giống như C tất cả các chương trình C++ đều có một và duy nhất một hàm main () và nếu chúng ta không nói gì có nghĩa là hàm main sẽ trả về một giá trị có kiểu int nên để tránh một vài rắc rối chúng ta nên xác định kiểu của hàm main là int và trả về 0 trước khi kết thúc hàm. Prototype của hàm main là : int main() có nghĩa là hàm này có thể nhận bất bao nhiêu tham số tùy ý.

Trong câu lệnh tiếp theo chúng ta sử dụng đối tượng **cout** (*console output*) để in ra một loạt các tham số thông qua các toán tử “<<”. Chúng ta đã biết trong ngôn ngữ C toán tử “<<” là toán tử dịch bit trái nhưng trong C++ ngoài ý nghĩa là một toán tử dịch bit trái nó còn là một toán tử của đối tượng **cout**, đó chính là một minh họa cho khả năng overload các toán tử của C++ mà chúng ta sẽ học sau này. Cũng cần chú ý là câu lệnh này được viết trên nhiều dòng, C++ cho phép một câu lệnh có thể viết trên nhiều dòng. Trình biên dịch nhận biết sự kết thúc một câu lệnh trong C++ bằng cách nhận biết sự có mặt của các dấu “;”.

endl là một hàm đặc biệt thuộc thư viện các luồng vào ra chuẩn nó kết thúc dòng hiện tại của **cout** là nhảy xuống dòng tiếp theo.

Đối tượng cout có khả năng xử lý nhiều tham số tương ứng với các toán tử “<<”. Nó xem các tham số đó như là một dãy các ký tự, nếu là các kiểu dữ liệu khác (ngoài kiểu xâu: các ký tự giữa hai dấu “” và “”) cout sẽ có hai cách thức xử lý. Thứ nhất nếu đó là các kiểu cơ bản chúng sẽ được chuyển thành một dãy các ký tự giữa hai dấu “”, còn nếu là một kiểu tự định nghĩa (lớp hoặc struct) thì có thể sẽ gọi tới hàm overload toán tử của kiểu đó “<<”.

Dòng lệnh cuối cùng là câu lệnh return 0 để phù hợp với prototype của hàm main được khai báo ban đầu.

Để tiến hành biên dịch chương trình trên chúng ta thực hiện lệnh:

Tcc -eHello hello.cpp. Kết quả nhận được là một file khả chạy có tên là hello.exe.

2. Biến, hằng và tầm hoạt động của các biến

2.1 Cú pháp khai báo biến (variable declaration)

Ý nghĩa của cụm từ “variable declaration” đã từng có những ý nghĩa trái ngược nhau và gây nhầm lẫn trong lịch sử, và việc hiểu đúng định nghĩa của cụm từ này là rất quan trọng trong việc hiểu đúng đắn mã chương trình. Một khai báo biến sẽ báo cho trình thông dịch biết các đặc điểm của một biến được khai báo. Mặc dù có thể đó là lần đầu tiên trình biên dịch bắt gặp biến đó trong quá trình biên dịch nhưng một khai báo biến đúng đắn sẽ đảm bảo rằng biến đó là tồn tại (đâu đó trong bộ nhớ) và nó là một biến có kiểu X.

Cú pháp khai báo biến hợp lệ trong C++ là:

<kiểu biến> tên biến;

Trong đó “kiểu biến” là một kiểu dữ liệu hợp lệ và tên biến là một tên hợp lệ theo như định nghĩa trong C.

Ví dụ: int a;

Khi gặp một khai báo như trên trong quá trình biên dịch, trình biên dịch sẽ ngay lập tức tạo ra một vùng nhớ (có thể có thêm giá trị khởi tạo) của biến kiểu số nguyên và gán nhãn là a (xác định hay định nghĩa biến). Tuy nhiên đôi khi chúng ta chỉ muốn đơn giản khai báo một biến là tồn tại (ở đâu đó trong toàn bộ chương trình chứ không

muốn ngay lập tức định nghĩa biến đó). Để giải quyết trường hợp này chúng ta sẽ dùng từ khóa **extern**, ví dụ:

```
extern int a;
```

Khai báo này sẽ báo cho trình biên dịch biết rằng biến có tên là a là tồn tại và nó đã hoặc sẽ được định nghĩa đâu đó trong chương trình.

Ví dụ:

```
// file: Declare.cpp
// Ví dụ khai báo và định nghĩa biến
extern int i; // khai báo và không định nghĩa
float b; // khai báo và định nghĩa
int i; // định nghĩa biến i

int main() {
    b = 1.0;
    i = 2;
}
```

Các biến có thể được khai báo ở bất kỳ một vị trí nào trong chương trình, điều này có đôi chút khác biệt so với các chương trình C.

2.2 Tầm hoạt động của các biến

Khái niệm tầm hoạt động của các biến cho chúng ta biết khu vực (phần chương trình) mà một biến nào đó có thể được sử dụng hợp lệ và khu vực nào thì việc truy cập tới một biến là không hợp lệ. Tầm hoạt động của một biến bắt đầu từ vị trí mà nó được khai báo cho tới dấu “}” đầu tiên khớp với dấu “{” ngay trước khai báo của biến đó. Có nghĩa là tầm hoạt động của một biến được xác định là trong cặp “{” và “}” gần nhất bao nó. Tất nhiên tầm hoạt động của các biến có thể chồng lên nhau.

2.3 Khai báo biến ngay trong cú pháp của các câu lệnh điều khiển

Như chúng ta đã biết trong các chương trình C++ việc khai báo biến là khá tự do. Các biến có thể được khai báo ở bất kỳ vị trí hợp lệ nào của chương trình miễn là chúng phải là xác định trước khi được sử dụng.

Trong ngôn ngữ C và hầu hết các ngôn ngữ thủ tục khác lập trình viên bắt buộc phải khai báo các biến tại phần đầu tiên của mỗi thủ tục. Do đó khi đọc các file mã nguồn C chúng ta luôn thấy một loạt khai báo các biến sẽ được dùng mỗi thủ tục ở phần đầu của thủ tục. Điều này sẽ rất bất tiện khi một thủ tục có nhiều biến hoặc dài vì việc kiểm soát biến (tên, giá trị khởi tạo, tầm hoạt) sẽ trở nên khó khăn.

Đi xa hơn cả việc cho phép khai báo bất kỳ vị trí nào hợp lệ trong chương trình C++ còn cho phép khai báo và khởi tạo các biến ngay bên trong biểu thức điều khiển của các vòng lặp **for**, **while**, **do** hoặc trong câu lệnh **if**, **switch**. Ví dụ:

```
for(int i=0;i<10;i++){
    ....
}
while(char c = cin.get() != 'q'){
    ....
}
if(char x = c == 'a' || c == 'b'){
    ....
}
```

```
switch(int i=cin.get()){
    case 'A': ...; break;
    .....
}
```

Mặc dù vậy việc khai báo như trên chỉ thường được dùng với các vòng lặp for vì đôi khi nó gây ra một số lỗi. Ví dụ câu lệnh:

```
while( (char c = cin.get()) !='q' ){
}
```

sẽ làm chúng ta ngạc nhiên với kết quả nhận được . Vì toán tử != có độ ưu tiên cao hơn toán tử gán = nên c sẽ nhận một giá trị có kiểu Bool và sau đó mới được convert sang kiểu char.

2.4 Các kiểu biến

Biến toàn cục (global variable)

Các biến toàn cục được định nghĩa bên ngoài tất cả các hàm và có thể được sử dụng trong tất cả các phần của chương trình (thậm chí ngay cả phần chương trình nằm trong một file mã nguồn khác). Các biến toàn cục không bị ảnh hưởng bởi các tầm hoạt động (chúng tồn tại cho tới khi chương trình kết thúc).

Khi cần tham chiếu tới các biến toàn cục trong một file mà nó chưa được khai báo (biến này được khai báo trong một file khác) chúng ta sử dụng từ khóa extern để chỉ ra rằng biến đó là một biến toàn cục được khai báo trong file khác .

Biến cục bộ (hay địa phương, local)

Các biến địa phương thường được khai báo trong một phạm vi hay tầm hoạt động nhất định, thường là trong một hàm. Các biến địa phương này còn được gọi là các biến tự động vì chúng ta có thể sử dụng chúng một cách tự nhiên trong tầm hoạt động của chúng và bản thân chúng cũng tự động “out of scope” bên ngoài phạm vi hoạt động . Chúng ta có thể sử dụng từ khóa auto để làm rõ hơn điều này.

Biến thanh ghi (register variable)

Các biến thanh ghi là một loại biến cục bộ . Để khai báo các biến thanh ghi chúng ta dùng từ khóa **register**. Mục đích của việc khai báo các biến register là báo cho trình biên dịch biết để nó có thể làm cho việc truy cập vào các biến này với tốc độ càng nhanh càng tốt. Việc tăng tốc độ truy cập biến là phụ thuộc vào cài đặt tùy nhiên như ngụ ý của từ register điều này thường được thực hiện bằng cách đặt biến vào một thanh ghi. Không có gì đảm bảo là biến được khai báo là register sẽ được đặt trong một thanh ghi hoặc thậm chí tốc độ truy cập sẽ nhanh hơn . Đó chỉ là một gợi ý cho trình biên dịch.

Không thể thực hiện các biến thanh ghi kiểu này , chúng cũng chỉ có thể là các biến địa phương, không thể là các biến toàn cục hoặc các biến tĩnh và nói chung chúng ta nên tránh dùng chúng.

Biến tĩnh (static variable)

Các biến tĩnh được khai báo bằng từ khóa static . Bình thường đối với một biến được khai báo cục bộ trong một hàm số , nó sẽ tự động bị loại bỏ khỏi bộ nhớ khi hàm được gọi thực hiện xong . Khi hàm được gọi thực hiện lại lần nữa , các biến cục bộ lại được khởi tạo lại và c ứ thế. Tuy nhiên đôi khi chúng ta muốn lưu lại các giá trị của một biến số đã có được trong các lần gọi thực hiện trước của hàm , khi đó việc dùng

biến static là hợp lý . Các biến static chỉ được khởi tạo lần đầu tiên khi hàm được gọi tới lần đầu tiên . Chúng ta có thể băn khoăn tự hỏi là vậy tại sao không dùng các biến toàn cục câu trả lời là các biến static có tầm hoạt động trong một thân hàm do đó chúng ta có thể thu hẹp các lỗi liên quan tới việc sử dụng biến này , có nghĩa khả năng lỗi là thấp hơn so với dùng biến toàn cục.

Ngoài ý nghĩa trên từ khóa static thường có một ý nghĩa khác đó là “không thể sử dụng ngoài một phạm vi nhất định” . Khi từ khóa static được dùng để khai báo một tên hàm hoặc một biến nằm ngoài tất cả các hàm trong một file mã nguồn thì có nghĩa là biến đó chỉ có tầm hoạt động trong file đó mà thôi . Khi đó chúng ta nói là biến đó có tầm hoạt động file.

2.5 Liên kết biến khi biên dịch

Để hiểu cách thức hoạt động của các chương trình C và C++ chúng ta cần phải hiểu quá trình liên kết diễn ra như thế nào . Có hình thức liên kết các biến khi biên dịch : liên kết trong và liên kết ngoài.

Liên kết trong có nghĩa là bộ nhớ (vùng lưu trữ) được tạo ra để biểu diễn định danh chỉ cho file đang được biên dịch . Các file khác có thể sử dụng định danh đó đối với liên kết trong , hoặc với một biến toàn cục . Liên kết trong thường được thực hiện với các biến static.

Liên kết ngoài có nghĩa là mỗi vùng nhớ được tạo ra để biểu diễn định danh cho tất cả các file đang được biên dịch. Các vùng nhớ này chỉ được tạo ra một lần và trình liên kết phải sắp xếp lại tất cả các tham chiếu tới vùng nhớ đó . Các tên hàm và các biến toàn cục có các liên kết ngoài và chúng có thể được truy cập trong các file khác bằng cách khai báo bằng từ khóa extern. Các biến định nghĩa ngoài các hàm (trừ các const) và các định nghĩa hàm là mặc định đối với liên kết ngoài . Chúng ta có thể buộc chúng thực hiện các liên kết trong bằng từ khóa static và chỉ rõ liên kết ngoài bằng từ khóa extern.

Các biến cục bộ chỉ được sử dụng tạm thời , trên stack khi các hàm được gọi tới . Trình liên kết không biết tới chúng và do đó không có quá trình liên kết nào được thực hiện.

2.6 Các hằng

Trong các trình biên dịch C cổ điển chúng ta có thể khai báo các hằng bằng cách sử dụng chỉ thị tiền xử lý, ví dụ:

```
#define PI 3.14159
```

Khi đó trong quá trình tiền xử lý bộ tiền xử lý sẽ thực hiện thay thế tất cả các ký hiệu PI mà nó gặp trong các file mã nguồn bằng giá trị 3.14159.

Chúng ta vẫn có thể sử dụng cách này trong C++ tuy nhiên có rất nhiều vấn đề đối với kiểu khai báo này . Chúng ta không thể thực hiện kiểm tra kiểu đối với PI , không thể lấy địa chỉ của PI (vì thế không thể dùng con trỏ trỏ vào biến này). PI cũng không thể là một biến có kiểu người dùng định nghĩa . Cũng không thể xác định được tầm hoạt động của PI.

C++ sử dụng từ khóa const để khai báo các hằng, cú pháp khai báo giống như khai báo biến chỉ khác là giá trị của hằng là không thay đổi.

Các hằng trong C++ đều phải khởi tạo trước khi sử dụng . Các giá trị hằng cho các kiểu built-in được biểu diễn như là các số thập phân, bát phân, số hexa hoặc các số dấu phẩy động (đáng buồn là các số nhị phân được cho là không quan trọng) hoặc là các ký tự.

Nếu không có các chỉ dẫn khai báo nào khác các hằng được coi là các số thập phân. Các hằng bắt đầu bởi số 0 được xem là các hằng trong hệ bát phân, còn 0x là các hằng trong hệ hexa. Các hằng dấu phẩy động được biểu diễn bởi phần thập phân và dạng mũ hóa ví dụ: 1e4, 1.4e4. Chúng ta có thể thêm các hậu tố f, F, L, l để chỉ rõ kiểu của các hằng loại này.

Các hằng ký tự được biểu diễn giữa hai dấu ' , nếu là ký tự đặc biệt thì có thêm dấu \ đứng trước.

Biến kiểu volatile

Trong khi từ khóa const có nghĩa là biến không thay đổi giá trị thì khai báo biến với từ khóa volatile có nghĩa là chúng ta không biết biến này sẽ thay đổi lúc nào và do đó trình biên dịch sẽ không thực hiện các tối ưu hóa dựa trên giả thiết về sự ổn định của biến này. Một biến volatile sẽ được đọc vào khi mà giá trị của nó được cần đến.

Một trường hợp đặc biệt của các biến volatile là khi chúng ta viết các chương trình đa luồng. Ví dụ khi chúng ta đang chờ đợi một cờ nào đó đang được xử lý bởi một luồng khác thì biến cờ đó bắt buộc phải là volatile.

Các biến volatile không có ảnh hưởng gì tới chương trình nếu chúng ta không thực hiện tối ưu hóa nó nhưng sẽ có thể có các lỗi rất tinh vi khi chúng ta tiến hành tối ưu hóa chương trình.

3. Hàm trong C++

Trong ngôn ngữ C cổ (không phải là ngôn ngữ C chuẩn mà chúng ta dùng hiện nay) chúng ta có thể thực hiện việc gọi hàm với số lượng tham số cũng như kiểu tham số tùy ý mà trình biên dịch sẽ không phản nản gì cả. Tất cả dường như đều tốt cho tới khi chúng ta chạy chương trình. Có thể chúng ta sẽ nhận được các kết quả rất khó hiểu mà không có bất cứ một dấu hiệu hay gợi ý nào về chúng. Đây có lẽ là một trong các lý do làm cho C trở thành một ngôn ngữ được đánh giá là ngôn ngữ Assembly cấp cao.

Ngôn ngữ C chuẩn và C++ ngày nay có một cơ chế gọi là nguyên mẫu hay bản mẫu hàm (function prototype). Với cơ chế này chúng ta cần khai báo kiểu của các tham số của hàm, kiểu của hàm khi khai báo và định nghĩa chúng. Sự khai báo hay mô tả rõ ràng này được gọi là biểu mẫu của hàm. Khi hàm được gọi trình biên dịch sẽ sử dụng biểu mẫu của hàm để kiểm tra xem các tham số được truyền có đúng kiểu, số lượng cũng như giá trị trả về của hàm có được xử lý đúng hay không. Nếu như có các lỗi trong quá trình kiểm tra xảy ra trình biên dịch sẽ thông báo ngay cho lập trình viên biết trong quá trình biên dịch.

Cú pháp khai báo một hàm như sau:

<kiểu trả về của hàm> <tên hàm>(<danh sách các kiểu tham số và tên ngăn cách nhau bởi dấu phẩy>);

Ví dụ: int max(int x, int y);

Về bản chất chúng ta không cần có các tên tham biến, chúng chỉ thực sự cần khi chúng ta sử dụng chúng trong việc định nghĩa các hàm. Tuy nhiên điều này cũng không phải là bắt buộc đối với C++ (trong C là bắt buộc). Chúng ta có thể có một tham số nào đó không có tên và nó sẽ không được sử dụng trong thân hàm (tất nhiên vì nó không có tên). Khi chúng ta gọi tới hàm đó chúng ta vẫn phải truyền đúng các tham số. Tuy nhiên tác giả của hàm đó sau đó vẫn có thể sử dụng tham số ở đâu đó mà không cần thiết phải thay đổi các lời gọi hàm. Điều này rất tiện khi chúng ta không muốn có các lời cảnh báo về việc không sử dụng một tham số nào đó trong thân hàm.

C và C++ có hai cách khác nhau để định nghĩa danh sách các tham số . Nếu chúng ta có một hàm func(), C++ sẽ hiểu là hàm này không có tham số , C lại hiểu là hàm này có thể có bất kỳ tham số nào . Một hàm func (void) sẽ được hiểu là không có tham số trong cả C và C++.

Một trường hợp nữa xảy ra là khi chúng ta không xác định được số tham số cũng như kiểu tham số của hàm mà chúng ta muốn khai báo (gọi là một danh sách tham biến: variable argument list). Khi đó chúng ta sẽ sử dụng ký pháp (...). Tuy nhiên nên hạn chế sử dụng nó trong C ++, chúng ta có nhiều cách khác để đạt được kết quả này mà không cần tới ký pháp đó.

Các giá trị trả về của hàm

Trong nguyên mẫu hàm chúng ta buộc phải chỉ rõ kiểu của hàm , nếu một hàm không có kiểu trả về thì kiểu của nó là void . Trong mỗi một thân hàm có kiểu bao giờ cũng có ít nhất một câu lệnh return . Khi gặp lệnh này trong quá trình thực hiện, hàm sẽ kết thúc. Trong các hàm không kiểu cũng có thể dùng return để thoát khỏi hàm .

Một trong các điểm mạnh của ngôn ngữ C và C ++ là một thư viện hàm rất phong phú và linh hoạt . Để sử dụng chúng , lập trình viên cần thực hiện include các file header chứa các prototype của chúng trong chương trình , phần còn lại sẽ tự do trình biên dịch và trình liên kết thực hiện.

Chúng ta có thể tạo ra các thư viện hàm riêng cho mình để sử dụng . Tuy nhiên hãy xem kỹ phần manual của trình biên dịch trước khi thực hiện .

4. Các cấu trúc điều khiển

Các câu lệnh điều khiển là điều mà mọi lập trình viên cần phải biết trước khi viết bất cứ một chương trình nào . Chúng ta có các câu lệnh điều khiển : **if-else, while, do, do-while, for** và câu lệnh lựa chọn **switch**.

Các câu lệnh điều kiện dựa trên kết quả đúng hoặc sai của một biểu thức điều kiện để xác định đường đi của chương trình . Trong C++ hai từ khóa true và false đã được đưa vào để biểu thị cho kết quả đúng hoặc sai của một biểu thức điều kiện , tuy nhiên các qui ước cũ vẫn có thể được dùng : một giá trị bất kỳ khác 0 sẽ được coi là đúng và một giá trị bằng 0 có nghĩa là sai.

4.1 Câu lệnh if-else

Câu lệnh điều kiện **if – else** có thể được sử dụng dưới hai dạng khác nhau : có hoặc không có phần mệnh đề **else**. Cú pháp của hai dạng này như sau:

if(expression)

statement

hoặc

if(expression)

statement

else

statement

Biểu thức expression cho một giá trị true hoặc false . Phần câu lệnh “statement” có thể là một câu lệnh đơn kết thúc bằng một dấu chấm phẩy cũng có thể là một câu lệnh hợp thành , một tập các câu lệnh đơn giữa hai dấu { và }. Chú ý là phần câu lệnh statement cũng có thể bao gồm các câu lệnh điều kiện **if – else**.

4.2 Vòng lặp không xác định while

Cú pháp:

while (expression)

statement

Biểu thức được tính toán lần đầu tiên tại thời điểm bắt đầu của vòng lặp và sau đó được tính lại mỗi khi lặp lại quá trình thực hiện câu lệnh. Điều kiện để dừng vòng lặp không xác định while là giá trị của biểu thức expression bằng false.

Như vậy điều cần chú ý ở đây là câu lệnh trong thân vòng lặp có thể không được thực hiện trong trường hợp biểu thức điều kiện cho giá trị false ngay lần đầu tính toán.

Đôi khi chúng ta không cần sử dụng biểu thức điều kiện để kết thúc vòng lặp while, đó cũng là trường hợp đơn giản nhất của biểu thức điều kiện.

4.3 Vòng lặp không xác định do – while

Sự khác biệt của vòng lặp do – while so với vòng lặp while là vòng lặp do – while thực hiện ít nhất một lần ngay cả khi biểu thức điều kiện cho giá trị false trong lần tính toán đầu tiên. Cú pháp của vòng lặp **do – while**:

do

Statement

while(expression);

Vì một vài lý do các lập trình viên thường ít sử dụng vòng lặp do – while hơn so với vòng lặp while.

4.4 Vòng lặp xác định for

Vòng lặp for thực hiện một thao tác khởi tạo trước khi thực hiện lần lặp đầu tiên. Sau đó thực hiện quá trình kiểm tra điều kiện thực hiện của vòng lặp, và tại cuối mỗi lần thực hiện của vòng lặp thực hiện một thao tác nhảy qua một số giá trị nào đó của biến điều khiển. Cú pháp:

for(initialization; conditional; step)

statement

Bất kỳ biểu thức nào trong các biểu thức initialization, conditional và step đều có thể là các biểu thức rỗng tuy nhiên trong trường hợp đó cần giữ lại các dấu chấm phẩy. Biểu thức khởi tạo chỉ được thực hiện lần đầu tiên trước khi vòng lặp được thực hiện. Biểu thức conditional sẽ được kiểm tra mỗi khi vòng lặp thực hiện và nếu nó nhận giá trị false ngay lần đầu tiên thân của vòng lặp sẽ không được thực hiện. Tại thời điểm kết thúc của thân vòng lặp biểu thức step sẽ được thực hiện.

Tuy có sự khác nhau so ng về bản chất các vòng lặp for, while và do – while có sự tương đồng và chúng đều có thể chuyển đổi cho nhau. Vòng lặp for được sử dụng nhiều hơn do một số nguyên nhân sau:

Trong vòng lặp for có sự khởi tạo ban đầu, đồng thời nó giữ cho các câu lệnh gần nhau hơn và dễ thấy từ đỉnh chu trình, đặc biệt là khi chúng ta có nhiều chu trình lồng nhau. Ví dụ:

Thuật toán sắp xếp **Shell – sort**: Ý tưởng của thuật toán này là ở mỗi bước thay vì so sánh và đổi chỗ hai phần tử kề nhau như trong phương pháp sắp xếp đơn giản chúng ta sẽ so sánh và đổi chỗ hai phần tử cách xa nhau. Điều này hướng tới việc loại bỏ quá nhiều sự mất trật tự một cách nhanh chóng cho nên ở các giai đoạn sau còn ít công việc phải làm. Khoảng cách giữa các phần tử so sánh cũng được giảm dần tới một lúc việc sắp xếp trở thành việc đổi chỗ hai phần tử kề nhau.

```
void shellSort(int * a, int n){  
    int gap, i, j, temp;
```

```

for(gap = n/2; gap > 0; gap /= 2)
    for(i = gap; i < n; i++)
        for(j = i-gap; j >=0 && a[i] > a[i + gap]; j -= gap){
            temp = a[j];
            a[j] = a[j + gap];
            a[j + gap] = temp;
        }
}

```

Một chú ý thứ hai là các toán tử đầu phẩy cũng thường được sử dụng với các biểu thức trong phần điều khiển của vòng lặp for ví dụ như khi để điều khiển nhiều biến chỉ số chẳng hạn:

```

char * reverse(char *s){
    int c, i, j;
    for(i = 0, j = strlen(s) - 1; i < j; i++, j--)
    {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

```

4.5 Các từ khóa break và continue

Chúng ta có thể thực hiện điều khiển việc thực hiện trong thân các vòng lặp bằng các câu lệnh **break** và **continue**. Câu lệnh **break** sẽ thoát khỏi thân vòng lặp và không thực hiện phần còn lại, câu lệnh continue quay trở lại thực hiện bước lặp tiếp theo (bỏ qua phần các câu lệnh nằm sau nó trong vòng lặp). Lệnh break là cần thiết để thoát khỏi các vòng lặp mà điều kiện thực hiện luôn luôn đúng chẳng hạn như while(true)....

4.6 Câu lệnh lựa chọn switch

Câu lệnh switch lựa chọn thực hiện các câu lệnh trong một nhóm các câu lệnh dựa trên giá trị của một biểu thức nguyên. Cú pháp của nó như sau:

```

switch(biến lựa chọn)
{
    case integral_value1: statement; break;
    case integral_value2: statement; break;
    (...)
    default: statement;
}

```

Biểu thức lựa chọn sau khi thực hiện sẽ cho một giá trị nguyên. Giá trị nguyên này được so sánh với mỗi giá trị hoặc một số giá trị nguyên, nếu trùng khớp câu lệnh tương ứng sẽ được thực hiện. Nếu không có sự trùng khớp nào xảy ra câu lệnh trong phần default sẽ được thực hiện.

Chú ý rằng câu lệnh break ngay sau mỗi phần lựa chọn case có thể không cần sử dụng tuy nhiên khi đó các câu lệnh tiếp sau lựa chọn đó sẽ được thực hiện cho tới khi gặp phải một lệnh break. Nếu có lệnh break các câu lệnh tiếp sau lựa chọn sẽ không được thực hiện, chương trình sẽ thoát khỏi thân lệnh switch.

4.7 Câu lệnh goto

Câu lệnh goto cũng là một câu lệnh cơ bản của C++ vì nó có trong C. Nói chung là chúng ta nên tránh dùng goto tuy vậy có một số trường hợp việc dùng goto cũng có thể chấp nhận được như khi chúng ta muốn thoát hoàn toàn ra khỏi tất cả các vòng lặp lồng nhau từ vòng lặp trong cùng.

4.8 Đệ qui

Đệ qui là một kỹ thuật thường được dùng để giải quyết các vấn đề có độ phức tạp không xác định khi mà chúng ta thường không cần phải lo lắng về kích thước bộ nhớ cần sử dụng. Các bài toán đệ qui thường được giải quyết theo chiến lược chia để trị.

5. Các kiểu dữ liệu cơ bản của C++

Các kiểu dữ liệu cơ bản của C++ hầu hết đều kế thừa của C ngoại trừ kiểu bool với hai hằng số true và false.

Đặc tả của ngôn ngữ C chuẩn cho các kiểu dữ liệu built-in không chỉ rõ cụ thể các kiểu dữ liệu này cần bao nhiêu bit. Thay vào đó nó qui định các giá trị max và min các bit mà mỗi kiểu dữ liệu có thể chứa. Khi đó tùy thuộc vào hệ thống nền mà chúng ta sử dụng các biến của cùng một chương trình sẽ có kích thước khác nhau khi biên dịch trên các hệ thống khác nhau. Ví dụ với các chương trình chạy trên DOS các biến kiểu int sẽ có kích thước là 2 byte tức 16 bit nhưng trên Linux kiểu int sẽ là 32 bit tức 4 byte. Các giá trị giới hạn này được định nghĩa trong hai file header hệ thống là limit.h và float.h.

Về cơ bản cả C và C++ đều có 4 kiểu dữ liệu built-in là char, int, float và double. Kiểu char có kích thước nhỏ nhất là 8 bit mặc dù thực tế có thể lớn hơn. Kiểu int có kích thước nhỏ nhất là 2 byte còn kiểu float và double là hai kiểu số thực có độ chính xác đơn và kép, chúng có format tuân theo chuẩn IEEE.

Như đã đề cập ở trên kiểu bool là một kiểu chuẩn của ngôn ngữ C++ với hai giá trị là true và false. Tuy nhiên rất nhiều chương trình vẫn dùng các giá trị kiểu int thay cho các giá trị kiểu bool nên trong các trường hợp cần đến một giá trị bool trình biên dịch thường thực hiện chuyển kiểu từ int sang bool hoặc có cảnh báo cho chúng ta để chính xác hóa các trường hợp này.

Ngoài 4 kiểu trên ra chúng ta có thể sử dụng các từ khóa bổ trợ sau để mở rộng khả năng lưu trữ của chúng. C++ cung cấp 4 từ khóa bổ trợ là: **long, short, signed và unsigned**.

long và short được dùng để chỉ định các giá trị max và min mà một kiểu dữ liệu sẽ lưu giữ. Một biến kiểu int sẽ có kích thước bằng kích thước nhỏ nhất của một biến kiểu short int. Các kiểu dữ liệu số nguyên có thể là: short int và long int.

Với các kiểu thực ta có long float và long double, không có các kiểu số thực với từ khóa short.

Các từ khóa signed và unsigned được dùng để chỉ định cho trình biên dịch cách thức sử dụng bit dấu với các kiểu nguyên và kiểu char. Với một kiểu signed bit cao nhất được dùng làm bit dấu, kiểu này chỉ cần thiết với char. Với kiểu unsigned không cần dùng một bit làm bit dấu nên số phần tử dương sẽ tăng lên gấp đôi.

Kiểu con trỏ và tham chiếu

Có thể có nhiều cách nói khác nhau về các biến con trỏ, một trong những điểm mạnh mẽ và mềm dẻo nhất của ngôn ngữ C, đồng thời cũng là nguyên nhân gây ra nhiều rắc rối với các chương trình viết bằng C. Con trỏ là một kiểu dữ liệu bình thường

như các kiểu dữ liệu khác chỉ có một điều đặc biệt là giá trị của nó là địa chỉ của các biến khác. Chính vì điều đặc biệt đó mà thông qua các biến con trỏ chúng ta có thể thực hiện các thao tác đối với một biến số khác ví dụ như thay đổi giá trị

Xét ví dụ sau đây:

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

```
int main(int argc, char *argv[])
{
    int n = 9;
    int * pn = &n;
    cout << pn << "\n" << *pn << "\n" << &pn << "\n" << &n;
    return 0;
}
```

output nhận được khi chạy chương trình trên là:

0x1491ffc0

9

0x1491ffbe

0x1491ffc0

Giá trị của pn đúng bằng địa chỉ của biến n và *pn bằng giá trị của n. Toán tử * được gọi là toán tử tham chiếu lại (dereference), nó có thể được dùng để khai báo biến con trỏ, tham chiếu tới giá trị của biến mà một biến con trỏ trỏ tới.

Hai ứng dụng cơ bản của con trỏ là:

- thay đổi giá trị của các đối tượng bên ngoài một hàm số. Đây là ứng dụng cơ bản nhất của các biến con trỏ.
- các kỹ thuật lập trình tinh vi khác mà chúng ta sẽ học sau.

Tham chiếu (reference)

Ấn tượng ban đầu của chúng ta về các tham chiếu là chúng không cần thiết. Chúng ta có thể viết các chương trình mà không cần tới các tham chiếu. Điều này nói chung là đúng trừ một số trường hợp mà chúng ta sẽ học sau này. Thực ra khái niệm truyền biến qua tham chiếu (pass by reference) không phải là một khái niệm chỉ có ở C++, nó cũng là một phần cơ bản trong một số ngôn ngữ khác.

Khái niệm về tham chiếu cũng có sự tương đồng với khái niệm con trỏ: chúng ta có thể truyền địa chỉ của một tham biến (trong một hàm) qua một tham chiếu. Tuy nhiên sự khác nhau giữa con trỏ và tham chiếu là truyền bằng tham chiếu có vẻ sạch sẽ hơn (cleaner) so với con trỏ. Tham chiếu cho phép các hàm có thể thay đổi giá trị của các đối tượng ngoài như con trỏ tuy nhiên trong cú pháp có sự khác nhau chút ít:

Trong danh sách tham số của hàm chúng ta dùng khai báo int &n để báo rằng chúng ta muốn truyền bằng tham chiếu và truy cập bình thường như một biến khác (con trỏ cần dùng dấu * để truy cập tới giá trị biến). Khi gọi hàm cú pháp đối với việc truyền bằng tham chiếu tương tự như truyền bằng giá trị (với con trỏ cần thêm 1 dấu & trước tên biến).

6. Một số toán tử trong C++

Tất cả các toán tử đều sinh ra một kết quả nào đó từ các toán hạng của chúng. Giá trị này được sinh ra mà không làm thay đổi giá trị của các toán hạng, trừ toán tử gán, toán tử tăng và giảm. Thay đổi giá trị của một toán hạng được gọi là 1 hiệu ứng phụ.

6.1 Toán tử gán (assignment operator)

Phép gán được thực hiện bằng toán tử `=`. Toán tử gán có nghĩa là lấy giá trị bên phải của toán tử (thường được gọi là *rvalue*) và copy giá trị đó sang bên trái của toán tử (thường được gọi là *lvalue*). Một *rvalue* có thể là bất kỳ giá trị hằng, biến, hoặc biểu thức có thể sinh ra một giá trị, nhưng một *lvalue* nhất thiết phải là một biến được đặt tên phân biệt (có nghĩa là phải có một vùng nhớ vật lý để chứa dữ liệu). Ví dụ chúng ta có thể gán một giá trị hằng số cho một biến chứ không thể gán một biến cho một giá trị hằng.

6.2 Các toán tử toán học

Các toán tử toán học gồm có phép cộng (+), phép trừ (-), phép nhân (*) và phép chia (/), phép lấy phần dư (%). Phép chia các số nguyên được thực hiện như là phép `div`, phép lấy phần dư không thực hiện được với các số dấu phẩy động.

Cả C và C++ đều có một cơ chế viết các câu lệnh tắt cho phép thực hiện đồng thời một phép gán và một toán tử toán học. Điều này được thực hiện bằng cách viết một toán tử trước một dấu `=`. Ví dụ: `x += 4;`

6.3 Các toán tử quan hệ

Các toán tử quan hệ thiết lập một quan hệ giữa các giá trị của toán hạng. Chúng sinh ra một giá trị có kiểu Boolean, là `true` nếu quan hệ đó là đúng và `false` nếu như quan hệ đó là sai. Các toán tử quan hệ gồm có: nhỏ hơn (<), lớn hơn (>), nhỏ hơn hoặc bằng (<=), lớn hơn hoặc bằng và bằng. Các toán tử này đều có thể sử dụng với các kiểu dữ liệu built-in của C và C++.

6.4 Các toán tử logic

Các toán tử logic `&&` và `||` cho ta kết quả là `true` hoặc `false` dựa trên mối quan hệ logic giữa các tham số của chúng. Chú ý rằng trong C và C++ `true` có nghĩa là một giá trị khác 0 và `false` có nghĩa là một giá trị bằng 0. Khi in ra màn hình `true` sẽ là 1 và `false` sẽ là 0.

Việc sử dụng các toán tử logic này cũng không có gì đặc biệt chỉ cần chú ý đối với các số dấu phẩy động, ví dụ:

```
float t = 1.22222e12123123, f = 1.22223e12123123;
```

```
cout << (t == f);
```

Sẽ cho ta một kết quả `true`.

6.5 Các toán tử bitwise

Các toán tử bitwise được sử dụng khi chúng ta muốn thao tác với các bit cụ thể của các biến có kiểu số (do các số thực dấu phẩy động có định dạng riêng nên các toán tử này chỉ làm việc với các biến kiểu **char, int, và long**). Các toán tử bitwise thực hiện các phép tính đại số Boolean trên các bit tương ứng của các tham số của nó.

Các toán tử bitwise gồm có toán tử `and`, `or`, `not` và `xor` được định nghĩa như trong đại số bool.

Các toán tử bitwise cũng có thể kết hợp với toán tử gán giống như các toán tử toán học.

6.6 Các toán tử dịch

Có hai toán tử dịch bit là toán tử dịch bit phải (>>) và toán tử dịch bit trái (<<). Các toán tử dịch bit sẽ thực hiện dịch tương ứng sang phải hoặc sang trái một số các bit

bằng số nằm n ngay sau toán tử . Nếu như số lượng các bit cần dịch lớn hơn số bit của toán hạng thì kết quả sẽ là không xác định . Nếu như toán hạng bên trái là một số thuộc kiểu unsigned thì phép toán dịch bit phải sẽ là một phép dịch h bit logic có nghĩa là các bit cao của nó sẽ là các bit 0. Nếu toán hạng bên trái là một số có dấu thì phép dịch phải có thể hoặc không thể là một phép dịch bit logic.

Các hàm bitwise thường rất hiệu quả do chúng thường được dịch trực tiếp thành mã assembly . Đôi khi một câu lệnh C hoặc C ++ đơn có thể sinh ra một dòng mã assembly.

6.7 Các toán tử một ngôi

Toán tử bitwise not không phải là toán tử một ngôi duy nhất . Ngoài toán tử này còn nhiều toán tử một ngôi khác chẳng hạn như dấu – cũng được xem như là một toán tử một ngôi.

Hai toán tử increment và decrement (-- và ++) cũng là các toán tử một ngôi , chúng khác với các toán tử một ngôi khác là chúng có hiệu ứng phụ.

Ngoài ra có thể thấy toán tử lấy địa chỉ của một biến số (&), toán tử tham chiếu lại (* và ->), toán tử new và delete cũng là các toán tử một ngôi của C++.

6.8 Toán tử 3 ngôi

Toán tử 3 ngôi if-else thường ít được sử dụng bởi vì nó đòi hỏi có 3 toán hạng. Đây thực sự là một toán tử bởi vì nó cho ta một giá trị chứ không giống với câu lệnh if -else bình thường. Nó bao gồm 3 biểu thức: nếu biểu thức đầu tiên (sau biểu thức đầu tiên là một dấu ?) cho ta một giá trị true thì biểu thức ngay sau dấu ? sẽ được thực hiện và kết quả của nó trở thành kết quả sinh ra bởi toán tử . Nếu như biểu thức đầu tiên cho một giá trị false thì biểu thức thứ 3 (sau dấu :) sẽ được thực hiện và kết quả của nó sẽ là kết quả của toán tử.

Ví dụ:

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

6.9 Toán tử dấu phẩy

Dấu phẩy không chỉ hạn chế sử dụng trong việc khai báo các biến , danh sách tham số của một hàm số mà nó còn là một toán tử được sử dụng để tách biệt các biểu thức . Trong trường hợp là một toán tử , kết quả của toán tử dấu phẩy sẽ là kết quả của việc thực hiện biểu thức cuối cùng . Các biểu thức khác cũng được thực hiện và có thể ảnh hưởng tới kết quả của việc thực hiện của toán tử này qua các hiệu ứng phụ của chúng .

Thường thì toán tử dấu phẩy cũng không được sử dụng nhiều vì mọi người thường có thói quen không xem đây là một toán tử.

6.10 Các lỗi thường gặp khi sử dụng các toán tử

Một lỗi thường gặp khi sử dụng các toán tử là chúng ta không sử dụng các cặp đóng mở ngoặc thường xuyên.

Ví dụ:

```
a = b++, c++, d++;
```

Câu lệnh trên hoàn toàn khác với câu lệnh:

```
a = (b++, c++, d++);
```

Tiếp theo là lỗi khi so sánh hai toán hạng bằng toán tử gán `=`. Tương tự chúng ta cũng hay nhầm lẫn khi sử dụng các toán tử bitwise và các toán tử logic tương tự với chúng.

6.11 Toán tử chuyển kiểu

Toán tử chuyển kiểu thường được sử dụng khi chúng ta muốn thực hiện một số toán tử của một kiểu dữ liệu nào đó với một biến thuộc một kiểu dữ liệu khác.

Ví dụ:

```
float f = 100.972;
```

```
int n = (int)f;
```

Đó là cách chuyển kiểu trong ngôn ngữ C. C++ cung cấp cho chúng ta một cách chuyển kiểu khác, tương đối giống với cách gọi hàm:

```
float f = 100.972;
```

```
int n = int(f);
```

Chuyển kiểu cung cấp cho chúng ta một cơ chế rất mạnh để xử lý các biến tuy nhiên đôi khi nó cũng gây ra nhiều lỗi làm chúng ta đau đầu. Vì để thực hiện chuyển kiểu đôi khi các biến cần có thêm bộ nhớ để chứa dữ liệu và điều này không phải bao giờ cũng suôn sẻ nhất là khi chúng ta thực hiện chuyển kiểu các con trỏ. Chuyển kiểu thường chỉ thực hiện với các biến hơn là các hằng.

Trong C++ chúng ta có 4 loại hình chuyển kiểu là `static_cast`, `const_cast`, `reinterpret_cast` và `dynamic_cast`.

6.12 Toán tử sizeof.

Toán tử `sizeof` cho chúng ta biết số lượng byte được sử dụng bởi một biến cụ thể. Nó cũng có thể cho ta biết kích thước cụ thể của một kiểu dữ liệu. Chú ý rằng `sizeof` là một toán tử chứ không phải là một hàm vì thế trường hợp sử dụng với các kiểu dữ liệu với các biến số chúng ta không cần có các dấu đóng, mở ngoặc.

7. Các kiểu dữ liệu người dùng định nghĩa

Các kiểu dữ liệu cơ bản và các biến thái của chúng là cần thiết tuy nhiên nếu chỉ dùng chúng thì cũng không thể tạo nên các chương trình có ý nghĩa được. C và C++ cung cấp cho chúng ta rất nhiều cơ chế khác nhau để xây dựng lên các kiểu tích hợp có ý nghĩa hơn, phức tạp hơn và phù hợp với nhu cầu của chương trình hơn. Kiểu dữ liệu người dùng định nghĩa quan trọng nhất của C là `struct`, và của C++ là `class`. Tuy nhiên các dễ nhất để định nghĩa một kiểu mới là dùng từ khóa **`typedef`** để đặt bí danh cho một kiểu sẵn có.

Thiết lập các tên bí danh với từ khóa typedef

`typedef` có nghĩa là “type definition” nhưng những gì mà từ khóa này thực sự làm không giống như đúng ngữ nghĩa của hai từ “type definition”. Cú pháp sử dụng với từ `typedef`:

```
typedef <một kiểu đã có sẵn> <tên kiểu muốn đặt>
```

Ví dụ:

```
typedef unsigned long ulong;
```

```
typedef struct str_list{
```

```
    int data;
```

```
    struct str_list * next;
```

```
}list;
```

Sau khi khai báo như trên chúng ta có thể khai báo trong chương trình như sau:

```
list * aList, anotherList;
```

Và trình biên dịch gặp từ ulong với một khai báo biến nó sẽ hiểu rằng đó chính là kiểu unsigned long. Chúng ta có thể cho rằng điều này có thể thực hiện dễ dàng bằng các thao tác thay thế trong giai đoạn tiền xử lý song thực ra không phải như vậy vì đôi khi trình biên dịch buộc phải hiểu rằng chúng ta muốn nó xử lý một tên nào đó như là một kiểu dữ liệu (ví dụ thực hiện so sánh hai biến chẳng hạn) vì thế nên từ khóa **typedef** là thực sự cần thiết. Từ khóa typedef thường được dùng khi chúng ta khai báo các cấu trúc mới hoặc đặt bí danh cho một kiểu con trỏ nào đó.

Kiểu dữ liệu cấu trúc với từ khóa struct

Kiểu dữ liệu cấu trúc là một cách cho phép các lập trình viên nhóm một nhóm các biến thuộc các kiểu dữ liệu khác nhau tạo thành một cấu trúc . Với một kiểu struct chúng ta có thể truy cập tới các thành phần dữ liệu qua các toán tử tham chiếu “.” và “->” với một con trỏ cấu trúc. Có một điều đặc biệt khi chúng ta khai báo và sử dụng các cấu trúc trong C++:

```
typedef struct{ // hoặc có thể là: typedef struct list
    int data;
    list *next;
}list;
main(){
    list * ptr;
}
```

Kiểu dữ liệu liệt kê (enum)

Kiểu dữ liệu liệt kê được khai báo bằng từ khóa enum và thường được dùng để làm chương trình sáng sủa hơn. Ví dụ:

```
enum Bool{true = 1, false = 0};
```

Thường trong C++ kiểu enum được dùng ít hơn do một số nguyên nhân ví dụ như kiểm tra kiểu, chẳng hạn chúng ta không thể thực hiện lệnh true++.

Kiểu hợp nhất (union)

Đôi khi trong chương trình chúng ta cần phải làm việc với nhiều kiểu dữ liệu khác nhau với cùng một biến số , khi đó chúng ta có thể thực hiện theo hai cách : một là dùng kiểu dữ liệu cấu trúc hoặc nếu có thể sử dụng kiểu hợp nhất để tiết kiệm bộ nhớ.

Kiểu union có cách khai báo giống hệt kiểu struct chỉ có khác ở cách sử dụng : tại một thời điểm chúng ta chỉ có thể truy cập tới một thành phần của một biến kiểu union và kích thước của một kiểu dữ liệu union chính bằng kích thước của thành phần có kích thước lớn nhất của kiểu.

Ví dụ:

```
union test_type{
    char c;
    int i;
    float f;
    double d;
};

#include <iostream.h>
```

```
union Packed { // khai báo giống như một lớp
    char i;
    short j;
    int k;
```

```

    long l;
    float f;
    double d;
    // kích thước bằng kích thước của kiểu double
}; // kết thúc bằng dấu ; giống như kiểu struct

int main() {
    cout << "sizeof(Packed) = "
          << sizeof(Packed) << endl;
    Packed x;
    x.i = 'c';
    cout << x.i << endl;
    x.d = 3.14159;
    cout << x.d << endl;
}

```

khi đó `sizeof(test_type)` sẽ bằng 8 bằng kích thước của kiểu `double`. Chú ý là sau khi thực hiện gán giá trị cho các thành phần của một biến có kiểu `union` thì các giá trị được gán trước đó của một thành phần khác sẽ bị sửa đổi. Ví dụ chúng ta chỉ cần thêm một lệnh: `cout << x.i << endl;` vào cuối chương trình trên sẽ nhận được kết quả là ‘n’ chứ không phải ‘c’.

Kiểu dữ liệu mảng

Mảng là một kiểu dữ liệu tích hợp rất hay được dùng, nó cho phép chúng ta kết hợp nhiều biến đơn lẻ có cùng kiểu thành một kiểu dữ liệu tích hợp. Việc truy cập tới các thành phần của một mảng được thực hiện bằng cách lấy chỉ mục (index) của nó: `[]`. Việc khai báo mảng có thể được thực hiện kèm với việc gán các giá trị cho các thành phần của nó.

Mảng và con trỏ.

Con trỏ là một công cụ tuyệt vời cho phép truy cập tới các thành phần của một mảng bất kỳ. Đặc biệt là khi chúng ta cần truyền một mảng làm tham số của một hàm. Ví dụ:

```

#include <iostream.h>
#include <string.h>

void func1(int a[], int size) {
    for(int i = 0; i < size; i++)
        a[i] = i * i - i;
}

void func2(int* a, int size) {
    for(int i = 0; i < size; i++)
        a[i] = i * i + i;
}

void print(int a[], string name, int size) {
    for(int i = 0; i < size; i++)
        cout << name << "[" << i << "] = "
              << a[i] << endl;
}

```

```

    }

    int main() {
        int a[5], b[5];
        print(a, "a", 5);
        print(b, "b", 5);
        // khởi tạo mảng:
        func1(a, 5);
        func1(b, 5);
        print(a, "a", 5);
        print(b, "b", 5);
        func2(a, 5);
        func2(b, 5);
        print(a, "a", 5);
        print(b, "b", 5);
    }

```

Một trường hợp đặc biệt của các hàm kiểu này chính là bản thân hàm `main`. Hàm `main` có hai tham số là `(int n, char * args[])`. Để thuận tiện cho việc sử dụng các tham số này chúng ta nên dùng các hàm chuyển kiểu chẳng hạn như: `atoi()`, `atof()`, `atol()`.

Chương 3: Con trỏ, tham chiếu và hàm. (3 tiết)

1. Hàm trong C++

Các hàm là công cụ chính cho phép xây dựng các chương trình lớn trong C và C++. Với các hàm nhỏ một chương trình lớn có thể được chia thành các chương trình nhỏ hơn, dễ giải quyết hơn. Với các ngôn ngữ lập trình khác nhau ngữ pháp ta dùng các thuật ngữ khác nhau để chỉ một chương trình con, trong C và C++ các chương trình con được gọi là các hàm.

1.1 Nguyên mẫu và định nghĩa hàm

Một hàm trong C và C++ thường được khai báo nguyên mẫu trước khi thực sự cài đặt (định nghĩa). Cú pháp khai báo nguyên mẫu của một hàm như sau:

<kiểu của hàm> <tên hàm>(<danh sách tham số>);

Trong đó <kiểu của hàm> là kiểu dữ liệu mà hàm trả về, <tên hàm> là tên mà chúng ta muốn đặt cho hàm (tên này được dùng để gọi hàm), danh sách tham số là danh sách các tham biến và kiểu của chúng được sử dụng với hàm. Ví dụ:

```
int max(int a, int b);
```

Thường các nguyên mẫu hàm được đặt trong các file `.h` mà người ta gọi là các file header và để gọi tới một hàm chúng ta cần có chỉ thị `#include` file header tương ứng với hàm mà chúng ta định sử dụng. Khi đó trong quá trình biên dịch trình biên dịch sẽ tự tìm các hàm chuẩn (được coi là một phần cơ bản của ngôn ngữ) còn với các hàm người dùng định nghĩa chúng ta cần chỉ rõ đường dẫn tới các file chứa phần cài đặt của hàm (thường là file `.cpp` hoặc một file thư viện tự tạo nào đó).

1.2 Hàm và các biến

Các biến nằm trong hàm cũng như các biến là tham số được truyền cùng với một hàm được gọi là các biến cục bộ của một hàm. Các biến nằm ngoài các hàm được gọi là các biến toàn cục.

1.3 Truyền tham số

Việc truyền tham số cho một hàm có thể được tiến hành theo 3 hình thức khác nhau:

Truyền theo giá trị

Đây là cách truyền các tham số mặc định của C và C++. Các biến được truyền theo cách này thực sự là các biến cục bộ của hàm. Khi hàm được gọi thực hiện sẽ xuất hiện bản copy của các biến này và hàm sẽ làm việc với chúng. Sau khi hàm được thực hiện xong các bản copy này sẽ được loại khỏi bộ nhớ của chương trình. Do hàm làm việc với các bản copy của các tham số nên các biến truyền theo giá trị không bị ảnh hưởng.

Truyền theo địa chỉ

Việc truyền theo địa chỉ được thực hiện khi chúng ta muốn thay đổi các biến được truyền cho một hàm. Thay vì làm việc với các bản copy của tham số trong trường hợp truyền theo địa chỉ hàm sẽ thao tác trực tiếp trên các biến được truyền vào thông qua địa chỉ của chúng. Ngoài mục đích là làm thay đổi các biến ngoài việc truyền theo địa chỉ còn được thực hiện khi chúng ta truyền một biến có kích thước lớn (một mảng chẳng hạn), khi đó việc truyền bằng địa chỉ sẽ tiết kiệm được không gian nhớ cần sử dụng. Tuy nhiên việc truyền tham số theo địa chỉ đôi khi gây ra rất nhiều lỗi khó kiểm soát.

Tham số là mảng một chiều : trong trường hợp tham số của một hàm là mảng một chiều chúng ta cần dùng thêm một biến kiểu int để chỉ định số phần tử của mảng, ví dụ:

```
void sort(int *a, int); // hay
```

```
void sort(int a[], int);
```

Tham số là mảng hai chiều : với mảng hai chiều chúng ta có thể sử dụng hai cách sau đây để thực hiện các thao tác:

```
void readData(int a[][5], int m, int n);
```

hoặc

```
void readData(int *a, int m, int n){
```

```
    for (int i=0;i<n;++i)
```

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

```
            cin >> *(a+m*i+j);
```

```
    return;
```

```
}
```

Khi đó chúng ta có thể gọi hàm này như sau :

```
int* a_ptr = new int [n*m] ;
```

```
readdata(a_ptr,n,m);
```

hoặc

```
int a[2][2];
```

```
readdata(&a[0][0],n,m);
```

Truyền theo tham chiếu

Việc thay đổi các biến ngoài cũng có thể được thực hiện bằng cách truyền theo tham chiếu. So với cách truyền theo địa chỉ truyền theo tham số an toàn hơn và do đó cũng kém linh hoạt hơn.

Ví dụ:

```
void swap(int &a, int &b);
```

1.4 Chồng hàm (overload) và tham số mặc định của hàm

Chồng hàm (overload)

C++ cho phép lập trình viên có khả năng viết các hàm có tên giống nhau, khả năng này được gọi là chồng hàm (overload hoặc polymorphism function mean many formed). Ví dụ chúng ta có thể có các hàm như sau:

```
int myFunction(int);
int myFunction(int, int);
int myFunction(int, int, int);
```

Các hàm overload cần thỏa mãn một điều kiện là danh sách các tham số của chúng phải khác nhau (về số lượng tham số và kiểu tham số). Kiểu các hàm overload có thể giống nhau hoặc khác nhau. Danh sách kiểu các tham số của một hàm được gọi là **chữ ký** (signature) của hàm đó.

Có sự tương tự khi sử dụng chồng hàm và các tham số mặc định và sự lựa chọn sử dụng tùy thuộc vào kinh nghiệm của lập trình viên. Với các hàm lớn và phức tạp chúng ta nên sử dụng chồng hàm, ngoài ra việc sử dụng các hàm chồng nhau cũng làm cho chương trình sáng sủa và dễ gỡ lỗi hơn.

Chú ý là không thể overload các hàm static.

Các tham số mặc định

Các biến được truyền làm tham số khi thực hiện gọi một hàm phải có kiểu đúng như nó đã được khai báo trong phần prototype của hàm. Chỉ có một trường hợp khác đó là khi chúng ta khai báo hàm với tham số có giá trị mặc định ví dụ:

```
int myFunction(int x=10);
```

Khi đó khi chúng ta thực hiện gọi hàm và không truyền tham số, giá trị 10 sẽ được dùng trong thân hàm. Vì trong prototype không cần tên biến nên chúng ta có thể thực hiện khai báo như sau:

```
int myFunction(int =10);
```

Trong phần cài đặt của hàm chúng ta vẫn tiến hành bình thường như các hàm khác:

```
int myFunction(int x){
    ...
}
```

Bất kỳ một tham số nào cũng có thể gán các giá trị mặc định chỉ có một hạn chế: nếu một tham số nào đó không được gán các giá trị mặc định thì các tham số đứng trước nó cũng không thể sử dụng các giá trị mặc định. Ví dụ với hàm:

```
int myFunction(int p1, int p2, int p3);
```

Nếu p3 không được gán các giá trị mặc định thì cũng không thể gán cho p2 các giá trị mặc định.

Các giá trị mặc định của tham số hàm thường được sử dụng trong các hàm cấu tử của các lớp.

1.5 Các vấn đề khác

Hàm inline

Các hàm inline được xác định bằng từ khóa inline. Ví dụ:

```
inline myFunction(int);
```

Khi chúng ta sử dụng các hàm trong một chương trình C hoặc C++ thường thì phần thân hàm sau khi được biên dịch sẽ là một tập các lệnh máy. Mỗi khi chương trình gọi tới hàm, đoạn mã của hàm sẽ được nạp vào stack để thực hiện sau đó trả về 1 giá trị nào đó nếu có và thân hàm được loại khỏi stack thực hiện của chương trình. Nếu hàm được gọi 10 lần sẽ có 10 lệnh nhảy tương ứng với 10 lần nạp thân hàm để thực hiện. Với chỉ

thì inline chúng ta muốn gợi ý cho trình biên dịch là thay vì nạp thân hàm như bình thường hãy chèn đoạn mã của hàm vào đúng chỗ mà nó được gọi tới trong chương trình. Điều này rõ ràng làm cho chương trình thực hiện nhanh hơn bình thường. Tuy nhiên inline chỉ là một gợi ý và không phải bao giờ cũng được thực hiện. Với các hàm phức tạp (chẳng hạn như có vòng lặp) thì không nên dùng inline. Các hàm inline do đó thường rất gọn gàng hạn như các hàm chỉ thực hiện một vài thao tác khởi tạo các biến (các hàm cấu tử của các lớp). Với các lớp khi khai báo các hàm inline chúng ta có thể không cần dùng từ khóa inline mà thực hiện cài đặt ngay sau khi khai báo là đủ.

Hàm đệ qui

Đệ qui là một cơ chế cho phép một hàm có thể gọi tới chính nó. Kỹ thuật đệ qui thường gắn với các vấn đề mang tính đệ qui hoặc được xác định đệ qui. Để giải quyết các bài toán có các chu trình lồng nhau người ta thường dùng đệ qui. Ví dụ như bài toán tính giai thừa, bài toán sinh các hoán vị của n phần tử

Sử dụng từ khóa const

Đôi khi chúng ta muốn truyền một tham số theo địa chỉ nhưng không muốn thay đổi tham số đó, để tránh các lỗi có thể xảy ra chúng ta có thể sử dụng từ khóa const. Khi đó nếu trong thân hàm chúng ta vô ý thay đổi nội dung của biến trình biên dịch sẽ báo lỗi. Ngoài ra việc sử dụng từ khóa const còn mang nhiều ý nghĩa khác liên quan tới các phương thức của lớp (chúng ta sẽ học trong chương 5).

2. Con trỏ, hàm và mảng

Một số khái niệm về con trỏ cũng đã được nhắc đến trong chương 2 nên không nhắc lại ở đây chúng ta chỉ chú ý một số vấn đề sau:

Cấp phát bộ nhớ cho biến con trỏ bằng toán tử new:

```
int *p = new int[2];
```

và xóa bỏ nó bằng toán tử delete.

Phân biệt khai báo `int a[]`; và `int *p`;

Trong trường hợp thứ nhất chúng ta có thể thực hiện khởi tạo các giá trị cho mảng a còn trong trường hợp thứ hai thì không thể.

Cần nhớ rằng tên của mảng chính là con trỏ trỏ tới phần tử đầu tiên của mảng do đó việc gán:

```
int *pa = &a[0];
```

và `pa = a`; là như nhau.

Cũng do trong C và C++ không kiểm soát số phần tử của một mảng nên để truyền một mảng cho hàm chúng ta thường dùng thêm các biến nguyên chỉ số lượng phần tử của mảng (xem lại ví dụ phần truyền biên).

Con trỏ hàm

3. Hàm và xử lý xâu

Việc xử lý xâu trong C++ chính là xử lý mảng, cụ thể là mảng ký tự nên tất cả các kỹ thuật được áp dụng với mảng và hàm bình thường cũng có thể được ứng dụng cho việc xử lý xâu.

Chương 4: Các dòng vào ra trong C++

Cho tới bây giờ chúng ta vẫn dùng cout để kết xuất dữ liệu ra màn hình và cin để đọc dữ liệu nhập vào từ bàn phím mà không hiểu một cách rõ ràng về cách thức hoạt động của chúng, trong phần này chúng ta sẽ học về các nội dung sau đây :

- Thế nào là các luồng và cách thức chúng được sử dụng
- Kiểm soát các thao tác nhập và kết xuất dữ liệu bằng cách sử dụng các luồng
- Cách đọc và ghi dữ liệu với các file sử dụng các luồng

1. Tổng quan về các luồng vào ra của C++

Về bản chất C++ không định nghĩa qui cách kết xuất và nhập dữ liệu trong hương trình. Hay nói một cách rõ ràng hơn các thao tác vào ra dữ liệu không phải là một phần cơ bản của ngôn ngữ C++. Chúng ta có thể thấy rõ điều này nếu so sánh với một số ngôn ngữ khác chẳng hạn Pascal. Trong Pascal để thực hiện các thao tác vào ra dữ liệu cơ bản chúng ta có thể sử dụng các lệnh chẳng hạn read hay write. Các lệnh này là một phần của ngôn ngữ Pascal.

Tuy nhiên ngày nay để thuận tiện cho việc vào ra dữ liệu trong các chương trình C++ người ta đã đưa vào thư viện chuẩn iostream. Việc không coi các thao tác vào ra dữ liệu là một phần cơ bản của ngôn ngữ và kiểm soát chúng trong các thư viện làm cho ngôn ngữ có tính độc lập về nền tảng cao. Một chương trình viết bằng C++ trên một hệ thống nền này có thể biên dịch lại và chạy tốt trên một hệ thống nền khác mà không cần thay đổi mã nguồn của chương trình. Các nhà cung cấp trình biên dịch chỉ việc cung cấp đúng thư viện tương thích với hệ thống và mọi thứ thế là ổn ít nhất là trên lý thuyết.

Chú ý: Thư viện là một tập các file OBJ có thể liên kết với chương trình của chúng ta khi biên dịch để cung cấp thêm một số chức năng (qua các hàm, hằng, biến được định nghĩa trong chúng). Đây là dạng cơ bản nhất của việc sử dụng lại mã chương trình.

Các lớp iostream coi luồng dữ liệu từ một chương trình tới màn hình như là một dòng (stream) dữ liệu gồm các byte (các ký tự) nối tiếp nhau. Nếu như đích của dòng này là một file hoặc màn hình thì nguồn thường là một phần nào đó trong chương trình. Hoặc có thể là dữ liệu được nhập vào từ bàn phím, các file và được rót vào các biến dùng để chứa dữ liệu trong chương trình.

Một trong các mục đích chính của các dòng là bao gói các vấn đề trong việc lấy và kết xuất dữ liệu ra file hay ra màn hình. Khi một dòng được tạo ra chương trình sẽ làm việc với dòng đó và dòng sẽ đảm nhiệm tất cả các công việc chi tiết cụ thể khác (làm việc với các file và việc nhập dữ liệu từ bàn phím).

Bộ đệm

Việc ghi dữ liệu lên đĩa là một thao tác tương đối đắt đỏ (về thời gian và tài nguyên hệ thống). Việc ghi và đọc dữ liệu từ các file trên đĩa chiếm rất nhiều thời gian và thường thì các chương trình sẽ bị chậm lại do các thao tác đọc và ghi dữ liệu trực tiếp lên đĩa cứng. Để giải quyết vấn đề này các luồng được cung cấp cơ chế sử dụng đệm. Dữ liệu được ghi ra luồng nhưng không được ghi ra đĩa ngay lập tức, thay vào đó bộ đệm của luồng sẽ được làm đầy từ từ và khi đầy dữ liệu nó sẽ thực hiện ghi tất cả lên đĩa một lần.

Điều này giống như một chiếc bình đựng nước có hai van, một van trên và một van dưới. Nước được đổ vào bình từ van trên, trong quá trình đổ nước vào bình van dưới được khóa kín, chỉ khi nào nước trong bình đã đầy thì van dưới mới mở và nước chảy ra khỏi bình. Việc thực hiện thao tác cho phép nước chảy ra khỏi bình mà không cần chờ cho tới khi nước đầy bình được gọi là “flush the buffer”.

2. Các luồng và các bộ đệm

C++ thực hiện cài đặt các luồng và các bộ đệm theo cách nhìn hướng đối tượng:

- Lớp streambuf quản lý bộ đệm và các hàm thành viên của nó cho phép thực hiện các thao tác quản lý bộ đệm: fill, empty, flush.
- Lớp ios là lớp cơ sở của các luồng vào ra, nó có một đối tượng streambuf trong vai trò của một biến thành viên.
- Các lớp istream và ostream kế thừa từ lớp ios và cụ thể hóa các thao tác vào ra tương ứng.
- Lớp iostream kế thừa từ hai lớp istream và ostream và có các phương thức vào ra để thực hiện kết xuất dữ liệu ra màn hình.
- Lớp fstream cung cấp các thao tác vào ra với các file.

3. Các đối tượng vào ra chuẩn

Thư viện iostream là một thư viện chuẩn được trình biên dịch tự động thêm vào mỗi chương trình nên để sử dụng nó chúng ta chỉ cần có chỉ thị include file header iostream.h vào chương trình. Khi đó tự động có 4 đối tượng được định nghĩa và chúng ta có thể sử dụng chúng cho tất cả các thao tác vào ra cần thiết.

- cin: quản lý việc vào dữ liệu chuẩn hay chính là bàn phím
- cout: quản lý kết xuất dữ liệu chuẩn hay chính là màn hình
- cerr: quản lý việc kết xuất (không có bộ đệm) các thông báo lỗi ra thiết bị báo lỗi chuẩn (là màn hình). Vì không có cơ chế đệm nên dữ liệu được kết xuất ra cerr sẽ được thực hiện ngay lập tức.
- clog: quản lý việc kết xuất (có bộ đệm) các thông báo lỗi ra thiết bị báo lỗi chuẩn (là màn hình). Thường được tái định hướng vào một file log nào đó trên đĩa.

4. Định hướng lại (Redirection)

Các thiết bị chuẩn (input, output, error) đều có thể được định hướng lại tới các thiết bị khác. Chẳng hạn error thường được tái định hướng tới một file còn các thiết bị vào ra chuẩn có thể sử dụng cơ chế đường ống bằng cách sử dụng các lệnh của hệ điều hành.

Thuật ngữ tái định hướng có nghĩa là thay đổi vị mặc định của dữ liệu vào và dữ liệu ra của chương trình. Trong DOS và Unix các toán tử này là > và <.

Thuật ngữ đường ống có nghĩa là sử dụng output của một chương trình làm input của một chương trình khác.

So với DOS các hệ thống Unix có các cơ chế linh hoạt hơn song về cơ bản thì ý tưởng là hoàn toàn giống nhau: lấy dữ liệu được kết xuất ra màn hình và ghi vào một file trên đĩa, hoặc ghép nối nó vào một chương trình khác. Tương tự dữ liệu vào của một chương trình cũng có thể được trích ra từ một file nào đó trên đĩa thay vì nhập vào từ bàn phím.

Tái định hướng lại các thiết bị vào ra chuẩn là một chức năng của hệ điều hành hơn là một chức năng của thư viện iostream và do đó C++ chỉ cung cấp phương tiện truy cập vào 4 thiết bị chuẩn, việc tái định hướng được thực hiện ra sao là tùy vào người dùng.

5. Nhập dữ liệu với cin

cin là một đối tượng toàn cục chịu trách nhiệm nhập dữ liệu cho chương trình. Để sử dụng cin cần có chỉ thị tiền xử lý include file header ios <iostream.h>. Toán tử >> được dùng với đối tượng cin để nhập dữ liệu cho một biến nào đó của chương trình.

Trước hết chúng ta thấy cin buộc phải là một biến toàn cục vì chúng ta không định nghĩa nó trong chương trình. Toán tử >> là một toán tử được overload và kết quả của việc sử dụng toán tử này là ghi tất cả nội dung trong bộ đệm của cin vào một biến cục bộ nào đó trong chương trình. Do >> là một toán tử được overload của cin nên nó có thể được dùng để nhập dữ liệu cho rất nhiều biến có kiểu khác nhau ví dụ:

```
int someVar;  
cin >> someVar;  
hàm toán tử sử dụng tương ứng sẽ được gọi đến là:  
istream & operator (int &);  
tham biến được truyền theo tham chiếu nên hàm toán tử có thể thực hiện thao tác trực tiếp trên biến.
```

Các chuỗi (strings)

cin cũng có thể làm việc với các biến chuỗi, hay các mảng ký tự, ví dụ:

```
char stdName[255];  
cin >> stdName;
```

Nếu chúng ta gõ vào chẳng hạn: hoai thì biến str sẽ là một mảng các ký tự: h, o, a, i, \0. Ký tự cuối cùng là một ký tự rỗng (null), cin tự động thêm ký tự này vào cuối chuỗi để đánh dấu vị trí kết thúc. Biến khai báo cần có đủ chỗ để chứa các ký tự được nhập vào (kể cả ký tự kết thúc chuỗi).

Tuy nhiên nếu chúng ta gõ vào một tên đầy đủ chẳng hạn: Phan Phi Hoai thì kết quả lại có thể không giống như mong đợi, chuỗi stdName sẽ có nội dung là: P, h, a, n, \0. Có kết quả này là do cách thức làm việc của toán tử >>. Khi thấy một ký tự dấu cách hoặc một ký tự xuống dòng được gõ vào thì nó sẽ xem như việc nhập tham số đã kết thúc và thêm ngay một ký tự kết thúc chuỗi vào vị trí đó. Chú ý là chúng ta có thể thực hiện nhập nhiều tham số một lúc ví dụ:

```
cin >> intVar >> floatVar;
```

Điều này là vì toán tử >> trả về một tham chiếu tới một đối tượng istream, bản thân cin cũng là một đối tượng istream nên kết quả trả về sau khi thực hiện một toán tử >> lại có thể là input cho toán tử tiếp theo.

```
(cin >> intVar) >> floatVar;
```

6. Các hàm thành viên khác của cin

Ngoài việc overload toán tử >> cin còn có rất nhiều hàm thành viên khác có thể được sử dụng để nhập dữ liệu.

Hàm get

Hàm get có thể sử dụng để nhập các ký tự đơn, khi đó chúng ta gọi tới hàm get() mà không cần có đối số, giá trị trả về là ký tự được nhập vào ví dụ:

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

```
int main()  
{  
    char ch;  
    while ( (ch = cin.get()) != EOF)  
    {  
        cout << "ch: " << ch << endl;
```

```

    }
    cout << "\nDone!\n";
    return 0;
}

```

Chương trình trên sẽ cho phép người dùng nhập vào các xâu có độ dài bất kỳ và in ra lần lượt các ký tự của xâu đó cho tới khi gặp ký tự điều khiển Ctrl-D hoặc Ctrl-Z.

Chú ý là không phải tất cả các cài đặt của istream đều hỗ trợ phiên bản này của hàm get().

Có thể gọi tới hàm get() để nhập các ký tự bằng cách truyền vào một biến kiểu char ví dụ:

```

char a, b;
cin.get(a).get(b);

```

Nói chung thì chúng ta nên dùng toán tử >> nếu cần thiết phải bỏ qua dấu cách và dùng get() với tham số là một biến kiểu char nếu cần thiết phải kiểm tra tất cả các ký tự kể cả ký tự dấu cách và không nên dùng get không có tham số.

Như chúng ta đã biết để nhập một xâu không có dấu cách có thể dùng toán tử >>, nhưng nếu muốn nhập các xâu có các dấu cách thì không thể dùng toán tử này được thay vào đó chúng ta có thể sử dụng hàm get với 3 tham số: tham số thứ nhất là một con trỏ trỏ tới một mảng ký tự, tham số thứ hai là số tối đa các ký tự có thể nhập vào cộng thêm 1, và tham số thứ 3 là ký tự báo hiệu kết thúc xâu. Tham số thứ 3 có giá trị mặc định là ký hiệu xuống dòng '\n'. Nếu như ký hiệu kết thúc xâu được nhập vào trước khi đạt tới số tối đa các ký tự có thể nhập vào thì cin sẽ thêm vào xâu câu một ký tự null và ký tự kết thúc xâu vẫn sẽ còn lại trong bộ đệm.

Ngoài cách nhập xâu bằng cách sử dụng hàm get() chúng ta có thể dùng hàm getline(). Hàm getline hoạt động tương tự như hàm get() chỉ trừ một điều là ký tự kết thúc sẽ được loại khỏi bộ đệm trong trường hợp nó được nhập trước khi đầy xâu.

Sử dụng hàm ignore

Đôi khi chúng ta muốn bỏ qua tất cả các ký tự còn lại của một dòng dữ liệu nào đó cho tới khi gặp ký tự kết thúc dòng (EOL) hoặc ký tự kết thúc file (EOF), hàm ignore của đối tượng cin nhằm phục vụ cho mục đích này. Hàm này có 2 tham số, tham số thứ nhất là số tối đa các ký tự sẽ bỏ qua cho tới khi gặp ký tự kết thúc được chỉ định bởi tham số thứ hai. Chẳng hạn với câu lệnh cin.ignore(80, '\n') thì tối đa 80 ký tự sẽ bị loại bỏ cho tới khi gặp ký tự xuống dòng, ký tự này sẽ được loại bỏ trước khi hàm ignore kết thúc công việc của nó. Ví dụ:

```

#include <iostream.h>

int main()
{
    char stringOne[255];
    char stringTwo[255];

    cout << "Nhập xâu thứ nhất:";
    cin.get(stringOne, 255);
    cout << "Xâu thứ nhất" << stringOne << endl;

    cout << "Nhập xâu thứ hai: ";
    cin.getline(stringTwo, 255);
    cout << "Xâu thứ hai: " << stringTwo << endl;
}

```

```

    cout << "\n\nNhập lại...\n";

    cout << "Nhập xâu thứ nhất: ";
    cin.get(stringOne,255);
    cout << "Xâu thứ nhất: " << stringOne<< endl;

    cin.ignore(255, '\n');

    cout << "Nhập xâu thứ hai: ";
    cin.getline(stringTwo,255);
    cout << "Xâu thứ hai: " << stringTwo<< endl;
    return 0;
}

```

cin có hai phương thức khác là peek và putback với mục đích làm cho công việc trở nên dễ dàng hơn. Hàm peek() cho ta biết ký tự tiếp theo nhưng không nhập chúng còn hàm putback() cho phép chèn một ký tự vào dòng input. Ví dụ:

```

while ( cin.get(ch) )
{
    if (ch == '!')
        cin.putback('$');
    else
        cout << ch;
    while (cin.peek() == '#')
        cin.ignore(1,'#');
}

```

Các hàm này thường được dùng để thực hiện phân tích các xâu hay các dữ liệu khác chẳng hạn trong các chương trình phân tích cú pháp của ngôn ngữ chẳng hạn.

7. Kết xuất dữ liệu với cout

Chúng ta đã từng sử dụng đối tượng cout cho việc kết xuất dữ liệu ra màn hình, ngoài ra chúng ta cũng có thể sử dụng cout để định dạng dữ liệu, căn lề các dòng kết xuất dữ liệu và ghi dữ liệu kiểu số dưới dạng số thập phân hay hệ hexa.

Xóa bộ đệm output

Việc xóa bộ đệm output được thực hiện khi chúng ta gọi tới hàm endl. Hàm này thực chất là gọi tới hàm flush() của đối tượng cout. Chúng ta có thể gọi trực tiếp tới hàm này:

```
cout << flush;
```

Các hàm liên quan

Tương tự như cin có các hàm get() và getline() cout có các hàm put() và write() phục vụ cho việc kết xuất dữ liệu. Hàm put() được dùng để ghi một ký tự ra thiết bị output và cũng như hàm get() của cin chúng ta có thể dùng hàm này liên tiếp:

```
cout.put('a').put('e');
```

Hàm write làm việc giống hệt như toán tử chèn chỉ trừ một điều là nó cần hai tham số: tham số thứ nhất là con trỏ xâu và tham số thứ hai là số ký tự sẽ in ra, ví dụ:

```
char str[] = "no pain no gain";
cout.write(str,3);
```

Các chỉ thị định dạng, các cờ và các thao tác với cout

Dòng output có rất nhiều cờ được sử dụng vào các mục đích khác nhau chẳng hạn như quản lý cơ sở của các biến sẽ được kết xuất ra màn hình, kích thước của các trường... Mỗi cờ trạng thái là một byte có các bit được gán cho các ý nghĩa khác nhau, các cờ này có thể được đặt các giá trị khác nhau bằng cách sử dụng các hàm.

Sử dụng hàm width của đối tượng cout

Độ rộng mặc định khi in ra các biến là vừa đủ để in dữ liệu của biến đó, điều này đôi khi làm cho output nhận được không đẹp về mặt thẩm mỹ. Để thay đổi điều này chúng ta có thể dùng hàm width, ví dụ:

```
cout.width(5);
```

Nếu như độ rộng thiết lập bằng hàm width nhỏ hơn độ rộng của biến trên thực tế thì nó sẽ không có tác dụng.

Thiết lập các ký tự lấp chỗ trống

Bình thường cout lấp các chỗ trống khi in ra một biến nào đó (với độ rộng được thiết lập bằng hàm width) bằng các dấu trống, nhưng chúng ta có thể thay đổi điều này bằng cách gọi tới hàm fill, ví dụ:

```
cout.fill('*');
```

Thiết lập các cờ

Các đối tượng ostream quản lý trạng thái của chúng bằng cách sử dụng các cờ. Chúng ta có thể thiết lập các cờ này bằng cách sử dụng hàm setf() với tham số là một hằng kiểu liệt kê đã được định nghĩa trước. Chẳng hạn với một biến kiểu float có giá trị là 20.000, nếu chúng ta sử dụng toán tử >> để in ra màn hình kết quả nhận được sẽ là 20, nếu chúng ta thiết lập cờ showpoint kết quả sẽ là 20.000.

Sau đây là một số cờ thường dùng:

Showpos	dấu của các biến kiểu số
Hex	In ra số dưới dạng hexa
Dec	In ra số dưới dạng cơ số 10
Oct	In ra số dưới dạng cơ số 8
Left	Căn lề bên trái
Right	Căn lề bên phải
Internal	Căn lề giữa
Precision	Hàm thiết lập độ chính xác của các biến thực: cout.precision(2);

Ngoài ra còn phải kể đến hàm setw() cũng có thể được dùng thay cho hàm width.

Chú ý là các hàm trên đều cần có chỉ thị include file header iomanip.h.

Các hàm width, fill và precision đều có một phiên bản không có tham số cho phép đọc các giá trị được thiết lập hiện tại (mặc định).

8. Các dòng vào ra và hàm printf

Hầu hết các cài đặt của C++ đều cung cấp các thư viện C chuẩn, trong đó bao gồm lệnh printf(). Mặc dù về mặt nào đó hàm printf dễ sử dụng hơn so với các dòng của C++ nhưng nó không thể theo kịp được các dòng: hàm printf không có đảm bảo về kiểu dữ liệu do đó khi chúng ta in ra một ký tự lại có thể là một số nguyên và ngược lại, hàm printf cũng không hỗ trợ các lớp do đó không thể overload để làm việc với các lớp.

Ngoài ra hàm `printf` cũng thực hiện việc định dạng dữ liệu dễ dàng hơn, ví dụ:
`printf("%15.5f", tf);`

Có rất nhiều lập trình viên C++ có xu hướng thích dùng hàm `printf()` nên cần phải chú ý kỹ tới hàm này.

9. Vào ra dữ liệu với các file

Các dòng cung cấp cho chúng ta một cách nhất quán để làm việc với dữ liệu được nhập vào từ bàn phím cũng như dữ liệu được nhập vào từ các file. Để làm việc với các file chúng ta tạo ra các đối tượng `ofstream` và `ifstream`.

ofstream

Các đối tượng cụ thể mà chúng ta dùng để đọc dữ liệu ra hoặc ghi dữ liệu vào được gọi là các đối tượng `ofstream`. Chúng được kế thừa từ các đối tượng `ostream`.

Để làm việc với một file trước hết chúng ta cần tạo ra một đối tượng `ofstream`, sau đó gắn nó với một file cụ thể trên đĩa, và để tạo ra một đối tượng `ofstream` chúng ta cần include file `fstream.h`.

Các trạng thái điều kiện

Các đối tượng `ostream` quản lý các cờ báo hiệu trạng thái sau khi chúng ta thực hiện các thao tác input và output chúng ta có thể kiểm tra các cờ này bằng cách sử dụng các hàm Boolean chẳng hạn như `eof()`, `bad()`, `fail()`, `good()`. Hàm `bad()` cho giá trị `TRUE` nếu một thao tác là không hợp lệ, hàm `fail()` cho giá trị `TRUE` nếu như hàm `bad()` cho giá trị `TRUE` hoặc một thao tác nào đó thất bại. Cuối cùng hàm `good()` cho giá trị `TRUE` khi và chỉ khi tất cả 3 hàm trên đều trả về `FALSE`.

Mở file

Để mở một file cho việc kết xuất dữ liệu chúng ta tạo ra một đối tượng `ofstream`:

```
ofstream fout("out.txt");
```

và để mở file nhập dữ liệu cũng tương tự:

```
ifstream fin("inp.txt");
```

Sau khi tạo ra các đối tượng này chúng ta có thể dùng chúng giống như các thao tác vẫn được thực hiện với `cout` và `cin` nhưng cần chú ý có hàm `close()` trước khi kết thúc chương trình.

Ví dụ:

```
ifstream fin("data.txt");
```

```
while(fin.get(ch))
```

```
    cout << ch;
```

```
fin.close();
```

Thay đổi thuộc tính mặc định khi mở file

Khi chúng ta mở một file để kết xuất dữ liệu qua một đối tượng `ofstream`, nếu file đó đã tồn tại nội dung của nó sẽ bị xóa bỏ còn nếu nó không tồn tại thì file mới sẽ được tạo ra. Nếu chúng ta muốn thay đổi các hành động mặc định này có thể truyền thêm một biến tường minh vào cấu tử của lớp `ofstream`.

Các tham số hợp lệ có thể là:

`ios::app` – append, mở rộng nội dung một file nếu nó đã có sẵn.

`ios::ate` -- đặt vị trí vào cuối file nhưng có thể ghi lên bất cứ vị trí nào trong file

`ios::trunc` -- mặc định

`ios::nocreate` -- nếu file không có sẵn thao tác mở file thất bại

`ios::noreplace` -- Nếu file đã có sẵn thao tác mở file thất bại.

Chú ý: nên kiểm tra khi thực hiện mở một file bất kỳ. Nên sử dụng lại các đối tượng `ifstream` và `ofstream` bằng hàm `open()`.

10. File text và file nhị phân

Một số hệ điều hành chẳng hạn DOS phân biệt các file nhị phân và các file text. Các file text lưu trữ dữ liệu dưới dạng text chẳng hạn một số sẽ được lưu thành một xâu, việc lưu trữ theo kiểu này có nhiều bất lợi song chúng ta có thể xem nội dung file bằng các chương trình rất đơn giản.

Để giúp phân biệt giữa các file text và nhị phân C++ cung cấp cờ `ios::binary`. Trên một số hệ thống cờ này thường được bỏ qua vì thường thì dữ liệu được ghi dưới dạng nhị phân.

Các file nhị phân không chỉ lưu các số và ký tự chúng có thể được sử dụng để lưu các cấu trúc. Để ghi một biến cấu trúc lên một file nhị phân chúng ta dùng hàm `write()`, ví dụ:

```
fout.write((char*) &Bear, sizeof Bear);
```

Để thực hiện việc đọc ngược lại chúng ta dùng hàm `read`.

```
fin.read((char*) &BearTwo, sizeof BearTwo);
```

Làm việc với máy in

Làm việc với máy in tương đối giống với làm việc với các file:

```
ofstream prn("PRN");
```

11. Tìm kiếm trong các dòng vào ra

Có hai cách tiếp cận để giải quyết vấn đề này, cách thứ nhất sử dụng vị trí tuyệt đối của dòng gọi là `streampos`, cách thứ hai là sử dụng hàm có cách thức làm việc giống như hàm `fseek()` của ngôn ngữ C.

Cách tiếp cận `streampos` đòi hỏi chúng ta trước tiên cần phải sử dụng một hàm chỉ chỗ (`tell`): `tellp()` cho một đối tượng ostream và `tellg()` một đối tượng istream. Hàm chỉ chỗ này sẽ trả về một giá trị `streampos` mà sau đó chúng ta có thể sử dụng với hàm `seekp()` đối với một đối tượng ostream và `seekg()` đối với một đối tượng istream khi chúng ta muốn nhảy tới vị trí đó của file.

Cách tiếp cận thứ hai sử dụng các phiên bản overload của các hàm `seekp()` và `seekg()`. Tham số thứ nhất được truyền vào là số byte di chuyển, đó có thể là một số nguyên âm hoặc nguyên dương.. Tham số thứ hai là hướng tìm kiếm:

<code>ios::beg</code>	Từ đầu dòng
<code>ios::cur</code>	Từ vị trí hiện tại
<code>ios::end</code>	Từ cuối dòng

Ví dụ minh họa:

```
//: C02:Seeking.cpp
// Seeking in iostreams
#include <iostream.h>
#include <fstream.h>

int main(int argc, char* argv[]) {
    ifstream in(argv[1]);
    in.seekg(0, ios::end); // Cuối file
    streampos sp = in.tellg(); // kích thước file
    cout << "file size = " << sp << endl;
    in.seekg(-sp/10, ios::end);
    streampos sp2 = in.tellg();
    in.seekg(0, ios::beg); // vị trí bắt đầu file
```



```

        cout << in.rdbuf(); // in ra cả file
        in.seekg(sp2); // di chuyển tới vị trí
streampos
        // in ra phần còn lại của file
        cout << endl << endl << in.rdbuf() << endl;
    } ///:~

```

Vì kiểu của `streampos` được định nghĩa là `long` nên `tellg()` sẽ cho ta biết kích thước của file. Hàm `rdbuf()` trả về con trỏ `streambuf()` của một đối tượng `istream` bất kỳ.

12. stringstream

Trước khi có các đối tượng `stringstream` chúng ta đã có các đối tượng cơ bản hơn là `stringstream`. Mặc dù không phải là một phần chính thức của C++ song vì đã từng được sử dụng trong một thời gian dài các trình biên dịch đều hỗ trợ `stringstream`. Tuy nhiên chúng ta nên dùng các đối tượng `stringstream` thay cho `stringstream` vì các lý do sau đây:

một đối tượng `stringstream` làm việc trực tiếp với bộ nhớ thay vì với các file hay output chuẩn. Nó cho phép chúng ta sử dụng các hàm đọc và định dạng để thao tác với các byte trong bộ nhớ. Để làm việc với các đối tượng `stringstream` chúng ta cần tạo ra các đối tượng `ostringstream` hoặc `istringstream` tương ứng.

Các đối tượng `stringstream` cũng làm việc với bộ nhớ song đơn giản hơn nhiều và đó chính là lý do nên dùng chúng thay vì `stringstream`.

Vấn đề cấp phát bộ nhớ

Cách tiếp cận dễ nhất để hiểu vấn đề này là khi người dùng cần phải cấp phát bộ nhớ cho các đối tượng cần sử dụng. Đối với các đối tượng `istringstream` chúng ta có một cách tiếp cận duy nhất với hai cấu tử:

```

istringstream(char * buf);
istringstream(char * buf, int size);

```

Cấu tử thứ nhất cho một đối tượng `stringstream` với một xâu có ký tự kết thúc, chúng ta có thể trích ra các ký tự cho tới khi gặp ký tự kết thúc, cấu tử thứ hai cho một đối tượng `stringstream` mà chúng ta có thể trích ra số ký tự bằng đúng giá trị của tham số thứ hai.

Ví dụ:

```

//: C02:Istring.cpp
// Input streams
#include <iostream.h>
#include <sstream.h>

int main() {
    istringstream s("47 1.414 This is a test");
    int i;
    float f;
    s >> i >> f;
    char buf2[100];
    s >> buf2;
    cout << "i = " << i << ", f = " << f;
    cout << " buf2 = " << buf2 << endl;
    cout << s.rdbuf(); // in ra phần còn lại...
} ///:~

```

Với các đối tượng `ostrstream` chúng ta có hai cách tiếp cận , cách thứ nhất thông qua một cấu tử của lớp:

```
ostrstream::ostrstream(char *, int, int = ios::out);
```

Tham số thứ nhất là một bộ đệm chứa các ký tự , tham số thứ hai là kích thước bộ đệm, và tham số thứ 3 là chế độ làm việc , nếu thiếu tham số này , các ký tự được định dạng bắt đầu từ địa chỉ của bộ đệm . Nếu tham số thứ 3 là `ios::ate` hay `ios::app` thì chương trình sẽ giả thiết như là ký tự bộ đệm đã chứa một xâu có ký tự kết thúc là `'\0'` và bất kỳ ký tự mới nào cũng sẽ được thêm vào bắt đầu từ ký tự kết thúc đó. Tham số thứ 2 được dùng để đảm bảo rằng không xảy ra trường hợp ghi đè lên cuối mảng .

Một điều quan trọng cần phải nhớ khi làm việc với một đối tượng `ostrstream` là ký tự kết thúc xâu không được tự động thêm vào cuối xâu, khi định dạng xong cần cùng `endl` để thực hiện điều đó.

Ví dụ:

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

int main() {
    const int sz = 100;
    cout << "Nhập vào 1 số nguyên, 1 số thực và\n"
    một xâu:";
    int i;
    float f;
    cin >> i >> f;
    cin >> ws; // bỏ đi các dấu cách
    char buf[sz];
    cin.getline(buf, sz); // Lấy phần còn lại của
    cin
    ostrstream os(buf, sz, ios::app);
    os << endl;
    os << "integer = " << i << endl; // chú ý nếu
    không có // tham số thứ 3 ios::app thì dòng
    lệnh sau sẽ ghi đè
    // lên nội dung của dòng lệnh trước
    os << "float = " << f << endl;
    os << ends;
    cout << buf;
    cout << os.rdbuf();
    cout << os.rdbuf();
} ///:~
```

Cách thứ hai để tạo ra một đối tượng `ostrstream`:

```
ostrstream a;
```

Cách khai báo này linh hoạt hơn, chúng ta không cần chỉ định kích thước cũng như bộ đệm của một đối tượng `ostrstream` tuy nhiên cũng rắc rối hơn.

Ví dụ làm thế nào chúng ta có thể lấy địa chỉ của vùng nhớ mà các ký tự của đối tượng `a` đã được định dạng:

```
char *cp = a.str();
```

Tuy nhiên vẫn còn vấn đề nảy sinh ở đây . Làm thế nào nếu chúng ta muốn thêm các ký tự vào `a`. Điều này có thể dễ dàng thực hiện nếu chúng ta biết trước là `a` đã được

cung cấp đủ để thực hiện thêm vào . Thông thường a sẽ hết bộ nhớ khi chúng ta thêm vào các ký tự và nó sẽ cố gắng xin cấp phát thêm vùng nhớ (on the heap) để làm việc. Điều này thường đòi hỏi di chuyển các khối c ủa bộ nhớ. Nhưng các đối tượng stream làm việc thông qua địa chỉ khối nhớ của chúng nên chúng có vẻ không tài năng lắm trong việc di chuyển các khối đó, vì chúng ta hy vọng chúng tập trung vào một chỗ.

Cách giải quyết củ a các đối tượng ostrstream trong trường hợp này là tự làm đóng băng chúng. Chừng nào mà chương trình chưa gọi tới hàm str () thì chúng ta có thể thêm bao nhiêu ký tự vào tùy thích , đối tượng ostrstream sẽ xin thêm bộ nhớ heap để làm việc và khi đối tượng không được sử dụng nữa vùng nhớ đó sẽ tự động được giải phóng.

Sau khi hàm str () được gọi tới chúng ta không thể thêm bất cứ ký tự nào vào một đối tượng ostrstream , điều này thực tế không được kiểm tra nhưng đối tượng ostrstream tương ứng sẽ không chịu trách nhiệm dọn dẹp vùng nhớ mà chúng ta đã xin thêm.

Để ngăn chặn một số trường hợp rò rỉ bộ nhớ có thể xảy ra chúng ta có thể gọi tới toán tử delete:

```
delete []a.str();
```

Cách thứ hai, không phổ biến lắm là chúng ta có thể giải phóng bộ nhớ đã chiếm dụng bằng cách gọi tới hàm freeze (). Hàm freeze () là một thành viên của lớp ostrstream, nó có một tham số mặc định bằng một dù ng để đóng băng nhưng nếu tham số có giá trị bằng 0 thì tác dụng của hàm lại là giải phóng bộ nhớ.

Sau khi giải phóng bộ nhớ chúng ta có thể thêm các ký tự vào đối tượng vừa được giải phóng tuy nhiên điều này thường gây ra các di chuyển các khối nhớ nên tốt nhất là không sử dụng các con trỏ tới đối tượng này trước đó mà chúng ta đã có bằng cách dùng hàm str().

Ví dụ;

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

int main() {
    ostrstream s;
    s << "'The time has come', the walrus said,";
    s << ends;
    cout << s.str() << endl;
    s.seekp(-1, ios::cur);
    s.rdbuf()->freeze(0);
    s << " 'To speak of many things'" << ends;
    cout << s.rdbuf();
} ///:~
```

Và chú ý hàm rdbuf() cho phép ta lấy nội dung phần còn lại , hoặc toàn bộ của một đối tượng iostream (giống như với file).

Còn rất nhiều chi tiết về các cờ và đối tượng iostream khác , nếu muốn tham khảo đầy đủ có thể tham khảo trong các sách tham khảo .

Một số câu hỏi:

Khi nào sử dụng các toán tử >>, << và khi nào dùng các hàm thành viên cho các thao tác vào ra dữ liệu?

Sự khác nhau giữa cerr và clog?

Tại sao lại sử dụng các dòng vào ra thay cho hàm printf()?

Chương 5: Đối tượng và lớp. (9 tiết)

1. Trừu tượng dữ liệu

C++ là một công cụ năng suất. Tại sao chúng ta lại phải cố gắng (một cố gắng, dù cho đó là một cố gắng dễ dàng) để chuyển từ một ngôn ngữ mà chúng ta đã biết và hiệu quả sang một ngôn ngữ mới kém hiệu quả hơn trong một khoảng thời gian nào đó, cho tới khi mà chúng ta thực sự nắm được nó. Đó là bởi vì chúng ta đã bị thuyết phục rằng chúng ta có thể đạt được những kết quả lớn nhờ vào việc sử dụng công cụ mới này.

Hiệu suất, nói theo ngôn ngữ tin học có nghĩa là ít người hơn có thể tạo ra các chương trình ấn tượng và phức tạp hơn trong thời gian ngắn hơn. Tất nhiên là còn có các yếu tố khác khi chúng ta cần lựa chọn một ngôn ngữ nào đó chẳng hạn như tính hiệu quả (bản chất của ngôn ngữ có làm cho chương trình chậm hay khó gỡ lỗi hay không), tính an toàn (ngôn ngữ có giúp chúng ta đảm bảo rằng nó sẽ thực hiện những gì mà chúng ta yêu cầu) và vân vân.

Tuy nhiên thực sự năng suất có nghĩa là một chương trình trước đây 3 người làm mất một tuần giờ đây một người có thể viết trong một hai ngày. Điều này mang lại nhiều lợi ích cho cả những người viết chương trình và khách hàng của họ. Và cách duy nhất để có được điều đó là sử dụng lại, tận dụng lại mã chương trình của người khác, đó chính là sử dụng thư viện.

Một thư viện đơn giản là một tập hợp các đoạn mã nguồn mà một ai đó đã viết và đóng gói chúng lại với nhau. Thường thường mỗi thư viện là một file có cấu trúc đặc biệt để trình liên kết có thể tìm tới đúng các đoạn mã hàm được gọi trong chương trình sử dụng chúng. Các thư viện này thường được cung cấp dưới dạng đã được đóng gói nhưng cũng có thể là có cả mã nguồn, điều này thường thích hợp với những hệ thống hoạt động trên nhiều nền tảng khác nhau chẳng hạn Unix.

Như vậy thư viện có lẽ là cách quan trọng nhất để cải tiến tính hiệu quả, và một trong các mục tiêu chính của C++ là làm cho việc sử dụng các thư viện trở nên dễ dàng hơn, và điều này cũng có nghĩa là việc sử dụng các thư viện trong C có một điều gì đó khó khăn.

Ví dụ 1: Một thư viện theo kiểu ngôn ngữ C

Một thư viện thường là một tập hợp các hàm, nhưng nếu chúng ta đã từng sử dụng một thư viện được cung cấp bởi một bên thứ 3 nào đó thì chúng ta có thể hiểu rằng không chỉ đơn giản là tập hợp các hàm mà thư viện còn có nhiều hơn thế, đó chính là các thuộc tính hay đặc điểm của dữ liệu. Khi chúng ta bắt đầu phải làm việc với một tập hợp các thuộc tính trong C chúng ta thường hay sử dụng một kiểu dữ liệu đặc biệt, đó là kiểu dữ liệu cấu trúc: struct, kiểu dữ liệu minh họa cho tư tưởng lập trình cấu trúc: chương trình = dữ liệu + thuật toán.

Start here

Khi chúng ta tiếp cận một vấn đề trong một ngôn ngữ lập trình hướng đối tượng, chúng ta sẽ không tìm cách chia vấn đề cần giải quyết thành các hàm con, các chương trình con như trong lập trình có cấu trúc mà điều cần thiết là chia nó thành các đối tượng.

Suy nghĩ theo kiểu các đối tượng thay vì các hàm có một ảnh hưởng rất lớn đến việc thiết kế chương trình. Vì thế giới thực bao gồm các đối tượng và có một sự liên quan mật thiết giữa các đối tượng trong một chương trình với các đối tượng trong thế giới thực.

1. Thế nào là một đối tượng?

Rất nhiều các đối tượng trong thế giới thực có hai đặc tính sau : chúng có một trạng thái (state) (các thuộc tính có thể thay đổi) và các năng lực (công việc mà chúng có thể thực hiện).

Đối tượng thực = Trạng thái (các thuộc tính)+ Các năng lực (hành vi)

Đối tượng lập trình = Dữ liệu + Các hàm

Kết quả của việc trừu tượng hóa các đối tượng của thế giới thực thành các đối tượng lập trình là sự kết hợp giữa dữ liệu và các hàm.

2. Các lớp và các đối tượng

Lớp là một kiểu dữ liệu mới được dùng để định nghĩa các đối tượng . Một lớp có vai trò như một kế hoạch hay một bản mẫu . Nó chỉ rõ dữ liệu nào (các thuộc tính - trạng thái) và các hàm (các năng lực) nào sẽ thuộc về các đối tượng của lớp đó. Việc viết hay tạo ra một lớp mới không sinh ra bất cứ một đối tượng nào trong chương trình.

Một lớp là sự trừu tượng hóa , tổng quát hóa các đối tượng có các thuộc tính giống nhau và các đối tượng là thể nghiệm của các lớp.

Ví dụ 1: Lớp point định nghĩa các điểm ảnh trong một chương trình đồ họa.

Mỗi điểm phải có hai trạng thái là hoành độ và tung độ , chúng ta có thể dùng hai biến kiểu int để biểu diễn chúng.

Trong chương trình của chúng ta các điểm phải có các khả năng sau :

- Các điểm có thể di chuyển trên màn hình : điều này được thực hiện qua hàm **move**
- Các điểm có thể in ra các tọa độ của chúng lên màn hình : hàm **print**
- Các điểm có thể trả lời câu hỏi xem chúng có đang ở vị trí gốc tọa độ hay không: hàm **isZero()**

```
class Point{ // khai báo lớp Point
    int x, y; // hoành độ và tung độ
public:
    void move(int,int);
    void print();
    bool isZero();
};
```

Ở đây có hai quan điểm trong việc khai báo một lớp : quan điểm thứ nhất cho rằng nên khai báo các biến và phương thức private trước , quan điểm thứ hai lại cho rằng nên khai báo các phương thức và biến public trước vì đây chính là phần giao diện cho phép các đối tượng của lớp thực hiện các giao tiếp với bên ngoài qua các thông điệp và đối với lập trình viên thì chỉ cần quan tâm tới phần này.

Các hàm và các biến trong một lớp được gọi là các thành viên của lớp . Trong ví dụ này chỉ có các nguyên mẫu hàm được đặt trong khai báo lớp , phần cài đặt sẽ được đặt ở các phần khác (có thể trong một file khác) của chương trình.

Nếu như phần định nghĩa hay cài đặt của một hàm được đặt ngay trong phần khai báo của lớp thì hàm đó được gọi là một hàm inline (macro).

Một số chú ý về hàm inline:

Chúng ta có thể sử dụng từ khóa inline để chỉ định một hàm là inline:

```
inline void Point::move(int newX, int newY){
    //
} //
```

Không phải lúc nào hàm inline cũng được thực hiện : có 2 trường hợp mà trình biên dịch sẽ từ chối chấp nhận một hàm là hàm inline trường hợp thứ nhất là khi hàm đó quá phức tạp, trường hợp thứ hai là nó gọi tới một hàm có địa chỉ không xác định , ví dụ:

```
class Forward {
    int i;
public:
    Forward() : i(0) {}
    // gọi tới hàm chưa được khai báo:
    int f() const { return g() + 1; }
    int g() const { return i; }
};

int main() {
    Forward frwd;
    frwd.f();
} //:~
```

Các hàm inline thường thấy chính là các cấu tử và các hủy tử , trong thân các hàm này chúng ta hay thực hiện một số thao tác khởi tạo và dọn dẹp ẩn.

// hàm di chuyển tới vị trí mới

```
void Point::move(int new_x, int new_y)
```

```
{
```

```
    x = new_x;
```

```
    y = new_y;
```

```
}
```

// in tọa độ ra màn hình

```
void Point::print()
```

```
{
```

```
    cout << "X= " << x << ", Y= " << y << endl;
```

```
}
```

// kiểm tra xem có trùng với gốc tọa độ không (0,0)

```
bool Point::is_zero()
```

```
{
```

```
    return (x == 0) && (y == 0); // if x=0 AND y=0 returns true
```

```
}
```

Vậy là chúng ta đã có một kiểu dữ liệu mới để xây dựng các đối tượng điểm chúng ta có thể tạo ra các đối tượng này trong chương trình khi cần thiết:

```
int main()
```

```
{
```

```
    Point point1, point2; // khai báo 2 đối tượng: point1 and point2
```

```
    point1.move(100,50); // di chuyển tới điểm (100,50)
```

```
    point1.print(); // in tọa độ ra màn hình
```

```
    point1.move(20,65); // point1 di chuyển tới điểm (20,65)
```

```
    point1.print(); // in tọa độ ra màn hình
```

```
    if(point1.is_zero()) // kiểm tra xem có trùng với gốc tọa độ không
```

```
        cout << "point1 is now on zero point(0,0)" << endl;
```

```
    else
```

```

        cout << "point1 is NOT on zero point(0,0)" << endl;
    point2.move(0,0); // point2 moves to (0,0)
    if(point2.is_zero()) // is point2 on (0,0)?
        cout << "point2 is now on zero point(0,0)" << endl;
    else
        cout << "point2 is NOT on zero point(0,0)" << endl;
    return 0;
}

```

Nhắc lại một số khái niệm và thuật ngữ:

Lớp: một lớp là kết quả của việc gom dữ liệu và các hàm. Khái niệm lớp rất giống với khái niệm cấu trúc được sử dụng trong C hay khái niệm bản ghi được sử dụng trong Pascal, nó là một kiểu dữ liệu được dùng để tạo ra các biến có thể được dùng trong một chương trình nào đó.

Đối tượng: Một đối tượng là một thể nghiệm của một lớp, điều này tương tự như đối với một biến được định nghĩa là một thể hiện của một kiểu dữ liệu, nó mang tính cụ thể và xác định.

Phương thức (Method) (member function) là một hàm được khai báo trong một lớp.

Thông điệp: đây là khái niệm tương đối giống với việc gọi tới một hàm tuy nhiên có sự khác nhau về cơ bản ở hai điểm sau đây:

- Một thông điệp phải được gửi tới cho một đối tượng nào đó dù có thể là tường minh hay cụ thể gọi là receiver.
- Hành động được thực hiện để đáp lại thông điệp được quyết định bởi receiver (đối tượng nhận thông điệp), các receiver có thể có các hành động khác nhau đối với cùng một thông điệp.

Ví dụ:

```
anObject.doSomething(...);
```

Nhiều người mới học C++ cho rằng hai khái niệm gọi hàm và gửi thông điệp trong các chương trình là giống nhau hoàn toàn.

Kết luận:

Sau phần này chúng ta cần chú ý các điểm sau:

Các chương trình được xây dựng theo kiểu hướng đối tượng bao gồm các đối tượng, và việc này thường được bắt đầu bằng việc xây dựng và thiết kế các lớp. Thay vì các lời gọi hàm trong một chương trình hướng đối tượng chúng ta thực hiện gửi các thông điệp tới cho các đối tượng để bảo chúng thực hiện một công việc nào đó.

3. Con trỏ và mảng các đối tượng

Các lớp có thể được dùng để khai báo các biến giống như các kiểu dữ liệu cơ bản khác, chẳng hạn chúng ta có thể khai báo các con trỏ tới các đối tượng:

```

int main()
{
    Point *pp1 = new Point; // cấp phát bộ nhớ cho 1 đối tượng, pp1 trỏ tới
    Point *pp2 = new Point; // cấp phát bộ nhớ cho 1 đối tượng, pp2 trỏ tới
    pp1->move(50,50);
    pp1->print();
    pp2->move(100,150);
    if(pp2->is_zero())
        cout << " Object pointed by pp2 is on zero." << endl;
    else

```

```

        cout << " Object pointed by pp2 is NOT on zero." << endl;
        delete pp1; // giải phóng bộ nhớ
        delete pp2;
        return 0;
}

```

Hoặc chúng ta cũng có thể khai báo mảng các đối tượng (có thể là mảng tĩnh hoặc mảng động):

```

int main()
{
    Point array[10]; // khai báo 1 mảng có 10 phần tử
    array[0].move(15,40);
    array[1].move(75,35);

    for (int i= 0; i < 10; i++)
        array[i].print();
    return 0;
}

```

4. Khai báo các lớp với các file header

Ví dụ:

File stack.h

```
#ifndef Stack_H
```

```
#define Stack_H
```

```

class Stack {
// LIFO objects
public:
    Stack(int MaxStackSize = 10);
    ~Stack() {delete [] stack;}
    bool IsEmpty() const {return top == -1;}
    bool IsFull() const {return top == MaxTop;}
    int top() const;
    void push(const int & x);
    int pop();
private:
    int top; // chỉ số phần tử ở đỉnh
    int MaxTop; // giá trị lớn nhất của biến top
    int *stack; // dữ liệu của stack
};
#endif

```

file stack.cpp:

```

Stack::Stack(int MaxStackSize)
{// cấu tử của lớp Stack
    MaxTop = MaxStackSize - 1;
    stack = new int[MaxStackSize];
    top = -1;
}

```



```

int Stack::top() const
{// trả về phần tử ở đỉnh
    if (IsEmpty()){
        cout << "OutOfBounds"; // stack rỗng
        return -1;
    }
    else return stack[top];
}

void Stack::push(const int& x)
{// Add x to stack.
    if (IsFull()) {
        cout << "NoMem"; // hết bộ nhớ
        return;
    }
    stack[++top] = x;
}

int Stack::pop()
{// loại bỏ phần tử ở đỉnh
    if (IsEmpty()){
        cout << "OutOfBounds"; // delete fails
        return -1;
    }
}

```

Việc sử dụng các file header để khai báo các lớp và các hàm nhằm mục đích chính là tách biệt phần giao diện và phần cài đặt của các lớp, các hàm. Như chúng ta đã biết các chương trình hiện nay thường được viết theo nhóm làm việc. Trong nhóm mỗi người được giao viết một phần của chương trình và do đó cần dùng phần chương trình của người khác, và việc sử dụng các file header cũng là một phần giải pháp để đạt được điều đó. Khi một lập trình viên thay đổi các cài đặt bên dưới của một hàm hay một lớp nào đó thì việc thay đổi này không làm ảnh hưởng tới phần chương trình do người khác viết dựa trên các lớp và hàm mà anh ta đưa ra.

Cấu trúc của một file header thường có 3 chỉ thị tiền xử lý chính, chỉ thị đầu tiên là chỉ thị kiểm tra sự tồn tại của một cờ định danh, nó có tác dụng kiểm tra xem nội dung của file đã được include vào một file nào đó hay chưa, nếu rồi thì phần nội dung tiếp theo của file sẽ được bỏ qua. Chỉ thị thứ hai là chỉ thị định nghĩa, xác định một định danh dùng cho việc kiểm tra được thực hiện trong quá trình biên dịch.

Chỉ thị cuối cùng là chỉ thị kết thúc nội dung của file.

Nếu không có các chỉ thị này chúng ta sẽ gặp rắc rối to với vấn đề include các file header trong các chương trình.

Nếu muốn tìm hiểu kỹ hơn về các chỉ thị và cấu trúc của file header chúng ta có thể tham khảo nội dung các file header chuẩn được cung cấp với trình biên dịch.

Việc biên dịch chương trình đối với các chương trình sử dụng lớp Stack được thực hiện như sau:

Trong chương trình có sử dụng lớp Stack chúng ta cần include file header stack.h. Để thử xem có lỗi không nhấn tổ hợp phím ALT -C để kiểm tra các lỗi biên dịch. Việc dịch tiếp theo (phần build file .exe) có thể được thực hiện theo hai cách: cách thứ nhất

là tạo một project và dịch như các chương trình chỉ có 1 file .cpp mà chúng ta đã biết; cách thứ hai là tạo một file .bat để thực hiện biên dịch, ví dụ với một chương trình gồm các file: main.cpp, stack.h, stack.cpp thì nội dung file build.bat sẽ có thể là:

```
tcc -c main.cpp
tcc -c stack.cpp
tcc -e<tên file.exe> main.obj stack.obj
```

Hoặc gọn hơn nữa là:

```
tcc -c *.cpp
tcc -e<tên file.exe> *.obj
del *.obj
```

Hoặc chúng ta cũng có thể thực hiện trực tiếp các lệnh này trên dòng lệnh trong các cửa sổ giả lập DOS.

5. Kiểm soát việc truy cập tới các biến và phương thức của lớp

Chúng ta có thể chia các lập trình viên thành hai nhóm : nhóm tạo các lớp (nhóm gồm những người tạo ra các kiểu dữ liệu mới) và các lập trình viên khách (các lập trình viên sử dụng các kiểu dữ liệu do nhóm thứ nhất tạo ra trong chương trình của họ).

Mục đích của nhóm thứ nhất là xây dựng một lớp bao gồm tất cả các thuộc tính và khả năng cần thiết. Lớp này sẽ cung cấp các hàm giao diện cần thiết , chỉ những gì cần thiết cho các lập trình viên sử dụng chúng và giữ bí mật các phần còn lại .

Mục đích của các lập trình viên khách là tập hợp một hộp công cụ với đầy các lớp để nhanh chóng sử dụng phát triển xây dựng chương trình ứng dụng của mình.

Lý do đầu tiên cho việc kiểm soát truy cập này là đảm bảo các lập trình viên khách không thể thò tay vào những phần mà họ không nên thò tay vào . Phần được ẩn đi là phần cần thiết cho cấu trúc bên trong của lớp và không phải là phần giao diện mà người dùng cần để giải quyết vấn đề của họ .

Lý do thứ hai là nếu như việc kiểm soát truy cập bị ẩn đi thì các lập trình viên khách không thể sử dụng nó , có nghĩa là người tạo ra các lớp có thể thay đổi các phần bị ẩn mà không cần phải lo lắng sẽ ảnh hưởng tới bất kỳ ai .

Sự bảo vệ này cũng ngăn chặn các thay đổi ngoài ý muốn đối với các trạng thái của các đối tượng.

Để phục vụ cho việc kiểm soát truy cập các thành viên của một lớp C ++ cung cấp 3 nhãn: public, private và protected.

Các thành viên đứng sau một nhãn sẽ được gán nhãn đó cho tới khi nhãn mới xuất hiện.

Các thành viên được gán nhãn private (mặc định) chỉ có thể được truy cập tới bởi các thành viên khác của lớp.

Mục đích chính của các thành viên được gán nhãn public là cung cấp cho các client danh sách các dịch vụ mà lớp đó hỗ trợ hay cung cấp. Tập các thành viên này tạo nên phần giao diện công cộng của một lớp.

Chúng ta có thể thấy rõ qua lớp Point trong ví dụ ở phần đầu , việc truy cập vào biến thành viên chẳng hạn x là bất hợp lệ.

Các cấu trúc trong một chương trình C++ có chế độ truy cập mặc định là public .

Từ khóa protected có ý nghĩa giống hết ý nghĩa của từ khóa private ngoại trừ một điều là các lớp kế thừa có thể truy cập vào các thành viên protected của lớp cơ sở ở mà nó kế thừa.

6. Các hàm bạn và các lớp bạn

Một hàm hoặc một thực thể lớp có thể được khai báo là một bạn bè của một lớp khác.

Một bạn bè của một lớp có quyền truy cập vào tất cả các thành viên (private, protected, public) của lớp. Ví dụ:

```
class A{
    friend class B;
private:
    int i;
    float f;
public:
    void fonk1(char *c);
};
class B{
    int j;
public:
    void fonk2(A &s){ cout << s.i;} // lớp B có thể truy cập tới mọi thành
    // viên của A nhưng ngược lại thì không thể vì A không là bạn của B
};
```

Một hàm bạn có quyền truy cập vào tất cả các thành viên của lớp :

```
class Point{
    friend void zero(Point &);
    int x,y;
public:
    bool move(int x, int y);
    void print();
    bool is_zero();
};
void zero(Point & p){
    p.x = p.y = 0;
}
```

Hoặc chúng ta cũng có thể khai báo một hàm thành viên của một lớp nào đó là hàm bạn của một lớp:

```
class X;

class Y {
    void f(X*);
};

class X { // Definition
private:
    int i;
public:
    void initialize();
    friend void g(X*, int); // hàm bạn toàn cục
    friend void Y::f(X*); // hàm bạn của lớp khác
    friend void h();
};
```

Trong ví dụ này chúng ta thấy lớp Y có một hàm thành viên `f()`, hàm này sẽ làm việc với một đối tượng thuộc lớp X. Điều này hơi phức tạp vì trình biên dịch C++ yêu cầu tất cả mọi thứ đều phải được khai báo trước khi chúng ta tham chiếu tới chúng, vì thế lớp Y phải khai báo thành viên của nó là hàm `Y::f(X*)` trước khi có thể khai báo hàm này là một hàm bạn của lớp X. Nhưng hàm `Y::f(X*)` có truy cập tới đối tượng thuộc lớp X vì thế nên lớp X cũng phải được khai báo trước đó.

Điều này khá nan giải, tuy nhiên chúng ta chú ý tới hàm `f()`, tham số truyền vào cho hàm là một biến con trỏ có nghĩa là chúng ta sẽ truyền một địa chỉ và thật may mắn là trình biên dịch luôn biết cách làm việc với các biến địa chỉ dù cho đó là địa chỉ của một đối tượng thuộc kiểu gì chẳng nữa dù cho nó không biết chính xác về định nghĩa hay đơn giản là kích thước của kiểu đó. Điều này cũng có nghĩa là nếu hàm thành viên chúng ta cần khai báo lớp X một cách đầy đủ trước khi muốn khai báo một hàm chẳng hạn như `Y::g(X)`.

Bằng cách truyền theo địa chỉ trình biên dịch cho phép chúng ta thực hiện một đặc tả kiểu không hoàn chỉnh (incomplete type specification) về lớp X trước khi khai báo `Y::f(X*)`. Điều này được thực hiện bằng cách khai báo:

```
class X;
```

Khai báo này đơn giản báo cho trình biên dịch biết là có một lớp có tên là X, và vì thế việc tham chiếu tới lớp đó sẽ là hợp lệ miễn là việc tham chiếu đó không đòi hỏi nhiều hơn 1 cái tên.

Một điều nữa mà chúng ta cần chú ý là trong trường hợp khai báo các lớp lồng nhau, lớp ngoài cũng không có quyền truy cập vào các thành phần private của lớp trong, để đạt được điều này chúng ta cũng cần thực hiện tương tự như trên, ví dụ:

```
#include <iostream.h>
#include <string.h> // memset()
const int sz = 20;

class Holder {
private:
    int a[sz];
public:
    void initialize();
    class Pointer;
    friend Pointer;
    class Pointer {
    private:
        Holder* h;
        int* p;
    public:
        void initialize(Holder* h);
        // Move around in the array:
        void next();
        void previous();
        void top();
        void end();
        // Access values:
        int read();
        void set(int i);
```

```

        };
};
void Holder::initialize() {
    memset(a, 0, sz * sizeof(int));
}
void Holder::Pointer::initialize(Holder* rv) {
    h = rv;
    p = rv->a;
}
void Holder::Pointer::next() {
    if(p < &(h->a[sz - 1])) p++;
}
void Holder::Pointer::previous() {
    if(p > &(h->a[0])) p--;
}
void Holder::Pointer::top() {
    p = &(h->a[0]);
}
void Holder::Pointer::end() {
    p = &(h->a[sz - 1]);
}
int Holder::Pointer::read() {
    return *p;
}
void Holder::Pointer::set(int i) {
    *p = i;
}
int main() {
    Holder h;
    Holder::Pointer hp, hp2;
    int i;
    h.initialize();
    hp.initialize(&h);
    hp2.initialize(&h);
    for(i = 0; i < sz; i++) {
        hp.set(i);
        hp.next();
    }
    hp.top();
    hp2.end();
    for(i = 0; i < sz; i++) {
        cout << "hp = " << hp.read()
        << ", hp2 = " << hp2.read() << endl;
        hp.next();
        hp2.previous();
    }
}

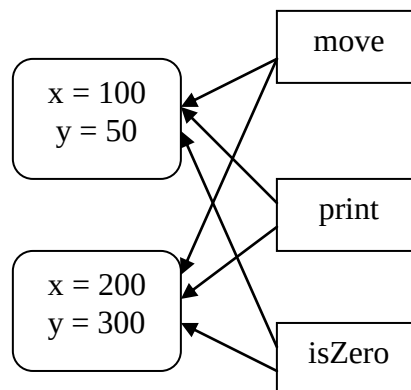
```

Các hàm friend không phải là thành viên của lớp nhưng chúng buộc phải xuất hiện trong khai báo lớp và do đó tất cả mọi người đều biết đó là một hàm ưu tiên và điều này thực sự là không an toàn . Bản thân C++ không phải là một ngôn ngữ hướng đối tượng hoàn toàn và việc sử dụng các lớp và hàm friend là nhằm giải quyết các vấn đề thực tế đồng thời cũng làm cho tính hướng đối tượng của ngôn ngữ giảm đi đáng kể.

7. Con trỏ this

Mỗi đối tượng có không gian dữ liệu riêng của nó trong bộ nhớ của máy tính . Khi mỗi đối tượng được định nghĩa, phân bộ nhớ được khởi tạo chỉ dành cho phần lưu dữ liệu của đối tượng đó.

Mã của các hàm thành viên chỉ được tạo ra một lần . Các đối tượng của cùng một lớp sẽ sử dụng chung mã của các hàm thành viên.



Vậy làm thế nào để trình biên dịch có thể đảm bảo được rằng việc tham chiếu này là đúng đắn. Để đảm bảo điều này trình biên dịch duy trì một con trỏ được gọi là con trỏ this. Với mỗi đối tượng trình biên dịch đều sinh ra một con trỏ this gắn với nó . Khi một hàm thành viên được gọi đến , con trỏ this chứa địa chỉ của đối tượng sẽ được sử dụng và chính vì vậy các hàm thành viên sẽ truy cập tới đúng các thành phần dữ liệu của đối tượng thông qua địa chỉ của đối tượng . Chúng ta cũng có thể sử dụng con trỏ this trong các chương trình một cách rõ ràng, ví dụ:

```

Point *Point::far_away(Point &p)
{
    unsigned long x1 = x*x;
    unsigned long y1 = y*y;
    unsigned long x2 = p.x * p.x;
    unsigned long y2 = p.y * p.y;
    if ( (x1+y1) > (x2+y2) ) return this; // trả về địa chỉ của đối tượng
    else return &p; // trả về đối tượng đang được tạo ra
}
  
```

8. Khởi tạo các đối tượng của lớp thông qua các hàm cấu tử

Như chúng ta đã biết các đối tượng trong mỗi chương trình C++ đều có hai loại thành viên: các dữ liệu thành viên và các hàm thành viên. Các hàm thành viên làm việc dựa trên hai loại dữ liệu : một loại được lấy từ bên ngoài thông qua việc gọi các thông điệp, loại kia chính là các dữ liệu bên trong thuộc về mỗi đối tượng và muốn sử dụng các dữ liệu bên trong này thông thường chúng ta cần phải thực hiện một thao tác gọi là

khởi tạo đối với chúng. Việc này có thể được thực hiện bằng cách viết một hàm public riêng biệt và sau đó người dùng có thể gọi chúng nhưng điều này sẽ phá vỡ các quy tắc về sự tách biệt giữa các lập trình viên tạo ra các lớp và những người dùng chúng hay nói một cách khác đây là một công việc quan trọng không thể giao cho các lập trình viên thuộc loại client đảm nhiệm.

C++ cung cấp một loại hàm đặc biệt cho phép chúng ta thực hiện điều này, các hàm đó được gọi là các hàm cấu tử (constructor). Nếu như lớp có một hàm cấu tử trình biên dịch sẽ tự động gọi tới nó khi một đối tượng nào đó của lớp được tạo ra, trước khi các lập trình viên client có thể sử dụng chúng. Việc gọi tới các cấu tử này không phụ thuộc vào việc sử dụng hay khai báo các đối tượng, nó được thực hiện bởi trình biên dịch vào thời điểm mà đối tượng được tạo ra.

Các hàm cấu tử này thường thực hiện các thao tác gán các giá trị khởi tạo cho các biến thành viên, mở các file input, thiết lập các kết nối tới các máy tính khác trên mạng...

Tên của các hàm cấu tử là tên của lớp, chúng buộc phải là các hàm public, vì nếu không trình biên dịch sẽ không thể gọi tới chúng. Các hàm cấu tử có thể có các tham số nếu cần thiết nhưng nó không trả về bất cứ giá trị nào và cũng không phải là hàm kiểu void.

Dựa vào các tham số đối với một cấu tử người ta chia chúng ra thành một số loại hàm cấu tử:

- **Cấu tử mặc định (default constructor)**

Cấu tử mặc định là cấu tử mà mọi tham số đều là mặc định hoặc không có tham số, cấu tử mặc định có thể được gọi mà không cần bất kỳ tham số nào.

Ví dụ:

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

```
class Point{ // Khai báo lớp Point
    int x,y;          // Properties: x and y coordinates
public:
    Point(); // Declaration of the default constructor
    bool move(int, int); // A function to move points
    void print(); // to print coordinates on the screen
};
```

```
Point::Point()
{
    cout << "Constructor is called..." << endl;
    x = 0; // Assigns zero to coordinates
    y = 0;
}
```

```
bool Point::move(int new_x, int new_y)
{
    if (new_x >=0 && new_y>=0){
        x = new_x; // assigns new value to x coordinate
        y = new_y; // assigns new value to y coordinate
        return true;
    }
```

```

    }
    return false;
}

void Point::print()
{
    cout << "X= " << x << ", Y= " << y << endl;
}

int main()
{
    Point p1,p2;           // Default construct is called 2 times
    Point *pp = new Point; // Default construct is called once
    p1.print();            // p1's coordinates to the screen
    p2.print();            // p2's coordinates to the screen
    pp->print();            // Coordinates of the object pointed by pp to the screen
    return 0;
}

```

- **Cấu tử có tham số**

Giống như các hàm thành viên, các cấu tử cũng có thể có các tham số, khi sử dụng các lớp với các cấu tử có tham số các lập trình viên cần cung cấp các tham số cần thiết.

Ví dụ:

```

class Point{           // Declaration Point Class
    int x,y;           // Properties: x and y coordinates
public:
    Point(int, int);    // Declaration of the constructor
    bool move(int, int); // A function to move points
    void print();        // to print coordinates on the screen
};

Point::Point(int x_first, int y_first)
{
    cout << "Constructor is called..." << endl;
    if ( x_first < 0 )    // If the given value is negative
        x = 0;          // Assigns zero to x
    else
        x = x_first;
    if ( y_first < 0 )    // If the given value is negative
        y = 0;          // Assigns zero to x
    else
        y = y_first;
}

```

Cấu tử Point(int, int) có nghĩa là chúng ta cần có hai tham số kiểu int khi khai báo một đối tượng của lớp Point. Ví dụ:

```

Point p1(20,100), p2(-10,45); // Constructor is called 2 times
Point *pp = new Point(10,50); // Constructor is called once

```

- **Cấu tử với các tham số có giá trị mặc định**

Giống như các hàm khác, các tham số của các cấu tử cũng có thể có các giá trị mặc định:

```
Point::Point(int x_first=0, int y_first=0)
{
    cout << "Constructor is called..." << endl;
    if ( x_first < 0 )           // If the given value is negative
        x = 0;                 // Assigns zero to x
    else
        x = x_first;
    if ( y_first < 0 )           // If the given value is negative
        y = 0;                 // Assigns zero to x
    else
        y = y_first;
}
```

Khi đó chúng ta có thể thực hiện khai báo các đối tượng của lớp Point như sau:

```
Point p1(19, 20); // x = 19, y = 20
```

```
Point p1(19); // x = 19, y = 0
```

Và hàm cấu tử trong đó tất cả các tham số đều có thể nhận các giá trị mặc định có thể được sử dụng như một cấu tử mặc định:

```
Point p3; // x = 0, y = 0
```

- **Chồng hàm cấu tử**

Một lớp có thể có nhiều cấu tử khác nhau bằng cách chồng hàm cấu tử:

```
class Point{
public:
    Point(); // Cấu tử mặc định
    Point(int, int); // Cấu tử có tham số
};
```

- **Khởi tạo mảng các đối tượng**

Khi một mảng các đối tượng được tạo ra, cấu tử mặc định của lớp sẽ được gọi đối với mỗi phần tử (là một đối tượng) của mảng. Ví dụ:

```
Point a[10]; // Cấu tử mặc định sẽ được gọi tới 10 lần
```

Cần chú ý là nếu lớp Point không có cấu tử mặc định thì khai báo như trên sẽ là sai.

Chúng ta cũng có thể gọi tới các cấu tử có tham số của lớp bằng cách sử dụng một danh sách các giá trị khởi tạo, ví dụ:

```
Point::Point(int x, int y=0);
```

```
Point a[] = {{20}, {30}, Point(20,40)}; // mảng có 3 phần tử
```

Nếu như lớp Point có thêm một cấu tử mặc định chúng ta cũng có thể khai báo như sau:

```
Point a[5] = {{20}, {30}, Point(20,40)}; // Mảng có 5 phần tử
```

- **Khởi tạo dữ liệu với các cấu tử**

Các hàm cấu tử có thể thực hiện khởi tạo các thành phần dữ liệu trong thân hàm hoặc bằng một cơ chế khác, cơ chế này đặc biệt được sử dụng khi khởi tạo các thành phần là hằng số. Ví dụ:

Chúng ta xem xét lớp sau đây:

```
class C{
    const int ci;
    int x;
    public:
        C(){
            x = 0; // đúng vì x không phải là hằng mà là biến
            ci = 0; // Sai vì ci là một hằng số
        }
};
```

Thậm chí ví dụ sau đây cũng không đúng:

```
class C{
    const int ci = 0;
    int x;
};
```

và giải pháp của chúng ta là sử dụng cơ chế khởi tạo (constructor initializer) của cấu tử:

```
class C{
    const int ci;
    int x;
    public:
        C():ci(0){
            x = 0;
        }
};
```

Cơ chế này cũng có thể được sử dụng để khởi tạo các thành phần không phải là hằng của lớp:

```
class C{
    const int ci;
    int x;
    public:
        C():ci(0),x(0){}
};
```

9. Hủy tử

Ý tưởng và khái niệm về hủy tử rất giống với cấu tử ngoại trừ việc hủy tử được tự động gọi đến khi một đối tượng không được sử dụng nữa (out of scope) (thường là các đối tượng cục bộ) hoặc một đối tượng động (sinh ra bởi việc sử dụng toán tử new) bị xóa khỏi bộ nhớ bằng toán tử delete.

Trái ngược với các hàm cấu tử các hàm hủy tử thường được gọi đến nhằm mục đích giải phóng vùng nhớ đang bị một đối tượng nào đó sử dụng, ngắt các kết nối, đóng các file hay ví dụ trong các chương trình đồ họa là xóa những gì mà đối tượng đã vẽ ra trên màn hình.

Hủy tử của một lớp cũng có tên trùng với tên lớp nhưng thêm một ký tự “~” đứng trước tên lớp. Một hàm hủy tử không có kiểu trả về (không phải là hàm kiểu void) và không nhận tham số, điều này có nghĩa là mỗi lớp chỉ có một hủy tử khác với cấu tử.

Ví dụ:

```
class String{
    int size;
```

```

    char *contents;
public:
    String(const char *); // Constructor
    void print();        // A member function
    ~String();           // Destructor
};

```

Thực tế thư viện chuẩn của C ++ có xây dựng một lớp string . Các lập trình viên không cần xây dựng lớp string riêng cho mình . Chúng ta xây dựng lớp String trong ví dụ trên chỉ để minh họa cho khái niệm về hàm hủy tử .

```

String::String(const char *in_data)
{
    cout<< "Constructor has been invoked" << endl;
    size = strlen(in_data);
    contents = new char[size +1]; // +1 for null character
    strcpy(contents, in_data); // input_data is copied to the contents
}

```

```

void String::print()
{
    cout<< contents << " " << size << endl;
}

```

```

// Destructor
// Memory pointed by contents is given back to the heap
String::~~String()
{
    cout << "Destructor has been invoked" << endl;
    delete[] contents;
}

```

```

//----- Main Function -----
int main()
{
    cout << "----- Start of Blok 1 -----" << endl;
    String string1("string 1");
    String string2("string 2");
    {
        cout << "----- Start of Blok 2 -----" << endl;
        string1.print();
        string2.print();
        String string3("string 3");
        cout << "----- End of Blok 2 -----" << endl;
    }
    cout << "----- End of Blok 1 -----" << endl;
    return 0;
}

```

Ví dụ:

```

// Listing 11.13
// Linked list simple implementation
#include <iostream.h>
// object to add to list
class CAT
{
    public:
        CAT() { itsAge = 1;}
        CAT(int age):itsAge(age){}
        ~CAT(){};
        int GetAge() const { return itsAge; }
    private:
        int itsAge;
};
// manages list, orders by cat's age!
class Node
{
    public:
        Node (CAT*);
        ~Node();
        void SetNext(Node * node) { itsNext = node; }
        Node * GetNext() const { return itsNext; }
        CAT * GetCat() const { return itsCat; }
        void Insert(Node *);
        void Display();
    private:
        CAT *itsCat;
        Node * itsNext;
};

Node::Node(CAT* pCat):itsCat(pCat),itsNext(0){}
Node::~~Node()
{
    cout << "Deleting node...\n";
    delete itsCat;
    itsCat = 0;
    delete itsNext;
    itsNext = 0;
}
// *****
// Insert
// Orders cats based on their ages
// Algorithm: If you are last in line, add the cat
// Otherwise, if the new cat is older than you
// and also younger than next in line, insert it after
// this one. Otherwise call insert on the next in line
// *****
void Node::Insert(Node* newNode)

```

```

{
    if (!itsNext)
        itsNext = newNode;
    else
    {
        int NextCatsAge = itsNext->GetCat()->GetAge();
        int NewAge = newNode->GetCat()->GetAge();
        int ThisNodeAge = itsCat->GetAge();
        if ( NewAge >= ThisNodeAge && NewAge < NextCatsAge )
        {
            newNode->SetNext(itsNext);
            itsNext = newNode;
        }
        else
            itsNext->Insert(newNode);
    }
}

void Node::Display()
{
    if (itsCat->GetAge() > 0)
    {
        cout << "My cat is ";
        cout << itsCat->GetAge() << " years old\n";
    }
    if (itsNext)
        itsNext->Display();
}

int main()
{
    Node *pNode = 0;
    CAT * pCat = new CAT(0);
    int age;
    Node *pHead = new Node(pCat);
    while (1)
    {
        cout << "New Cat's age? (0 to quit): ";
        cin >> age;
        if (!age)
            break;
        pCat = new CAT(age);
        pNode = new Node(pCat);
        pHead->Insert(pNode);
    }
    pHead->Display();
    delete pHead;
    cout << "Exiting...\n\n";
}

```

```

    return 0;
}

```

10. Cấu tử copy

Cấu tử copy là một cấu tử đặc biệt và nó được dùng để copy nội dung của một đối tượng sang một đối tượng mới trong quá trình xây dựng đối tượng mới đó.

Tham số input của nó là một tham chiếu tới các đối tượng cùng kiểu. Nó nhận tham số như là một tham chiếu tới đối tượng sẽ được copy sang đối tượng mới.

Cấu tử copy thường được tự động sinh ra bởi trình biên dịch nếu như tác giả không định nghĩa cho tác phẩm tương ứng.

Nếu như trình biên dịch sinh ra nó, cấu tử copy sẽ làm việc theo kiểu học máy, nó sẽ copy từng byte từng byte một. Đối với các lớp đơn giản không có biến con trỏ thì điều này là đủ, nhưng nếu có một con trỏ là thành viên của lớp thì việc copy theo kiểu học máy này (byte by byte) sẽ làm cho cả hai đối tượng (mới và cũ) cùng trỏ vào một địa chỉ. Do đó khi chúng ta thay đổi đối tượng mới sẽ làm ảnh hưởng tới đối tượng cũ và ngược lại. Đây không phải là điều chúng ta muốn, điều chúng ta muốn là thay vì copy địa chỉ con trỏ cấu tử copy sẽ copy nội dung mà biến con trỏ trỏ tới và để đạt được điều đó chúng ta buộc phải tự xây dựng cấu tử copy đối với các lớp có các thành viên là biến con trỏ.

Ví dụ:

```

#include <iostream.h>
#include <string.h>

```

```

class String{
    int size;
    char *contents;
public:
    String(const char *);           // Constructor
    String(const String &);        // Copy Constructor
    void print();                  // Prints the string on the screen
    ~String();                     // Destructor
};

```

```

// Constructor
// copies the input character array to the contents of the string
String::String(const char *in_data)
{
    cout<< "Constructor has been invoked" << endl;
    size = strlen(in_data);
    contents = new char[size + 1]; // +1 for null character
    strcpy(contents, in_data);     // input_data is copied to the contents
}

```

```

// Copy Constructor
String::String(const String &object_in)
{
    cout<< "Copy Constructor has been invoked" << endl;
    size = object_in.size;
}

```

```

        contents = new char[size + 1];    // +1 for null character
        strcpy(contents, object_in.contents);
    }

void String::print()
{
    cout<< contents << " " << size << endl;
}

// Destructor
// Memory pointed by contents is given back to the heap
String::~String()
{
    cout << "Destructor has been invoked" << endl;
    delete[] contents;
}

//----- Main Function -----
int main()
{

    String my_string("string 1");
    my_string.print();
    String other = my_string;           // Copy constructor is invoked
    String more(my_string);             // Copy constructor is invoked
    other.print();
    more.print();
    return 0;
}

```

11. Đối tượng hằng và các hàm thành viên hằng

Chúng ta có thể sử dụng từ khóa const để chỉ ra rằng một đối tượng là không thể thay đổi (not modifiable) hay là đối tượng hằng. Tất cả các cố gắng nhằm thay đổi nội dung của các đối tượng hằng đều gây ra các lỗi. Ví dụ:

```
const ComplexT cz(0,1);
```

C++ hoàn toàn ngăn cấm việc gọi tới các hàm thành viên của các đối tượng hằng trừ khi các hàm thành viên đó là các hàm hằng, có nghĩa là các hàm không thay đổi các thành phần bên trong của đối tượng.

Để khai báo một hàm như vậy chúng ta cũng sử dụng từ khóa const:

Ví dụ:

```

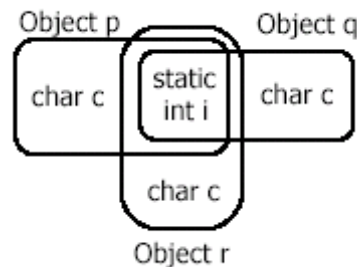
class Point{                                // Declaration Point Class
    int x,y;                                // Properties: x and y coordinates
public:
    Point(int, int);                        // Declaration of the constructor
    bool move(int, int);                    // A function to move points
    void print() const;                     // constant function: prints coordinates on the screen
};

```

Khi đó chúng ta có thể khai báo các đối tượng hằng của lớp Point và gọi tới hàm print của nó.

12. Các thành viên tĩnh của lớp

Thông thường mỗi đối tượng của một lớp đều có một bản copy riêng tất cả các thành viên dữ liệu của lớp. Trong các trường hợp cụ thể đôi khi chúng ta muốn là tất cả các đối tượng của lớp sẽ chia sẻ cùng một thành viên dữ liệu nào đó. Và đó chính là lý do tồn tại của các thành viên dữ liệu tĩnh hay còn gọi là biến của lớp.



```
class A{
    char c;
    static int i;
};
int main(){
    ...
    A p, q, r;
    ...
}
```

Các thành viên tĩnh tồn tại ngay cả khi không có đối tượng nào của lớp được tạo ra trong chương trình. Chúng cũng có thể là các thành phần public hoặc private.

Để truy cập vào các thành phần dữ liệu tĩnh **public** của một lớp khi không có đối tượng nào của lớp tồn tại chúng ta sẽ sử dụng tên lớp và toán tử "::", ví dụ: A::i = 5;

Để truy cập vào các thành phần dữ liệu tĩnh **private** của một lớp khi không có đối tượng nào của lớp tồn tại, chúng ta cần có một hàm thành viên tĩnh **public**.

Các biến tĩnh bắt buộc phải được khởi tạo một lần (và chỉ một lần) trước khi chúng được sử dụng.

```
#include <iostream.h>
class A{
    char c;
    static int number;    // Number of created objects (static data)
public:
    static void setNum(){number=0;}    // Static function to initialize number
    A(){number++; cout<< "\n"<< "Constructor "<< number;} //Constructor
    ~A(){number--; cout<< "\n"<< "Destructor "<< number;} //Destructor
};
```

```
int A::number;    // Allocating memory for number
```

// Chú ý nếu không có đoạn này chương trình sẽ báo lỗi

// ----- Main function -----


```

int main()
{
    cout<<"\n Entering 1. BLOCK.....";
    A::setNum();
    // The static function is called
    A a,b,c;
    {
        cout<<"\n Entering 2. BLOCK.....";
        A d,e;
        cout<<"\n Exiting 2. BLOCK.....";
    }
    cout<<"\n Exiting 1. BLOCK.....";
    return 0;
}

```

13. Sử dụng các đối tượng trong vai trò là tham số của hàm

Các đối tượng nên được truyền và trả về theo kiểu tham chiếu trừ khi có các lý do đặc biệt đòi hỏi chúng ta phải truyền hoặc trả về chúng theo kiểu truyền biến và trả về theo kiểu giá trị. Việc truyền tham biến và giá trị của hàm đặc biệt không hiệu quả khi làm việc với các đối tượng. Chúng ta hãy nhớ lại đối tượng được truy cập hoặc trả lại theo giá trị phải được copy vào stack và dữ liệu có thể rất lớn, và do đó có thể làm lãng phí bộ nhớ. Bản thân việc copy này cũng tốn thời gian. Nếu như lớp chứa một cấu trúc copy thì trình biên dịch sẽ sử dụng hàm này để copy đối tượng vào stack.

Chúng ta nên truyền tham số theo kiểu tham chiếu vì chúng ta không muốn có các bản copy được tạo ra. Và để ngăn chặn việc hàm thành viên có thể vô tình làm thay đổi đối tượng ban đầu chúng ta sẽ khai báo tham số hình thức có kiểu là hằng tham chiếu (**const reference**).

Ví dụ:

```

ComplexT & ComplexT::add(constComplexT &z){
    ComplexT result;
    result.re = re + z.re;
    result.im = im + z.im;
    return result; // Sai các biến cục bộ không thể trả về qua tham chiếu.
}

```

Ví dụ trên có thể sửa lại cho đúng như sau:

```

ComplexT ComplexT::add(constComplexT &z){
    ComplexT result;
    result.re = re + z.re;
    result.im = im + z.im;
    return result; // Sai các biến cục bộ không thể trả về qua tham chiếu.
}

```

Tuy nhiên do có một đối tượng tạm thời được tạo ra như vậy các hàm hủy tử và cấu trúc sẽ được gọi đến. Để tránh việc tạo ra một đối tượng tạm thời (để tiết kiệm thời gian và bộ nhớ) người ta có thể làm như sau:

```

ComplexT ComplexT::add(constComplexT &z){
    double re_new, im_new;
    re_new = re + z.re;
    im_new = im + z.im;
}

```

```

    return ComplexT(re_new, im_new);
}

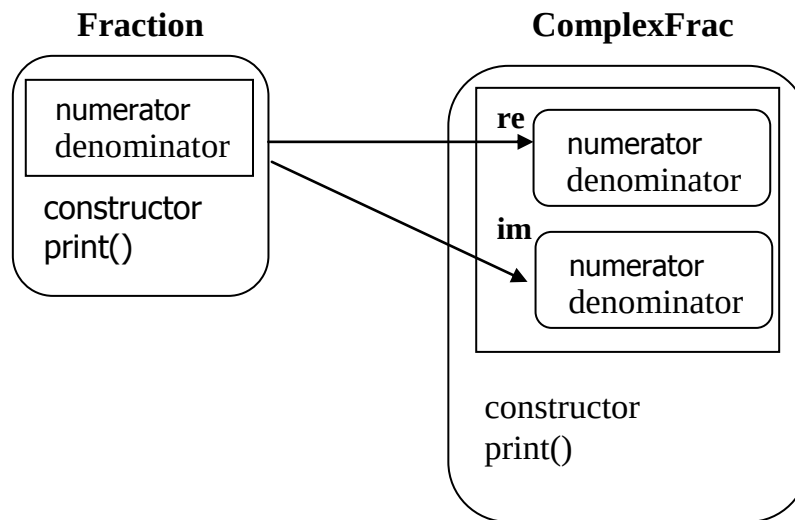
```

Chỉ có đối tượng trả về trên stack là được tạo ra (là thứ luôn cần thiết khi trả về theo giá trị).

Đây có thể là một cách tiếp cận tốt hơn : chúng ta sẽ tạo ra và hủy bỏ các phần tử dữ liệu riêng biệt , cách này sẽ nhanh hơn là tạo ra và hủy bỏ cả một đối tượng hoàn chỉnh.

14. Các đối tượng chồng nhau: Các lớp là thành viên của các lớp khác.

Một lớp có thể sử dụng các đối tượng của lớp khác như là các thành viên dữ liệu của nó. Trong ví dụ dưới đây chúng ta sẽ thiết kế một lớp (ComplexFrac) để biểu diễn các số phức. Các thành viên dữ liệu của lớp này là các phân số và là các đối tượng của lớp Fraction.



Mối quan hệ giữa hai lớp Fraction và ComplexFrac được gọi là “**has a relation**”. Ở đây lớp ComplexFrac có một (thực ra là hai đối tượng của lớp Fraction).

Trong lớp ComplexFrac tác giả của lớp phải cung cấp các tham biến cần thiết cho các cấu tử của các đối tượng của lớp Fraction (đối tượng mà nó có).

Việc xây dựng các đối tượng g trong một lớp sẽ thực hiện theo thứ tự xuất hiện của các đối tượng và trước dấu “{” bắt đầu của hàm cấu tử của lớp chứa.

Ví dụ:

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

```

class Fraction{    // A class to define fractions
    int numerator,denominator;
public:
    Fraction(int, int);
    void print() const;
};

```

```

Fraction::Fraction(int num, int denom)    // CONSTRUCTOR
{
    numerator=num;
    if (denom==0) denominator=1;
}

```

```

        else denominator=denom;
        cout << "Constructor of Fraction" << endl;
    }

void Fraction ::print() const
{
    cout << numerator << "/" << denominator << endl;
}

class ComplexFrac{                // Complex numbers: has two fractions
    Fraction re,im;                // member objects
public:
    ComplexFrac(int,int); // Constructor
    void print() const;
};

// Constructor
// first, constructors of the member objects must be called
ComplexFrac::ComplexFrac(int re_in,int im_in):re(re_in,1),im(im_in,1)
{
    cout << "Constructor of ComplexFrac" << endl;
}

// Prints complex numbers on the screen
// print function of the member objects are called
void ComplexFrac::print() const
{
    re.print();
    im.print();
}

//----- Main Function -----
int main()
{
    ComplexFrac cf(2,5);           // A complex number is created
    cf.print();                    // Complex number is printed on the screen
    return 0;
}

```

Khi đối tượng không còn cần nữa (go out of scope) thì các hủy tử của chúng sẽ được gọi theo thứ tự ngược lại với các hủy tử: hủy tử của các đối tượng thuộc lớp chứa sẽ được gọi tới trước sau đó mới là hủy tử của các đối tượng bên trong lớp đó.

15. Chồng toán tử

Chúng ta đã biết rằng có thể thực hiện chồng hàm , bản thân các toán tử cũng là các hàm và vì thế nên hoàn toàn có thể thực hiện chồng các toán tử (hay các hàm toán tử) chẳng hạn như các toán tử +, -, *, >= hay ==, khi đó chúng sẽ gọi tới các hàm khác nhau tùy thuộc vào các toán hạng của chúng . Ví dụ với toán tử +, biểu thức a + b sẽ

gọi tới một hàm cộng hai số nguyên nếu a và b thuộc kiểu `int` nhưng sẽ gọi tới một hàm khác nếu chúng là các đối tượng của một lớp nào đó mà chúng ta mới tạo ra.

Chồng toán tử (operator overloading) là một đặc điểm thuận tiện khác của C++ làm cho các chương trình dễ viết hơn và cũng dễ hiểu hơn.

Thực chất chồng toán tử không thêm bất cứ một khả năng mới nào vào C++. Tất cả những gì chúng ta có thể thực hiện với một toán tử chồng đều có thể thực hiện được với một hàm nào đó. Tuy nhiên việc chồng các toán tử làm cho chương trình dễ viết, dễ đọc và dễ bảo trì hơn.

Chồng toán tử là cách duy nhất để gọi một nào đó hàm theo cách khác với cách thông thường. Xem xét theo khía cạnh này chúng ta không có bất cứ lý do nào để thực hiện chồng một toán tử nào đó trừ khi nó làm cho việc cài đặt các lớp trong chương trình dễ dàng hơn và đặc biệt là dễ đọc hơn (lý do này quan trọng hơn cả).

Các hạn chế của chồng toán tử

Hạn chế thứ nhất là chúng ta không thể thực hiện cài đặt các toán tử không có trong C++. Chẳng hạn không thể cài hàm toán tử `**` để thực hiện lấy lũy thừa. **Chúng ta chỉ có thể thực hiện chồng các toán tử thuộc loại built-in của C++.**

Thậm chí một số các toán tử sau đây: toán tử dấu chấm (`.`), toán tử phân giải tầm hoạt động (`::`), toán tử điều kiện (`?:`), toán tử `sizeof` cũng không thể overload.

Các toán tử của C++ có thể chia thành hai loại là toán tử một ngôi và toán tử hai ngôi. Và nếu một toán tử thuộc kiểu binary thì toán tử được chồng của nó cũng là toán tử hai ngôi và tương tự đối với toán tử một ngôi. Độ ưu tiên của toán tử cũng như số lượng hay cú pháp của các toán hạng là không đổi đối với hàm chồng toán tử. Ví dụ như toán tử `*` bao giờ cũng có độ ưu tiên cao hơn toán tử `+`. Tất cả các toán tử được sử dụng trong biểu thức chỉ nhận các kiểu dữ liệu built-in không thể thay đổi. Chẳng hạn chúng ta không bao giờ chồng toán tử `+` để biểu thức `a = 3 + 5` hay `1 << 4` có ý nghĩa khác đi.

Ít nhất thì một toán hạng phải thuộc kiểu dữ liệu người dùng định nghĩa (lớp).

Ví dụ:

```
class ComplexT{
    double re,im;
public:
    ComplexT(double re_in=0,double im_in=1);    // Constructor
    ComplexT operator+(const ComplexT & ) const; // Function of operator +
    void print() const;
};
ComplexT ComplexT::operator+(const ComplexT &c) const
{
    double re_new, im_new;
    re_new=re+c.re;
    im_new=im+c.im;
    return ComplexT(re_new,im_new);
}
int main()
{
    ComplexT z1(1,1),z2(2,2),z3;
    z3=z1+z2;    // like z3 = z1.operator+(z2);
    z3.print();
}
```

```

    return 0;
}

```

Chồng toán tử gán (=)

Việc gán một đối tượng này cho một đối tượng khác cùng kiểu (cùng thuộc một lớp) là một công việc mà hầu hết mọi người (các lập trình viên) đều mong muốn là có thể thực hiện một cách dễ dàng nên trình biên dịch sẽ tự động sinh ra một hàm để thực hiện điều này đối với mỗi lớp được người dùng tạo ra nếu họ không có ý định cài đặt hàm đó:

```

type::operator(const type &);

```

Hàm này thực hiện theo cơ chế gán thành phần, có nghĩa là nó sẽ thực hiện gán từng biến thành viên của đối tượng này cho một đối tượng khác có cùng kiểu (cùng lớp). Nếu như đối với các lớp không có gì đặc biệt, thao tác này là đủ thì chúng ta cũng không cần thiết phải thực hiện cài đặt hàm toán tử này, chẳng hạn ví dụ cài đặt hàm toán tử gán đối với lớp ComplexT là không cần thiết:

```

void ComplexT::operator=(const ComplexT & z){
    re = z.re;
    im = z.im;
}

```

Nói chung thì chúng ta thường có xu hướng tự cài đặt lấy hàm toán tử gán đối với các lớp được sử dụng trong chương trình và đặc biệt là với các lớp tinh vi hơn chẳng hạn:

```

class String{
    int size;
    char *contents;
public:
    String(); //default constructor
    String(const char *); // constructor
    String(const String &); // copy constructor
    const String& operator=(const String &); // assignment operator
    void print() const ;
    ~String(); // Destructor
}

```

Chú ý là trong trường hợp trên hàm toán tử = có kiểu là void do đó chúng ta không thể thực hiện các phép gán nối tiếp nhau kiểu như (a = b = c;).

```

const String& String::operator=(const String &in_object)
{
    cout<< "Assignment operator has been invoked" << endl;
    size = in_object.size;
    delete[] contents; // delete old contents
    contents = new char[size+1];
    strcpy(contents, in_object.contents);
    return *this; // returns a reference to the object
}

```

Sự khác biệt giữa hàm toán tử gán và cấu tử copy là ở chỗ cấu tử copy sẽ thực sự tạo ra một đối tượng mới trước khi copy dữ liệu sang cho nó còn hàm toán tử gán thì chỉ thực hiện việc copy dữ liệu sang cho một đối tượng có sẵn.

Chồng toán tử chỉ số []

Các qui luật chung chúng ta đã trình bày được áp dụng đối với mọi toán tử . Vì thế chúng ta không cần thiết phải bàn luận về từng loại toán tử . Tuy nhiên chúng ta sẽ khảo sát một vài toán tử được người ta cho là thú vị. Và một trong các toán tử đó chính là toán tử chỉ số.

Toán tử này có thể được khai báo theo hai cách như sau:

```
class C{
    returntype & operator [](paramtype);
    hoặc:
    const returntype & operator [](paramtype) const;
};
```

Cách khai báo thứ nhất được sử dụng khi việc chồng toán tử chỉ số làm thay đổi thuộc tính của đối tượng. Cách khai báo thứ hai được sử dụng đối với một đối tượng hằng; trong trường hợp này , toán tử chỉ số được chồng có thể truy cập nhưng không thể làm thay đổi các thuộc tính của đối tượng.

Nếu c là một đối tượng của lớp C, biểu thức

c[i]
sẽ được dịch thành
c.operator[](i)

Ví dụ: chúng ta sẽ cài đặt hàm chồng toán tử chỉ số cho lớp String . Toán tử sẽ được sử dụng để truy cập vào ký tự thứ i của chuỗi . Nếu i nhỏ hơn 0 và lớn hơn độ dài của chuỗi thì ký tự đầu tiên và cuối cùng sẽ được truy cập.

```
char & String::operator[](int i)
{
    if(i < 0)
        return contents[0];           // return first character
    if(i >= size)
        return contents[size-1];       // return last character
    return contents[i];                 // return i th character
}
```

Chồng toán tử gọi hàm ()

Toán tử gọi hàm là duy nhất , nó duy nhất ở chỗ cho phép có bất kỳ một số lượng tham số nào.

```
class C{
    returntype operator()(paramtypes);
};
```

Nếu c là một đối tượng của lớp C, biểu thức

c(i,j,k)
sẽ được thông dịch thành:
c.operator()(i,j,k);

Ví dụ toán tử gọi hàm được chồng để in ra các số phức ra màn hình . Trong ví dụ này toán tử gọi hàm không nhận bất cứ một tham số nào.

```
void ComplexT::operator()() const
{
    cout << re << " , " << im << endl;
}
```

Ví dụ: toán tử gọi hàm được dùng để copy một phần nội dung của một chuỗi tới một vị trí bộ nhớ xác định.

Trong ví dụ này toán tử gọi hàm nhận hai tham số : địa chỉ của bộ nhớ đích và số lượng ký tự cần sao chép.

```
void String::operator()(char * dest, int num) const
{
    // numbers of characters to be copied may not exceed the size
    if (num>size) num=size;
    for (int k=0; k< num; k++)
        dest[k]=contents[k];
}
int main()
{
    String s1("Example Program");
    char *c=new char[8];      // Destination memory
    s1(c,7);                  // Function call operator is invoked
    c[7]='\0';                // End of String (null)
    cout << c << endl;
    delete[] c;
    return 0;
}
```

Chồng các toán tử một ngôi

Các toán tử một ngôi chỉ nhận một toán hạng để làm việc , một vài ví dụ điển hình về chúng chẳng hạn như: ++, --, - và !.

Các toán tử một ngôi không nhận tham số , chúng thao tác trên chính đối tượng gọi tới chúng. Thông thường toán tử này xuất hiện bên trái của đối tượng chẳng hạn như --obj, ++obj...

Ví dụ: Chúng ta định nghĩa toán tử ++ cho lớp ComplexT để tăng phần thực của số phức lên 1 đơn vị 0,1.

```
void ComplexT::operator++()
{
    re = re + 0.1;
}
int main(){
    ComplexT z(0.2, 1);
    ++z;
}
```

Để có thể thực hiện gán giá trị được tăng lên cho một đối tượng mới , hàm toán tử cần trả về một tham chiếu tới một đối tượng nào đó:

```
const ComplexT & ComplexT::operator++()
{
    re = re + 0.1;
    return this;
}
```

```
int main(){
    ComplexT z(0.2, 1), z1;
```

```

    z1 = ++z;
}

```

Chúng ta nhớ lại rằng các toán tử ++ và - - có hai dạng sử dụng theo kiểu đứng trước và đứng sau toán hạng và chúng có các ý nghĩa khác nhau . Việc khai báo nh ư trong hai ví dụ trên sẽ chồng toán tử ở dạng đứng trước toán hạng . Các khai báo có dạng operator(int) sẽ chồng dạng đứng sau của toán tử.

```

ComplexT ComplexT::operator++(int)
{
    ComplexT temp;
    temp=*this;    // saves old value
    re=re+0.1;
    return temp;    // return old value
}

```

Lớp String:

```

enum bool{true = 1, false = 0};
class String{
    int size;
    char *contents;
public:
    String();                                //default constructor
    String(const char *);                    // constructor
    String(const String &); // copy constructor
    const String& operator=(const String &); // assignment operator
    bool operator==(const String &); // assignment operator
    bool operator!=(const String &rhs){return !(*this==rhs);}; // assignment
operator
    void print() const ;
    ~String(); // Destructor
    friend ostream & operator <<(ostream &, const String &);
};
// Creates an empty string (only NULL character)
String::String(){
    size = 0;
    contents = new char[1];
    strcpy(contents, "");
}
String::String(const char *in_data){
    size = strlen(in_data);                // Size of input data
    contents = new char[size + 1]; // allocate mem. for the string, +1 is for
NULL
    strcpy(contents, in_data);
}

String::String(const String &in_object){
    size = in_object.size;
    contents = new char[size+1];
    strcpy(contents, in_object.contents);
}

```



```

// Assignment operator
const String& String::operator=(const String &in_object){
    size = in_object.size;
    delete[] contents;
    // delete old contents
    contents = new char[size+1];
    strcpy(contents, in_object.contents);
    return *this;          // returns a reference to the object
}

bool String::operator==(const String &rhs){
    if(size == rhs.size){
        for(int i=0;i<=size&&(contents[i]==rhs.contents[i]);i++);
        if(i>size)
            return true;
    }
    return false;
}

// This method prints strings on the screen
void String::print() const{
    cout<< contents << " " << size << endl;
}

//Destructor
String::~String(){
    delete[] contents;
}

ostream & operator <<(ostream & out, const String & rhs){
    out << rhs.contents;
    return out;
}

```

Chương 6: Kế thừa. (6 tiết)

Kế thừa là một cách trong lập trình hướng đối tượng để có thể thực hiện được khả năng “sử dụng lại mã chương trình” . Sử dụng lại mã chương trình có nghĩa là dùng một lớp sẵn có trong một khung cảnh chương trình khác . Bằng cách sử dụng lại các lớp chúng ta có thể làm giảm thời gian và công sức cần thiết để phát triển một chương trình đồng thời làm cho chương trình phần mềm có khả năng và qui mô lớn hơn cũng như tính tin cậy cao hơn.

1. Sử dụng lại mã chương trình

Cách tiếp cận đầu tiên nhằm mục đích sử dụng lại mã chương trình đơn giản là viết lại các đoạn mã đã có . Chúng ta có một đoạn mã chương trình nào đó đã sử dụng tốt trong một chương trình cũ nào đó , nhưng không thực sự đáp ứng được yêu cầu của chúng ta trong một dự án mới.

Chúng ta sẽ paste đoạn mã cũ đó vào một file mã nguồn mới , thực hiện một vài sửa đổi để nó phù hợp với môi trường mới . Tất nhiên chúng ta lại phải thực hiện gỡ lỗi đoạn mã đó từ đầu và thường thì chúng ta lại thấy tiếc là tại sao không viết hẳn một đoạn mã chương trình mới.

Để làm giảm các lỗi có thể có khi thay sửa đổi mã chương trình, các lập trình viên cố gắng tạo ra các đoạn mã có thể được sử dụng lại mà không cần bận khoăn về khả năng gây lỗi của chúng và đó được gọi là các hàm.

Các hàm thư viện là một bước tiến nữa nhằm sử dụng lại mã chương trình tuy nhiên các thư viện có nhược điểm là chúng mô hình hóa thế giới thực không được tốt lắm vì chúng không bao gồm các dữ liệu quan trọng. Và thường xuyên chúng ta cần thay đổi chúng để có thể phù hợp với môi trường mới và tất nhiên sự thay đổi này lại dẫn đến các lỗi có thể phát sinh.

2. Sử dụng lại mã chương trình trong OOP

Một cách tiếp cận đầy sức mạnh để sử dụng lại mã chương trình trong lập trình hướng đối tượng là thư viện lớp. Vì các lớp mô hình hóa các thực thể của thế giới thực khá sát nên chúng cần ít các thay đổi hơn các hàm để có thể phù hợp với hoàn cảnh mới. Khi một lớp đã được tạo ra và kiểm thử cẩn thận, nó sẽ là một đơn vị mã nguồn có ích. Và nó có thể được sử dụng theo nhiều cách khác nhau:

Cách đơn giản nhất để sử dụng lại một lớp là sử dụng một đối tượng của lớp đó một cách trực tiếp. Thư viện chuẩn của C++ có rất nhiều đối tượng và lớp có ích chẳng hạn `cin` và `cout` là hai đối tượng kiểu đó.

Cách thứ hai là đặt một đối tượng của lớp đó vào trong một lớp khác. Điều này được gọi là “tạo ra một đối tượng thành viên”.

Lớp mới có thể được xây dựng bằng cách sử dụng số lượng bất kỳ các đối tượng thuộc các lớp khác theo bất kỳ cách thức kết hợp nào để đạt được các chức năng mà chúng ta mong muốn trong lớp mới. Vì chúng ta xây dựng lên lớp mới (composing) từ các lớp cũ nên ý tưởng này được gọi là composition và nó cũng thường được đề cập tới như là một quan hệ “has a”.

Cách thứ ba để sử dụng lại một lớp là kế thừa. Kế thừa là một quan hệ kiểu “is a” hoặc “a kind of”.

3. Cú pháp kế thừa

OOP cung cấp một cơ chế để thay đổi một lớp mà không làm thay đổi mã nguồn của nó. Điều này đạt được bằng cách dùng kế thừa để sinh ra một lớp mới từ một lớp cũ. Lớp cũ được gọi là lớp cơ sở sẽ không bị sửa đổi, nhưng lớp mới (được gọi là lớp dẫn xuất) có thể sử dụng tất cả các đặc điểm của lớp cũ và các đặc điểm thêm khác của riêng nó. Nếu có một quan hệ cùng loài (kind of) giữa hai đối tượng thì chúng ta có thể sinh một đối tượng này từ đối tượng kia bằng cách sử dụng kế thừa.

Ví dụ chúng ta đều biết Lớp Animal bao gồm tất cả các loài động vật, lớp Fish là một loài động vật nên các đối tượng của lớp Fish có thể được sinh ra từ một đối tượng của lớp Animal.

Ví dụ kế thừa đơn giản nhất đòi hỏi phải có 2 lớp: một lớp cơ sở và một lớp dẫn xuất. Lớp cơ sở không có yêu cầu gì đặc biệt, lớp dẫn xuất ngược lại cần chỉ rõ nó được sinh ra từ lớp cơ sở và điều này được thực hiện bằng cách sử dụng một dấu: sau tên lớp dẫn xuất sau đó tới một từ khóa chẳng hạn `public` và tên lớp cơ sở:

Ví dụ: chúng ta cần mô hình hóa các giáo viên và hiệu trưởng trong trường học. Trước hết giả sử rằng chúng ta có một lớp định nghĩa các giáo viên, sau đó chúng ta có thể sử dụng lớp này để mô hình hóa hiệu trưởng vì hiệu trưởng cũng là một giáo viên:

```
class Teacher{
protected:
    String name;
    int age, numOfStudents;
```

```

    public:
        void setName(const String & new_name){name = new_name;}
};
class Principal: public Teacher{
    String school_name;
    int numOfTeachers;
    public:
        void setSchool(const & String s_name){school_name = s_name;}
};
int main(){
    Teacher t1;
    Principal p1;
    p1.setName("Principal 1");
    t1.setName("Teacher 1");
    p1.setSchool("Elementary School");
    return 0;
}

```

Một đối tượng dẫn xuất kế thừa tất cả các thành phần dữ liệu và các hàm thành viên của lớp cơ sở. Vì thế đối tượng con (dẫn xuất) p1 không chỉ chứa các phần tử dữ liệu school_name và numOfTeachers mà còn chứa cả các thành phần dữ liệu name, age và numOfStudents.

Đối tượng p1 không những có thể truy cập vào hàm thành viên riêng của nó là setSchool() mà còn có thể sử dụng hàm thành viên của lớp cơ sở mà nó kế thừa là hàm setName().

Các thành viên thuộc kiểu private của lớp cơ sở cũng được kế thừa bởi lớp dẫn xuất nhưng chúng không nhìn thấy ở lớp kế thừa. Lớp kế thừa chỉ có thể truy cập vào các thành phần này qua các hàm public giao diện của lớp cơ sở.

4. Định nghĩa lại các thành viên

Một vài thành viên (hàm hoặc dữ liệu) của lớp cơ sở có thể không phù hợp với lớp dẫn xuất. Các thành viên này nên được định nghĩa lại trong lớp dẫn xuất. Chẳng hạn lớp Teacher có một hàm thành viên in ra các thuộc tính của các giáo viên lên màn hình. Nhưng hàm này là không đủ đối với lớp Principal vì các hiệu trưởng có nhiều thuộc tính hơn các giáo viên bình thường. Vì thế hàm này sẽ được định nghĩa lại:

```

class Teacher{
    protected:
        String name;
        int age, numOfStudents;
    public:
        void setName(const String & new_name){name = new_name;}
        void print() const;
};
void Teacher::print() const{
    cout << "Name: " << name << " Age: " << age << endl;
    cout << "Number of Students: " << numOfStudents << endl;
};

class Principal: public Teacher{

```

```

String school_name;
int numOfTeachers;
public:
    void setSchool(const & String s_name){school_name = s_name;}
    void print() const;
};

```

```

void Principal::print() const{
    cout << "Name: " << name << " Age: " << age << endl;
    cout << "Number of Students: " << numOfStudents << endl;
    cout << "Name of the school: " << school_name << endl;
};

```

Hàm print() của lớp Principal override (hoặc hide) hàm print() của lớp Teacher. Lớp Principal giờ đây có hai hàm print(). Hàm print() của lớp cơ sở có thể được truy cập bằng cách sử dụng toán tử "::".

```

void Principal::print() const{
    Teacher::print();
    cout << "Name of the school: " << school_name << endl;
};

```

Chú ý: overloading khác với overriding. Nếu chúng ta thay đổi signature hoặc kiểu trả về của một hàm thành viên thuộc lớp cơ sở thì lớp dẫn xuất sẽ có hai hàm thành viên có tên giống nhau nhưng đó không phải là overloading mà là overriding.

Và nếu như tác giả của lớp dẫn xuất định nghĩa lại một hàm thành viên, thì điều đó có nghĩa là họ muốn thay đổi giao diện của lớp cơ sở. Trong trường hợp này hàm thành viên của lớp cơ sở sẽ bị che đi.

Ví dụ:

```

class A{                                // Base class
public:
    int ia1,ia2;
    void fa1();
    int fa2(int);
};
class B: public A{                      // Derived class
public:
    float ia1;                          // overrides ia1
    float fa1(float);                  // overloads fa1
};
void A::fa1(){
    cout << "fa1 of A has been called" << endl;
}

int A::fa2(int i){
    cout << "fa2 of A has been called" << endl;
    return i;
}

```

```

float B::fa1(float f){
    cout << "fa1 of A has been called" << endl;
}

```

```

    return f;
}

int main(){
    B b;
    int j=b.fa2(1);           // A::fa2
    b.ia1=4;                  // float fa1 of B
    b.ia2=3;                  // ia2 of A. If it is public
    float y=b.fa1(3.14);      // OK, fa1 of B is called
    //b.fa1();                // ERROR! fa1 of B needs a floar argument
    b.A::fa1();
    b.A::fa1();
    b.A::ia1=1;
    return 0;
}

```

5. Kiểm soát truy cập

Hãy nhớ rằng khi chưa sử dụng kế thừa, các hàm thành viên của lớp có thể truy cập vào bất cứ thành viên nào của lớp cho dù đó là public, private nhưng các đối tượng của lớp đó chỉ có thể truy cập vào các thành phần public.

Khi kế thừa được đưa ra các khả năng truy cập tới các thành viên khác đã ra đời. Các hàm thành viên của lớp dẫn xuất có thể truy cập vào các thành viên public và protected của lớp cơ sở, trừ các thành viên private. Các đối tượng của lớp dẫn xuất chỉ có thể truy cập vào các thành viên public của lớp cơ sở.

Chúng ta có thể xem rõ hơn trong bảng sau đây:

	Truy cập từ chính lớp đó	Truy cập từ lớp dẫn xuất	Truy cập từ đối tượng của lớp
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	Yes	No	No

Chúng ta có thể định nghĩa lại hai lớp Teacher và Principal như sau:

```

class Teacher{
    private:
        String name;
    protected:
        int age, numOfStudents;
    public:
        void setName(const String & new_name){name = new_name;}
        void print() const;
};

void Teacher::print() const{
    cout << "Name: " << name << " Age: " << age << endl;
    cout << "Number of Students: " << numOfStudents << endl;
};

```

```

class Principal: public Teacher{
private:
    String school_name;
    int numOfTeachers;
public:
    void setSchool(const & String s_name){school_name = s_name;}
    void print() const;
    int getAge() const{return age;}
    const String & getName(){return name;}
};
int main(){
    Teacher t1;
    Principal p1;
    t1.numOfStudents = 100; // sai
    t1.setName("Ali Bilir");
    p1.setSchool("Istanbul Lisesi");
    return 0;
}

```

Sự khác nhau giữa các thành viên private và các thành viên protected

Nói chung dữ liệu của lớp nên là private . Các thành viên dữ liệu public có thể bị sửa đổi bất kỳ lúc nào trong chương trình nên được tránh sử dụng . Các thành viên dữ liệu protected có thể bị sửa đổi bởi các hàm trong bất kỳ lớp kế thừa nào . Bất kỳ người dùng nào cũng có thể kế thừa một lớp nào đó và truy cập vào các thành viên dữ liệu protected của lớp cơ sở. Do đó sẽ an toàn và tin cậy hơn nếu các lớp kế thừa không thể truy cập vào các dữ liệu của lớp cơ sở một cách trực tiếp .

Nhưng trong các hệ thống thời gian thực , nơi mà tốc độ là rất quan trọng , các lời gọi hàm truy cập vào các thành viên private có thể làm chậm chương trình . Trong các hệ thống như vậy dữ liệu có thể được định nghĩa là protected để các lớp kế thừa có thể truy cập tới chúng trực tiếp và nhanh hơn .

Ví dụ:

```

class A{
private:
    int i;
public:
    void access(int new_i){
        if( new_i > 0 && new_i <= 100)
            i = new_i;
    }
};
class B: public A{
private:
    int k;
public:
    void set(int new_i, int new_k){
        A::access(new_i); // an toàn nhưng chậm
        ....
    }
};

```

```

class A{
    protected:
        int i;
    public:
        .....
};
class B: public A{
    private:
        int k;
    public:
        void set(int new_i, int new_k){
            i = new_i; // nhanh
            ....
        }
};

```

6. Các kiểu kế thừa

Kế thừa public

Đây là kiểu kế thừa mà chúng ta hay dùng nhất:

```

class Base{
};
class Derived: public Base{
};

```

Kiểu kế thừa này được gọi là public inheritance hay public derivation . Quyền truy cập của các thành viên của lớp cơ sở không thay đổi . Các đối tượng của lớp dẫn xuất có thể truy cập vào các thành viên public của lớp cơ sở . Các thành viên public của lớp cơ sở cũng sẽ là các thành viên public của lớp kế thừa.

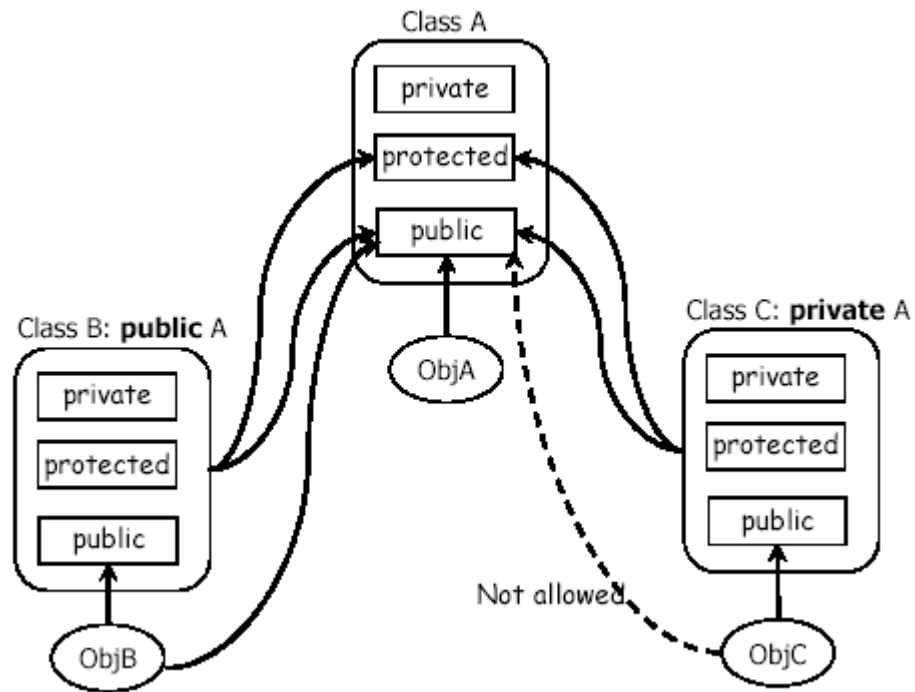
Kế thừa private

```

class Base{
};
class Derived: private Base{
};

```

Kiểu kế thừa này có tên gọi là private inheritance . Các thành viên public của lớp cơ sở trở thành các thành viên private của lớp dẫn xuất . Các đối tượng của lớp dẫn xuất không thể truy cập vào các thành viên của lớp cơ sở . Các hàm thành viên của lớp dẫn xuất có thể truy cập vào các thành viên public và protected của lớp cơ sở.



7. Định nghĩa lại các đặc tả truy cập

Các đặc tả truy cập của các thành viên public của lớp cơ sở có thể được định nghĩa lại trong lớp kế thừa. Khi chúng ta kế thừa theo kiểu private, tất cả các thành viên public của lớp cơ sở sẽ trở thành private. Nếu chúng ta muốn chúng vẫn là public trong lớp kế thừa chúng ta sẽ sử dụng từ khóa using (chú ý là Turbo C++ 3.0 không hỗ trợ từ khóa này) và tên thành viên đó (không có danh sách tham số và kiểu trả về) trong phần public của lớp kế thừa:

```
class Base{
    private:
        int k;
    public:
        int i;
        void f();
};
class Derived: public Base{
    private:
        int m;
    public:
        using Base::f;
        void fb1();
};
int main(){
    Base b;
    Derived d;
    b.i = 5;
    d.i = 0; // Sai
    b.f();
    d.f(); // Ok
    return 0;
}
```


}

8. Các hàm không thể kế thừa

Một vài hàm sẽ cần thực hiện các công việc khác nhau trong lớp cơ sở và lớp dẫn xuất. Chúng là các hàm toán tử gán $=$, hàm hủy tử và tất cả các hàm cấu tử. Chúng ta hãy xem xét một hàm cấu tử, đối với lớp cơ sở hàm cấu tử của nó có trách nhiệm khởi tạo các thành viên dữ liệu và cấu tử của lớp dẫn xuất có trách nhiệm khởi tạo các thành viên dữ liệu của lớp dẫn xuất. Và bởi vì các cấu tử của lớp dẫn xuất và lớp cơ sở tạo ra các dữ liệu khác nhau nên chúng ta không thể sử dụng hàm cấu tử của lớp cơ sở cho lớp dẫn xuất và do đó các hàm cấu tử là không thể kế thừa.

Tương tự như vậy toán tử gán của lớp dẫn xuất phải gán các giá trị cho dữ liệu của lớp dẫn xuất, và toán tử gán của lớp cơ sở phải gán các giá trị cho dữ liệu của lớp cơ sở. Chúng làm các công việc khác nhau vì thế toán tử này không thể kế thừa một cách tự động.

9. Các hàm cấu tử và kế thừa

Khi chúng ta định nghĩa một đối tượng của một lớp dẫn xuất, cấu tử của lớp cơ sở sẽ được gọi tới trước cấu tử của lớp dẫn xuất. Điều này là bởi vì đối tượng của lớp cơ sở là một đối tượng con - một phần - của đối tượng lớp dẫn xuất, và chúng ta cần xây dựng nó từng phần trước khi xây dựng toàn bộ nội dung của nó. Ví dụ:

```
class Parent
{
    public:
        Parent(){ cout << endl<< "  Parent constructor"; }
};
class Child : public Parent
{
    public:
        Child(){ cout << endl<<"  Child constructor"; }
};

int main()
{
    cout << endl<<"Starting";
    Child ch;           // create a Child object
    cout << endl<<"Terminating";
    return 0;
}
```

Nếu như cấu tử của lớp cơ sở là một hàm có tham số thì nó cũng cần phải được gọi tới trước cấu tử của lớp dẫn xuất:

```
class Teacher{
    String name;
    int age, numOfStudents;
    public:
        Teacher(const String & new_name): name(new_name){}
};
class Principal: public Teacher{
    int numOfTeachers;
```

```

    public:
        Principal(const String &, int);
};
Principal::Principal(const String & new_name, int numOT):Teacher(new_name){
    NumOfTeachers = numOT;
}

```

Hãy nhớ lại rằng toán tử khởi tạo cấu tử cũng có thể được dùng để khởi tạo các thành viên:

```

Principal::Principal(const String & new_name, int numOT)
    :Teacher(new_name),NumOfTeachers(numOT){
}
int main(){
    Principal p1("Ali Baba", 20);
    return 0;
}

```

Nếu lớp cơ sở có một cấu tử và hàm cấu tử này cần có các tham số thì lớp dẫn xuất phải có một cấu tử gọi tới cấu tử đó với các giá trị tham số thích hợp.

Các hàm hủy tử được gọi tới một cách tự động. Khi một đối tượng của lớp dẫn xuất ra ngoài tầm hoạt động các cấu tử sẽ được gọi tới theo thứ tự ngược lại với thứ tự của các hàm cấu tử. Đối tượng dẫn xuất sẽ thực hiện các thao tác dọn dẹp trước sau đó là đối tượng cơ sở.

Ví dụ:

```

class Parent {
public:
    Parent() { cout << "Parent constructor" << endl; }
    ~Parent() { cout << "Parent destructor" << endl; }
};

class Child : public Parent {
public:
    Child() { cout << "Child constructor" << endl; }
    ~Child() { cout << "Child destructor" << endl; }
};
int main(){
    cout << "Start" << endl;
    Child ch;                // create a Child object
    cout << "End" << endl;
    return 0;
}

```

10. Composition và Inheritance

Mỗi khi chúng ta đặt một thể nghiệm (instance) dữ liệu vào trong một lớp là chúng ta đã tạo ra một mối quan hệ “có”. Nếu có một lớp Teacher và một trong các phần tử dữ liệu trong lớp này là tên của giáo viên thì chúng ta có thể nói rằng một đối tượng Teacher có một tên. Mối quan hệ này được gọi là mối quan hệ tổng hợp (composition) vì mỗi đối tượng của lớp Teacher được tạo thành từ các biến khác. Hoặc là chúng ta có thể nhớ lại rằng mỗi đối tượng thuộc lớp ComplexFrac đều có hai thành viên thuộc lớp Fraction.

Quan hệ tổng hợp trong OOP mô hình hóa mối quan hệ trong thế giới thực trong đó các đối tượng được xây dựng từ tổ hợp của các đối tượng khác.

Kế thừa trong OOP là sự ánh xạ ý tưởng mà chúng ta gọi là tổ hợp quát hóa trong thế giới thực. Ví dụ nếu chúng ta mô hình hóa các công nhân, quản lý và nhà nghiên cứu trong một nhà máy thì chúng ta có thể nói rằng các kiểu cụ thể này đều thuộc về một kiểu người mang tính chung hơn là “người làm thuê”. Trong đó mỗi người làm thuê có các đặc điểm cụ thể sau đây: định danh (ID), tên, tuổi và vân vân. Nhưng một nhà quản lý chẳng hạn ngoài các thuộc tính chung đó còn có thêm một số thuộc tính chuyên biệt khác chẳng hạn như tên phòng ban mà anh ta quản lý... Nhà quản lý là một người làm thuê và giữa hai lớp (kiểu) người “nhà quản lý” và “người làm thuê” có một quan hệ kế thừa.

Chúng ta có thể sử dụng cả hai kiểu quan hệ này trong chương trình để thực hiện mục đích sử dụng lại mã chương trình. Quan hệ tổng hợp hay tổ hợp thường hay được sử dụng khi mà chúng ta muốn sử dụng các thuộc tính của một lớp trong một lớp khác chứ không phải là giao diện của lớp đó. Khi đó lớp mới sẽ sử dụng các thuộc tính của lớp cơ sở để xây dựng nên giao diện của nó và người sử dụng lớp mới sẽ làm việc với giao diện mà chúng ta tạo ra cho lớp mới này. Ví dụ:

```
class Engine {
public:
    void start() const {}
    void rev() const {}
    void stop() const {}
};
class Wheel {
public:
    void inflate(int psi) const {}
};
class Window {
public:
    void rollup() const {}
    void rolldown() const {}
};
class Door {
public:
    Window window;
    void open() const {}
    void close() const {}
};
class Car {
public:
    Engine engine;
    Wheel wheel[4];
    Door left, right; // 2-door
};
int main() {
    Car car;
    car.left.window.rollup();
    car.wheel[0].inflate(72);
}
```

```
}
```

11. Đa kế thừa

Đa kế thừa là trường hợp mà một lớp kế thừa các thuộc tính từ hai hoặc nhiều hơn các lớp cơ sở, ví dụ:

```
class Base1{      // Base 1
public:
    int a;
    void fa1(){cout << "Base1 fa1" << endl;}
    char *fa2(int){cout << "Base1 fa2" << endl;return 0;}
};
class Base2{      // Base 2
public:
    int a;
    char *fa2(int, char){cout << "Base2 fa2" << endl;return 0;}
    int fc(){cout << "Base2 fc" << endl;return 0;}
};
class Deriv : public Base1 , public Base2{
public:
    int a;
    float fa1(float){cout << "Deriv fa1" << endl;return 1.0;}
    int fb1(int){cout << "Deriv fb1" << endl;return 0;}
};

int main(){
    Deriv d;
    d.a=4;                      //Deriv::a
    d.Base2::a=5;               //Base2::a
    float y=d.fa1(3.14);        // Deriv::fa1
    int i=d.fc();               // Base2::fc
    //char *c = d.fa2(1);       // ERROR
    return 0;
}
```

Chú ý là câu lệnh `char * c = d.fa2(1);` là sai vì trong kế thừa các hàm không được overload mà chúng bị override chúng ta cần phải viết là `: char * c = d.Base1::fa1(1);` hoặc `char * c = d.Base::fa2(1,"Hello");`

12. Lặp lại lớp cơ sở trong đa kế thừa và lớp cơ sở ảo

Chúng ta hãy xét ví dụ sau:

```
class Gparent{};
class Mother: public Gparent{};
class Father: public Gparent{};
class Child: public Mother, public Father{};
```

Cả hai lớp `Mother` và `Father` đều kế thừa từ lớp `Gparent` và lớp `Child` kế thừa từ hai lớp `Mother` và `Father`. Hãy nhớ lại rằng mỗi đối tượng được tạo ra nhờ kế thừa đều chứa một đối tượng con của lớp cơ sở. Mỗi đối tượng của lớp `Mother` và `Father` đều chứa các đối tượng con của lớp `Gparent` và một đối tượng của lớp `Child` sẽ chứa các đối tượng con của hai lớp `Mother` và `Father` vì thế một đối tượng của lớp `Child` sẽ chứa

hai đối tượng con của lớp Gparent , một được kế thừa từ lớp Mother và một từ lớp Father.

Đây là một trường hợp lạ vì có hai đối tượng con trong khi chỉ nên có 1.

Ví dụ giả sử có một phần tử dữ liệu trong lớp Gparent:

```
class Gparent{
protected:
    int gdata;
```

```
};
```

Và chúng ta sẽ truy cập vào phần tử dữ liệu này trong lớp Child:

```
class Child: public Mother, public Father{
public:
    void Cfunc(){
        int item = gdata; // Sai
    }
};
```

Trình biên dịch sẽ phàn nàn rằng việc truy cập tới phần tử dữ liệu gdata là mập mờ và lỗi. Nó không biết truy cập tới phần tử gdata nào : của đối tượng con Gparent trong đối tượng con Mother hay của đối tượng con Gparent trong đối tượng con Father .

Để giải quyết trường hợp này chúng ta sẽ sử dụng một từ khóa mới , **virtual**, khi kế thừa Mother và Father từ lớp Gparent:

```
class Gparent{};
class Mother: virtual public Gparent{};
class Father: virtual public Gparent{};
class Child: public Father, public Mother{};
```

Từ khóa virtual báo cho trình biên dịch biết là chỉ kế thừa duy nhất một đối tượng con từ một lớp trong các lớp dẫn xuất . Việc sử dụng từ khóa virtual giải quyết được vấn đề nhập nhằng trên song lại làm nảy sinh rất nhiều vấn đề khác .

Nói chung thì chúng ta nên tránh dùng đa kế thừa mặc dù có thể chúng ta đã là một chuyên gia lập trình C ++, và nên suy nghĩ xem tại sao lại phải dùng đa kế thừa trong các trường hợp hiếm hoi thực sự cần thiết.

Để tìm hiểu kỹ hơn về đa kế thừa chúng ta có thể xem chương 6: đa kế thừa của sách tham khảo: "Thinking in C++, 2nd Edition".

13. Con trỏ và các đối tượng

Các đối tượng được lưu trong bộ nhớ nên các con trỏ cũng có thể trỏ tới các đối tượng giống như chúng có thể trỏ tới các biến có kiểu cơ bản.

Các toán tử new và delete được cũng được sử dụng bình thường đối với các con trỏ trỏ tới các đối tượng của một lớp . Toán tử new thực hiện cấp phát bộ nhớ và trả về điểm bắt đầu của vùng nhớ nếu thành công, nếu thất bại nó trả về 0. Khi chúng ta dùng toán tử new nó không chỉ thực hiện cấp phát bộ nhớ mà còn tạo ra đối tượng bằng cách gọi tới cấu tử của lớp tương ứng. Toán tử delete được dùng để giải phóng vùng nhớ mà một con trỏ trỏ tới chiếm giữ.

Danh sách liên kết các đối tượng

Một lớp có thể chứa một con trỏ tới các đối tượng của chính lớp đó . Con trỏ này có thể được sử dụng để xây dựng các cấu trúc dữ liệu cố định hạn như một danh sách liên kết các đối tượng của một lớp:

```
class Teacher{
```

```

    friend class Teacher_list;
    String name;
    int age, numOfStudents;
    Teacher * next;    // Pointer to next object of teacher
public:
    Teacher(const String &, int, int); // Constructor
    void print() const;
    const String& getName() const {return name;}
    ~Teacher() {                // only to show that the destructor is called
        cout<<" Destructor of teacher" << endl;
    }
};

Teacher::Teacher(const String &new_name,int a,int nos){
    name = new_name;
    age=a;
    numOfStudents=nos;
    next=0;
}

void Teacher::print() const{
    cout <<"Name: "<< name<<" Age: "<< age<< endl;
    cout << "Number of Students: " <<numOfStudents << endl;
}

class Teacher_list{ // linked list for teachers
    Teacher *head;
public:
    Teacher_list(){head=0;}
    bool append(const String &,int,int);
    bool del(const String &);
    void print() const ;
    ~Teacher_list();
};

// Append a new teacher to the end of the list
// if there is no space returns false, otherwise true
bool Teacher_list::append(const String & n, int a, int nos){
    Teacher *previous, *current, *new_teacher;
    new_teacher=new Teacher(n,a,nos);
    if (!new_teacher) return false; // if there is no space return false
    if(head)    // if the list is not empty
    {
        previous=head;
        current=head->next;
        while(current)    // search for the end of the list
        {
            previous=current;
            current=current->next;
        }
        previous->next=new_teacher;
    }
}

```

```

        else          // if the list is empty
            head=new_teacher;
        return true;
    }
    // Delete a teacher with the given name from the list
    // if the teacher is not found returns false, otherwise true
    bool Teacher_list::del(const String & n)
    {
        Teacher *previous, *current;
        if(head)      // if the list is not empty
        {
            if (n==head->getName()) //1st element is to be deleted
            {
                previous=head;
                head=head->next;
                delete previous;
                return true;
            }
            previous=head;
            current=head->next;
            while( (current) && (n!=current->getName()) ) // search for the end of the
list
            {
                previous=current;
                current=current->next;
            }
            if (current==0) return false;
            previous->next=current->next;
            delete current;
            return true;
        } //if (head)
        else          // if the list is empty
            return false;
    }
    // Prints all elements of the list on the screen
    void Teacher_list::print() const
    {
        Teacher *tempPtr;
        if (head)
        {
            tempPtr=head;
            while(tempPtr)
            {
                tempPtr->print();
                tempPtr=tempPtr->next;
            }
        }
        else

```

```

        cout << "The list is empty" << endl;
    }

    // Destructor
    // deletes all elements of the list
    Teacher_list::~Teacher_list()
    {
        Teacher *temp;
        while(head)    // if the list is not empty
        {
            temp=head;
            head=head->next;
            delete temp;
        }
    }

    // ----- Main Function -----
    int main()
    {
        Teacher_list theList;
        theList.print();
        theList.append("Teacher1",30,50);
        theList.append("Teacher2",40,65);
        theList.append("Teacher3",35,60);
        theList.print();
        if (!theList.del("TeacherX")) cout << " TeacherX not found" << endl;
        theList.print();
        if (!theList.del("Teacher1")) cout << " Teacher1 not found" << endl;
        theList.print();
        return 0;
    }

```

Trong ví dụ trên lớp Teacher phải có một con trỏ trỏ tới đối tượng tiếp theo trong lớp danh sách và lớp danh sách phải được khai báo như là một lớp bạn , để người dùng của lớp này cps thể xây dựng lên các danh sách liên kết . Nếu như lớp này được viết bởi những người làm việc trong một nhóm thì chẳng có vấn đề gì nhưng thường thì chúng ta muốn xây dựng danh sách các đối tượng đã đư ợc xây dựng chẳng hạn các danh sách các đối tượng thuộc các lớp thư viện chẳng hạn , và tất nhiên là các lớp này không có các con trỏ tới đối tượng tiếp theo cùng lớp với nó . Để xây dựng các danh sách như vậy chúng ta sẽ xây dựng các lớp lá , mỗi đối tượng của nút lá sẽ lưu giữ các địa chỉ của một phần tử trong danh sách:

```

class Teacher_node{
    friend class Teacher_list;
    Teacher * element;
    Teacher_node * next;
    Teacher_node(const String &,int,int);    // constructor
    ~Teacher_node();                        // destructor
};

```



```

Teacher_node::Teacher_node(const String & n, int a, int nos){
    element = new Teacher(n,a,nos);
    next = 0;
}

Teacher_node::~~Teacher_node(){
    delete element;
}

// *** class to define a linked list of teachers ***
class Teacher_list{ // linked list for teachers
    Teacher_node *head;
public:
    Teacher_list(){head=0;}
    bool append(const String &,int,int);
    bool del(const String &);
    void print() const ;
    ~Teacher_list();
};

// Append a new teacher to the end of the list
// if there is no space returns false, otherwise true
bool Teacher_list::append(const String & n, int a, int nos){
    Teacher_node *previous, *current;
    if(head) // if the list is not empty
    {
        previous=head;
        current=head->next;
        while(current) // search for the end of the list
        {
            previous=current;
            current=current->next;
        }
        previous->next = new Teacher_node(n, a, nos);
        if (!(previous->next)) return false; // If memory is full
    }
    else // if the list is empty
    {
        head = new Teacher_node(n, a, nos); // Memory for new node
        if (!head) return false; // If memory is full
    }
    return true;
}

// Delete a teacher with the given name from the list
// if the teacher is not found returns false, otherwise true
bool Teacher_list::del(const String & n){
    Teacher_node *previous, *current;
    if(head) // if the list is not empty
    {

```

```

        if (n==(head->element)->getName()) //1st element is to be deleted
        {
            previous=head;
            head=head->next;
            delete previous;
            return true;
        }
        previous=head;
        current=head->next;
        while( (current) && (n != (current->element)->getName()) )
// search for the end of the list
        {
            previous=current;
            current=current->next;
        }
        if (current==0) return false;
        previous->next=current->next;
        delete current;
        return true;
    } //if (head)
    else // if the list is empty
        return false;
}

// Prints all elements of the list on the screen
void Teacher_list::print() const{
    Teacher_node *tempPtr;
    if (head){
        empPtr=head;
        while(tempPtr){
            tempPtr->element->print();
            empPtr=tempPtr->next;
        }
    }else
        ut << "The list is empty" << endl;
}

// Destructor
// deletes all elements of the list
Teacher_list::~~Teacher_list(){
    Teacher_node *temp;
    while(head) // if the list is not empty
    {
        temp=head;
        head=head->next;
        delete temp;
    }
}

```

14. Con trỏ và kế thừa

Nếu như một lớp dẫn xuất Derived có một lớp cơ sở public Base thì một con trỏ của lớp Derived có thể được gán cho một biến con trỏ của lớp Base mà không cần có các thao tác chuyển kiểu tường minh nhưng thao tác ngược lại cần phải được chỉ rõ ràng, tường minh.

Ví dụ một con trỏ của lớp Teacher có thể trỏ tới các đối tượng của lớp Principal. Một đối tượng Principal thì luôn là một đối tượng Teacher nhưng điều ngược lại thì không phải luôn đúng.

```
class Base{};
class Derived: public Base{};
Derived d;
Base * bp = &d; // chuyển kiểu không tường minh
Derived * dp = bp; // lỗi
dp = static_cast<Derived*>(bp);
```

Nếu như là kế thừa private thì chúng ta không thể thực hiện việc chuyển kiểu không tường minh từ lớp dẫn xuất về lớp cơ sở vì trong trường hợp đó một thành phần public của lớp cơ sở chỉ có thể được truy cập qua một con trỏ lớp cơ sở chứ không thể qua một con trỏ lớp dẫn xuất:

```
class Base{
    int m1;
    public: int m2;
};
class Derived: public Base{};
Derived d;
d.m2 = 5; // Lỗi
Base * bp = &d; // chuyển kiểu không tường minh, lỗi
bp = static_cast<Derived*>(&d);
bp->m2 = 5;
```

Việc kết hợp con trỏ với kế thừa cho phép chúng ta có thể xây dựng các danh sách liên kết hỗn hợp có khả năng lưu giữ các đối tượng thuộc các lớp khác nhau, chúng ta sẽ học kỹ phần này trong chương sau.

Chương 7: Dynamic Binding và Polymorphism. (6 tiết)

Dynamic Binding

1. Một số đặc điểm của ràng buộc động

- Khi thiết kế một hệ thống thường các nhà phát triển hệ thống gặp một trong số các tình huống sau đây:

- Hiểu rõ về các giao diện lớp mà họ muốn mà không hiểu biết chính xác về cách trình bày hợp lý nhất.
- Hiểu rõ về thuật toán mà họ muốn sử dụng song lại không biết cụ thể các thao tác nào nên được cài đặt.

Trong cả hai trường hợp thường thì các nhà phát triển mong muốn trì hoãn một số các quyết định cụ thể càng lâu càng tốt. Mục đích là giảm các cố gắng đòi hỏi để thay đổi cài đặt khi đã có đủ thông tin để thực hiện một quyết định có tính chính xác hơn.

Vì thế sẽ rất tiện lợi nếu có một cơ chế cho phép trừu tượng hóa việc “đặt chỗ trước”.

- Che dấu thông tin và trừu tượng dữ liệu cung cấp các khả năng “place – holder” phụ thuộc thời điểm biên dịch và thời điểm liên kết . Ví dụ: các thay đổi về việc representation đòi hỏi phải biên dịch lại hoặc liên kết lại .
- Ràng buộc động cho phép thực hiện khả năng “place – holder” một cách linh hoạt . Ví dụ trì hoãn một số quyết định cụ thể cho tới thời điểm chương trình được thực hiện mà không làm ảnh hưởng tới cấu trúc mã chương trình hiện tại.

- Ràng buộc động không mạnh bằng các con trỏ hàm nhưng nó mang tính tổng hợp hơn và làm giảm khả năng xuất hiện lỗi hơn vì một số lý do chẳng hạn trình biên dịch sẽ thực hiện kiểm tra kiểu tại thời điểm biên dịch.

- Ràng buộc động cho phép các ứng dụng có thể gọi tới các phương thức mang tính chung chung qua các con trỏ tới lớp cơ sở . Tại thời điểm chương trình thực hiện các lời gọi hàm này sẽ được chỉ định tới các phương thức cụ thể được cài đặt tại các lớp dẫn xuất thích hợp.

Polymorphism

Trong lập trình hướng đối tượng có 3 khái niệm chính là:

- Các lớp
- Kế thừa
- Đa thể, được cài đặt trong ngôn ngữ C++ bằng các hàm ảo.

Trong cuộc sống thực tế, thường có một tập các đối tượng khác nhau có các chỉ thị (instruction) (message) giống nhau, nhưng lại thực hiện các hành động khác nhau . Ví dụ như là hai lớp các đối tượng giáo viên và hiệu trưởng trong một trường học .

Giả sử ông bộ trưởng bộ giáo dục muốn gửi một chỉ thị xuống cho tất cả các nhân viên “In thông tin cá nhân của ông ra và gửi cho tôi” . Các loại nhân viên khác nhau của bộ giáo dục (giáo viên bình thường và hiệu trưởng) sẽ in ra các thông tin khác nhau. Nhưng ông bộ trưởng không cần gửi các thông điệp khác nhau cho các nhóm nhân viên khác nhau của ông ta . Chỉ cần một thông điệp cho tất cả các nhân viên vì tất cả các nhân viên đều biết in ra thông tin hay lý lịch cá nhân của mình như thế nào .

Đa thể (polymorphism) có nghĩa là “take many shapes” . Câu lệnh hay chỉ thị đơn của bộ trưởng chính là một trường hợp đa thể vì nó sẽ có dạng khác nhau đối với các loại nhân lực khác nhau.

Thường thường đa thể xảy ra trong các lớp có mối liên hệ kế thừa lẫn nhau . Trong C++ đa thể có nghĩa là một lời gọi tới một hàm thành viên sẽ tạo ra một hàm khác nhau để thực hiện phụ thuộc vào loại đối tượng có hàm thành viên được gọi tới.

Điều này nghe có vẻ giống như là overload hàm , nhưng thực ra không phải, đa thể mạnh hơn là chồng hàm về mặt kỹ thuật . Một sự khác nhau giữa đa thể và chồng hàm đó là cách thức lựa chọn hàm để thực hiện.

Với chồng hàm sự lựa chọn được thực hiện bởi trình biên dịch vào thời điểm biên dịch. Với đa thể việc lựa chọn hàm để thực hiện được thực hiện khi chương trình đang chạy.

1. Các hàm thành viên bình thường được truy cập qua các con trỏ

Ví dụ đầu tiên sẽ cho chúng ta thấy điều gì sẽ xảy ra khi một lớp cơ sở và các lớp dẫn xuất đều có các hàm có cùng tên và các hàm này được truy cập thông qua các con trỏ nhưng không sử dụng các hàm ảo (không phải đa thể).

```
class Teacher{ // Base class
```

```

String *name;
int numOfStudents;
public:
    Teacher(const String &, int); // Constructor of base
    void print() const{
        cout << "Name: " << name << endl;
        cout << " Num of Students:" << numOfStudents << endl;
    }
};

void Teacher::print() const    // Non-virtual function{
    cout << "Name: " << name << endl;
    cout << " Num of Students:" << numOfStudents << endl;
}

class Principal : public Teacher{ // Derived class
    String *SchoolName;
public:
    Principal(const String &, int , const String &);
    void print() const{
        teacher::print();
        cout << " Name of School:" << SchoolName << endl;
    }
};

void Principal::print() const    // Non-virtual function{
    Teacher::print();
    cout << " Name of School:" << SchoolName << endl;
}

int main()
{
    Teacher t1("Teacher 1",50);
    Principal p1("Principal 1",40,"School");
    Teacher *ptr;
    char c;
    cout << "Teacher or Principal "; cin >> c;
    if (c=='t') ptr=&t1;
    else ptr=&p1;
    ptr->print(); // which print ??
}

```

Lớp Principal kế thừa từ lớp cơ sở Teacher . Cả hai lớp đều có hàm thành viên print(). Trong hàm main () chương trình tạo ra các đối tượng của hai lớp Teacher và Principal và một con trỏ trỏ tới lớp Teacher . Sau đó nó truyền địa chỉ của đối tượng thuộc lớp dẫn xuất vào con trỏ của lớp cơ sở bằng lệnh:

ptr = &p1; // địa chỉ của lớp dẫn xuất trong con trỏ trỏ tới lớp cơ sở.

Hãy nhớ rằng hoàn toàn hợp lệ khi thực hiện gán một địa chỉ của một đối tượng thuộc lớp dẫn xuất cho một con trỏ của lớp cơ sở , vì các con trỏ tới các đối tượng của

một lớp dẫn xuất hoàn toàn tương thích về kiểu với các con trỏ tới các đối tượng của lớp cơ sở.

Bây giờ câu hỏi đặt ra là khi thực hiện câu lệnh:

```
ptr->print();
```

thì hàm nào sẽ được gọi tới ? Là hàm print () của lớp dẫn xuất hay hàm print () của lớp cơ sở.

Hàm print() của lớp cơ sở sẽ được thực hiện trong cả hai trường hợp . Trình biên dịch sẽ bỏ qua nội dung của con trỏ ptr và chọn hàm thành viên khớp với kiểu của con trỏ.

2. Các hàm thành viên ảo được truy cập qua các con trỏ

Bây giờ chúng ta sẽ thay đổi một chút trong chương trình : đặt thêm từ khóa **virtual** trước khai báo của hàm print() trong lớp cơ sở.

```
class Teacher{
    // Base class
    String name;
    int numOfStudents;
public:
    Teacher(const String & new_name,int nos){ // Constructor of base
        name=new_name;numOfStudents=nos;
    }
    virtual void print() const;           // print is a virtual function
};

void Teacher::print() const               // virtual function
{
    cout << "Name: "<< name << endl;
    cout << " Num of Students:"<< numOfStudents << endl;
}

class Principal : public Teacher{        // Derived class
    String SchoolName;
public:
    Principal(const String & new_name,int nos, const String & sn)
        :Teacher(new_name,nos)
    {
        SchoolName=sn;
    }
    void print() const;
};

void Principal::print() const             // Non-virtual function
{
    Teacher::print();
    cout << " Name of School:"<< SchoolName << endl;
}

int main()
```

```

{
    Teacher t1("Teacher 1",50);
    Principal p1("Principal 1",40,"School");
    Teacher *ptr;
    char c;
    cout << "Teacher or Principal "; cin >> c;
    if (c=='t') ptr=&t1;
        else ptr=&p1;
    ptr->print();           // which print compare with example e81.cpp
    return 0;
}

```

Giờ thì các hàm khác nhau sẽ được thực hiện , phụ thuộc vào nội dung của con trỏ ptr. Các hàm được gọi dựa trên nội dung của con trỏ ptr , chứ không dựa trên kiểu của con trỏ. Đó chính là cách thức làm việc của đa thể . Chúng ta đã làm cho hàm print () trở thành đa thể bằng cách gán cho nó kiểu hàm ảo.

3. Ràng buộc động

Ràng buộc động hay ràng buộc muộn là một khái niệm gắn liền với khái niệm đa thể. Chúng ta hãy xem xét câu hỏi sau : làm thế nào trình biên dịch biết được hàm nào để biên dịch ? Trong ví dụ trước trình biên dịch không có vấn đề gì khi nó gặp câu lệnh:

```
ptr->print();
```

Câu lệnh này sẽ được biên dịch thành một lời gọi tới hàm print () của lớp cơ sở . Nhưng trong ví dụ sau (7.1) trình biên dịch sẽ không nội dung của lớp nào được ptr trỏ tới. Đó có thể là nội dung của một đối tượng thuộc lớp Teacher hoặc lớp Principal . Phiên bản nào của hàm print () sẽ được gọi tới ? Trên thực tế tại thời điểm biên dịch chương trình trình biên dịch sẽ không biết làm thế nào vì thế nó sẽ sắp xếp sao cho việc quyết định chọn hàm nào để thực hiện được trì hoãn cho tới khi chương trình thực hiện.

Tại thời điểm chương trình được thực hiện khi lời gọi hàm được thực hiện mã mà trình biên dịch đặt vào trong chương trình sẽ tìm đúng kiểu của đối tượng mà địa chỉ của nó được lưu trong con trỏ ptr và gọi tới hàm print () thích hợp của lớp Teacher hay của lớp Principal phụ thuộc vào lớp của đối tượng.

Chọn lựa một hàm để thực hiện tại thời điểm chương trình thực hiện được gọi là ràng buộc muộn hoặc ràng buộc động (Binding có nghĩa là kết nối lời gọi hàm với hàm). Kết nối các hàm theo cách bình thường, trong khi biên dịch , được gọi là ràng buộc trước (early binding) hoặc ràng buộc tĩnh (static binding). Ràng buộc động đòi hỏi chúng ta cần xài sang hơn một chút (lời gọi hàm đòi hỏi khoảng 10 phần trăm mã hàm) nhưng nó cho phép tăng năng lực cũng như sự linh hoạt của các chương trình lên gấp bội.

4. Ràng buộc động làm việc như thế nào

Hãy nhớ lại rằng, lưu trữ trong bộ nhớ, một đối tượng bình thường – không có hàm thành viên ảo chỉ chứa các thành phần dữ liệu của chính nó ngoài ra không có gì khác . Khi một hàm thành viên được gọi tới với một đối tượng nào đó trình biên dịch sẽ truyền địa chỉ của đối tượng cho hàm . Địa chỉ này là luôn sẵn sàng đối với các hàm thông qua con trỏ this , con trỏ được các hàm sử dụng để truy cập vào các thành viên dữ liệu của các đối tượng trong phần cài đặt của hàm . Địa chỉ này thường được sinh bởi trình biên dịch mỗi khi một hàm thành viên được gọi tới ; nó không được chứa

trong đối tượng và không chiếm bộ nhớ. Con trỏ this là kết nối duy nhất giữa các đối tượng và các hàm thành viên bình thường của nó.

Với các hàm ảo, công việc có vẻ phức tạp hơn đôi chút. Khi một lớp dẫn xuất với các hàm ảo được chỉ định, trình biên dịch sẽ tạo ra một bảng – một mảng – các địa chỉ hàm được gọi là bảng ảo. Trong ví dụ 71 các lớp Teacher và Principal đều có các bảng hàm ảo của riêng chúng. Có một entry (lỗi vào) trong mỗi bảng hàm ảo cho mỗi một hàm ảo của lớp. Các đối tượng của các lớp với các hàm ảo chứa một con trỏ tới bảng hàm ảo của lớp. Các đối tượng này lớn hơn đôi chút so với các đối tượng bình thường.

Trong ví dụ khi một hàm ảo được gọi tới với một đối tượng của lớp Teacher hoặc Principal trình biên dịch thay vì chỉ định hàm nào sẽ được gọi sẽ tạo ra mã trước hết tìm bảng hàm ảo của đối tượng và sau đó sử dụng bảng hàm ảo đó để truy cập vào địa chỉ hàm thành viên thích hợp. Vì thế đối với các hàm ảo đối tượng tự nó quyết định xem hàm nào được gọi thay vì giao công việc này cho trình biên dịch.

Ví dụ: Giả sử các lớp Teacher và Principal chứa hai hàm ảo:

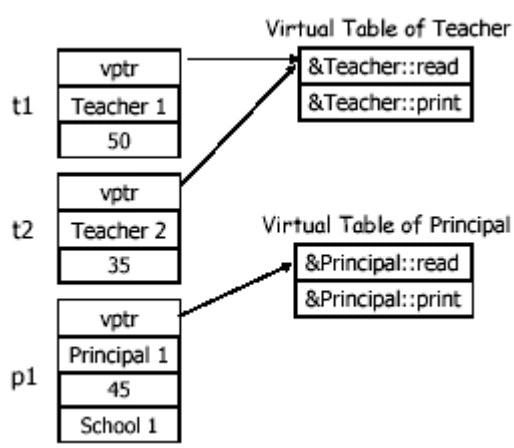
```
class Principal : public Teacher{ // Derived class
    tring *SchoolName;
public:
    void read(); // Virtual function
    void print() const; // Virtual function
};
class Teacher{ // Base class
    String *name;
    int numOfStudents;
public:
    virtual void read(); // Virtual function
    virtual void print() const; // Virtual function
};
```

Khi đó ta có các bảng hàm ảo sau:

Bảng hàm ảo của lớp Teacher	Bảng hàm ảo của lớp Principal
&Teacher::read	&Principal::read
&Teacher::print	&Principal::print

Các đối tượng của lớp Teacher và Principal sẽ chứa một con trỏ tới các bảng hàm ảo của chúng.

```
int main(){
    Teacher t1("Teacher 1", 50);
    Teacher t2("Teacher 2", 35);
    Principal p1("Principal 1", 45, "School 1");
}
```

5. Don't try this with object

Cần ghi nhớ rằng kỹ thuật hàm ảo chỉ làm việc với các con trỏ trỏ tới các đối tượng và với các tham chiếu, chứ không phải bản thân các đối tượng.

```
int main()
{
    Teacher t1("Teacher 1",50);
    Principal p1("Principal 1",40,"School");
    t1.print(); // not polymorphic
    p1.print(); // not polymorphic
    return 0;
}
```

Việc gọi tới các hàm ảo hơi mất thời gian đôi chút vì đó thực chất là việc gọi gián tiếp thông qua bảng hàm ảo. Không nên khai báo các hàm là ảo nếu không cần thiết.

6. Danh sách liên kết các đối tượng và đa thể

Các cách thức chung nhất để sử dụng các hàm ảo là với một mảng các con trỏ trỏ tới các đối tượng và các danh sách liên kết các đối tượng.

Chúng ta xem xét ví dụ sau đây:

Ví dụ 7.3:

```
class Teacher{
    // Base class
    String name;
    int numOfStudents;
public:
    Teacher(const String & new_name,int nos){ // Constructor of base
        name=new_name;numOfStudents=nos;
    }
    virtual void print() const;                // print is a virtual function
};

void Teacher::print() const                    // virtual function
{
    cout << "Name: "<< name << endl;
    cout << " Num of Students:"<< numOfStudents << endl;
}
```

```

class Principal : public Teacher{                                // Derived class
    String SchoolName;
public:
    Principal(const String & new_name,int nos, const String & sn)
        :Teacher(new_name,nos)
    {
        SchoolName=sn;
    }
    void print() const;
};

void Principal::print() const                                    // Non-virtual function
{
    Teacher::print();
    cout << " Name of School:"<< SchoolName << endl;
}

// *** A class to define nodes of the list ***
class List_node{
    friend class List;
    const Teacher * element;
    List_node * next;
    List_node(const Teacher &);    // constructor
};

List_node::List_node(const Teacher & n){
    element = &n;
    next = 0;
}

// *** class to define a linked list of teachers and principals ***
class List{ // linked list for teachers
    List_node *head;
public:
    List(){head=0;}
    Bool append(const Teacher &);
    void print() const ;
    ~List();
};

// Append a new teacher to the end of the list
// if there is no space returns False, otherwise True
Bool List::append(const Teacher & n)
{
    List_node *previous, *current;
    if(head)    // if the list is not empty
    {
        previous=head;

```

```

        current=head->next;
        while(current)          // search for the end of the list
        {
            previous=current;
            current=current->next;
        }
        previous->next = new List_node(n);
        if (!(previous->next)) return False;           // If memory is full
    }
    else          // if the list is empty
    {
        head = new List_node(n); // Memory for new node
        if (!head) return False; // If memory is full
    }
    return True;
}

// Prints all elements of the list on the screen
void List::print() const
{
    List_node *tempPtr;
    if (head)
    {
        tempPtr=head;
        while(tempPtr)
        {
            (tempPtr->element)->print();           // POLYMORPHISM
            tempPtr=tempPtr->next;
        }
    }
    else
        cout << "The list is empty" << endl;
}

// Destructor
// deletes all elements of the list
List::~~List()
{
    List_node *temp;
    while(head) // if the list is not empty
    {
        temp=head;
        head=head->next;
        delete temp;
    }
}

// ----- Main Function -----

```

```

int main()
{
    Teacher t1("Teacher 1",50);
    Principal p1("Principal 1",40,"School1");
    Teacher t2("Teacher 2",60);
    Principal p2("Principal 2",100,"School2");
    List theList;
    theList.print();
    theList.append(t1);
    theList.append(p1);
    theList.append(t2);
    theList.append(p2);
    theList.print();
    return 0;
}

```

7. Các lớp trừu tượng

Để viết các hàm đa thể chúng ta cần phải có các lớp dẫn xuất. Nhưng đôi khi chúng ta không cần phải tạo ra bất kỳ một đối tượng thuộc lớp cơ sở nào cả. Lớp cơ sở tồn tại chỉ như là một điểm khởi đầu cho việc kế thừa của các lớp khác. Kiểu lớp cơ sở như thế được gọi là một lớp trừu tượng, có nghĩa là không có một đối tượng thực sự nào của lớp được tạo ra từ lớp đó.

Các lớp trừu tượng làm nảy sinh rất nhiều tình huống mới. Một nhà máy rất nhiều xe thể thao hoặc một xe tải hoặc một xe cứu thương, nhưng nó không thể tạo ra một chiếc xe chung chung nào đó. Nhà máy phải biết loại xe nào mà nó cần tạo ra trước khi thực sự tạo ra nó. Tương tự chúng ta có thể thấy sparrow (chim sẻ), wren (chim hồng tước), robin (chim két cổ đỏ) nhưng chúng ta không thể thấy một con chim chung chung nào đó.

Thực tế một lớp sẽ là một lớp ảo chỉ trong con mắt của con người. Trình biên dịch sẽ bỏ qua các quyết định của chúng ta về việc biến một lớp nào đó thành lớp ảo.

8. Các hàm ảo thực sự

Sẽ là tốt hơn nếu, đã quyết định tạo ra một lớp trừu tượng cơ sở, chúng ta có thể (hướng dẫn) (instruct) chỉ thị cho trình biên dịch ngăn chặn một cách linh động bất cứ người nào sao cho họ không thể tạo ra bất cứ đối tượng nào của lớp đó. Điều này sẽ cho phép chúng ta tự do hơn trong việc thiết kế lớp cơ sở vì chúng ta sẽ không phải lập kế hoạch cho bất kỳ đối tượng thực sự nào của lớp đó, mà chỉ cần quan tâm tới các dữ liệu và hàm sẽ được sử dụng trong các lớp dẫn xuất. Có một cách để báo cho trình biên dịch biết một lớp là trừu tượng: chúng ta định nghĩa ít nhất một hàm ảo thực sự trong khai báo lớp.

Một hàm ảo thực sự là một hàm ảo không có thân hàm. Thân của hàm ảo trong lớp cơ sở sẽ được loại bỏ và ký pháp =0 sẽ được thêm vào khai báo hàm:

9. Ví dụ 1

```

class generic_shape{ // Abstract base class
protected:
    int x,y;

```

```

    public:
        generic_shape(int x_in,int y_in){ x=x_in; y=y_in;} // Constructor
        virtual void draw() const =0; //pure virtual function
};

class Line:public generic_shape{ // Line class
protected:
    int x2,y2; // End coordinates of line
public:
    Line(int x_in,int y_in,int x2_in,int y2_in):generic_shape(x_in,y_in)
    {
        x2=x2_in;
        y2=y2_in;
    }
    void draw() const { line(x,y,x2,y2); } // virtual draw function
};

```

//line là một hàm thư viện vẽ một đường thẳng lên màn hình

```

class Rectangle:public Line{ // Rectangle class
public:
    Rectangle(int x_in,int y_in,int x2_in,int
y2_in):Line(x_in,y_in,x2_in,y2_in){}
    void draw() const { rectangle(x,y,x2,y2); } // virtual draw
};

```

```

class Circle:public generic_shape{ // Circle class
protected:
    int radius;
public:
    Circle(int x_cen,int y_cen,int r):generic_shape(x_cen,y_cen)
    {
        radius=r;
    }
    void draw() const { circle(x,y, radius); } // virtual draw
};

```

//rectangle và circle là các hàm thư viện vẽ các hình chữ nhật và hình tròn lên màn hình

```

int main()
{
    Line Line1(1,1,100,250);
    Circle Circle1(100,100,20);
    Rectangle Rectangle1(30,50,250,140);
    Circle Circle2(300,170,50);
    show(Circle1); // show function can take different shapes as argument
    show(Line1);
    show(Circle2);
    show(Rectangle1);
}

```

```

        return 0;
    }

    // hàm vẽ các hình khác nhau
    void show(generic_shape &shape)
    { // Which draw function will be called?
        shape.draw(); // It 's unknown at compile-time
    }

```

Nếu chúng ta viết một lớp cho một hình mới bằng cách kế thừa nó từ các lớp đã có chúng ta không cần phải thay đổi hàm show . Hàm này có thể thực hiện chức năng với các lớp mới.

10. Ví dụ 2

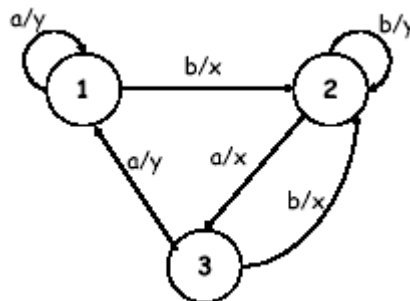
Trong ví dụ này chúng ta sẽ xem xét một “Máy trạng thái hữu hạn” (Finite State Machine) FSM.

Chúng ta có các trạng thái: {1, 2, 3}

Input: {a, b}, x để thoát

Output: {x, y}

Các trạng thái của FSM được định nghĩa bằng cách sử dụng một cấu trúc lớp . Mỗi trạng thái sẽ được kế thừa từ lớp cơ sở.



```

// A Finite State Machine with 3 states
#include<iostream.h>

// *** Base State (Abstract Class) ***
class State{
protected:
    State * const next_a, * const next_b; // Pointers to next state
    char output;
public:
    State( State & a, State & b):next_a(&a),next_b(&b){}
    virtual State* transition(char)=0;
};

// *** State1 ***
class State1:public State{
public:
    State1( State & a, State & b):State(a,b){}
    State* transition(char);
};

```

```

// *** State2 ***
class State2:public State{
public:
    State2( State & a, State & b):State(a,b){}
    State* transition(char);
};

/** State3 */
class State3:public State{
public:
    State3( State & a, State & b):State(a,b){}
    State* transition(char);
};

State* State1::transition(char input)
{
    cout << endl << "Current State: 1";
    switch(input){
        case 'a': output='y';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: 1";
                    return next_a;

        case 'b': output='x';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: 2";
                    return next_b;

        default : cout << endl << "Undefined input";
                    cout << endl << "Next State: Unchanged";
                    return this;
    }
}

State* State2::transition(char input)
{
    cout << endl << "Current State: 2";
    switch(input){
        case 'a': output='x';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: 3";
                    return next_a;

        case 'b': output='y';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: 2";
                    return next_b;

        default : cout << endl << "Undefined input";
                    cout << endl << "Next State: Unchanged";
                    return this;
    }
}

```

```

    }
}

State* State3::transition(char input)
{
    cout << endl << "Current State: 3";
    switch(input){
        case 'a': output='y';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: State1";
                    return next_a;
        case 'b': output='x';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: 2";
                    return next_b;
        default : cout << endl << "Undefined input";
                    cout << endl << "Next State: Unchanged";
                    return this;
    }
}

// *** Finite State Machine ***
// This class has 3 State objects as members
class FSM{
    State1 s1;
    State2 s2;
    State3 s3;
    State * current;
public:
    FSM():s1(s1,s2),s2(s3,s2),s3(s1,s2),current(&s1)
    {}
    void run();
};

void FSM::run()
{
    char in;
    cout << endl << "The finite state machine starts ...";
    do{
        cout << endl << "Give the input value (a or b; x:EXIT) ";
        cin >> in;
        if (in != 'x')
            current = current->transition(in);
        else
            current = 0;           // EXIT
    }while(current);
    cout << endl << "The finite state machine stops ..." << endl;;
}

```



```
int main()
{
    FSM machine1;
    machine1.run();
    return 0;
}
```

Hàm transition của mỗi trạng thái xác định hành vi của FSM . Nó nhận giá trị input như là tham số, kiểm tra input sinh ra giá trị output tùy thuộc vào giá trị input và trả về địa chỉ của trạng thái tiếp theo.

Hàm chuyển đổi (transition) của trạng thái hiện tại được gọi . Giá trị trả về của hàm này sẽ xác định trạng thái tiếp theo của FSM.

11. Cấu tử ảo và hủy tử ảo

Khi chúng ta tạo ra một đối tượng chúng ta thường là đã biết kiểu của đối tượng đang được tạo ra và chúng ta có thể chỉ định điều này cho trình biên dịch . Vì thế chúng ta không cần có các cấu tử ảo.

Cũng như vậy một hàm cấu tử của một đối tượng thiết lập cơ chế ảo của nó (bảng hàm ảo) trước tiên. Chúng ta không nhìn thấy đoạn mã chương trình này , tất nhiên, cũng như chúng ta không nhìn thấy đoạn mã khởi tạo vùng nhớ cho một đối tượng.

Các hàm ảo không thể thậm chí tồn tại trừ khi hàm cấu tử hoàn thành công việc của nó vì thế các hàm cấu tử không thể là các hàm ảo .

Hàm hủy tử ảo:

Ví dụ:

```
// non-virtual function used as base class destructor
#include <iostream.h>
```

```
class Base{
public:
    ~Base() { cout << "Base destructor" << endl; } // Destructor is not
virtual
};
```

```
class Derived : public Base{
public:
    ~Derived() { cout << "Derived destructor" << endl; } // Non-virtual
};
```

```
int main()
{
    Base* pb; // pb can point to objects of Base and Derived
    pb = new Derived; // pb points to an object of Derived
    delete pb;
    cout << "Program terminates" << endl;
    return 0;
}
```

Hãy nhớ lại rằng một đối tượng của lớp dẫn xuất thường chứa dữ liệu từ cả lớp cơ sở và lớp dẫn xuất. Để đảm bảo rằng các dữ liệu này được goodbye một cách hoàn hảo

có thể cần có những lời gọi tới hàm hủy tử cho cả lớp cơ sở và lớp dẫn xuất . Nhưng output của ví dụ trên là:

Base Destructor

Program terminates

Trong chương trình này bp là một con trỏ của lớp cơ sở (kiểu Base). Vì thế nó có thể trỏ tới các đối tượng thuộc lớp Base và Derived. Trong ví dụ trên bp trỏ tới một đối tượng thuộc lớp Derived nhưng trong khi xóa con trỏ này chỉ hàm hủy tử của lớp Base là được gọi tới.

Vấn đề tương tự cũng đã được bắt gặp với các hàm bình thường trong phần trước (các hàm không là hàm hủy tử). Nếu như một hàm không phải là hàm ảo chỉ có phiên bản của lớp cơ sở là được gọi tới thậm chí nếu nội dung của con trỏ là một địa chỉ của một đối tượng của lớp dẫn xuất. Vì thế trong ví dụ trên hàm hủy tử của lớp Derived sẽ không bao giờ được gọi tới. Điều này có thể là một tai họa nếu như hàm này thực hiện một vài công việc cao thủ nào đó. Để sửa chữa chúng ta chỉ cần làm cho hàm hủy tử này trở thành hàm ảo và thế là mọi thứ sẽ trở lại bình thường.

12. Hàm toán tử ảo

Như chúng ta đã thấy khái niệm về ràng buộc động có nghĩa là kiểu động (dynamic type) của đối tượng thực sự sẽ quyết định hàm nào sẽ được gọi thực hiện. Nếu chẳng hạn chúng ta thực hiện lời gọi: $p \rightarrow f(x)$ trong đó p là một con trỏ x là tham số và f là một hàm ảo, thì chính là kiểu của đối tượng mà p trỏ tới sẽ xác định biến thể (variant) nào của hàm f sẽ được gọi thực hiện. Các toán tử cũng là các hàm thành viên nếu như chúng ta có một biểu thức:

a XX b

Trong đó XX là một ký hiệu toán tử thì điều này cũng giống như chúng ta có câu lệnh sau:

a.operatorXX(b);

Nếu như hàm toán tử operatorXX được khai báo là một hàm ảo thì chúng ta sẽ thực hiện lời gọi hàm như sau:

(*p) XX (*q);

Trong đó p và q là hai con trỏ và khi đó kiểu của đối tượng mà p trỏ tới sẽ xác định biến thể nào của hàm toán tử operatorXX sẽ được gọi tới để thực hiện. Việc con trỏ q trỏ tới đối tượng thuộc lớp nào là không quan trọng, nhưng C++ chỉ cho phép ràng buộc động đối với toán tử đầu tiên. Điều này có thể làm nảy sinh rắc rối nếu chúng ta muốn viết các toán tử đối xứng với hai tham số có cùng kiểu. Chẳng hạn giả sử chúng ta muốn xây dựng toán tử $>$ để so sánh hai đối tượng cùng thuộc kiểu Teacher (có nghĩa là có thể là thuộc lớp Teacher hoặc lớp Principal) trong các ví dụ trước. Nếu chúng ta có hai con trỏ $t1$ và $t2$:

Teacher *t1, *t2;

Chúng ta muốn rằng có thể thực hiện so sánh bằng câu lệnh sau:

if(*t1 > *t2)

....

Có nghĩa là chúng ta muốn thực hiện ràng buộc động đối với hàm toán tử $>$ sao cho có thể gọi tới các biến thể khác nhau của nó tùy thuộc vào kiểu của các đối tượng mà hai con trỏ $t1$ và $t2$ trỏ tới. Ví dụ nếu chúng cùng trỏ tới các đối tượng thuộc lớp Teacher chúng ta có thể so sánh theo hai điều kiện 1 là tên và số sinh viên quản lý, còn nếu chúng cùng trỏ tới hai đối tượng thuộc lớp Principal chúng ta sẽ so sánh thêm tiêu chí thứ 3 là tên trường.

Để có thể thực hiện điều này theo nguyên tắc của ràng buộc động chúng ta sẽ khai báo hàm toán tử > là hàm ảo ở lớp cơ sở (Teacher):

```
class Teacher{
..
public:
    virtual Bool operator > (Teacher & rhs);
};
```

Sau đó trong lớp dẫn xuất chúng ta có khai báo tiếp như sau:

```
class Principal: public Teacher{
..
public:
    Bool operator > (Principal & rhs);
};
```

Theo nguyên lý thông thường về các hàm ảo chúng ta mong muốn là toán tử > sẽ hoạt động tốt với các đối tượng thuộc lớp Principal (hay chính xác hơn là các con trỏ trỏ vào các đối tượng thuộc lớp đó). Tuy nhiên thật không may đây lại là một lỗi, định nghĩa lớp như thế sẽ không thể biên dịch được, thậm chí ngay cả trong trường hợp mà có thể biên dịch được thì chúng ta cũng không thể sinh bất cứ đối tượng nào thuộc lớp Principal vì sẽ làm xuất hiện lỗi biên dịch. Lý do là vì trình biên dịch hiểu rằng lớp Principal là một lớp trừu tượng và nguyên nhân là ở tham số trong khai báo của hàm toán tử >. Điều này là vì khai báo hàm toán tử trong lớp Principal không khớp với kiểu tham số trong khai báo của lớp cơ sở Teacher do đó hàm toán tử > của lớp cơ sở Teacher sẽ không được kế thừa hay bị ẩn đi và điều này có nghĩa là lớp Principal vẫn có một hàm toán tử > là hàm ảo thực sự do đó nó là lớp ảo.

Để sửa chữa lỗi này chúng ta sẽ thực hiện khai báo lại như sau:

```
class Principal: public Teacher{
..
public:
    Bool operator > (Teacher & rhs);
};
```

Bây giờ thì trình biên dịch không kêu ca phàn nàn gì nữa và chúng ta xem xét phần cài đặt hàm:

```
Bool Principal::operator > (Teacher & rhs){
    if(name > rhs.name)
        return True;
    else if((name == rhs.name) && (numOfStudents > rhs.numOfStudents))
        return True;
    else if((name == rhs.name) && (numOfStudents == rhs.numOfStudents) &&
(schoolName > rhs.schoolName))
        return True;
    return False;
};
```

Tuy nhiên ở đây chúng ta lại gặp phải lỗi biên dịch. Trình biên dịch sẽ kêu ca rằng thành viên schoolName không phải là một thành viên của lớp Teacher. Điều này là chính xác và để khắc phục nó chúng ta cần thực hiện một thao tác chuyển đổi kiểu (casting) cho tham số truyền vào của hàm toán tử >:

```
Bool Principal::operator > (Teacher & rhs){
```

```

Principal & r = Dynamic_cast<Principal&>(rhs);
if(name > r.name)
    return True;
else if((name == r.name) && (numOfStudents > r.numOfStudents))
    return True;
else if((name == r.name) && (numOfStudents == r.numOfStudents) &&
(schoolName > r.schoolName))
    return True;
return False;
};

```

Chúng ta cũng thực hiện hoàn toàn tương tự với các toán tử khác hay các đối tượng khác có vai trò tương tự như lớp Principal.

Chương 8: Bản mẫu (Template)

I. Các bản mẫu lớp

Khi viết các chương trình chúng ta luôn có nhu cầu sử dụng các cấu trúc có khả năng lưu trữ và xử lý một tập các đối tượng nào đó. Các đối tượng này có thể cùng kiểu – khi đó chúng ta có tập các đối tượng đồng nhất, hoặc chúng có thể có kiểu khác nhau khi đó ta có các tập đối tượng không đồng nhất hay hỗn hợp. Để xây dựng lên các cấu trúc đó chúng ta có thể sử dụng mảng hay các cấu trúc dữ liệu chẳng hạn như danh sách, hàng đợi, hoặc là cây. Một lớp có thể được dùng để xây dựng nên các collection object được gọi là một lớp chứa. Các lớp Stack, Queue hoặc Set đều là các ví dụ điển hình về lớp chứa. Vấn đề với các lớp chứa mà chúng ta đã biết này là chúng được xây dựng chỉ để chứa các đối tượng kiểu cơ bản (int, char *, ...). Như vậy nếu chúng ta muốn xây dựng một hàng đợi chẳng hạn để chứa các đối tượng thuộc lớp Person chẳng hạn thì lớp Queue này lại không thể sử dụng được, giải pháp là chúng ta lại xây dựng một lớp mới chẳng hạn là Person_Queue. Đây là một phương pháp không hiệu quả và nó đòi hỏi chúng ta phải xây dựng lại hoàn toàn các cấu trúc mới với các kiểu dữ liệu (lớp) mới.

C++ cung cấp một khả năng cho phép chúng ta không phải lặp lại công việc tạo mới này bằng cách tạo ra các lớp chung. Một bản mẫu sẽ được viết cho lớp, và trình biên dịch sẽ tự động sinh ra các lớp khác nhau cần thiết từ bản mẫu này. Lớp chứa cần sử dụng sẽ chỉ phải viết một lần duy nhất. Ví dụ nếu chúng ta có một lớp bản mẫu là List đã được xây dựng xong thì trình biên dịch có thể, bằng cách sử dụng thông tin này, sinh ra lớp List<int> và List<Person>, tương ứng là một danh sách các số nguyên và danh sách các đối tượng của lớp Person.

Chúng ta cũng có thể xây dựng lên các hàm chung. Nếu chẳng hạn chúng ta muốn viết một hàm để sắp xếp một mảng, chúng ta có thể xây dựng một bản mẫu hàm. Trình biên dịch sẽ sử dụng các thông tin này để sinh ra các hàm khác nhau có khả năng sắp xếp các mảng khác nhau.

Các bản mẫu không luôn là một phần của C++. Kế thừa được sử dụng để xây dựng các object collection. Nếu chúng ta muốn xây dựng một danh sách các đối tượng Person chúng ta sẽ trừu tượng hóa cho lớp Person là một lớp dẫn xuất của một lớp được định nghĩa trước Listable. Lớp Person sau đó sẽ kế thừa khả năng có thể là một phần của danh sách. Nhưng chúng ta sẽ có vấn đề nếu như lớp Person cần một vài thuộc tính của riêng nó. Có thể chúng ta muốn là nó sẽ chứa trong một cấu trúc cây nào đó hoặc có thể xuất các thông tin ra màn hình qua toán tử << của cout chẳng hạn. Để giải quyết các vấn đề như kiểu trên khái niệm đa kế thừa đã được đưa ra, đây là

một khái niệm có thể mang lại rất nhiều khả năng mạnh mẽ nhưng cũng làm nảy sinh rất nhiều vấn đề rắc rối. Một trong những khó khăn khi sử dụng kế thừa để xây dựng các object collection đó là chúng ta cần phải quyết định trước cách thức hoạt động và các tài nguyên của một lớp cụ thể nào đó. Ví dụ như với lớp Person chẳng hạn chúng ta cần phải biết là chúng ta sẽ sử dụng nó trong các danh sách hay các cây, có khả năng nào như thế nào... Ngoài ra các lập trình viên cần phải biết rõ chi tiết về cách thức kế thừa của các lớp để có thể biết được các thuộc tính mà nó sẽ có.

Và hóa ra việc sử dụng các lớp chung là một cách tốt, tốt hơn so với việc sử dụng kế thừa trong việc xây dựng các object collection. Viết các chương trình hướng đối tượng không phải có nghĩa là lúc nào chúng ta cũng phải sử dụng kế thừa. Điều này rất rõ ràng nếu chúng ta xem xét tình huống xây dựng một lớp Person và một lớp danh sách cơ sở, lớp Person kế thừa từ lớp danh sách cơ sở đó (như ví dụ trong chương 6 về phân kế thừa) nhưng sẽ là tự nhiên nếu chúng ta xây dựng một lớp danh sách riêng để chứa các đối tượng thuộc lớp Person. Và với khái niệm bản mẫu chúng ta có một lớp List chung, để chứa các đối tượng thuộc lớp Person chúng ta chỉ cần khai báo một đối tượng thuộc lớp List<Person>.

Khái niệm bản mẫu được đưa vào khá muộn trong ngôn ngữ C++. Có rất nhiều lớp chứa trong bản phác thảo chuẩn. Chúng đều là các lớp chung sử dụng các bản mẫu. Các lớp chứa đều không sử dụng khái niệm kế thừa. Ngoài ra còn có một tập các hàm chung trong bản phác thảo chuẩn có thể thực hiện các thuật toán trên các object collection. Chẳng hạn như các hàm tìm kiếm và các hàm sắp xếp.

Trong chương này chúng ta sẽ xem xét cách thức các bản mẫu được sử dụng để xây dựng các lớp chung và các hàm. Thật không may là cú pháp của bản mẫu không được dễ hiểu và đẹp lắm trong C++ mặc dù vậy về bản chất các bản mẫu là những thứ rất có ích.

1. Các bản mẫu và thử nghiệm

Chúng ta xem xét lớp Matrix thể hiện các ma trận không sử dụng bản mẫu như sau:

```
class Matrix{
public:
    Matrix(int i=0, int j=0):r(i), c(j), a(new double[r*c]){}
    Matrix(const Matrix &m):a(0){*this = m;}
    ~Matrix(){delete []a;}
    int num_row() {return r;} // cho biết số hàng của ma trận
    int num_col() {return c;} // cho biết số cột của ma trận
    Matrix & operator = (const Matrix &); // toán tử gán
    double & operator()(int i, int j); // toán tử chỉ số
private:
    int r,c;
    double *a;
};
```

Như chúng ta đã biết ma trận là một bảng các hàng và các cột. Mỗi phần tử riêng biệt của ma trận trong lớp Matrix trên có kiểu là double. Để biểu diễn các phần tử của 1 ma trận có r hàng và c cột chúng ta sử dụng một mảng một chiều có (r * c) phần tử và một con trỏ a để quản lý mảng một chiều này. Các phần tử của ma trận được lưu trong mảng một chiều này theo nguyên tắc hàng 1 trước, sau đó đến hàng 2 ... Lớp Matrix có hai cấu tử, 1 cấu tử với hai tham số khởi tạo hàng và cột, phần dữ liệu không

có gì, một cấu tử copy bình thường. Còn lại các phương thức khác có lẽ không cần giải thích nhiều.

```
Matrix & Matrix::operator = (const Matrix & m){
    if(this != &m){
        r = m.r;
        c = m.c;
        delete [] a;
        a = new double[r*c];
        for(int i=0; i<r*c; i++)
            a[i] = m.a[i];
    }
    return *this;
}
```

```
double & Matrix::operator () (int i, int j){
    if(i<1 || i>r || j<1 || j>c)
        return 0;
    return a[(i-1)*c + j-1];
}
```

Lớp Matrix này tất nhiên có thể hoạt động tốt nhưng chỉ với các ma trận mà các phần tử có kiểu double. Vì thế chúng ta có thể viết lại lớp Matrix để nó trở thành một bản mẫu:

```
template <class T>
class Matrix{
public:
    Matrix(int i=0, int j=0):r(i), c(j), a(new T[r*c]){ }
    Matrix(const Matrix &m):a(0){*this = m;}
    ~Matrix(){delete []a;}
    int num_row() {return r;} // cho biết số hàng của ma trận
    int num_col() {return c;} // cho biết số cột của ma trận
    Matrix & operator = (const Matrix &); // toán tử gán
    T & operator()(int i, int j); // toán tử chỉ số
private:
    int r,c;
    T *a;
};
```

Có hai thay đổi trong cài đặt của chúng ta so với lớp Matrix trước. Đầu tiên là dòng trước khai báo lớp Matrix. Từ khóa template báo cho trình biên dịch biết rằng lớp Matrix là một bản mẫu và không phải là một lớp bình thường. Các tham số bản mẫu (template parameters) hay các tham số chung được đặt giữa hai dấu < và >. Trong trường hợp này chỉ có một tham số bản mẫu và nó được đặt tên là T. Từ khóa class cũng chỉ ra rằng T là một kiểu dữ liệu bất kỳ nào đó. (chú ý là T cũng có thể là một kiểu dữ liệu built-in chẳng hạn như int hoặc char * mặc dù từ khóa class vẫn được sử dụng).

Thay đổi thứ hai là chúng ta thay mọi chỗ nào có từ khóa double bằng T. Ví dụ biến thành viên a sẽ có kiểu là T * thay vì double *.

Việc sử dụng lớp Matrix cũng hơi khác chút ít:

```
Matrix<int> mi(2,3); // khai báo một ma trận các số kiểu int có kích thước là 2 x 3
```

Matrix<Person> mp(10,3); // khai báo một ma trận các số kiểu int có kích thước là 10 x 3

Trình biên dịch đã sử dụng bản mẫu Matrix để sinh ra hai lớp thường chứa các kiểu dữ liệu khác nhau. Chúng ta gọi đó là hai thể nghiệm của lớp Matrix (có thể dùng thuật ngữ các lớp sinh hoặc là các đặt tử). Trong trường hợp thứ nhất dường như kiểu tham số T sẽ được thay bằng kiểu int ở bất kỳ vị trí nào nó xuất hiện, và trong trường hợp thứ hai là kiểu Person. Cách dễ nhất để giải thích là trình biên dịch sẽ tự động sinh ra các đoạn mã tương ứng thay thế kiểu tham số lớp và biên dịch như một lớp thường (thực tế thì công việc này phức tạp hơn nhiều nhưng có lẽ là những gì mà trình biên dịch tự mình thực hiện). Mỗi một khai báo Matrix <X>, trong đó X là một kiểu nào đó sẽ tương ứng với lớp mới được trình biên dịch sinh ra. Tên lớp này được sử dụng bình thường như các tên lớp khác và các tham số hàm cũng như các biến của lớp có thể được khai báo một cách bình thường.

Trong lớp Matrix chúng ta có hai hàm thành viên operator = và operator () được định nghĩa riêng rẽ. Do lớp Matrix là một bản mẫu nên các hàm thành viên của nó cũng là các bản mẫu.

```
template <class T>
Matrix<T> & Matrix<T>::operator = (const Matrix & m){
    if(this != &m){
        r = m.r;
        c = m.c;
        delete [] a;
        a = new T[r*c];
        for(int i=0; i< r*c; i++)
            a[i] = m.a[i];
    }
    return *this;
}
```

```
template <class T>
T & Matrix<T>::operator () (int i, int j){
    if(i<1 || i>r || j<1 || j>c)
        return 0; //return a[0];
    return a[(i-1)*c + j-1];
}
```

Chúng ta thấy rằng cả hai khai báo này đều bắt đầu bằng dòng <template class T> giống như khai báo lớp Matrix ban đầu. Vì tất cả các thể nghiệm của lớp bản mẫu đều có các thể nghiệm riêng của các hàm thành viên của nó nên chúng ta buộc phải viết là Matrix<T>:: trước tên hàm khi chúng ta muốn chỉ ra tên đầy đủ của một hàm thành viên. Nếu chúng ta muốn chỉ rõ tên của bản mẫu lớp trong định nghĩa của nó (lớp) chúng ta không cần viết cả <T> và tên lớp mà chỉ cần tên lớp là đủ. Có thể thấy rõ chú ý này với tham số của hàm toán tử = chúng ta chỉ cần viết là Matrix & chứ không cần Matrix<T>, tuy nhiên với phiên bản 3.0 chúng ta cần chỉ rõ tham số là Matrix <T> thì mới biên dịch được.

Kết luận:

```
template <class T>
class C{
    return_type f(parameters);
```

```

    ....
};

template <class T>
return_type C<T>::f(parameters){
    ...
}

```

T là một tham số kiểu và được dùng như một kiểu bình thường trong lớp.

Cả định nghĩa lớp và định nghĩa hàm của một lớp bản mẫu nên đặt trong một file header của lớp đó.

Các thử nghiệm của một lớp bản mẫu được tạo ra khi khai báo:

C<type_name> khi đó T sẽ được thay bằng type_name.

Biểu thức C<type_name> được dùng như một tên lớp bình thường.

Với một lớp bất kỳ chúng ta nên có hai file tương ứng : một file khai báo (.h) (header) và một file định nghĩa (.cpp). Chúng ta cũng được học rằng định nghĩa lớp nên đặt trong file header và định nghĩa hàm nên đặt trong file .cpp. Với các lớp bản mẫu chúng ta có đôi chút thay đổi trong khuyến cáo này . Vì trình biên dịch phải tự động sinh ra một lớp nào đó từ lớp bản mẫu nên nó phải truy cập tới toàn bộ định nghĩa và cài đặt của lớp bản mẫu . Chính vì thế chúng ta đặt tất cả cài đặt của một lớp trong file header . Điều này có nghĩa là không có file định nghĩa .cpp. Thực chất các trình biên dịch khác nhau giải quyết vấn đề này theo các cách khác nhau nhưng khuyến cáo này có thể làm việc với mọi trình biên dịch .

Giả sử chúng ta cần có một hàm có khả năng cộng hai ma trận trong trường hợp chúng chứa các số nguyên chẳng hạn . Để giải quyết vấn đề này với lớp bản mẫu Matrix chúng ta có thể cài đặt một hàm add như sau:

```

Matrix<int> add(Matrix<int> a, Matrix<int> b){
    if((a.num_row()!=b.num_row()) || (a.num_col()!=b.num_col()))
        return Matrix<int>(0);
    Matrix<int> c(a);
    for(int i=1;i<=c.num_row();i++)
        for(int j=1;j<=c.num_col();j++)
            c(i,j) += b(i,j);
    return c;
}

```

Và đây là một đoạn chương trình hoàn chỉnh đọc vào một ma trận kiểu int và nhân đôi nó.

```

int main(){
    int x, y;
    cout << "Number of rows:"; cin >> x;
    cout << "Number of columns:"; cin >> y;
    Matrix<int> mi(x,y);
    for(int i=1;i<=mi.num_row();i++)
        for(int j=1;j<=mi.num_col();j++)
            cin >> mi(i,j);
    mi = add(mi,mi);
    for(i=1;i<=mi.num_row();i++){
        for(j=1;j<=mi.num_col();j++)

```



```

        cout << setw(4) << mi(i,j) ;
        cout << endl;
    }
    return 0;
}

```

2. Các thành phần tĩnh

Cũng như các lớp thường khác một lớp bản mẫu cũng có các thành viên dữ liệu và hàm thành viên tĩnh. Chẳng hạn chúng ta có thể thêm vào lớp bản mẫu Matrix một biến thành viên tĩnh mn để kiểm soát số các ma trận có các phần tử thuộc một kiểu nào đó và một hàm thành viên tĩnh để đọc giá trị này. (Bình thường các hàm cấu tử và hủy tử sẽ thay đổi để chúng có thể đếm và thay đổi giá trị của biến thành viên tĩnh này nhưng chúng ta sẽ không xem xét cụ thể ở đây, tất cả các cấu tử được sử dụng cũng như các hủy tử).

```

template <class T>
class Matrix{
public:
    ...
    static int num_matrices();
private:
    ...
    static int mn;
};

```

Đối với các lớp thường chúng ta biết rằng các thành viên tĩnh là chung đối với các đối tượng của một lớp, nó chỉ có một bản (edition) duy nhất. Đối với các lớp bản mẫu, mỗi lớp thể nghiệm của lớp bản mẫu sẽ có một bản riêng của biến thành viên tĩnh. Chẳng hạn nếu chúng ta có 3 lớp thể nghiệm của lớp Matrix là Matrix<int>, Matrix<double>, Matrix<float> chúng ta sẽ có 3 biến tĩnh, mỗi biến cho một lớp thể nghiệm.

Các hàm thành viên tĩnh được định nghĩa tương tự như với các hàm không tĩnh ví dụ:

```

template <class T>
int Matrix<T>::num_matrices(){
    return mn;
}

```

Các biến thành viên tĩnh phải được định nghĩa bên ngoài lớp, điều này cũng đúng với các biến thành viên tĩnh của một lớp bản mẫu. Vì sẽ có một thể nghiệm của biến thành viên tĩnh cho tất cả các thể nghiệm của lớp bản mẫu nên định nghĩa của biến thành viên tĩnh cũng phải là một bản mẫu:

```

template <class T>
int Matrix<T>::mn = 0;

```

Chú ý về các thành viên tĩnh của các lớp bản mẫu:

Tồn tại dưới dạng một bản đối với mỗi thể nghiệm của lớp

Các hàm thành viên tĩnh được định nghĩa như các hàm bình thường của lớp bản mẫu.

Các biến thành viên tĩnh là các bản mẫu và phải được định nghĩa riêng biệt, ví dụ đối với một thành viên tĩnh v của lớp bản mẫu C sẽ được định nghĩa như sau:

```

template <class T>

```

type C<T>::v = giá trị khởi tạo;

Nếu có một hàm thành viên tĩnh f của một lớp thường C , thì f có thể được gọi theo 2 cách: x.f() hoặc C::f() trong đó x là một đối tượng thuộc C . Ví dụ với lớp Matrix ta có thể gọi như sau:

```
cout << "Num of int matrices:" << mi.num_matrices();
```

hoặc:

```
cout << "Num of int matrices:" << Matrix<int>::num_matrices();
```

3. Các lớp bạn và lớp trợ giúp

Nếu chúng ta đã có một lớp non-generic, nó có thể được chuyển thành một lớp bản mẫu bằng một cách tương đối dễ dàng.

Ví dụ: lớp Stack:

```
template <class T>
class Stack{
public:
    Stack():first(0){};
    ~Stack();
    void push(T d);
    T pop();
    Bool isEmpty();
    Bool isFull();
private:
    Element<T> *first;
    Stack(const Stack &){};
    Stack & operator=(const Stack &){};
};
```

Để cài đặt lớp Stack chúng ta sử dụng một lớp trợ giúp là lớp Element , lớp định nghĩa các phần tử riêng biệt trong một danh sách liên kết . Lớp Element cũng là một lớp bản mẫu vì các phần tử của danh sách sẽ chứa các phần tử dữ liệu có kiểu khác nhau đối với mỗi lớp thể nghiệm của lớp Stack . Như vậy trong khai báo của lớp Element chúng ta sẽ sử dụng một tham số bản mẫu khác chẳng hạn U , nhưng do trong khai báo lớp Stack biến first có kiểu là Element <T> nên Stack thể nghiệm tương ứng sẽ vẫn chứa dữ liệu có kiểu T , có nghĩa là kiểu dữ liệu của Stack là do nó quy ết định mặc dù Element là một lớp bản mẫu trợ giúp.

```
template <class U>
class Element{
    friend class Stack<U>;
    Element *next;
    U data;
    Element(Element *n, U d):next(n),data(d){}
};
```

Chú ý rằng lớp Stack phải điều khiển được các thành phần hữ dữ liệu trong lớp Element nên nó phải là một lớp bạn của Element . Thứ hai là tương thích về kiểu nên trong khai báo lớp Element Stack sẽ là Stack <U>, nếu không trình biên dịch sẽ không hiểu.

Bây giờ chúng ta sẽ thực hiện cài đặt các phương thức của lớp Stack:

```
template <class T>
void Stack<T>::push(T d){
```

```

    first = new Element<T>(first,d);
}

```

```

template <class T>
T Stack<T>::pop(){
    Element<T> * p = first;
    T d = p->data;
    first = first->next;
    delete p;
    return d;
}

```

```

template <class T>
Stack<T>::~~Stack(){
    while(!isEmpty()){
        Element<T> * p = first;
        first = first->next;
        delete p;
    }
}

```

```

template <class T>
Stack<T>::~~Stack(){
    while(!isEmpty()){
        Element<T> * p = first;
        first = first->next;
        delete p;
    }
}

```

Và đây là đoạn chương trình chính sử dụng lớp Stack trên:

```

int main()
{
    int n;
    cout << "Ngai muon xem dang nhi phan cua so:";
    cin >> n;
    Stack<int> stkint;
    while(n){
        stkint.push(n%2);
        n/=2;
    }
    cout << "Ket qua:";
    while(!stkint.isEmpty())
        cout << stkint.pop();
    /* cần include "mstring.h"
    Stack<String> stks;
    stks.push("ajg ajgd");
    stks.push(" a 128");
    while(!stks.isEmpty())

```

```

        cout << stks.pop() << endl;
    */
    return 0;
}

```

4. Các tham số bản mẫu

Các lớp bản mẫu mà chúng ta xem xét từ đầu tới giờ chỉ có một tham số bản mẫu nhưng trên thực tế một lớp bản mẫu hoặc một hàm bản mẫu có thể có số tham số bản mẫu bất kỳ. Chúng ta chia các tham số bản mẫu ra làm hai phần : các kiểu tham số bản mẫu và các giá trị tham số bản mẫu. Các kiểu tham số bản mẫu được chỉ ra bởi từ khóa class đứng trước và có thể là bất kỳ kiểu nào còn các giá trị tham số giống như các tham số hàm bình thường nhưng có thể không phải là kiểu dấu phẩy động.

Chúng ta sẽ xem xét ví dụ sau : cài đặt một stack bằng mảng , có hai tham số bản mẫu được dùng, ngoại trừ T còn có một tham số int để chỉ định số phần tử lớn nhất mà stack có thể chứa:

```

template <class T, int size>
class Stack{
public:
    Stack():n(0){};
    ~Stack();
    void push(const T & d){s[n++] = d};
    T pop(){return s[n--]};
    Bool isEmpty(){return (n<=0)?True:False;};
    Bool isFull();
private:
    T s[size];
};

```

Khi chúng ta tạo ra một thể nghiệm của lớp bản mẫu này chúng ta cần cung cấp các giá trị cho cả hai tham số bản mẫu:

```
Stack<int, 200> x; // ngăn xếp có tối đa 200 phần tử
```

Khi giá trị được gán cho tham số giá trị cần phải có một biểu thức hằng chỉ ra rằng có 1 giá trị có thể tính tại thời điểm biên dịch chương trình.

Chú ý: Các tham số bản mẫu

```
template <param 1, param 2, param 3, ...>
```

Trong đó param N có thể có dạng:

```
class T hoặc
```

```
type_name V
```

Một tham số giá trị có thể không phải là kiểu số dấu phẩy động

Có thể sử dụng các tham số mặc định.

Giống như các tham số hàm bình thường các tham số bản mẫu cũng có thể nhận các giá trị mặc định ví dụ chúng ta có thể chuyển khai báo Stack trên thành :

```
template <class T=double, int size = 100>
```

```
class Stack{
```

```
....
```

```
};
```

Nếu một tham số bản mẫu có giá trị mặc định thì các tham số bản mẫu sau đó cũng bắt buộc phải có giá trị mặc định. Nếu lớp bản mẫu có tham số bản mẫu mặc định ta có thể bỏ qua các tham số đó khi tạo ra các thể nghiệm lớp chẳng hạn :

Stack<Person> sp; // một ngăn xếp Person có kích thước max là 100
Stack<> sf; //ngăn xếp double có max phần tử là 100.

5. Các lớp thuộc tính

Khi chúng ta tạo ra một thể nghiệm của một lớp bản mẫu nào đó có một tham số kiểu T, kiểu T sẽ được thay thế ở tất cả các vị trí trong lớp. Việc thay thế này có thể làm việc tốt với tất cả các kiểu đối với một lớp bản mẫu nào đó nhưng không phải với tất cả các lớp bản mẫu đều như vậy. Để minh họa chúng ta sẽ lấy ví dụ về lớp bản mẫu Tree:

```
#include <iostream.h>
#include <bool.h>
template <class T>
class Node{
    friend class Tree<T>;
    T data;
    Tree<T> * left, * right;
    Node(T d):data(d), left(new Tree<T>), right(new Tree<T>){};
    ~Node(){delete left; delete right;};
};
```

```
template <class D>
class Tree{
    public:
        Tree():root(0){};
        Tree(D d){ root = new Node<D>(d);};
        Tree(const Tree & t){copy(t);};
        ~Tree(){delete root;};
        Bool empty()const{return (root==0)?True:False;};
        D & value() const{return root->data;};
        Tree & l_child() const {return *root->left;};
        Tree & r_child() const {return *root->right;};
        Tree & operator =(const Tree & t);
        Bool operator==(const Tree & t) const;
        void inorder() const;
        void put_in(D d);
        D* search(D d);
    private:
        Node<D> *root;
        void copy(const Tree & t);
};
```

```
template <class D>
void Tree<D>::copy(const Tree<D> & t){
    if(t.empty())
        root = 0;
    else{
        root = new Node<D>(t.value());
        l_child().copy(t.l_child());
        r_child().copy(t.r_child());
    }
```

```

    }
}

template <class D>
Tree<D> & Tree<D>::operator=(const Tree<D> & t){
    if(root!=t.root){
        delete root;
        copy(t);
    }
    return *this;
}

template <class D>
Bool Tree<D>::operator==(const Tree<D> & t)const{
    if((empty()&&t.empty())||
        (!empty()&&!t.empty()&&l_child()==t.l_child()&&r_child()==t.r_child()))
        return True;
    else
        return False;
}

template <class D>
void Tree<D>::inorder() const{
    if(!empty()){
        l_child().inorder();
        cout << value() << " ";
        r_child().inorder();
    }
}

int main()
{
    Tree<int> t;
    t = Tree<int>(10);
    t.l_child() = Tree<int>(20);
    t.l_child().r_child() = Tree<int>(12);
    t.r_child() = Tree<int>(30);
    t.inorder();
    return 0;
}

```

Tuy nhiên nếu chúng ta có thêm một lớp Person n chẳng hạn được khai báo như sau:

```

class Person{
private:
    char pno[10];
    char name[30];
public:
    char * getPno() const;
    char * getName() const;
}

```

```
};
```

Và trong chương trình chính chúng ta khai báo:

```
Tree<Person> tp;
```

Thì lập tức trình biên dịch sẽ báo lỗi, lý do là do trong các hàm của lớp bản mẫu Tree chúng ta đã dùng một số hàm mặc định có với kiểu int nhưng chưa có với lớp Person:

```
Person(char * PNO, char *n){strcpy(pno,PNO);strcpy(name,n);};
```

```
Bool operator>(const Person &);
```

```
Bool operator<(const Person &);
```

```
Bool operator==(const Person &);
```

```
friend ostream & operator <<(ostream & out, const Person &);
```

Sau khi cài đặt các hàm này mọi việc lại trở lại như xưa.

Ở đây điều cần chú ý là: bình thường với các kiểu dữ liệu built-in một số hàm (chẳng hạn các hàm toán tử, cấu trúc copy) mặc định có nên việc chương trình dùng chúng là không sao, nhưng với các kiểu dữ liệu do người dùng định nghĩa không có các hàm này nên cần phải cài đặt chúng nếu chương trình có dùng đến.

6. Các lớp chứa (bản mẫu) chuẩn

Trong đặc tả chuẩn của ngôn ngữ C++ có rất nhiều các lớp chứa đã được chuẩn hóa, hay được coi là chuẩn (1 phần cơ bản của ngôn ngữ). Chẳng hạn như các lớp List, Set, Queue, Stack Tất cả chúng đều là các lớp chứa và sử dụng các iterator. Tuy nhiên do thời gian và điều kiện chúng ta không thể xem xét hết các lớp chuẩn này. Nếu sinh viên nào muốn tìm hiểu thêm về các lớp chứa có thể liên hệ với giáo viên để photo tài liệu hoặc xem phần help về các lớp chứa đi kèm với bản Turbo C++ 3.0 for Dos của hãng Borland.

II. Các bản mẫu hàm

1. Các định nghĩa và thể nghiệm

Các lớp không phải là những thứ duy nhất có thể làm bản mẫu. Các hàm cũng có thể là các bản mẫu, khi đó chúng được gọi là các hàm bản mẫu.

Ví dụ 1: Trong các chương trình chúng ta thường hay phải thực hiện việc so sánh hai biến thuộc cùng một kiểu xem biến nào lớn hơn, thay vì thực hiện một câu lệnh if chúng ta có thể viết một hàm trả về phần tử nhỏ hơn ví dụ:

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

Việc dùng hàm min này không có gì đáng nói nhưng nếu chúng ta muốn sử dụng hàm min với hai biến không phải kiểu int thì rắc rối to. Và trong trường hợp đó chúng ta muốn sử dụng bản mẫu hàm:

```
template <class T>  
const T & min(const T & a, const T & b){  
    return (a<b)?a:b;  
}
```

cũng giống như các lớp bản mẫu trong các hàm bản mẫu cũng có sự thay thế kiểu khi sử dụng chúng với các tình huống cụ thể:

```
int a, y;  
long int m, n;  
...
```

```
cout << min(x,y);
cout << min(m,n);
```

tương ứng với dòng thứ nhất trình biên dịch sẽ sinh ra một thể nghiệm của hàm và thay thế T bởi kiểu int, dòng thứ hai là một thể nghiệm với T được thay bằng long int.

Một thể nghiệm của một bản mẫu hàm không nhất thiết phải được tạo ra một cách tường minh. Nó sẽ được tự động sinh ra khi có một lời gọi hàm. Trình biên dịch sẽ cố gắng khớp các tham số thực sự của hàm với các tham số hình thức và xác định kiểu mà các tham số kiểu chung sẽ nhận. Nếu chúng có thể khớp, trình biên dịch sẽ kiểm tra xem đã có một thể nghiệm nào cho các tham số thực sự chưa. Nếu chưa có thì mới sinh ra một thể nghiệm mới.

Để điều này có thể thực hiện được tất cả các tham số kiểu chung đều phải xuất hiện trong đặc tả kiểu ít nhất 1 lần của các tham số thường của hàm. Ví dụ trong hàm min T xuất hiện cả trong đặc tả kiểu cho tham số a và tham số b. Khi các thể nghiệm của các bản mẫu hàm được tạo ra sẽ không có nhiều chuyển kiểu tự động như so với các hàm bình thường mà sự khớp giữa các tham số thực sự với các tham số hình thức còn đòi hỏi ngặt nghèo hơn ví dụ việc gọi hàm min như sau:

```
cout << min(m,y) sẽ lập tức sinh lỗi.
```

Trên thực tế trong một lời gọi hàm có thể chỉ định một cách tường minh kiểu mà các tham số kiểu chung có thể nhận và cú pháp thực hiện điều này cũng giống như với các thể nghiệm của các lớp bản mẫu:

```
cout << min<double>(m, y);
```

Việc chuyển đổi kiểu sẽ được thực hiện vì min<double> là một hàm cụ thể, có nghĩa là không có thao tác kiểm tra khớp kiểu ở đây, đoạn mã tương ứng với min<double> sẽ được sinh ra.

Hàm thứ hai mà chúng ta xem xét là một hàm hoán vị giá trị cho hai biến kiểu bất kỳ:

```
template <class T>
void swap(T &a, T &b){
    T temp = a;
    a = b;
    b = temp;
}
```

Điều cần chú ý với hàm swap này là các kiểu sử dụng với chúng cần có hàm toán tử gán.

Các bản mẫu hàm đặc biệt hay được sử dụng với các thao tác hay dùng với các mảng ví dụ:

```
template <class T>
T & min_element(T f[], int n){
    int m = 0;
    for(int i=1;i<n;i++){
        if(f[i] < f[m])
            m = i;
    }
    return f[m];
}
```

Hàm sắp xếp cũng là một hàm hay dùng vì thế sẽ rất tiện lợi nếu chúng ta có một hàm có khả năng sắp xếp các mảng có kiểu bất kỳ:

```
template <class T>
void sort(T f[], int n){
```



```

    for(int i=0;i<n;i++)
        swap(f[i],min_element(&f[i], n-k);
}

```

Khi một thể nghiệm của hàm sort được tạo ra với một kiểu dữ liệu cụ thể T, các thể nghiệm của các hàm swap và min_element cũng tự động được sinh ra với cùng kiểu.

Tổng kết: các bản mẫu hàm

```

template <class T1, class T2>
return_type f(parameters)
{...}

```

trong đó T1 và T2, ... là các tham số và được sử dụng như các kiểu bình thường khác trong thân hàm. Tất cả các tham số kiểu phải xuất hiện trong đặc tả trong danh sách tham số.

Các thể nghiệm của bản mẫu hàm được gọi tới một cách tự động khi có câu lệnh: f(các tham số thực sự);

trình biên dịch sẽ khớp các tham số thực sự với các tham số hình thức và xác định kiểu nào sẽ thay thế cho T.

Chúng ta có thể chỉ định tường minh kiểu của T:

```

f<type>(các tham số thực sự);

```

Một thể nghiệm đặc biệt đối với một kiểu cụ thể V có thể được tạo ra. Thân hàm sẽ được viết lại và T sẽ được thay thế bằng V.

Cũng giống như các bản mẫu lớp chúng ta có thể tạo ra một thể nghiệm đặc biệt của bản mẫu hàm với một kiểu đặc biệt nào đó. Hàm min_element sẽ không làm việc chẳng hạn trong trường hợp chúng ta có một mảng các chuỗi, khi đó chúng ta cần có một phiên bản đặc biệt của hàm này để làm việc với các chuỗi mà các phần tử mảng có kiểu char*:

```

char * & min_element(char * f[], int n){
    int m = 0;
    for(int i=1;i<n;i++)
        if(strcmp(f[i],f[m])<0)
            m = i;
    return f[m];
}

```

Mọi vị trí của T bây giờ được thay bằng char*. Khi gặp một lời gọi tới thể nghiệm hàm có kiểu là char* thay vì sinh ra một thể nghiệm từ bản mẫu trình biên dịch sẽ trực tiếp biên dịch đoạn mã được viết riêng để làm việc với kiểu này.

Tất nhiên cũng giống như các bản mẫu lớp chúng ta có thể sử dụng nhiều tham số bản mẫu với bản mẫu hàm, đồng thời có thể dùng của các tham số bản mẫu kiểu và tham số bản mẫu giá trị.

2. Các hàm chuẩn chung – thư viện thuật toán

Cũng như các lớp bản mẫu có rất nhiều hàm bản mẫu trong đặc tả chuẩn của ngôn ngữ C++. Chúng có khả năng thực hiện một số các thao tác khác nhau trên các lớp chứa và các data collection khác. Chẳng hạn chúng ta có thể thực hiện tìm kiếm, sắp xếp và thực hiện các thay đổi dữ liệu khác. Các bản mẫu hàm này được định nghĩa trong file header algorithm.

Tất cả các hàm bản mẫu này đều sử dụng các iterator để quản lý dữ liệu.

Thư viện STL là thư viện bản mẫu chuẩn đang được sử dụng rộng rãi nhất hiện nay.

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VỚI C++

CHƯƠNG 1

GIỚI THIỆU VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

1.1 LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (OOP) LÀ GÌ ?

Lập trình hướng đối tượng (Object-Oriented Programming, viết tắt là OOP) là một phương pháp mới trên bước đường tiến hóa của việc lập trình máy tính, nhằm làm cho chương trình trở nên linh hoạt, tin cậy và dễ phát triển. Tuy nhiên để hiểu được OOP là gì, chúng ta hãy bắt đầu từ lịch sử của quá trình lập trình – xem xét OOP đã tiến hóa như thế nào.

1.1.1 Lập trình tuyến tính



Máy tính đầu tiên được lập trình bằng mã nhị phân, sử dụng các công tắc cơ khí để nạp chương trình. Cùng với sự xuất hiện của các thiết bị lưu trữ lớn và bộ nhớ máy tính có dung lượng lớn nên các ngôn ngữ lập trình cấp cao đầu tiên được đưa vào sử dụng. Thay vì phải suy nghĩ trên một dãy các bit và byte, lập trình viên có thể viết một loạt lệnh gần với tiếng Anh và sau đó chương trình dịch thành ngôn ngữ máy. Các ngôn ngữ lập trình cấp cao đầu tiên được thiết kế để lập các chương trình làm các công việc tương đối đơn giản như tính toán. Các chương trình ban đầu chủ yếu liên quan đến tính toán và không đòi hỏi gì nhiều ở ngôn ngữ lập trình. Hơn nữa phần lớn các chương trình này tương đối ngắn, thường ít hơn 100 dòng. Khi khả năng của máy tính tăng lên thì khả năng để triển khai các chương trình phức tạp hơn cũng

tăng lên. Các ngôn ngữ lập trình ngày trước không còn thích hợp đối với việc lập trình đòi hỏi cao hơn. Các phương tiện cần thiết để sử dụng lại các phần mã chương trình đã viết hầu như không có trong ngôn ngữ lập trình tuyến tính. Thật ra, một đoạn lệnh thường phải được chép lại mỗi khi chúng ta dùng trong nhiều chương trình do đó chương trình dài dòng, logic của chương trình khó hiểu. Chương trình được điều khiển để nhảy đến nhiều chỗ mà thường không có sự giải thích rõ ràng, làm thế nào để chương trình đến chỗ cần thiết hoặc tại sao như vậy.

Ngôn ngữ lập trình tuyến tính không có khả năng kiểm soát phạm vi nhìn thấy của các dữ liệu. Mọi dữ liệu trong chương trình đều là dữ liệu toàn cục nghĩa là chúng có thể bị sửa đổi ở bất kỳ phần nào của chương trình. Việc dò tìm các thay đổi không mong muốn đó của các phần tử dữ liệu trong một dãy mã lệnh dài và vòng vèo đã từng làm cho các lập trình viên rất mất thời gian.

1.1.2 Lập trình cấu trúc:

Rõ ràng là các ngôn ngữ mới với các tính năng mới cần phải được phát triển để có thể tạo ra các ứng dụng tinh vi hơn. Vào cuối các năm trong 1960 và 1970, ngôn ngữ lập trình có cấu trúc ra đời. Các chương trình có cấu trúc được tổ chức theo các công việc mà chúng thực hiện.

Về bản chất, chương trình chia nhỏ thành các chương trình con riêng rẽ (còn gọi là hàm hay thủ tục) thực hiện các công việc rời rạc trong quá trình lớn hơn, phức tạp hơn. Các hàm này được giữ càng độc lập với nhau càng nhiều càng tốt, mỗi hàm có dữ liệu và logic riêng. Thông tin được chuyển giao giữa các hàm thông qua các tham số, các hàm có thể có các biến cục bộ mà không một ai nằm bên ngoài phạm vi của hàm lại có thể truy xuất được chúng. Như vậy, các hàm có thể được xem là các chương trình con được đặt chung với nhau để xây dựng nên một ứng dụng.

Mục tiêu là làm sao cho việc triển khai các phần mềm dễ dàng hơn đối với các lập trình viên mà vẫn cải thiện được tính tin cậy và dễ bảo quản chương trình. Một chương trình có cấu trúc được hình thành bằng cách bẻ gãy các chức năng cơ bản của chương trình thành các mảnh nhỏ mà sau đó trở thành các hàm. Bằng cách cô lập các công việc vào trong các hàm, chương trình có cấu trúc có thể làm giảm khả năng của một hàm này ảnh hưởng đến một hàm khác. Việc này cũng làm cho việc tách các vấn đề trở nên dễ dàng hơn. Sự gói gọn này cho phép chúng ta có thể viết các chương trình sáng sủa hơn và giữ được điều khiển trên từng hàm. Các biến toàn cục không còn nữa và được thay thế bằng các tham số và biến cục bộ có phạm vi nhỏ hơn và dễ kiểm soát hơn. Cách tổ chức tốt hơn này nói lên rằng chúng ta có khả năng quản lý logic của cấu trúc chương trình, làm cho việc triển khai và bảo dưỡng chương trình nhanh hơn và hữu hiệu hơn và hiệu quả hơn.

Một khái niệm lớn đã được đưa ra trong lập trình có cấu trúc là **sự trừu tượng hóa (Abstraction)**.

Sự trừu tượng hóa có thể xem như khả năng quan sát một sự việc mà không cần xem xét đến các chi tiết bên trong của nó. Trong một chương trình có cấu trúc, chúng ta chỉ cần biết một hàm đã cho có thể làm được một công việc cụ thể gì là đủ. Còn làm thế nào mà công việc đó lại thực hiện được là không quan trọng, chừng nào hàm còn tin cậy được thì còn có thể dùng nó mà không cần phải biết nó thực hiện đúng đắn chức năng của mình như thế nào. Điều này gọi là **sự trừu tượng hóa theo chức năng (Functional abstraction)** và là nền tảng của lập trình có cấu trúc.

Ngày nay, các kỹ thuật thiết kế và lập trình có cấu trúc được sử dụng rãi. Gần như mọi ngôn ngữ lập trình đều có các phương tiện cần thiết để cho phép lập trình có cấu trúc. Chương trình có cấu trúc dễ viết, dễ bảo dưỡng hơn các chương trình không cấu trúc.

Sự nâng cấp như vậy cho các kiểu dữ liệu trong các ứng dụng mà các lập trình viên đang viết cũng đang tiếp tục diễn ra. Khi độ phức tạp của một chương trình tăng lên, sự phụ thuộc của nó vào các kiểu dữ liệu cơ bản mà nó xử lý cũng tăng theo. Vấn đề trở rõ ràng là cấu trúc dữ liệu trong chương trình quan trọng chẳng kém gì các phép toán thực hiện trên chúng. Điều này càng trở rõ ràng hơn khi kích thước của chương trình càng tăng. Các kiểu dữ liệu được xử lý trong nhiều hàm khác nhau bên trong một chương trình có cấu trúc. Khi có sự thay đổi trong các dữ liệu này thì cũng cần phải thực hiện cả các thay đổi ở mọi nơi có các thao tác tác động trên chúng. Đây có thể là một công việc tốn thời gian và kém hiệu quả đối với các chương trình có hàng ngàn dòng lệnh và hàng trăm hàm trở lên.

Một yếu điểm nữa của việc lập trình có cấu trúc là khi có nhiều lập trình viên làm việc theo nhóm cùng một ứng dụng nào đó. Trong một chương trình có cấu trúc, các lập trình viên được phân công viết một tập hợp các hàm và các kiểu dữ liệu. Vì có nhiều lập trình viên khác nhau quản lý các hàm riêng, có liên quan đến các kiểu dữ liệu dùng chung nên các thay đổi mà lập trình viên tạo ra trên một phần tử dữ liệu sẽ làm ảnh hưởng đến công việc của tất cả các người còn lại trong nhóm. Mặc dù trong bối cảnh làm việc theo nhóm, việc viết các chương trình có cấu trúc thì dễ dàng hơn nhưng sai sót trong việc trao đổi thông tin giữa các thành viên trong nhóm có thể dẫn tới hậu quả là mất rất nhiều thời gian để sửa chữa chương trình.

1.1.3 Sự trừu tượng hóa dữ liệu:

Sự trừu tượng hóa dữ liệu (Data abstraction) tác động trên các dữ liệu cũng tương tự như sự trừu tượng hóa theo chức năng. Khi có trừu tượng hóa dữ liệu, các cấu trúc dữ liệu và các phần tử có thể được sử dụng mà không cần bận tâm đến các chi tiết cụ thể. Chẳng hạn như các số dấu chấm động đã được trừu tượng hóa trong tất cả các ngôn ngữ lập trình, Chúng ta không cần quan tâm cách biểu diễn nhị phân chính xác nào cho số dấu chấm động khi gán một giá trị, cũng không cần biết tính bất thường của phép nhân nhị phân khi nhân các giá trị dấu chấm động. Điều quan trọng là các số dấu chấm động hoạt động đúng đắn và hiểu được.

Sự trừu tượng hóa dữ liệu giúp chúng ta không phải bận tâm về các chi tiết không cần thiết. Nếu lập trình viên phải hiểu biết về tất cả các khía cạnh của vấn đề, ở mọi lúc và về tất cả các hàm của chương trình thì chỉ ít hàm mới được viết ra, may mắn thay trừu tượng hóa theo dữ liệu đã tồn tại sẵn trong mọi ngôn ngữ lập trình đối với các dữ liệu phức tạp như số dấu chấm động. Tuy nhiên chỉ mới gần đây, người ta mới phát triển các ngôn ngữ cho phép chúng ta định nghĩa các kiểu dữ liệu trừu tượng riêng.

1.1.4 Lập trình hướng đối tượng:

Khái niệm hướng đối tượng được xây dựng trên nền tảng của khái niệm lập trình có cấu trúc và sự trừu tượng hóa dữ liệu. Sự thay đổi căn bản ở chỗ, một chương trình hướng đối tượng được thiết kế xoay quanh dữ liệu mà chúng ta có thể làm việc trên đó, hơn là theo bản thân chức năng của chương trình. Điều này hoàn toàn tự nhiên một khi chúng ta hiểu rằng mục tiêu của chương trình là xử lý dữ liệu. Suy cho cùng, công việc mà máy tính thực hiện vẫn thường được gọi là xử lý dữ liệu. Dữ liệu và thao tác liên kết với nhau ở một mức cơ bản (còn có thể gọi là mức thấp), mỗi thứ đều đòi hỏi ở thứ kia có mục tiêu cụ thể, các chương trình hướng đối tượng làm tường minh mối quan hệ này.

Lập trình hướng đối tượng liên kết cấu trúc dữ liệu với các thao tác, theo cách mà tất cả thường nghĩ về thế giới quanh mình. Chúng ta thường gán một số các hoạt động cụ thể với một loại hoạt động nào đó và đặt các giả thiết của mình trên các quan hệ đó.

Ví dụ 1.1: Chúng ta biết rằng một chiếc xe có các bánh xe, di chuyển được và có thể đổi hướng của nó bằng cách quẹo tay lái. Tương tự như thế, một cái cây là một loại thực vật có thân gỗ và lá. Một chiếc xe không phải là một cái cây, mà cái cây không phải là một chiếc xe, chúng ta có thể giả thiết rằng cái mà chúng ta có thể làm được với một chiếc xe thì không thể làm được với một cái cây.

Chẳng hạn, thật là vô nghĩa khi muốn lái một cái cây, còn chiếc xe thì lại chẳng lớn thêm được khi chúng ta tưới nước cho nó.

Lập trình hướng đối tượng cho phép chúng ta sử dụng các quá trình suy nghĩ như vậy với các khái niệm trừu tượng được sử dụng trong các chương trình máy tính. Một mẫu tin (record) nhân sự có thể được đọc ra, thay đổi và lưu trữ lại; còn số phức thì có thể được dùng trong các tính toán. Tuy vậy không thể nào lại viết một số phức vào tập tin làm mẫu tin nhân sự và ngược lại hai mẫu tin nhân sự lại không thể cộng với nhau được. Một chương trình hướng đối tượng sẽ xác định đặc điểm và hành vi cụ thể của các kiểu dữ liệu, điều đó cho phép chúng ta biết một cách chính xác rằng chúng ta có thể có được những gì ở các kiểu dữ liệu khác nhau.

Chúng ta còn có thể tạo ra các quan hệ giữa các kiểu dữ liệu tương tự nhưng khác nhau trong một

chương trình hướng đối tượng. Người ta thường tự nhiên phân loại ra mọi thứ, thường đặt mối liên hệ giữa các khái niệm mới với các khái niệm đã có, và thường có thể thực hiện suy diễn giữa chúng trên các quan hệ đó. Hãy quan niệm thế giới theo kiểu cấu trúc cây, với các mức xây dựng chi tiết hơn kế tiếp nhau cho các thế hệ sau so với các thế hệ trước. Đây là phương pháp hiệu quả để tổ chức thế giới quanh chúng ta. Các chương trình hướng đối tượng cũng làm việc theo một phương thức tương tự, trong đó chúng cho phép xây dựng các cơ cấu dữ liệu và thao tác mới dựa trên các cơ cấu có sẵn, mang theo các tính năng của các cơ cấu nền mà chúng dựa trên đó, trong khi vẫn thêm vào các tính năng mới.

Lập trình hướng đối tượng cho phép chúng ta tổ chức dữ liệu trong chương trình theo một cách tương tự như các nhà sinh học tổ chức các loại thực vật khác nhau. Theo cách nói lập trình đối tượng, xe hơi, cây cối, các số phức, các quyển sách đều được gọi là các **lớp (Class)**.

Một lớp là một bản mẫu mô tả các thông tin cấu trúc dữ liệu, lần các thao tác hợp lệ của các phần tử dữ liệu. Khi một phần tử dữ liệu được khai báo là phần tử của một lớp thì nó được gọi là một **đối tượng (Object)**. Các hàm được định nghĩa hợp lệ trong một lớp được gọi là các **phương thức (Method)** và chúng là các hàm duy nhất có thể xử lý dữ liệu của các đối tượng của lớp đó. Một **thực thể (Instance)** là một vật thể có thực bên trong bộ nhớ, thực chất đó là một đối tượng (nghĩa là một đối tượng được cấp phát vùng nhớ).

Mỗi một đối tượng có riêng cho mình một bản sao các phần tử dữ liệu của lớp còn gọi là các **biến thực thể (Instance variable)**. Các phương thức định nghĩa trong một lớp có thể được gọi bởi các đối tượng của lớp đó. Điều này được gọi là gửi một **thông điệp (Message)** cho đối tượng. Các thông điệp này phụ thuộc vào đối tượng, chỉ đối tượng nào nhận thông điệp mới phải làm việc theo thông điệp đó. Các đối tượng đều độc lập với nhau vì vậy các thay đổi trên các biến thể hiện của đối tượng này không ảnh hưởng gì trên các biến thể hiện của các đối tượng khác và việc gửi thông điệp cho một đối tượng này không ảnh hưởng gì đến các đối tượng khác.

1.2 MỘT SỐ KHÁI NIỆM MỚI TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Trong phần này, chúng ta tìm hiểu các khái niệm như sự đóng gói, tính kế thừa và tính đa hình. Đây là các khái niệm căn bản, là nền tảng tư tưởng của lập trình hướng đối tượng. Hiểu được khái niệm này, chúng ta bước đầu tiếp cận với phong cách lập trình mới, phong cách lập trình dựa vào đối tượng làm nền tảng mà trong đó quan điểm che dấu thông tin thông qua sự đóng gói là quan điểm trung tâm của vấn đề.

1.2.1 Sự đóng gói (Encapsulation)

Sự đóng gói là cơ chế ràng buộc dữ liệu và thao tác trên dữ liệu đó thành một thể thống nhất, tránh được các tác động bất ngờ từ bên ngoài. Thể thống nhất này gọi là đối tượng.

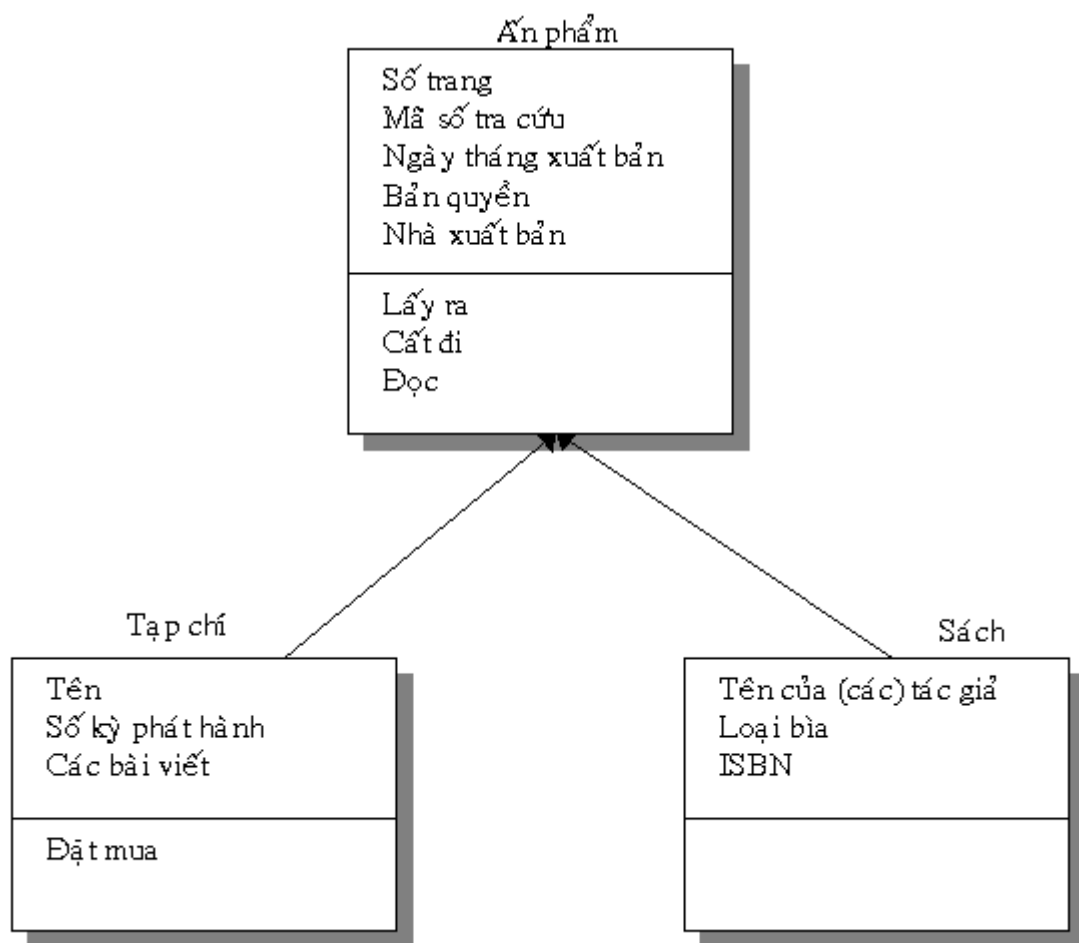
Trong một đối tượng, dữ liệu hay thao tác hay cả hai có thể là **riêng (private)** hoặc **chung (public)** của đối tượng đó. Thao tác hay dữ liệu riêng là thuộc về đối tượng đó chỉ được truy cập bởi các thành phần của đối tượng, điều này nghĩa là thao tác hay dữ liệu riêng không thể truy cập bởi các phần khác của chương trình tồn tại ngoài đối tượng. Khi thao tác hay dữ liệu là chung, các phần khác của chương trình có thể truy cập nó mặc dù nó được định nghĩa trong một đối tượng. Các thành phần chung của một đối tượng dùng để cung cấp một giao diện có điều khiển cho các thành phần riêng của đối tượng. Cơ chế đóng gói là phương thức tốt để thực hiện cơ chế che dấu thông tin so với các ngôn ngữ lập trình cấu trúc.

1.2.2 Tính kế thừa (Inheritance)

Chúng ta có thể xây dựng các lớp mới từ các lớp cũ thông qua sự kế thừa. Một lớp mới còn gọi là **lớp dẫn xuất (derived class)**, có thể thừa hưởng dữ liệu và các phương thức của **lớp cơ sở (base class)** ban đầu. Trong lớp này, có thể bổ sung các thành phần dữ liệu và các phương thức mới vào

những thành phần dữ liệu và các phương thức mà nó thừa hưởng từ lớp cơ sở. Mỗi lớp (kể cả lớp dẫn xuất) có thể có một số lượng bất kỳ các lớp dẫn xuất. Qua cơ cấu kế thừa này, dạng hình cây của các lớp được hình thành. Dạng cây của các lớp trông giống như các cây gia phả vì thế các lớp cơ sở còn được gọi là **lớp cha (parent class)** và các lớp dẫn xuất được gọi là **lớp con (child class)**.

Ví dụ 1.2: Chúng ta sẽ xây dựng một tập các lớp mô tả cho thư viện các ấn phẩm. Có hai kiểu ấn phẩm: tạp chí và sách. Chúng ta có thể tạo một ấn phẩm tổng quát bằng cách định nghĩa các thành phần dữ liệu tương ứng với số trang, mã số tra cứu, ngày tháng xuất bản, bản quyền và nhà xuất bản. Các ấn phẩm có thể được lấy ra, cất đi và đọc. Đó là các phương thức thực hiện trên một ấn phẩm. Tiếp đó chúng ta định nghĩa hai lớp dẫn xuất tên là tạp chí và sách. Tạp chí có tên, số kỳ phát hành và chứa nhiều bài của các tác giả khác nhau. Các thành phần dữ liệu tương ứng với các yếu tố này được đặt vào định nghĩa của lớp tạp chí. Tạp chí cũng cần có một phương thức nữa đó là đặt mua. Các thành phần dữ liệu xác định cho sách sẽ bao gồm tên của (các) tác giả, loại bìa (cứng hay mềm) và số hiệu ISBN của nó. Như vậy chúng ta có thể thấy, sách và tạp chí có chung các đặc trưng ấn phẩm, trong khi vẫn có các thuộc tính riêng của chúng.



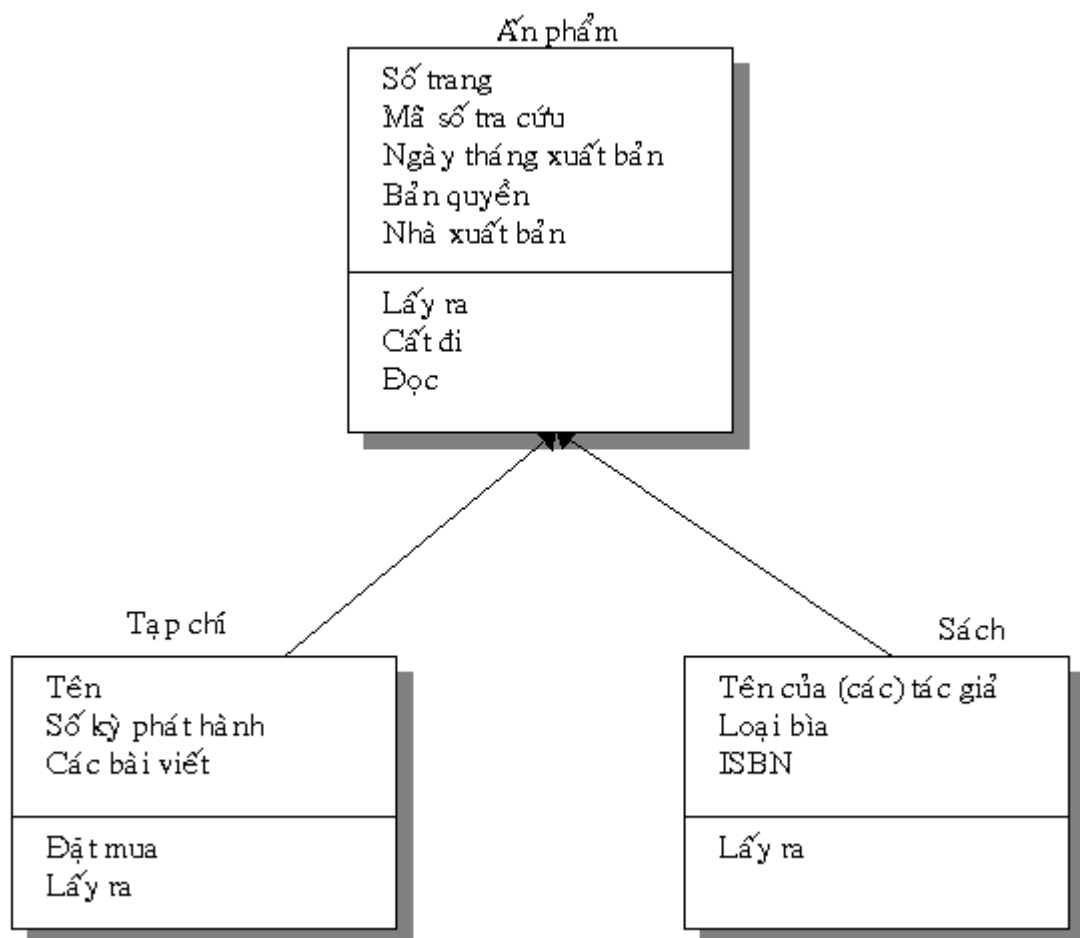
Hình 1.1: Lớp ấn phẩm và các lớp dẫn xuất của nó.

Với tính kế thừa, chúng ta không phải mất công xây dựng lại từ đầu các lớp mới, chỉ cần bổ sung để có được trong các lớp dẫn xuất các đặc trưng cần thiết.

1.2.3 Tính đa hình (Polymorphism)

Đó là khả năng để cho một thông điệp có thể thay đổi cách thực hiện của nó theo lớp cụ thể của đối tượng nhận thông điệp. Khi một lớp dẫn xuất được tạo ra, nó có thể thay đổi cách thực hiện các phương thức nào đó mà nó thừa hưởng từ lớp cơ sở của nó. Một thông điệp khi được gọi đến một đối tượng của lớp cơ sở, sẽ dùng phương thức đã định nghĩa cho nó trong lớp cơ sở. Nếu một lớp dẫn xuất định nghĩa lại một phương thức thừa hưởng từ lớp cơ sở của nó thì một thông điệp có cùng tên với phương thức này, khi được gọi tới một đối tượng của lớp dẫn xuất sẽ gọi phương thức đã định nghĩa cho lớp dẫn xuất.

Ví dụ 1.3: Xét lại ví dụ 1.2, chúng ta thấy rằng cả tạp chí và sách đều phải có khả năng lấy ra. Tuy nhiên phương pháp lấy ra cho tạp chí có khác so với phương pháp lấy ra cho sách, mặc dù kết quả cuối cùng giống nhau. Khi phải lấy ra tạp chí, thì phải sử dụng phương pháp lấy ra riêng cho tạp chí (dựa trên một bản tra cứu) nhưng khi lấy ra sách thì lại phải sử dụng phương pháp lấy ra riêng cho sách (dựa trên hệ thống phiếu lưu trữ). Tính đa hình cho phép chúng ta xác định một phương thức để lấy ra một tạp chí hay một cuốn sách. Khi lấy ra một tạp chí nó sẽ dùng phương thức lấy ra dành riêng cho tạp chí, còn khi lấy ra một cuốn sách thì nó sử dụng phương thức lấy ra tương ứng với sách. Kết quả là chỉ cần một tên phương thức duy nhất được dùng cho cả hai công việc tiến hành trên hai lớp dẫn xuất có liên quan, mặc dù việc thực hiện của phương thức đó thay đổi tùy theo từng lớp. Tính đa hình dựa trên sự nối kết (Binding), đó là quá trình gán một phương thức với một hàm thực sự. Khi các phương thức kiểu đa hình được sử dụng thì trình biên dịch chưa thể xác định hàm nào tương ứng với phương thức nào sẽ được gọi. Hàm cụ thể được gọi sẽ tùy thuộc vào việc phân tử nhận thông điệp lúc đó là thuộc lớp nào, do đó hàm được gọi chỉ xác định được vào lúc chương trình chạy. Điều này gọi là sự kết nối muộn (Late binding) hay kết nối lúc chạy (Runtime binding) vì nó xảy ra khi chương trình đang thực hiện.



Hình 1.2: Minh họa tính đa hình đối với lớp ẩn phẩm và các lớp dẫn xuất của nó.

1.3 CÁC NGÔN NGỮ VÀ VAI ỨNG DỤNG CỦA OOP

Xuất phát từ tư tưởng của ngôn ngữ SIMULA67, trung tâm nghiên cứu Palo Alto (PARC) của hãng XEROR đã tập trung 10 năm nghiên cứu để hoàn thiện ngôn ngữ OOP đầu tiên với tên gọi là Smalltalk. Sau đó các ngôn ngữ OOP lần lượt ra đời như Eiffel, CLOS, Loops, Flavors, Object Pascal, Object C, C++, Delphi, Java...

Chính XEROR trên cơ sở ngôn ngữ OOP đã đề ra tư tưởng giao diện biểu tượng trên màn hình (icon base screen interface), kể từ đó Apple Macintosh cũng như Microsoft Windows phát triển giao diện đồ họa như ngày nay. Trong Microsoft Windows, tư tưởng OOP được thể hiện một cách rõ nét nhất đó là "chúng ta click vào đối tượng", mỗi đối tượng có thể là control menu, control menu box, menu bar, scroll bar, button, minimize box, maximize box, ... sẽ đáp ứng công việc tùy theo đặc tính của đối tượng. Turbo Vision của hãng Borland là một ứng dụng OOP tuyệt vời, giúp lập trình viên không quan tâm đến chi tiết của chương trình giao diện mà chỉ cần thực hiện các nội dung chính của vấn đề.

2.1 LỊCH SỬ CỦA C++

Vào những năm đầu thập niên 1980, người dùng biết C++ với tên gọi "C with Classes" được mô tả trong hai bài báo của Bjarne Stroustrup (thuộc AT&T Bell Laboratories) với nhan đề "Classes: An Abstract Data Type Facility for the C Language" và "Adding Classes to C : An Exercise in Language Evolution". Trong công trình này, tác giả đã đề xuất khái niệm lớp, bổ sung việc kiểm tra kiểu tham số của hàm, các chuyển đổi kiểu và một số mở rộng khác vào ngôn ngữ C. Bjarne Stroustrup nghiên cứu mở rộng ngôn ngữ C nhằm đạt đến một ngôn ngữ mô phỏng (simulation language) với những tính năng hướng đối tượng. Trong năm 1983, 1984, ngôn ngữ "C with Classes" được thiết kế lại, mở rộng hơn rồi một trình biên dịch ra đời. Và chính từ đó, xuất hiện tên gọi "C++". Bjarne Stroustrup mô tả ngôn ngữ C++ lần đầu tiên trong bài báo có nhan đề "Data Abstraction in C". Sau một vài hiệu chỉnh C++ được công bố rộng rãi trong quyển "The C++ Programming Language" của Bjarne Stroustrup xuất hiện đánh dấu sự hiện diện thực sự của C++, người lập trình chuyên nghiệp từ đây đã có một ngôn ngữ đủ mạnh cho các dự án thực tiễn của mình. Về thực chất C++ giống như C nhưng bổ sung thêm một số mở rộng quan trọng, đặc biệt là ý tưởng về đối tượng, lập trình định hướng đối tượng. Thật ra các ý tưởng về cấu trúc trong C++ đã xuất phát vào các năm 1970 từ Simula 70 và Algol 68. Các ngôn ngữ này đã đưa ra các khái niệm về lớp và đơn thể. Ada là một ngôn ngữ phát triển từ đó, nhưng C++ đã khẳng định vai trò thực sự của mình.

2.2 CÁC MỞ RỘNG CỦA C++

2.2.1 Các từ khóa mới của C++

Để bổ sung các tính năng mới vào C, một số từ khóa (keyword) mới đã được đưa vào C++ ngoài các từ khóa có trong C. Các chương trình bằng C nào sử dụng các tên trùng với các từ khóa cần phải thay đổi trước khi chương trình được dịch lại bằng C++. Các từ khóa mới này là :

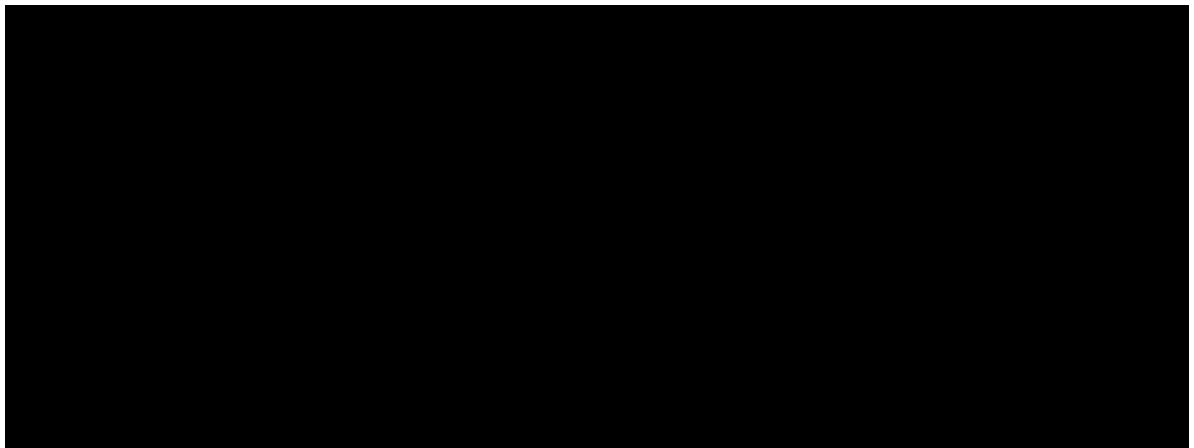
asm	catch	class	delete	friend	inline
new	operator	private	protected	public	template
this	throw	try	virtual		

2.2.2 Cách ghi chú thích

C++ chấp nhận hai kiểu chú thích. Các lập trình viên bằng C đã quen với cách chú thích bằng /*...*/. Trình biên dịch sẽ bỏ qua mọi thứ nằm giữa /*...*/. [Ví dụ 2.1](#): Trong chương trình sau :

```
CT2_1.CPP
1: /*
2: Chương trình in các số từ 0 đến 9.
3: */
4: #include <iostream.h>
5: int main()
6: {
7:     int I;
8:     for(I = 0; I < 10 ; ++ I) // 0 - 9
9:         cout<<I<<"\n"; // In ra 0 - 9
10:    return 0;
11: }
```

Mọi thứ nằm giữa /*...*/ từ dòng 1 đến dòng 3 đều được chương trình bỏ qua. Chương trình này còn minh họa cách chú thích thứ hai. Đó là cách chú thích bắt đầu bằng // ở dòng 8 và dòng 9. Chúng ta [chạy ví dụ 2.1](#), kết quả ở hình 2.1.



Hình 2.1: Kết quả của ví dụ 2.1

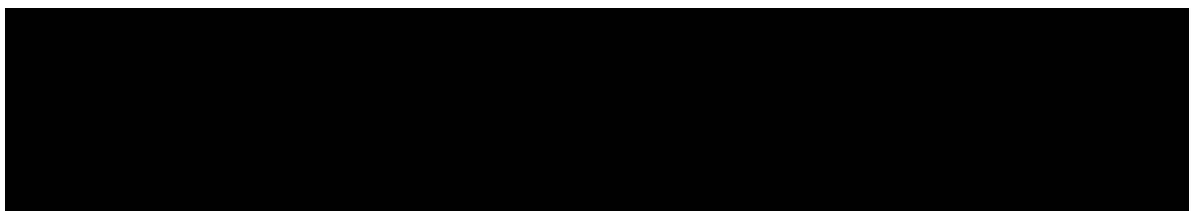
Nói chung, kiểu chú thích /*...*/ được dùng cho các khối chú thích lớn gồm nhiều dòng, còn kiểu // được dùng cho các chú thích một dòng.

2.2.3 Dòng nhập/xuất chuẩn

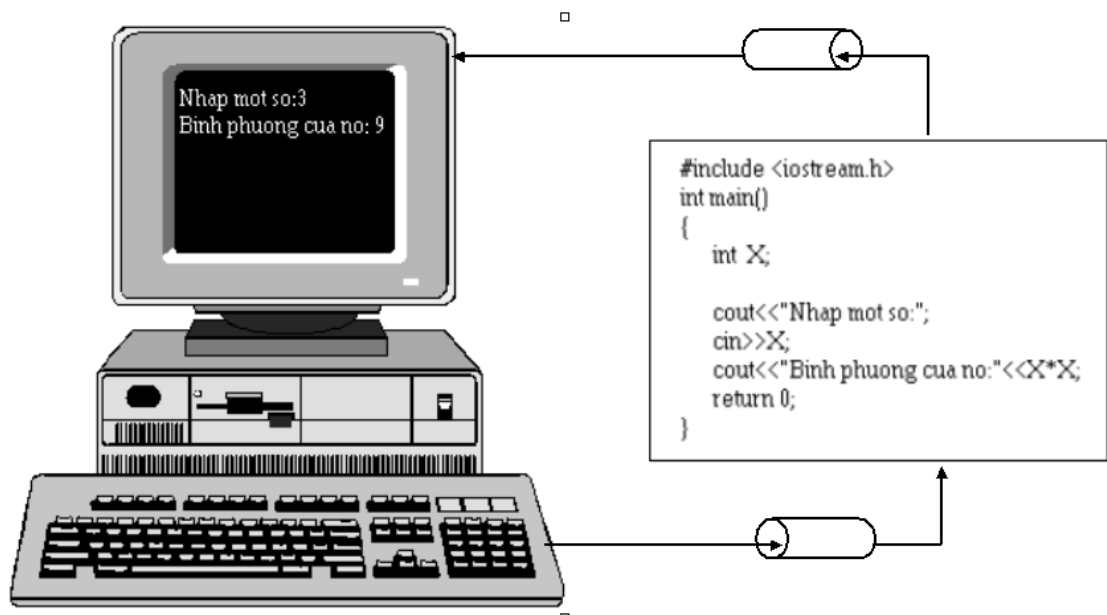
Trong chương trình C, chúng ta thường sử dụng các hàm nhập/xuất dữ liệu là `printf()` và `scanf()`. Trong C++ chúng ta có thể dùng dòng nhập/xuất chuẩn (standard input/output stream) để nhập/xuất dữ liệu thông qua hai biến đối tượng của dòng (stream object) là **cout** và **cin**.
Ví dụ 2.2: Chương trình nhập vào hai số. Tính tổng và hiệu của hai số vừa nhập.

```
CT2_2.CPP
1: #include <iostream.h>
2: int main()
3: {
4:     int X, Y;
5:     cout<< "Nhap vao mot so X:";
6:     cin>>X;
7:     cout<< "Nhap vao mot so Y:";
8:     cin>>Y;
9:     cout<<"Tong cua chung:"<<X+Y<<"\n";
10:    cout<<"Hieu cua chung:"<<X-Y<<"\n";
11:    return 0;
12: }
```

Để thực hiện dòng xuất chúng ta sử dụng biến **cout** (console output) kết hợp với toán tử chèn (insertion operator) `<<` như ở các dòng 5, 7, 9 và 10. Còn dòng nhập chúng ta sử dụng biến **cin** (console input) kết hợp với toán tử trích (extraction operator) `>>` như ở các dòng 6 và 8. Khi sử dụng **cout** hay **cin**, chúng ta phải kéo file `iostream.h` như dòng 1. Chúng ta sẽ tìm hiểu kỹ về dòng nhập/xuất ở chương 8. Chúng ta chạy ví dụ 2.2, kết quả ở hình 2.2.



Hình 2.2: Kết quả của ví dụ 2.2



Hình 2.3: Dòng nhập/xuất dữ liệu

2.2.4 Cách chuyển đổi kiểu dữ liệu

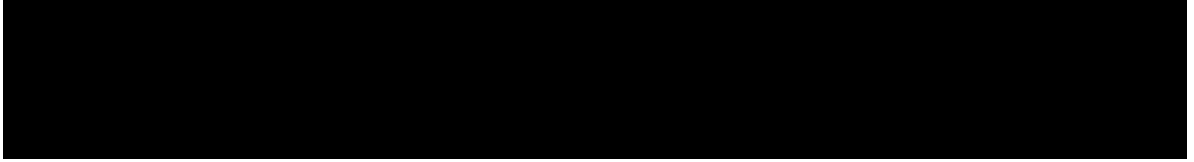
Hình thức chuyển đổi kiểu trong C tương đối tối nghĩa, vì vậy C++ trang bị thêm một cách chuyển đổi kiểu giống như một lệnh gọi hàm. [Ví dụ 2.3:](#)

```

1: #include <iostream.h>
2: int main()
3: {
4:     int X = 200;
5:     long Y = (long) X; //Chuyển đổi kiểu theo cách của C
6:     long Z = long(X); // Chuyển đ ổi kiểu theo cách mới của C++
7:     cout<< "X = "<<X<<"\n";
8:     cout<< "Y = "<<Y<<"\n";
9:     cout<< "Z = "<<Z<<"\n";
10:    return 0;
11: }

```

Chúng ta [chạy ví dụ 2.3](#) , kết quả ở hình 2.4.



Hình 2.4: Kết quả của ví dụ 2.3

2.2.5 Vị trí khai báo biến

Trong chương trình C đòi hỏi tất cả các khai báo bên trong một phạm vi cho trước phải được đặt ở ngay đầu của phạm vi đó. Điều này có nghĩa là tất cả các khai báo toàn cục phải đặt trước tất cả các hàm và các khai báo cục bộ phải được tiến hành trước tất cả các lệnh thực hiện. Ngược lại C++ cho phép chúng ta khai báo linh hoạt bất kỳ vị trí nào trong một phạm vi cho trước (không nhất thiết phải ngay đầu của phạm vi), chúng ta xen kẽ việc khai báo dữ liệu với các câu lệnh thực hiện.

Ví dụ 2.4: Chương trình mô phỏng một máy tính đơn giản



```
1: #include <iostream.h>
2: int main()
3: {
4:     int X;
5:     cout<< "Nhap vao so thu nhat:";
6:     cin>>X;
7:     int Y;
8:     cout<< "Nhap vao so thu hai:";
9:     cin>>Y;
10:    char Op;
11:    cout<<"Nhap vao toan tu (+-*/):";
12:    cin>>Op;
13:    switch(Op)
14:    {
15:        case '+':
16:            cout<<"Ket qua:"<<X+Y<<"\n";
17:            break;
18:        case '-':
19:            cout<<"Ket qua:"<<X-Y<<"\n";
20:            break;
21:        case '*':
22:            cout<<"Ket qua:"<<long(X)*Y<<"\n";
23:            break;
24:        case '/':
25:            if (Y)
26:                cout<<"Ket qua:"<<float(X)/Y<<"\n";
27:            else
28:                cout<<"Khong the chia duoc!" <<"\n"; 9; 9;
29:            break;
```

Trong chương trình chúng ta xen kê khai báo biến với lệnh thực hiện ở dòng 4 đến dòng 12. Chúng ta [chạy ví dụ 2.4](#), kết quả ở hình 2.5.



Hình 2.5: Kết quả của ví dụ 2.4

Khi khai báo một biến trong chương trình, biến đó sẽ có hiệu lực trong phạm vi của chương trình đó kể từ vị trí nó xuất hiện. Vì vậy chúng ta không thể sử dụng một biến được khai báo bên dưới nó.

2.2.6 Các biến **const**

Trong ANSI C, muốn định nghĩa một hằng có kiểu nhất định thì chúng ta dùng biến **const** (vì nếu dùng **#define** thì tạo ra các hằng không có chứa thông tin về kiểu). Trong C++, các biến **const** linh hoạt hơn một cách đáng kể:

C++ xem **const** cũng như **#define** nếu như chúng ta muốn dùng hằng có tên trong chương trình. Chính vì vậy chúng ta có thể dùng **const** để quy định kích thước của một mảng như đoạn mã sau:

```
const int ArraySize = 100;  
int X[ArraySize];
```

Khi khai báo một biến **const** trong C++ thì chúng ta phải khởi tạo một giá trị ban đầu nhưng đối với ANSI C thì không nhất thiết phải làm như vậy (vì trình biên dịch ANSI C tự động gán trị zero cho biến **const** nếu chúng ta không khởi tạo giá trị ban đầu cho nó).

Phạm vi của các biến **const** giữa ANSI C và C++ khác nhau. Trong ANSI C, các biến **const** được khai báo ở bên ngoài mọi hàm thì chúng có phạm vi toàn cục, điều này nghĩa là chúng có thể nhìn thấy cả ở bên ngoài file mà chúng được định nghĩa, trừ khi chúng được khai báo là **static**. Nhưng trong C++, các biến **const** được hiểu mặc định là **static**.

2.2.7 Về **struct**, **union** và **enum**

Trong C++, các **struct** và **union** thực sự các kiểu **class**. Tuy nhiên có sự thay đổi đối với C++. Đó là tên của **struct** và **union** được xem luôn là tên kiểu giống như khai báo bằng lệnh **typedef** vậy. Trong C, chúng ta có thể có đoạn mã sau :

```
struct Complex  
{  
    float Real;  
    float Imaginary;  
};
```



```
.....  
struct Complex C;
```

Trong C++, vấn đề trở nên đơn giản hơn:

```
struct Complex  
{  
    float Real;  
    float Imaginary;  
};  
.....  
Complex C;
```

Quy định này cũng áp dụng cho cả **union** và **enum**. Tuy nhiên để tương thích với C, C++ vẫn chấp nhận cú pháp cũ. Một kiểu **union** đặc biệt được thêm vào C++ gọi là union nặc danh (anonymous union). Nó chỉ khai báo một loạt các trường(field) dùng chung một vùng địa chỉ bộ nhớ. Một union nặc danh không có tên tag, các trường có thể được truy xuất trực tiếp bằng tên của chúng. Chẳng hạn như đoạn mã sau:

```
union  
{  
    int Num;  
    float Value;  
};
```

Cả hai Num và Value đều dùng chung một vị trí và không gian bộ nhớ. Tuy nhiên không giống như kiểu **union** có tên, các trường của union nặc danh thì được truy xuất trực tiếp, chẳng hạn như sau:

```
Num = 12;  
Value = 30.56;
```

2.2.8 Toán tử định phạm vi

Toán tử định phạm vi (scope resolution operator) ký hiệu là ::, nó được dùng truy xuất một phần tử bị che bởi phạm vi hiện thời.

[Ví dụ 2.5 :](#)

```

1: #include <iostream.h>

2: int X = 5;

3: int main()

4: {

5:     int X = 16;

6:     cout<< "Bien X ben trong = "<<X<<"\n";

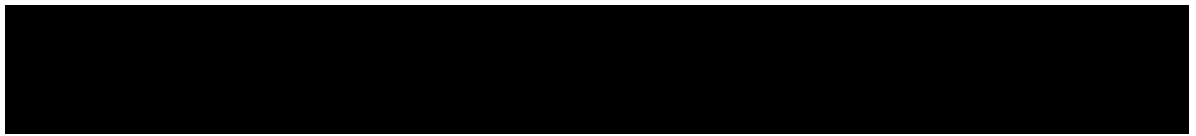
7:     cout<< "Bien X ben ngoai = "<<::X<<"\n";

8:     return 0;

9: }

```

Chúng ta [chạy ví dụ 2.5](#), kết quả ở hình 2.6



Hình 2.6: Kết quả của ví dụ 2.5

Toán tử định phạm vi còn được dùng trong các định nghĩa hàm của các phương thức trong các lớp, để khai báo lớp chủ của các phương thức đang được định nghĩa đó. Toán tử định phạm vi còn có thể được dùng để phân biệt các thành phần trùng tên của các lớp cơ sở khác nhau.

2.2.9 Toán tử new và delete

Trong các chương trình C, tất cả các cấp phát động bộ nhớ đều được xử lý thông qua các hàm thư viện như **malloc()**, **calloc()** và **free()**. C++ định nghĩa một phương thức mới để thực hiện việc cấp phát động bộ nhớ bằng cách dùng hai toán tử **new** và **delete**. Sử dụng hai toán tử này sẽ linh hoạt hơn rất nhiều so với các hàm thư viện của C. Đoạn chương trình sau dùng để cấp phát vùng nhớ động theo lối cổ điển của C.

```

int *P;

P = malloc(sizeof(int));

if (P==NULL)

    printf("Khong con du bo nho de cap phat\n");

else

```

```

{

    *P = 290;

    printf("%d\n", *P);

    free(P);

}

```

Trong C++, chúng ta có thể viết lại đoạn chương trình trên như sau:

```

int *P;

P = new int;

if (P==NULL)

    cout<<"Khong con du bo nho de cap phat\n";

else

{

    *P = 290;

    cout<<*P<<"\n";

    delete P;

}

```

Chúng ta nhận thấy rằng, cách viết của C++ sáng sủa và dễ sử dụng hơn nhiều. Toán tử **new** thay thế cho hàm **malloc()** hay **calloc()** của C có cú pháp như sau :

```

new type_name

new ( type_name )

new type_name initializer

new ( type_name ) initializer

```

Trong đó :

type_name: Mô tả kiểu dữ liệu được cấp phát. Nếu kiểu dữ liệu mô tả phức tạp, nó có thể được đặt bên trong các dấu ngoặc.

initializer: Giá trị khởi động của vùng nhớ được cấp phát.

Nếu toán tử **new** cấp phát không thành công thì nó sẽ trả về giá trị **NULL**.

Còn toán tử **delete** thay thế hàm **free()** của C, nó có cú pháp như sau :

delete *pointer*

delete [] *pointer*

Chúng ta có thể vừa cấp phát vừa khởi động như sau :

```
int *P;

P = new int(100);

if (P!=NULL)
{
    cout<<*P<<"\n";
    delete P;
}

else

    cout<<"Khong con du bo nho de cap phat\n";
```

Để cấp phát một mảng, chúng ta làm như sau :

```
int *P;

P = new int[10]; //Cấp phát mảng 10 số nguyên

if (P!=NULL)
{
    for(int I = 0;I<10;++I)

        P[I]= I;

        for(I = 0;I<10;++I)

            cout<<P[I]<<"\n";

    delete []P;
}
```

```
else
```

```
cout<<"Khong con du bo nho de cap phat\n";
```

Chú ý: Đối với việc cấp phát mảng chúng ta không thể vừa cấp phát vừa khởi động giá trị cho chúng, chẳng hạn đoạn chương trình sau là sai :

```
int *P;
```

```
P = new (int[10])(3); //Sai !!!
```

Ví dụ 2.6: Chương trình tạo một mảng động, khởi động mảng này với các giá trị ngẫu nhiên và sắp xếp chúng.



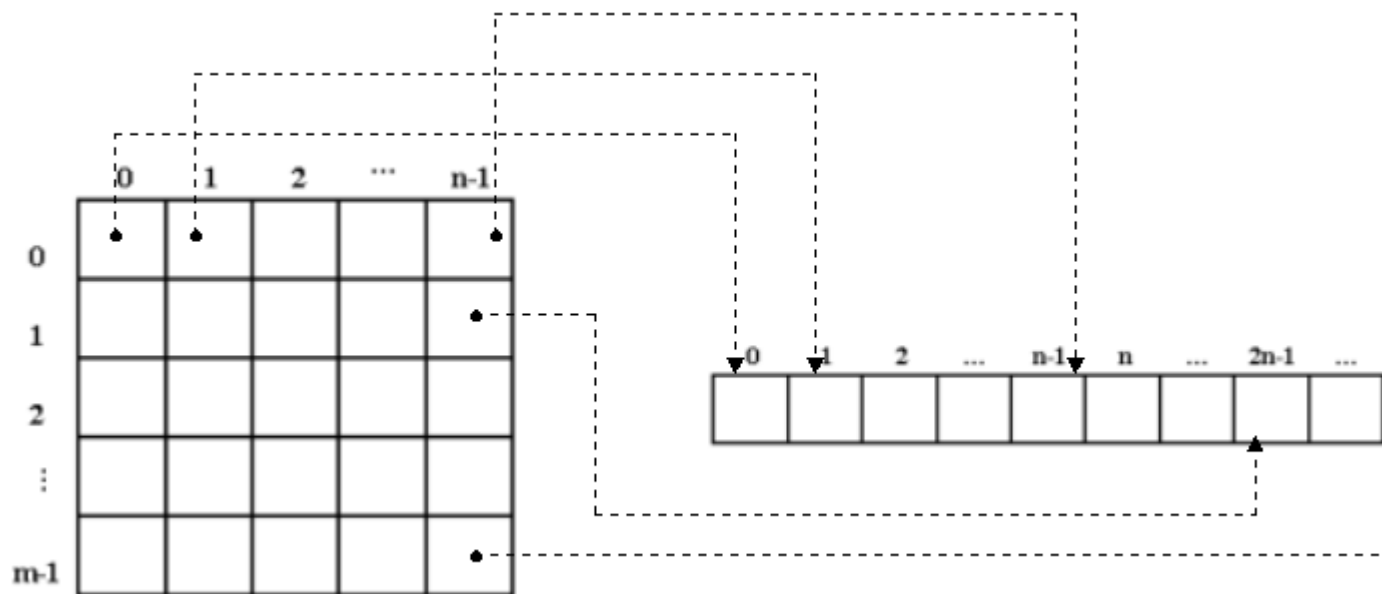
```
1: #include <iostream.h>
2: #include <time.h>
3: #include <stdlib.h>
4: int main()
5: {
6:     int N;
7:     cout<<"Nhap vao so phan tu cua mang:";
8:     cin>>N;
9:     int *P=new int[N];
10:    if (P==NULL)
11:    {
12:        cout<<"Khong con bo nho de cap phat\n";
13:        return 1;
14:    }
15:    srand((unsigned)time(NULL));
16:    for(int I=0;I<N;++I)
17:        P[I]=rand()%100; //Tạo các số ngẫu nhiên từ 0 đến 99
18:    cout<<"Mang truoc khi sap xep\n";
19:    for(I=0;I<N;++I)
20:        cout<<P[I]<<" ";
21:    for(I=0;I<N-1;++I)
22:        for(int J=I+1;J<N;++J)
23:            if (P[I]>P[J])
24:            {
25:                int Temp=P[I];
26:                P[I]=P[J];
27:                P[J]=Temp;
28:            }
29:    cout<<"\nMang sau khi sap xep\n";
```

Chúng ta [chạy ví dụ 2.6](#), kết quả ở hình 2.7



Hình 2.7: Kết quả của ví dụ 2.6

[Ví dụ 2.7](#): Chương trình cộng hai ma trận trong đó mỗi ma trận được cấp phát động. Chúng ta có thể xem mảng hai chiều như mảng một chiều như hình 2.8



Hình 2.8: Mảng hai chiều có thể xem như mảng một chiều.

Gọi X là mảng hai chiều có kích thước m dòng và n cột. A là mảng một chiều tương ứng. Nếu $X[i][j]$ chính là $A[k]$ thì $k = i*n + j$

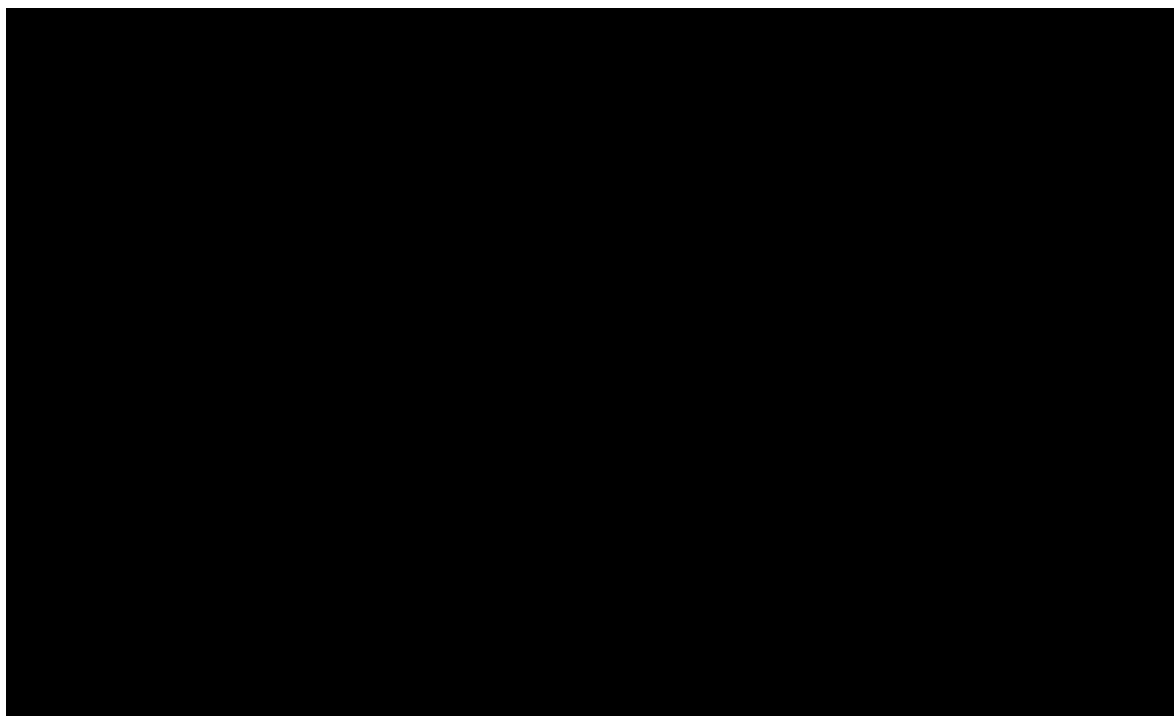
Chúng ta có chương trình như sau :

```

1: #include <iostream.h>
2: #include <conio.h>
3: //prototype
4: void AddMatrix(int * A,int *B,int*C,int M,int N);
5: int AllocMatrix(int **A,int M,int N);
6: void FreeMatrix(int *A);
7: void InputMatrix(int *A,int M,int N,char Symbol);
8: void DisplayMatrix(int *A,int M,int N);
9:
10: int main()
11: {
12:     int M,N;
13:     int *A = NULL,*B = NULL,*C = NULL;
14:
15:     clrscr();
16:     cout<<"Nhap so dong cua ma tran:";
17:     cin>>M;
18:     cout<<"Nhap so cot cua ma tran:";
19:     cin>>N;
20:     //Cấp phát vùng nhớ cho ma trận A
21:     if (!AllocMatrix(&A,M,N))
22:     { //endl: Xuất ra kí tự xuống dòng ('\n')
23:         cout<<"Khong con du bo nho!"<<endl;
24:         return 1;
25:     }
26:     //Cấp phát vùng nhớ cho ma trận B
27:     if (!AllocMatrix(&B,M,N))
28:     {
29:         cout<<"Khong con du bo nho!"<<endl;

```


Chúng ta [chạy ví dụ 2.7](#) , kết quả ở hình 2.9



Hình 2.9: Kết quả của ví dụ 2.7

Một cách khác để cấp phát mảng hai chiều A gồm M dòng và N cột như sau:

```
int ** A = new int *[M];

int * Tmp = new int[M*N];

for(int I=0;I<M;++I)
{
    A[I]=Tmp;
    Tmp+=N;
}

//Thao tác trên mảng hai chiều A

.....

delete [] *A;

delete [] A;
```

Toán tử **new** còn có một thuận lợi khác, đó là tất cả các lỗi cấp phát động đều có thể bắt được bằng một hàm xử lý lỗi do người dùng tự định nghĩa. C++ có định nghĩa một con trỏ (pointer) trỏ đến hàm đặc biệt. Khi toán tử **new** được sử dụng để cấp phát động và một lỗi xảy ra do cấp phát, C++ tự gọi đến hàm được chỉ bởi con trỏ này. Định nghĩa của con trỏ này như sau:

```
typedef void (*pvf)();
```

```
pvf _new_handler(pvf p);
```

Điều này có nghĩa là con trỏ **_new_handler** là con trỏ trỏ đến hàm không có tham số và không trả về giá trị. Sau khi chúng ta định nghĩa hàm như vậy và gán địa chỉ của nó cho **_new_handler** chúng ta có thể bắt được tất cả các lỗi do cấp phát động.

[Ví dụ 2.8:](#)



```
1: #include <iostream.h>
2: #include <stdlib.h>
3: #include <new.h>
4:
5: void MyHandler();
6:
7: unsigned long I = 0; 9;
8: void main()
9: {
10:     int *A;
11:     _new_handler = MyHandler;
12:     for( ; ; ++I)
13:         A = new int;
14:
15: }
16:
17: void MyHandler()
18: {
19:     cout<<"Lan cap phat thu "<<I<<endl;
20:     cout<<"Khong con du bo nho!"<<endl;
21:     exit(1);
22: }
```

Sử dụng con trỏ **new_handler** chúng ta phải include file new.h như ở dòng 3. Chúng ta [chạy ví dụ 2.8](#), kết quả ở hình 2.10.

Hình 2.10: Kết quả của ví dụ 2.8

Thư viện cũng còn có một hàm được định nghĩa trong new.h là hàm có prototype sau :

```
void ( * set_new_handler(void (* my_handler)() ))();
```

Hàm **set_new_handler()** dùng để gán một hàm cho **_new_handler**.

[Ví dụ 2.9:](#)



```
1: #include <iostream.h>
2: #include <new.h>
3: #include <stdlib.h>
4:
5: void MyHandler();
6:
7: int main(void)
8: {
9:
10:     char *Ptr;
11:
12:     set_new_handler(MyHandler);
13:     Ptr = new char[64000u];
14:     set_new_handler(0); //Thiết lập lại giá trị mặc định
15:     return 0;
16: }
17:
18: void MyHandler()
19: {
20:     cout <<endl<<"Khong con du bo nho";
21:     exit(1);
22: }
```

Chúng ta [chạy ví dụ 2.9](#), kết quả ở hình 2.11



Hình 2.11: Kết quả của ví dụ 2.9

2.2.10 Hàm inline

Một chương trình có cấu trúc tốt sử dụng các hàm để chia chương trình thành các đơn vị độc lập có logic riêng. Tuy nhiên, các hàm thường phải chứa một loạt các xử lý điểm vào (entry point): tham số phải được đẩy vào stack, một lệnh gọi phải được thực hiện và sau đó việc quay trở về cũng phải được thực hiện bằng cách giải phóng các tham số ra khỏi stack. Khi các xử lý điểm vào chậm chạp thường các lập trình viên C phải sử dụng cách chép lại các đoạn chương trình nếu muốn tăng hiệu quả. Để tránh khỏi phải xử lý điểm vào, C++ trang bị thêm từ khóa **inline** để loại việc gọi hàm. Khi đó trình biên dịch sẽ không biên dịch hàm này như một đoạn chương trình riêng biệt mà nó sẽ được chèn thẳng vào các chỗ mà hàm này được gọi. Điều này làm giảm việc xử lý điểm vào mà vẫn cho phép một chương trình được tổ chức dưới dạng có cấu trúc. Cú pháp của hàm **inline** như sau :

inline *data_type* *function_name* (*parameters*)

```
{  
  
.....  
  
}
```

Trong đó:

data_type: Kiểu trả về của hàm.

Function_name: Tên của hàm.

Parameters: Các tham số của hàm.

Ví dụ 2.10: Tính thể tích của hình lập phương

```

1: #include <iostream.h>

2: inline float Cube(float S)

3: {

4:     return S*S*S;

5: }

6:

7: int main()

8: {

9:     cout<<"Nhap vao chieu dai canh cua hinh lap phuong:";

10:    float Side;

11:    cin>>Side;

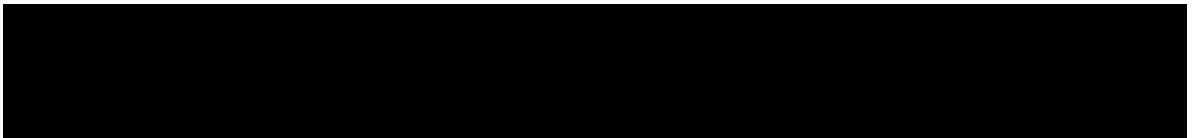
12:    cout<<"The tich cua hinh lap phuong = "<<Cube(Side);

13:    return 0;

14: }

```

Chúng ta [chạy ví dụ 2.10](#), kết quả ở hình 2.12



Hình 2.12: Kết quả của ví dụ 2.10

Chú ý: Sử dụng hàm **inline** sẽ làm cho chương trình lớn lên vì trình biên dịch chèn đoạn chương trình vào các chỗ mà hàm này được gọi. Do đó thường các hàm **inline** thường là các hàm nhỏ, ít phức tạp.

Các hàm **inline** phải được định nghĩa trước khi sử dụng. Ở ví dụ 2.10 chúng ta sửa lại như sau thì chương trình sẽ bị báo lỗi:

[Redacted code block]

```

1: #include <iostream.h>

2: float Cube(float S);

3: int main()

4: {

5:     cout<<"Nhap vao chieu dai canh cua hinh lap phuong:";

6:     float Side;

7:     cin>>Side;

8:     cout<<"The tich cua hinh lap phuong = "<<Cube(Side);

9:     return 0;

10: }

11:

12: inline float Cube(float S)

13: {

14:     return S*S*S;

15: }

```

Các hàm đệ quy không được là hàm **inline**.

2.2.11 Các giá trị tham số mặc định

Một trong các đặc tính nổi bật nhất của C++ là khả năng định nghĩa các giá trị tham số mặc định cho các hàm. Bình thường khi gọi một hàm, chúng ta cần gọi một giá trị cho mỗi tham số đã được định nghĩa trong hàm đó, chẳng hạn chúng ta có đoạn chương trình sau:

```
void MyDelay(long Loops); //prototype
```

.....

```
void MyDelay(long Loops)
```

```
{
```

```
for(int I = 0; I < Loops; ++I)
```

```
;
```



```
}
```

Mỗi khi hàm *MyDelay()* được gọi chúng ta phải gởi cho nó một giá trị cho tham số *Loops*. Tuy nhiên, trong nhiều trường hợp chúng ta có thể nhận thấy rằng chúng ta luôn luôn gọi hàm *MyDelay()* với cùng một giá trị *Loops* nào đó. Muốn vậy chúng ta sẽ dùng giá trị mặc định cho tham số *Loops*, giả sử chúng ta muốn giá trị mặc định cho tham số *Loops* là 1000. Khi đó đoạn mã trên được viết lại như sau :

```
void MyDelay(long Loops = 1000); //prototype
```

```
.....
```

```
void MyDelay(long Loops)
```

```
{
```

```
for(int I = 0; I < Loops; ++I)
```

```
;
```

```
}
```

Mỗi khi gọi hàm *MyDelay()* mà không gởi một tham số tương ứng thì trình biên dịch sẽ tự động gán cho tham số *Loops* giá trị 1000.

```
MyDelay(); // Loops có giá trị là 1000
```

```
MyDelay(5000); // Loops có giá trị là 5000
```

Giá trị mặc định cho tham số có thể là một hằng, một hàm, một biến hay một biểu thức.

Ví dụ 2.11: Tính thể tích của hình hộp



```

1: #include <iostream.h>

2: int BoxVolume(int Length = 1, int Width = 1, int Height = 1);

3:

4: int main()

5: {

6:     cout << "The tích hình hộp mac dinh: "

7:         << BoxVolume() << endl << endl

8:         << "The tích hình hộp voi chieu dai=10,do rong=1,chieu cao=1:"

9:         << BoxVolume(10) << endl << endl

10:        << "The tích hình hộp voi chieu dai=10,do rong=5,chieu cao=1:"

11:        << BoxVolume(10, 5) << endl << endl

12:        << "The tích hình hộp voi chieu dai=10,do rong=5,chieu cao=2:"

13:        << BoxVolume(10, 5, 2)<< endl;

14:     return 0;

15: }

16: //Tính thể tích của hình hộp

17: int BoxVolume(int Length, int Width, int Height)

18: {

19:     return Length * Width * Height;

20: }

```

Chúng ta [chạy ví dụ 2.11](#), kết quả ở hình 2.13



Hình 2.13: Kết quả của ví dụ 2.11

Chú ý:

Các tham số có giá trị mặc định chỉ được cho trong prototype của hàm và không được lặp lại trong định nghĩa hàm (Vì trình biên dịch sẽ dùng các thông tin trong prototype chứ không phải trong định nghĩa hàm để tạo một lệnh gọi).

Một hàm có thể có nhiều tham số có giá trị mặc định. Các tham số có giá trị mặc định cần phải được nhóm lại vào các tham số cuối cùng (hoặc duy nhất) của một hàm. Khi gọi hàm có nhiều tham số có giá trị mặc định, chúng ta chỉ có thể bỏ bớt các tham số theo thứ tự từ phải sang trái và phải bỏ liên tiếp nhau, chẳng hạn chúng ta có đoạn chương trình như sau:

```
int MyFunc(int a= 1, int b , int c = 3, int d = 4); //prototype sai!!!
```

```
int MyFunc(int a, int b = 2 , int c = 3, int d = 4); //prototype đúng
```

```
.....
```

```
MyFunc(); // Lỗi do tham số a không có giá trị mặc định
```

```
MyFunc(1); // OK, các tham số b, c và d lấy giá trị mặc định
```

```
MyFunc(5, 7); // OK, các tham số c và d lấy giá trị mặc định
```

```
MyFunc(5, 7, , 8); // Lỗi do các tham số bị bỏ phải liên tiếp nhau
```

2.2.12 Phép tham chiếu

Trong C, hàm nhận tham số là con trỏ đòi hỏi chúng ta phải thận trọng khi gọi hàm. Chúng ta cần viết hàm hoán đổi giá trị giữa hai số như sau:

```
void Swap(int *X, int *Y);
```

```
{
```

```
int Temp = *X;
```

```
*X = *Y;
```

```
*Y = *Temp;

}
```

Để hoán đổi giá trị hai biến A và B thì chúng ta gọi hàm như sau:

```
Swap(&A, &B);
```

Rõ ràng cách viết này không được thuận tiện lắm. Trong trường hợp này, C++ đưa ra một kiểu biến rất đặc biệt gọi là biến tham chiếu (reference variable). Một biến tham chiếu giống như là một bí danh của biến khác. Biến tham chiếu sẽ làm cho các hàm có thay đổi nội dung các tham số của nó được viết một cách thanh thoát hơn. Khi đó hàm *Swap()* được viết như sau:

```
void Swap(int &X, int &Y);

{

int Temp = X;

X = Y;

Y = Temp ;

}
```

Chúng ta gọi hàm như sau :

```
Swap(A, B);
```

Với cách gọi hàm này, C++ tự gởi địa chỉ của A và B làm tham số cho hàm *Swap()*. Cách dùng biến tham chiếu cho tham số của C++ tương tự như các tham số được khai báo là **Var** trong ngôn ngữ Pascal. Tham số này được gọi là tham số kiểu tham chiếu (reference parameter). Như vậy biến tham chiếu có cú pháp như sau :

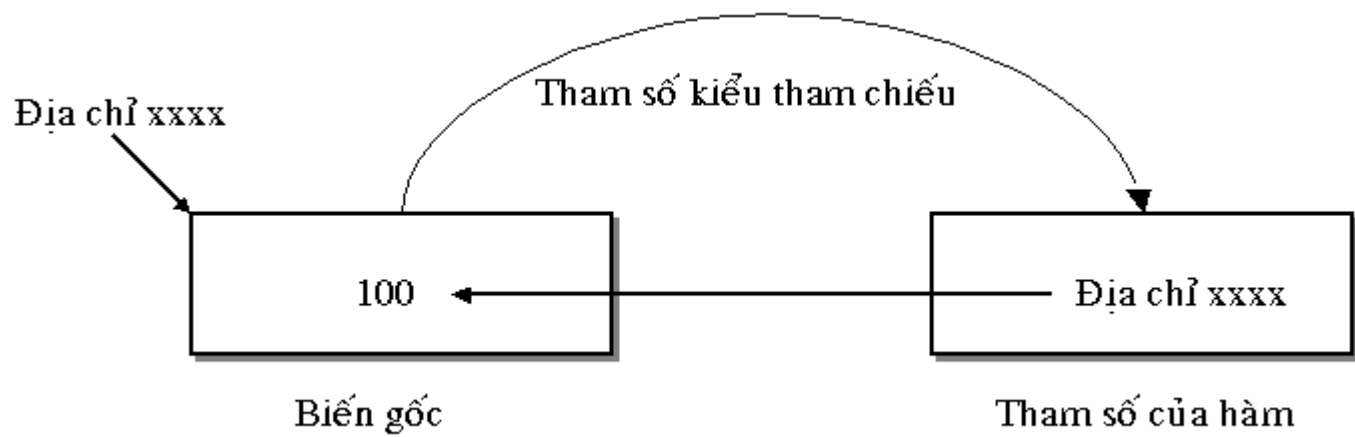
data_type & variable_name;

Trong đó:

data_type: Kiểu dữ liệu của biến.

variable_name: Tên của biến

Khi dùng biến tham chiếu cho tham số chỉ có địa chỉ của nó được gởi đi chứ không phải là toàn bộ cấu trúc hay đối tượng đó như hình 2.14, điều này rất hữu dụng khi chúng ta gởi cấu trúc và đối tượng lớn cho một hàm.



Hình 2.14: Một tham số kiểu tham chiếu nhận một tham chiếu tới một biến được chuyển cho tham số của hàm.

Ví dụ 2.12: Chương trình hoán đổi giá trị của hai biến.

```

1: #include <iostream.h>

2: //prototype

3 void Swap(int &X,int &Y);

4:

5: int main()

6: {

7:     int X = 10, Y = 5;

8:     cout<<"Truoc khi hoan doi: X = "<<X<<","Y = "<<Y<<endl;

9:     Swap(X,Y);

10:    cout<<"Sau khi hoan doi: X = "<<X<<","Y = "<<Y<<endl;

11:    return 0;

12: }

13:

14: void Swap(int &X,int &Y)

15: {

16:     int Temp=X;

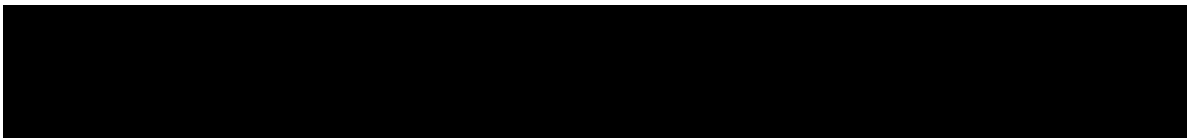
17:     X=Y;

18:     Y=Temp;

19: }

```

Chúng ta [chạy ví dụ 2.12](#), kết quả ở hình 2.15



Hình 2.15: Kết quả của ví dụ 2.12

Đôi khi chúng ta muốn gọi một tham số nào đó bằng biến tham chiếu cho hiệu quả, mặc dù chúng ta không muốn giá trị của nó bị thay đổi thì chúng ta dùng thêm từ khóa **const** như sau :

int MyFunc(const int & X);

Hàm *MyFunc()* sẽ chấp nhận một tham số *X* gọi bằng tham chiếu nhưng **const** xác định rằng *X*

không thể bị thay đổi. Biến tham chiếu có thể sử dụng như một bí danh của biến khác (bí danh đơn giản như một tên khác của biến gốc), chẳng hạn như đoạn mã sau :

```
int Count = 1;
```

```
int & Ref = Count; //Tạo biến Ref như là một bí danh của biến Count
```

```
++Ref; //Tăng biến Count lên 1 (sử dụng bí danh của biến Count)
```

Các biến tham chiếu phải được khởi động trong phần khai báo của chúng và chúng ta không thể gán lại một bí danh của biến khác cho chúng. Chẳng hạn đoạn mã sau là sai:

```
int X = 1;
```

```
int & Y; //Lỗi: Y phải được khởi động.
```

Khi một tham chiếu được khai báo như một bí danh của biến khác, mọi thao tác thực hiện trên bí danh chính là thực hiện trên biến gốc của nó. Chúng ta có thể lấy địa chỉ của biến tham chiếu và có thể so sánh các biến tham chiếu với nhau (phải tương thích về kiểu tham chiếu).

Ví dụ 2.13: Mọi thao tác trên trên bí danh chính là thao tác trên biến gốc của nó.

```
CT2_13.CPP
1: #include <iostream.h>
2: int main()
3: {
4:     int X = 3;
5:     int &Y = X; //Y là bí danh của X
6:     int Z = 100;
7:
8:     cout<<"X="<<X<<endl<<"Y="<<Y<<endl;
9:     Y *= 3;
10:    cout<<"X="<<X<<endl<<"Y="<<Y<<endl;
11:    Y = Z;
12:    cout<<"X="<<X<<endl<<"Y="<<Y<<endl;
13:    return 0;
14: }
```

Chúng ta [chạy ví dụ 2.13](#), kết quả ở hình 2.16

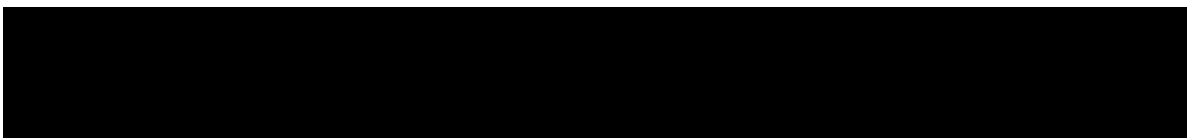


Hình 2.16: Kết quả của ví dụ 2.13

Ví dụ 2.14: Lấy địa chỉ của biến tham chiếu

```
CT2_14.CPP
1: #include <iostream.h>
2: int main()
3: {
4:     int X = 3;
5:     int &Y = X; //Y là bí danh của X
6:
7:     cout<<"Địa chỉ của X = "<<&X<<endl;
8:     cout<<"Địa chỉ của bí danh Y= "<<&Y<<endl;
9:     return 0;
10: }
```

Chúng ta chạy ví dụ 2.14, kết quả ở hình 2.17



Hình 2.17: Kết quả của ví dụ 2.14

Chúng ta có thể tạo ra biến tham chiếu với việc khởi động là một hằng, chẳng hạn như đoạn mã sau
:int & Ref = 45;

Trong trường hợp này, trình biên dịch tạo ra một biến tạm thời chứa trị hằng và biến tham chiếu chính là bí danh của biến tạm thời này. Điều này gọi là tham chiếu độc lập (independent reference). Các hàm có thể trả về một tham chiếu, nhưng điều này rất nguy hiểm. Khi hàm trả về một tham

chiều tới một biến cục bộ của hàm thì biến này phải được khai báo là **static**, ngược lại tham chiếu tới nó thì khi hàm kết thúc biến cục bộ này sẽ bị bỏ qua. Chẳng hạn như đoạn chương trình sau:

```
int & MyFunc()
{
    static int X = 200; //Nếu không khai báo là static thì điều này rất nguy hiểm.
    return X;
}
```

Khi một hàm trả về một tham chiếu, chúng ta có thể gọi hàm ở phải bên trái của một phép gán.

Ví dụ 2.15:



```

1: #include <iostream.h>

2:

3: int X = 4;

4: //prototype

5: int & MyFunc();

6:

7: int main()

8: {

9:     cout<<"X="<<X<<endl;

10:    cout<<"X="<<MyFunc()<<endl;

11:    MyFunc() = 20; //Nghĩa là X = 20

12:    cout<<"X="<<X<<endl;

13:    return 0;

14: }

15:

16: int & MyFunc()

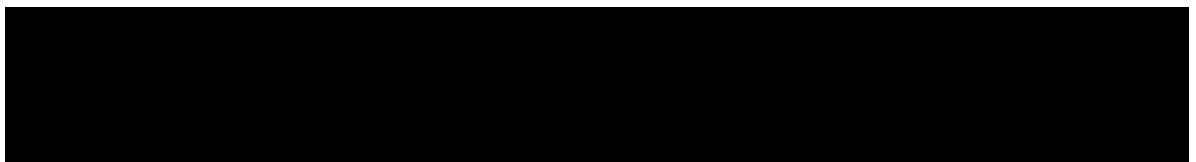
17: {

18:     return X;

19: }

```

Chúng ta [chạy ví dụ 2.15](#), kết quả ở hình 2.18



Hình 2.18: Kết quả của ví dụ 2.15

Chú ý: Mặc dù biến tham chiếu trông giống như là biến con trỏ nhưng chúng không thể là biến con trỏ do đó chúng không thể được dùng cấp phát động.

Chúng ta không thể khai báo một biến tham chiếu chỉ đến biến tham chiếu hoặc biến con trỏ chỉ

đến biến tham chiếu. Tuy nhiên chúng ta có thể khai báo một biến tham chiếu về biến con trỏ như đoạn mã sau:

```
int X;
```

```
int *P = &X;
```

```
int * &Ref = P;
```

2.2.13 **Phép đa năng hóa (Overloading) :**

Với ngôn ngữ C++, chúng ta có thể đa năng hóa các hàm và các toán tử (operator). Đa năng hóa là phương pháp cung cấp nhiều hơn một định nghĩa cho tên hàm đã cho trong cùng một phạm vi. Trình biên dịch sẽ lựa chọn phiên bản thích hợp của hàm hay toán tử dựa trên các tham số mà nó được gọi.

2.2.13.1 **Đa năng hóa các hàm (Functions overloading) :**

Trong ngôn ngữ C cũng như mọi ngôn ngữ máy tính khác, mỗi hàm đều phải có một tên phân biệt. Đôi khi đây là một điều phiền toái. Chẳng hạn như trong ngôn ngữ C, có rất nhiều hàm trả về trị tuyệt đối của một tham số là số, vì cần thiết phải có tên phân biệt nên C phải có hàm riêng cho mỗi kiểu dữ liệu số, do vậy chúng ta có tới ba hàm khác nhau để trả về trị tuyệt đối của một tham số :

```
int abs(int i);
```

```
long labs(long l);
```

```
double fabs(double d);
```

Tất cả các hàm này đều cùng thực hiện một chức năng nên chúng ta thấy điều này nghịch lý khi phải có ba tên khác nhau. C++ giải quyết điều này bằng cách cho phép chúng ta tạo ra các hàm khác nhau có cùng một tên. Đây chính là đa năng hóa hàm. Do đó trong C++ chúng ta có thể định nghĩa lại các hàm trả về trị tuyệt đối để thay thế các hàm trên như sau :

```
int abs(int i);
```

```
long abs(long l);
```

```
double abs(double d);
```

Ví dụ 2.16:

```
1: #include <iostream.h>
2: #include <math.h>
3:
4: int MyAbs(int X);
5: long MyAbs(long X);
6: double MyAbs(double X);
7:
8: int main()
9: {
10:     int X = -7;
11:     long Y = 2000001;
12:     double Z = -35.678;
13:     cout<<"Tri tuyet doi cua so nguyen (int) "<<X<<" la "
14:         <<MyAbs(X)<<endl;
15:     cout<<"Tri tuyet doi cua so nguyen (long int) "<<Y<<" la "
16:         <<MyAbs(Y)<<endl;
17:     cout<<"Tri tuyet doi cua so thuc "<<Z<<" la "
18:         <<MyAbs(Z)<<endl;
19:     return 0;
20: }
21:
22: int MyAbs(int X)
23: {
24:     return abs(X);
25: }
26:
27: long MyAbs(long X)
28: {
29:     return labs(X);
```

Chúng ta [chạy ví dụ 2.16](#) , kết quả ở hình 2.19



Hình 2.19: Kết quả của ví dụ 2.16

Trình biên dịch dựa vào sự khác nhau về số các tham số, kiểu của các tham số để có thể xác định chính xác phiên bản cài đặt nào của hàm *MyAbs()* thích hợp với một lệnh gọi hàm được cho, chẳng hạn như:

```
MyAbs(-7); //Gọi hàm int MyAbs(int)
```

```
MyAbs(-7l); //Gọi hàm long MyAbs(long)
```

```
MyAbs(-7.5); //Gọi hàm double MyAbs(double)
```

Quá trình tìm được hàm được đa năng hóa cũng là quá trình được dùng để giải quyết các trường hợp nhập nhằng của C++. Chẳng hạn như nếu tìm thấy một phiên bản định nghĩa nào đó của một hàm được đa năng hóa mà có kiểu dữ liệu các tham số của nó trùng với kiểu các tham số đã gọi tới trong lệnh gọi hàm thì phiên bản hàm đó sẽ được gọi. Nếu không trình biên dịch C++ sẽ gọi đến phiên bản nào cho phép chuyển kiểu dễ dàng nhất.

```
MyAbs('c'); //Gọi int MyAbs(int)
```

```
MyAbs(2.34f); //Gọi double MyAbs(double)
```

Các phép chuyển kiểu có sẵn sẽ được ưu tiên hơn các phép chuyển kiểu mà chúng ta tạo ra (chúng ta sẽ xem xét các phép chuyển kiểu tự tạo ở chương 3).

Chúng ta cũng có thể lấy địa chỉ của một hàm đã được đa năng hóa sao cho bằng một cách nào đó chúng ta có thể làm cho trình biên dịch C++ biết được chúng ta cần lấy địa chỉ của phiên bản hàm nào có trong định nghĩa. Chẳng hạn như:

```
int (*pf1)(int);
```

```
long (*pf2)(long);
```

```
int (*pf3)(double);
```

```
pf1 = MyAbs; //Trỏ đến hàm int MyAbs(int)
```

```
pf2 = MyAbs; //Trỏ đến hàm long MyAbs(long)
```

```
pf3 = MyAbs; //Lỗi!!! (không có phiên bản hàm nào để đối sánh)
```

■ Các giới hạn của việc đa năng hóa các hàm:

- Bất kỳ hai hàm nào trong tập các hàm đa năng phải có các tham số khác nhau.
- Các hàm đa năng hóa với danh sách các tham số cùng kiểu chỉ dựa trên kiểu trả về của hàm thì trình biên dịch báo lỗi. Chẳng hạn như, các khai báo sau là không hợp lệ:

```
void Print(int X);
```

```
int Print(int X);
```

Không có cách nào để trình biên dịch nhận biết phiên bản nào được gọi nếu giá trị trả về bị bỏ qua. Như vậy các phiên bản trong việc đa năng hóa phải có sự khác nhau ít nhất về kiểu hoặc số tham số mà chúng nhận được.

- Các khai báo bằng lệnh **typedef** không định nghĩa kiểu mới. Chúng chỉ thay đổi tên gọi của kiểu đã có. Chúng không ảnh hưởng tới cơ chế đa năng hóa hàm. Chúng ta hãy xem xét đoạn mã sau:

```
typedef char * PSTR;
```

```
void Print(char * Mess);
```

```
void Print(PSTR Mess);
```

Hai hàm này có cùng danh sách các tham số, do đó đoạn mã trên sẽ phát sinh lỗi.

- Đối với kiểu mảng và con trỏ được xem như đồng nhất đối với sự phân biệt khác nhau giữa các phiên bản hàm trong việc đa năng hóa hàm. Chẳng hạn như đoạn mã sau sẽ phát sinh lỗi:

```
void Print(char * Mess);
```

```
void Print(char Mess[]);
```

Tuy nhiên, đối với mảng nhiều chiều thì có sự phân biệt giữa các phiên bản hàm trong việc đa năng hóa hàm, chẳng hạn như đoạn mã sau hợp lệ:

```
void Print(char Mess[]);
```

```
void Print(char Mess[][7]);
```

```
void Print(char Mess[][9][42]);
```

- **const** và các con trỏ (hay các tham chiếu) có thể dùng để phân biệt, chẳng hạn như đoạn mã sau hợp lệ:

```
void Print(char *Mess);
```

```
void Print(const char *Mess);
```

2.2.13.2 Đa năng hóa các toán tử (Operators overloading) :

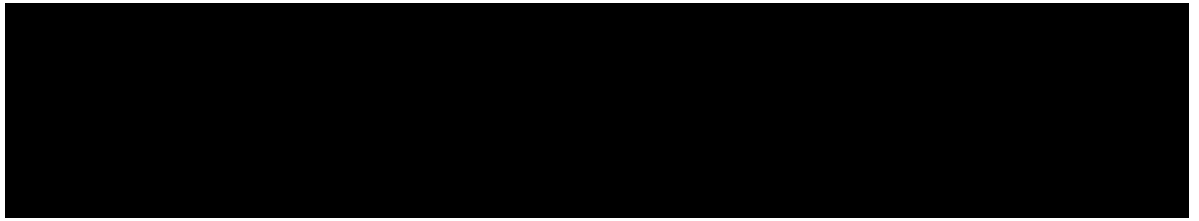
Trong ngôn ngữ C, khi chúng ta tự tạo ra một kiểu dữ liệu mới, chúng ta thực hiện các thao tác liên quan đến kiểu dữ liệu đó thường thông qua các hàm, điều này trở nên không thoải mái.

Ví dụ 2.17: Chương trình cài đặt các phép toán cộng và trừ số phức



```
1: #include <stdio.h>
2: /* Định nghĩa số phức */
3: typedef struct
4: {
5:     double Real;
6:     double Imaginary;
7: }Complex;
8:
9: Complex SetComplex(double R,double I);
10: Complex AddComplex(Complex C1,Complex C2);
11: Complex SubComplex(Complex C1,Complex C2);
12: void DisplayComplex(Complex C);
13:
14: int main(void)
15: {
16:     Complex C1,C2,C3,C4;
17:
18:     C1 = SetComplex(1.0,2.0);
19:     C2 = SetComplex(-3.0,4.0);
20:     printf("\nSo phuc thu nhat:");
21:     DisplayComplex(C1);
22:     printf("\nSo phuc thu hai:");
23:     DisplayComplex(C2);
24:     C3 = AddComplex(C1,C2); //Hoi bất tiện !!!
25:     C4 = SubComplex(C1,C2);
26:     printf("\nTong hai so phuc nay:");
27:     DisplayComplex(C3);
28:     printf("\nHieu hai so phuc nay:");
29:     DisplayComplex(C4);
```


Chúng ta [chạy ví dụ 2.17](#), kết quả ở hình 2.20



Hình 2.20: Kết quả của ví dụ 2.17

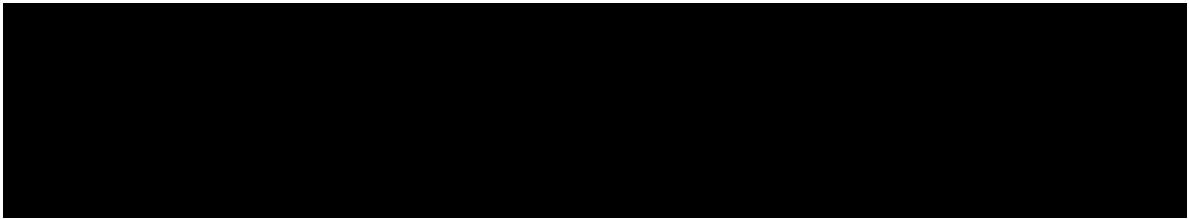
Trong chương trình ở ví dụ 2.17, chúng ta nhận thấy với các hàm vừa cài đặt dùng để cộng và trừ hai số phức $1+2i$ và $-3+4i$; người lập trình hoàn toàn không thoải mái khi sử dụng bởi vì thực chất thao tác cộng và trừ là các toán tử chứ không phải là hàm. Để khắc phục yếu điểm này, trong C++ cho phép chúng ta có thể định nghĩa lại chức năng của các toán tử đã có sẵn một cách tiện lợi và tự nhiên hơn rất nhiều. Điều này gọi là đa năng hóa toán tử. Khi đó chương trình ở ví dụ 2.17 được viết như sau:

[Ví dụ 2.18:](#)



```
1: #include <iostream.h>
2: // Định nghĩa số phức
3: typedef struct
4: {
5:     double Real;
6:     double Imaginary;
7: }Complex;
8:
9: Complex SetComplex(double R,double I);
10: void DisplayComplex(Complex C);
11: Complex operator + (Complex C1,Complex C2);
12: Complex operator - (Complex C1,Complex C2);
13:
14: int main(void)
15: {
16:     Complex C1,C2,C3,C4;
17:
18:     C1 = SetComplex(1.0,2.0);
19:     C2 = SetComplex(-3.0,4.0);
20:     cout<<"\nSo phuc thu nhat:";
21:     DisplayComplex(C1);
22:     cout<<"\nSo phuc thu hai:";
23:     DisplayComplex(C2);
24:     C3 = C1 + C2;
25:     C4 = C1 - C2;
26:     cout<<"\nTong hai so phuc nay:";
27:     DisplayComplex(C3);
28:     cout<<"\nHieu hai so phuc nay:";
29:     DisplayComplex(C4);
```

Chúng ta [chạy ví dụ 2.18](#), kết quả ở hình 2.21



Hình 2.21: Kết quả của ví dụ 2.18

Như vậy trong C++, các phép toán trên các giá trị kiểu số phức được thực hiện bằng các toán tử toán học chuẩn chứ không phải bằng các tên hàm như trong C. Chẳng hạn chúng ta có lệnh sau:

```
C4 = AddComplex(C3, SubComplex(C1,C2));
```

thì ở trong C++, chúng ta có lệnh tương ứng như sau:

```
C4 = C3 + C1 - C2;
```

Chúng ta nhận thấy rằng cả hai lệnh đều cho cùng kết quả nhưng lệnh của C++ thì dễ hiểu hơn. C++ làm được điều này bằng cách tạo ra các hàm định nghĩa cách thực hiện của một toán tử cho các kiểu dữ liệu tự định nghĩa. Một hàm định nghĩa một toán tử có cú pháp sau:

```
data_type operator operator_symbol ( parameters )  
  
{  
  
.....  
  
}
```

Trong đó:

data_type: Kiểu trả về.

operator_symbol: Ký hiệu của toán tử.

parameters: Các tham số (nếu có).

Trong chương trình ví dụ 2.18, toán tử + là toán tử gồm hai toán hạng (gọi là toán tử hai ngôi; toán tử một ngôi là toán tử chỉ có một toán hạng) và trình biên dịch biết tham số đầu tiên là ở bên trái toán tử, còn tham số thứ hai thì ở bên phải của toán tử. Trong trường hợp lập trình viên quen thuộc với cách gọi hàm, C++ vẫn cho phép bằng cách viết như sau:

```
C3 = operator + (C1,C2);
```

$C4 = \text{operator} - (C1, C2);$

Các toán tử được đa năng hóa sẽ được lựa chọn bởi trình biên dịch cũng theo cách thức tương tự như việc chọn lựa giữa các hàm được đa năng hóa là khi gặp một toán tử làm việc trên các kiểu không phải là kiểu có sẵn, trình biên dịch sẽ tìm một hàm định nghĩa của toán tử nào đó có các tham số đối sánh với các toán hạng để dùng. Chúng ta sẽ tìm hiểu kỹ về việc đa năng hóa các toán tử trong chương 4.

Các giới hạn của đa năng hóa toán tử:

- Chúng ta không thể định nghĩa các toán tử mới.
- Hầu hết các toán tử của C++ đều có thể được đa năng hóa. Các toán tử sau không được đa năng hóa là :

Toán tử	Ý nghĩa
::	Toán tử định phạm vi.
.*	Truy cập đến con trỏ là trường của struct hay thành viên của class .
.	Truy cập đến trường của struct hay thành viên của class .
?:	Toán tử điều kiện
sizeof	

và chúng ta cũng không thể đa năng hóa bất kỳ ký hiệu tiền xử lý nào.

- Chúng ta không thể thay đổi thứ tự ưu tiên của một toán tử hay không thể thay đổi số các toán hạng của nó.
- Chúng ta không thể thay đổi ý nghĩa của các toán tử khi áp dụng cho các kiểu có sẵn.
- Đa năng hóa các toán tử không thể có các tham số có giá trị mặc định.

Các toán tử có thể đa năng hoá:

+	-	*	/	%	^
!	=	<	>	+=	-=
^=	&=	=	<<	>>	<<=
<=	>=	&&		++	--

()	[]	new	delete	&	
~	*=	/=	%=	>>=	==
!=	,	->	->*		

Các toán tử được phân loại như sau :

- Các toán tử một ngôi : * & ~ ! ++ -- **sizeof** (*data_type*)

Các toán tử này được định nghĩa chỉ có một tham số và phải trả về một giá trị cùng kiểu với tham số của chúng. Đối với toán tử **sizeof** phải trả về một giá trị kiểu **size_t** (định nghĩa trong `stddef.h`).

Toán tử (*data_type*) được dùng để chuyển đổi kiểu, nó phải trả về một giá trị có kiểu là *data_type*.

- Các toán tử hai ngôi:

* / % + - >> << > <

>= <= == != & | ^ && ||

Các toán tử này được định nghĩa có hai tham số.

- Các phép gán: = += -= *= /= %= >>= <<= ^= |=

Các toán tử gán được định nghĩa chỉ có một tham số. Không có giới hạn về kiểu của tham số và kiểu trả về của phép gán.

- Toán tử lấy thành viên : ->
- Toán tử lấy phần tử theo chỉ số: []
- Toán tử gọi hàm: ()

3.1 DẪN NHẬP

Bây giờ chúng ta bắt đầu tìm hiểu về lập trình hướng đối tượng trong C++. Trong các phần sau, chúng ta cũng tìm hiểu về các kỹ thuật của thiết kế hướng đối tượng (Object-Oriented Design OOD): Chúng ta phân tích một vấn đề cụ thể, xác định các đối tượng nào cần để cài đặt hệ thống, xác định các thuộc tính nào mà đối tượng phải có, xác định hành vi nào mà đối tượng cần đưa ra, và chỉ rõ làm thế nào các đối tượng cần tương tác với đối tượng khác để thực hiện các mục tiêu tổng thể của hệ thống.

Chúng ta nhắc lại các khái niệm và thuật ngữ chính của định hướng đối tượng. OOP *đóng gói* dữ liệu (các thuộc tính) và các hàm (hành vi) thành gói gọi là *các đối tượng*. Dữ liệu và các hàm của đối tượng có sự liên hệ mật thiết với nhau. Các đối tượng có các đặc tính của việc che dấu thông tin. Điều này nghĩa là mặc dù các đối tượng có thể biết làm thế nào liên lạc với đối tượng khác thông qua các giao diện hoàn toàn xác định, bình thường các đối tượng không được phép biết làm thế nào các đối tượng khác được thực thi, các chi tiết của sự thi hành được giấu bên trong các đối tượng.

Trong C và các ngôn ngữ lập trình thủ tục, lập trình có khuynh hướng định hướng hành động, trong khi ý tưởng trong lập trình C++ là định hướng đối tượng. Trong C, đơn vị của lập trình là hàm; trong C++, đơn vị của lập trình là *lớp* (class) .

Các lập trình viên C tập trung vào viết các hàm. Các nhóm của các hành động mà thực hiện vài công việc được tạo thành các hàm, và các hàm được nhóm thành các chương trình. Dữ liệu thì rất quan trọng trong C, nhưng quan điểm là dữ liệu tồn tại chính trong việc hỗ trợ các hàm động mà hàm thực hiện. Các động từ trong một hệ thống giúp cho lập trình viên C xác định tập các hàm mà sẽ hoạt động cùng với việc thực thi hệ thống.

Các lập trình viên C++ tập trung vào việc tạo ra "các kiểu do người dùng định nghĩa" (user-defined types) gọi là các lớp. Các lớp cũng được tham chiếu như "các kiểu do lập trình viên định nghĩa" (programmer-defined types). Mỗi lớp chứa dữ liệu cũng như tập các hàm mà xử lý dữ liệu. Các thành phần dữ liệu của một lớp được gọi là "các thành viên dữ liệu" (data members). Các thành phần hàm của một lớp được gọi là "các hàm thành viên" (member functions). Giống như thực thể của kiểu có sẵn như **int** được gọi là một biến, một thực thể của kiểu do người dùng định nghĩa (nghĩa là một lớp) được gọi là một đối tượng. Các danh từ trong một hệ thống giúp cho lập trình viên C++ xác định tập các lớp. Các lớp này được sử dụng để tạo các đối tượng mà sẽ hoạt động cùng với việc thực thi hệ thống.

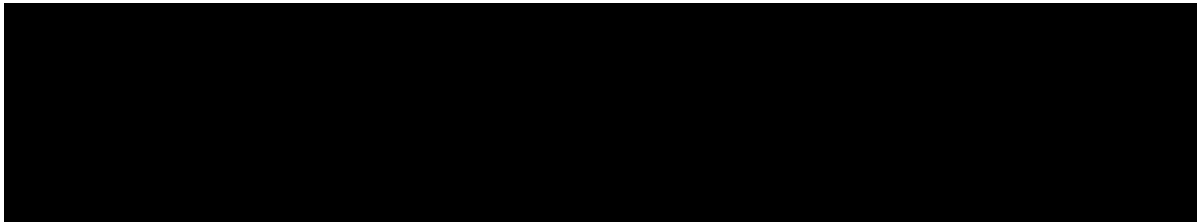
Các lớp trong C++ được tiến hóa tự nhiên của khái niệm **struct** trong C. Trước khi tiến hành việc trình bày các lớp trong C++, chúng ta tìm hiểu về cấu trúc, và chúng ta xây dựng một kiểu do người dùng định nghĩa dựa trên một cấu trúc.

3.2 CÀI ĐẶT MỘT KIỂU DO NGƯỜI DÙNG ĐỊNH NGHĨA VỚI MỘT STRUCT

Ví dụ 3.1: Chúng ta xây dựng kiểu cấu trúc *Time* với ba thành viên số nguyên: *Hour*, *Minute* và *second*. Chương trình định nghĩa một cấu trúc *Time* gọi là *DinnerTime*. Chương trình in thời gian dưới dạng giờ quân đội và dạng chuẩn.

```
1: #include <iostream.h>
2:
3: struct Time
4: {
5:     int Hour; // 0-23
6:     int Minute; // 0-59
7:     int Second; // 0-59
8: };
9:
10: void PrintMilitary(const Time &); //prototype
11: void PrintStandard(const Time &); //prototype
12:
13: int main()
14: {
15:     Time DinnerTime;
16:
17:     //Thiết lập các thành viên với giá trị hợp lệ
18:     DinnerTime.Hour = 18;
19:     DinnerTime.Minute = 30;
20:     DinnerTime.Second = 0;
21:
22:     cout << "Dinner will be held at ";
23:     PrintMilitary(DinnerTime);
24:     cout << " military time," << endl << "which is ";
25:     PrintStandard(DinnerTime);
26:     cout << " standard time." << endl;
27:
28:     //Thiết lập các thành viên với giá trị không hợp lệ
29:     DinnerTime.Hour = 29;
```

Chúng ta [chạy ví dụ 3.1](#), kết quả ở hình 3.1



Hình 3.1: Kết quả của ví dụ 3.1

Có một vài hạn chế khi tạo các kiểu dữ liệu mới với các cấu trúc ở phần trên. Khi việc khởi tạo không được yêu cầu, có thể có dữ liệu chưa khởi tạo và các vấn đề nảy sinh. Ngay cả nếu dữ liệu được khởi tạo, nó có thể khởi tạo không chính xác. Các giá trị không hợp lệ có thể được gán cho các thành viên của một cấu trúc bởi vì chương trình trực tiếp truy cập dữ liệu. Chẳng hạn ở ví dụ 3.1 ở dòng 29 đến dòng 31, chương trình gán các giá trị không hợp lệ cho đối tượng *DinnerTime*. Nếu việc cài đặt của **struct** thay đổi, tất cả các chương trình sử dụng **struct** phải thay đổi. Điều này do lập trình viên trực tiếp thao tác kiểu dữ liệu. Không có "giao diện" để bảo đảm lập trình viên sử dụng dữ liệu chính xác và bảo đảm dữ liệu còn lại ở trạng thái thích hợp. Mặt khác, cấu trúc trong C không thể được in như một đơn vị, chúng được in khi các thành viên được in. Các cấu trúc trong C không thể so sánh với nhau, chúng phải được so sánh thành viên với thành viên.

Phần sau cài đặt lại cấu trúc *Time* ở ví dụ 3.1 như một lớp và chứng minh một số thuận lợi để việc tạo ra cái gọi là các kiểu dữ liệu trừu tượng (Abstract Data Types – ADT) như các lớp. Chúng ta sẽ thấy rằng các lớp và các cấu trúc có thể sử dụng gần như giống nhau trong C++. Sự khác nhau giữa chúng là thuộc tính truy cập các thành viên.

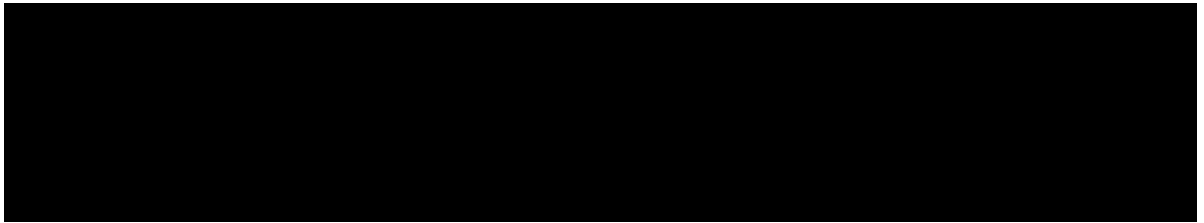
3.2 CÀI ĐẶT MỘT KIỂU DO NGƯỜI DÙNG ĐỊNH NGHĨA VỚI MỘT STRUCT

[Ví dụ 3.1](#): Chúng ta xây dựng kiểu cấu trúc *Time* với ba thành viên số nguyên: *Hour*, *Minute* và *second*. Chương trình định nghĩa một cấu trúc *Time* gọi là *DinnerTime*. Chương trình in thời gian dưới dạng giờ quân đội và dạng chuẩn.




```
1: #include <iostream.h>
2:
3: struct Time
4: {
5:     int Hour; // 0-23
6:     int Minute; // 0-59
7:     int Second; // 0-59
8: };
9:
10: void PrintMilitary(const Time &); //prototype
11: void PrintStandard(const Time &); //prototype
12:
13: int main()
14: {
15:     Time DinnerTime;
16:
17:     //Thiết lập các thành viên với giá trị hợp lệ
18:     DinnerTime.Hour = 18;
19:     DinnerTime.Minute = 30;
20:     DinnerTime.Second = 0;
21:
22:     cout << "Dinner will be held at ";
23:     PrintMilitary(DinnerTime);
24:     cout << " military time," << endl << "which is ";
25:     PrintStandard(DinnerTime);
26:     cout << " standard time." << endl;
27:
28:     //Thiết lập các thành viên với giá trị không hợp lệ
29:     DinnerTime.Hour = 29;
```

Chúng ta [chạy ví dụ 3.1](#), kết quả ở hình 3.1



Hình 3.1: Kết quả của ví dụ 3.1

Có một vài hạn chế khi tạo các kiểu dữ liệu mới với các cấu trúc ở phần trên. Khi việc khởi tạo không được yêu cầu, có thể có dữ liệu chưa khởi tạo và các vấn đề nảy sinh. Ngay cả nếu dữ liệu được khởi tạo, nó có thể khởi tạo không chính xác. Các giá trị không hợp lệ có thể được gán cho các thành viên của một cấu trúc bởi vì chương trình trực tiếp truy cập dữ liệu. Chẳng hạn ở ví dụ 3.1 ở dòng 29 đến dòng 31, chương trình gán các giá trị không hợp lệ cho đối tượng *DinnerTime*. Nếu việc cài đặt của **struct** thay đổi, tất cả các chương trình sử dụng **struct** phải thay đổi. Điều này do lập trình viên trực tiếp thao tác kiểu dữ liệu. Không có "giao diện" để bảo đảm lập trình viên sử dụng dữ liệu chính xác và bảo đảm dữ liệu còn lại ở trạng thái thích hợp. Mặt khác, cấu trúc trong C không thể được in như một đơn vị, chúng được in khi các thành viên được in. Các cấu trúc trong C không thể so sánh với nhau, chúng phải được so sánh thành viên với thành viên.

Phần sau cài đặt lại cấu trúc *Time* ở ví dụ 3.1 như một lớp và chứng minh một số thuận lợi để việc tạo ra cái gọi là các kiểu dữ liệu trừu tượng (Abstract Data Types – ADT) như các lớp. Chúng ta sẽ thấy rằng các lớp và các cấu trúc có thể sử dụng gần như giống nhau trong C++. Sự khác nhau giữa chúng là thuộc tính truy cập các thành viên.

3.3 CÀI ĐẶT MỘT KIỂU DỮ LIỆU TRỪU TƯỢNG VỚI MỘT LỚP

Các lớp cho phép lập trình viên mô hình các đối tượng mà có các thuộc tính (biểu diễn như các thành viên dữ liệu – Data members) và các hành vi hoặc các thao tác (biểu diễn như các hàm thành viên – Member functions). Các kiểu chứa các thành viên dữ liệu và các hàm thành viên được định nghĩa thông thường trong C++ sử dụng từ khóa **class**, có cú pháp như sau:

```
class <class-name>
{
    <member-list> //Thân của lớp
};
```

Trong đó:

class-name: tên lớp.

member-list: đặc tả các thành viên dữ liệu và các hàm thành viên.

Các hàm thành viên đôi khi được gọi là các phương thức (methods) trong các ngôn ngữ lập trình hướng đối tượng khác, và được đưa ra trong việc đáp ứng các message gửi tới một đối tượng. Một message tương ứng với việc gọi hàm thành viên.

Khi một lớp được định nghĩa, tên lớp có thể được sử dụng để khai báo đối tượng của lớp theo cú pháp sau:

<class-name> <object-name>;

Chẳng hạn, cấu trúc *Time* sẽ được định nghĩa dưới dạng lớp như sau:

```
class Time
{
    public:
        Time();

        void SetTime(int, int, int)

        void PrintMilitary();

        void PrintStandard()

    private:
        int Hour; // 0 - 23

        int Minute; // 0 - 59

        int Second; // 0 - 59

};
```

Trong định nghĩa lớp *Time* chứa ba thành viên dữ liệu là *Hour*, *Minute* và *Second*, và cũng trong lớp này, chúng ta thấy các nhãn **public** và **private** được gọi là các thuộc tính xác định truy cập thành viên (member access specifiers) gọi tắt là thuộc tính truy cập.

Bất kỳ thành viên dữ liệu hay hàm thành viên khai báo sau **public** có thể được truy cập bất kỳ nơi nào mà chương trình truy cập đến một đối tượng của lớp. Bất kỳ thành viên dữ liệu hay hàm thành viên khai báo sau **private** chỉ có thể được truy cập bởi các hàm thành viên của lớp. Các thuộc tính truy cập luôn luôn kết thúc với dấu hai chấm (:) và có thể xuất hiện nhiều lần và theo thứ tự bất kỳ trong định nghĩa lớp. Mặc định thuộc tính truy cập là **private**.

Định nghĩa lớp chứa các prototype của bốn hàm thành viên sau thuộc tính truy cập **public** là *Time()*, *SetTime()*, *PrintMilitary()* và *PrintStandard()*. Đó là các hàm thành viên public (public member function) hoặc giao diện (interface) của lớp. Các hàm này sẽ được sử dụng bởi các client (nghĩa là các phần của một chương trình mà là các người dùng) của lớp xử lý dữ liệu của lớp. Có thể nhận thấy trong định nghĩa lớp *Time*, hàm thành viên *Time()* có cùng tên với tên lớp *Time*, nó được gọi là hàm xây dựng (constructor function) của lớp *Time*.

Một constructor là một hàm thành viên đặc biệt mà khởi động các thành viên dữ liệu của một đối tượng của lớp. Một constructor của lớp được gọi tự động khi đối tượng của lớp đó được tạo.

Thông thường, các thành viên dữ liệu được liệt kê trong phần **private** của một lớp, còn các hàm thành viên được liệt kê trong phần **public**. Nhưng có thể có các hàm thành viên **private** và thành viên dữ liệu **public**.

Khi lớp được định nghĩa, nó có thể sử dụng như một kiểu trong phần khai báo như sau:

```
Time Sunset, // Đối tượng của lớp Time
```

```
ArrayTimes[5], // Mảng các đối tượng của lớp Time
```

```
*PTime, // Con trỏ trỏ đến một đối tượng của lớp Time
```

```
&DinnerTime = Sunset; // Tham chiếu đến một đối tượng của lớp Time
```

Ví dụ 3.2: Xây dựng lại lớp *Time* ở ví dụ 3.1



```

1: #include <iostream.h>

2:

3: class Time

4: {

5:     public:

6:     Time(); //Constructor

7:     void SetTime(int, int, int); //Thiết lập Hour, Minute va Second

8:     void PrintMilitary(); //In thời gian dưới dạng giờ quân đội

9:     void PrintStandard(); //In thời gian dưới dạng chuẩn

10:    private:

11:    int Hour; // 0 - 23

12:    int Minute; // 0 - 59

13:    int Second; // 0 - 59

14: };

15:

16: //Constructor khởi tạo mỗi thành viên dữ liệu với giá trị zero

17: //Bảo đảm tất cả các đối tượng bắt đầu ở một trạng thái thích hợp

18: Time::Time()

19: {

20:     Hour = Minute = Second = 0;

21: }

22:

23: //Thiết lập một giá trị Time mới sử dụng giờ quânđội

24: //Thực hiện việc kiểm tra tính hợp lệ trên các giá trị dữ liệu

25: //Thiết lập các giá trị không hợp lệ thành zero

26: void Time::SetTime(int H, int M, int S)

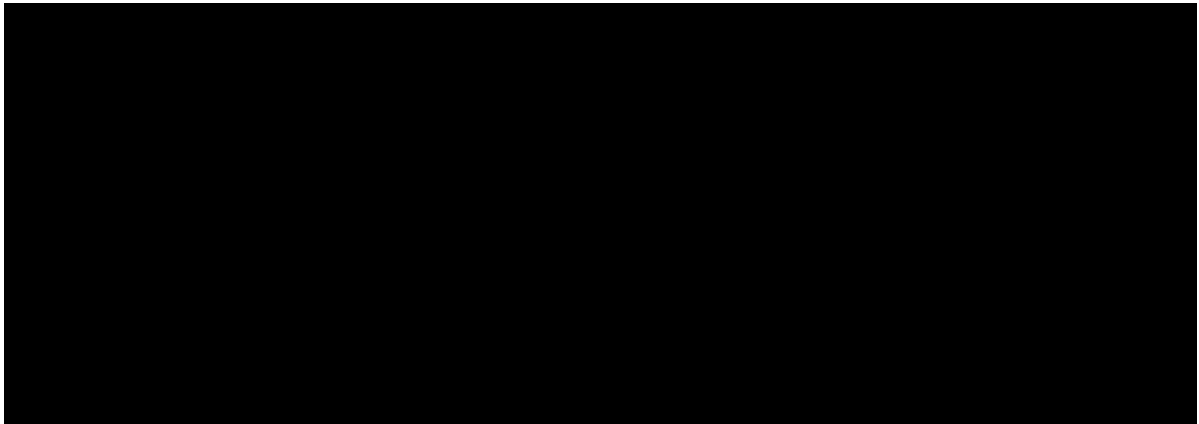
27: {

28:     Hour = (H >= 0 && H < 24) ? H : 0;

29:     Minute = (M >= 0 && M < 60) ? M : 0;

```

Chúng ta [chạy ví dụ 3.2](#), kết quả ở hình 3.2



Hình 3.2: Kết quả của ví dụ 3.2

Trong ví dụ 3.2, chương trình thuyết minh một đối tượng của lớp *Time* gọi là *T* (dòng 52). Khi đó constructor của lớp *Time* tự động gọi và rõ ràng khởi tạo mỗi thành viên dữ liệu **private** là zero. Sau đó thời gian được in dưới dạng giờ quân đội và dạng chuẩn để xác nhận các thành viên này được khởi tạo thích hợp (dòng 54 đến 57). Kế tới thời gian được thiết lập bằng cách sử dụng hàm thành viên *SetTime()* (dòng 59) và thời gian lại được in ở hai dạng (dòng 60 đến 63). Cuối cùng hàm thành viên *SetTime()* (dòng 65) thử thiết lập các thành viên dữ liệu với các giá trị không hợp lệ, và thời gian lại được in ở hai dạng (dòng 66 đến 70).

Chúng ta nhận thấy rằng, tất cả các thành viên dữ liệu của một lớp không thể khởi tạo tại nơi mà chúng được khai báo trong thân lớp. Các thành viên dữ liệu này phải được khởi tạo bởi constructor của lớp hay chúng có thể gán giá trị bởi các hàm thiết lập.

Khi một lớp được định nghĩa và các hàm thành viên của nó được khai báo, các hàm thành viên này phải được định nghĩa. Mỗi hàm thành viên của lớp có thể được định nghĩa trực tiếp trong thân lớp (hiển nhiên bao gồm prototype hàm của lớp), hoặc hàm thành viên có thể được định nghĩa sau thân lớp. Khi một hàm thành viên được định nghĩa sau định nghĩa lớp tương ứng, tên hàm được đặt trước bởi tên lớp và toán tử định phạm vi (::). Chẳng hạn như ở ví dụ 3.2 gồm các dòng 18, 26, 34 và 42. Bởi vì các lớp khác nhau có thể có các tên thành viên giống nhau, toán tử định phạm vi "ràng buộc" tên thành viên tới tên lớp để nhận dạng các hàm thành viên của một lớp.

Mặc dù một hàm thành viên khai báo trong định nghĩa một lớp có thể định nghĩa bên ngoài định nghĩa lớp này, hàm thành viên đó vẫn còn bên trong phạm vi của lớp, nghĩa là tên của nó chỉ được biết tới các thành viên khác của lớp ngoại trừ tham chiếu thông qua một đối tượng của lớp, một tham chiếu tới một đối tượng của lớp, hoặc một con trỏ tới một đối tượng của lớp.

Nếu một hàm thành viên được định nghĩa trong định nghĩa một lớp, hàm thành viên này chính là hàm inline. Các hàm thành viên định nghĩa bên ngoài định nghĩa một lớp có thể là hàm inline bằng cách sử dụng từ khóa **inline**.

Hàm thành viên cùng tên với tên lớp nhưng đặt trước là một ký tự ngã (~) được gọi là destructor của lớp này. Hàm destructor làm "công việc nội trợ kết thúc" trên mỗi đối tượng của lớp trước khi vùng nhớ cho đối tượng được phục hồi bởi hệ thống.

Ví dụ 3.3: Lấy lại ví dụ 3.2 nhưng hai hàm *PrintMilitary()* và *PrintStandard()* là các hàm inline.



```

1: #include <iostream.h>

2:

3: class Time

4: {

5: public:

6: Time(); ; //Constructor

7: void SetTime(int, int, int); // Thiết lập Hour, Minute và
Second

8: void PrintMilitary() // In thời gian dưới dạng giờ quânđội

9: {

10: cout << (Hour < 10 ? "0" : "") << Hour << ":"

11:         << (Minute < 10 ? "0" : "") << Minute << ":"

12:         << (Second < 10 ? "0" : "") << Second;

13: }

14: void PrintStandard(); // In thời gian dưới dạng chuẩn

15: private:

16: int Hour; // 0 - 23

17: int Minute; // 0 - 59

18: int Second; // 0 - 59

19: };

20: //Constructor khởi tạo mỗi thành viên dữ liệu với giá trị
zero

21: //Bảo đảm tất cả các đối tượng bắt đầu ở một trạng thái thích
hợp

22: Time::Time()

23: {

24: Hour = Minute = Second = 0;

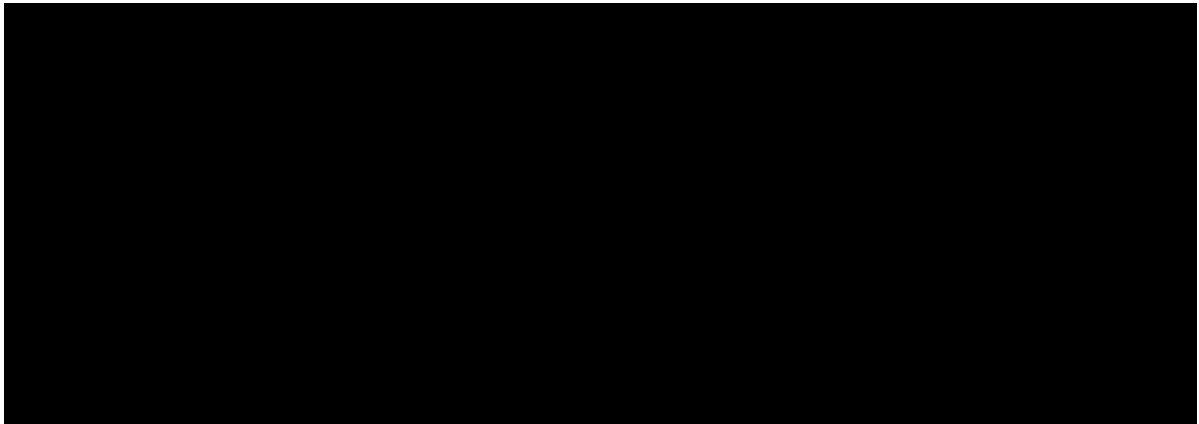
25: }

26:

27: #9; //Thiết lập một g61 trị Time mới sử dụng giờ quân đội

```


Chúng ta [chạy ví dụ 3.3](#), kết quả ở hình 3.3



Hình 3.3: Kết quả của ví dụ 3.3

3.4 PHẠM VI LỚP VÀ TRUY CẬP CÁC THÀNH VIÊN LỚP

Các thành viên dữ liệu của một lớp (các biến khai báo trong định nghĩa lớp) và các hàm thành viên (các hàm khai báo trong định nghĩa lớp) thuộc vào phạm vi của lớp.

Trong một phạm vi lớp, các thành viên của lớp được truy cập ngay lập tức bởi tất cả các hàm thành viên của lớp đó và có thể được tham chiếu một cách dễ dàng bởi tên. Bên ngoài một phạm vi lớp, các thành viên của lớp được tham chiếu thông qua hoặc một tên đối tượng, một tham chiếu đến một đối tượng, hoặc một con trỏ tới đối tượng.

Các hàm thành viên của lớp có thể được đa năng hóa (overload), nhưng chỉ bởi các hàm thành viên khác của lớp. Để đa năng hóa một hàm thành viên, đơn giản cung cấp trong định nghĩa lớp một prototype cho mỗi phiên bản của hàm đa năng hóa, và cung cấp một định nghĩa hàm riêng biệt cho mỗi phiên bản của hàm.

Các hàm thành viên có phạm vi hàm trong một lớp – các biến định nghĩa trong một hàm thành viên chỉ được biết tới hàm đó. Nếu một hàm thành viên định nghĩa một biến cùng tên với tên một biến trong phạm vi lớp, biến phạm vi lớp được dấu bởi biến phạm vi hàm bên trong phạm vi hàm. Như thế một biến bị dấu có thể được truy cập thông qua toán tử định phạm vi.

Các toán tử được sử dụng để truy cập các thành viên của lớp được đồng nhất với các toán tử sử dụng để truy cập các thành viên của cấu trúc. Toán tử lựa chọn thành viên dấu chấm (.) được kết hợp với một tên của đối tượng hay với một tham chiếu tới một đối tượng để truy cập các thành viên của đối tượng. Toán tử lựa chọn thành viên mũi tên (->) được kết hợp với một con trỏ tới một truy cập để truy cập các thành viên của đối tượng.

Ví dụ 3.4: Chương trình sau minh họa việc truy cập các thành viên của một lớp với các toán tử lựa chọn thành viên.



```
1: #include <iostream.h>
2:
3: class Count
4: {
5:     public:
6:     int X;
7:     void Print()
8:     {
9:         cout << X << endl;
10:    }
11: };
12:
13: int main()
14: {
15:     Count Counter, //Tạo đối tượng Counter
16:     *CounterPtr = &Counter, //Con trỏ trỏ tới Counter
17:     &CounterRef = Counter; //Tham chiếu tới Counter
18:
19:     cout << "Assign 7 to X and Print using the object's name: ";
20:     Counter.X = 7; //Gán 7 cho thành viên dữ liệu X
21:     Counter.Print(); //Gọi hàm thành viên Print
22:
23:     cout << "Assign 8 to X and Print using a reference: ";
24:     CounterRef.X = 8; //Gán 8 cho thành viên dữ liệu X
25:     CounterRef.Print(); //Gọi hàm thành viên Print
26:
27:     cout << "Assign 10 to X and Print using a pointer: ";
28:     CounterPtr->X = 10; //Gán 10 cho thành viên dữ liệu X
```

Chúng ta [chạy ví dụ 3.4](#), kết quả ở hình 3.4



Hình 3.4: Kết quả của ví dụ 3.4

3.5 ĐIỀU KHIỂN TRUY CẬP TỚI CÁC THÀNH VIÊN

Các thuộc tính truy cập **public** và **private** (và **protected** chúng ta sẽ xem xét sau) được sử dụng để điều khiển truy cập tới các thành viên dữ liệu và các hàm thành viên của lớp. Chế độ truy cập mặc định đối với lớp là **private** vì thế tất cả các thành viên sau phần header của lớp và trước nhãn đầu tiên là **private**. Sau mỗi nhãn, chế độ mà được kéo theo bởi nhãn đó áp dụng cho đến khi gặp nhãn kế tiếp hoặc cho đến khi gặp dấu móc phải (}) của phần định nghĩa lớp. Các nhãn **public**, **private** và **protected** có thể được lặp lại nhưng cách dùng như vậy thì hiếm có và có thể gây khó hiểu.

Các thành viên **private** chỉ có thể được truy cập bởi các hàm thành viên (và các hàm friend) của lớp đó. Các thành viên **public** của lớp có thể được truy cập bởi bất kỳ hàm nào trong chương trình.

Mục đích chính của các thành viên **public** là để biểu thị cho client của lớp một cái nhìn của các dịch vụ (services) mà lớp cung cấp. Tập hợp này của các dịch vụ hình thành giao diện public của lớp. Các client của lớp không cần quan tâm làm thế nào lớp hoàn thành các thao tác của nó. Các thành viên **private** của lớp cũng như các định nghĩa của các hàm thành viên **public** của nó thì không phải có thể truy cập tới client của một lớp. Các thành phần này hình thành sự thi hành của lớp.

Ví dụ 3.5: Chương trình sau cho thấy rằng các thành viên **private** chỉ có thể truy cập thông qua giao diện public sử dụng các hàm thành viên **public**.



```
1: #include <iostream.h>
2:
3: class MyClass
4: {
5: private:
6: int X,Y;
7: public:
8: void Print();
9: };
10:
11: void MyClass::Print()
12: {
13: cout <<X<<Y<<endl;
14: }
15:
16: int main()
17: {
18: MyClass M;
19:
20: M.X = 3; //Error: 'MyClass::X' is not accessible
21: M.Y = 4; //Error: 'MyClass::Y' is not accessible
22: M.Print();
23: return 0;
24: }
```

Khi chúng ta biên dịch chương trình này, compiler phát sinh ra hai lỗi tại hai dòng 20 và 21 như sau:



Hình 3.5: Thông báo lỗi của ví dụ 3.5

Thuộc tính truy cập mặc định đối với các thành viên của lớp là **private**. Thuộc tính truy cập các thành viên của một lớp có thể được thiết lập rõ ràng là **public**, **protected** hoặc **private**. Thuộc tính truy cập mặc định đối với các thành viên của **struct** là **public**. Thuộc tính truy cập các thành viên của một **struct** cũng có thể được thiết lập rõ ràng là **public**, **protected** hoặc **private**.

Truy cập đến một dữ liệu **private** cần phải được điều khiển cẩn thận bởi việc sử dụng của các hàm thành viên, gọi là các hàm truy cập (access functions).

3.6 CÁC HÀM TRUY CẬP VÀ CÁC HÀM TIỆN ÍCH

Không phải tất cả các hàm thành viên đều là **public** để phục vụ như bộ phận giao diện của một lớp. Một vài hàm còn lại là **private** và phục vụ như các hàm tiện ích (utility functions) cho các hàm khác của lớp.

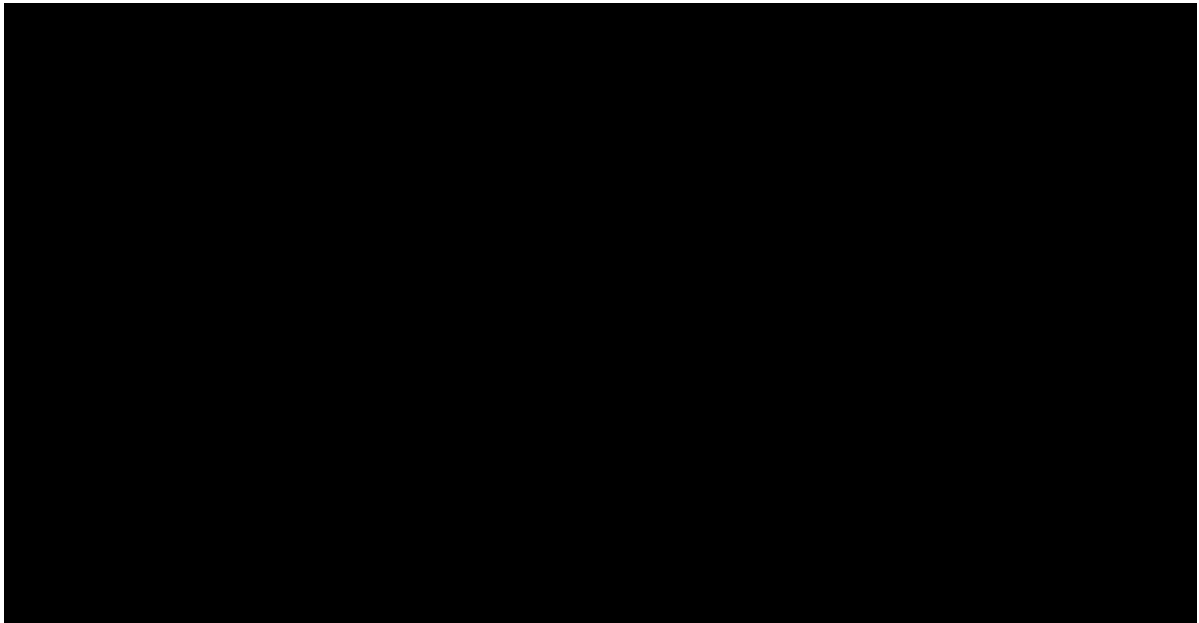
Các hàm truy cập có thể đọc hay hiển thị dữ liệu. Sử dụng các hàm truy cập để kiểm tra tính đúng hoặc sai của các điều kiện – các hàm như thế thường được gọi là các hàm khẳng định (predicate functions). Một ví dụ của hàm khẳng định là một hàm *IsEmpty()* của lớp container - một lớp có khả năng giữ nhiều đối tượng - giống như một danh sách liên kết, một stack hay một hàng đợi. Một chương trình sẽ kiểm tra hàm *IsEmpty()* trước khi thử đọc mục khác từ đối tượng container. Một hàm tiện ích không là một phần của một giao diện của lớp. Hơn nữa nó là một hàm thành viên **private** mà hỗ trợ các thao tác của các hàm thành viên **public**. Các hàm tiện ích không dự định được sử dụng bởi các client của lớp.

Ví dụ 3.6: Minh họa cho các hàm tiện ích.



```
1: #include <iostream.h>
2: #include <iomanip.h>
3:
4: class SalesPerson
5: {
6: public:
7: SalesPerson(); //constructor
8: void SetSales(int, double); //Người dùng cung cấp các hình của
9: //những hàng bán của một
tháng
10: void PrintAnnualSales();
11:
12: private:
13: double Sales[12]; //12 hình của những hàng bán hằng tháng
14: double TotalAnnualSales(); //Hàm tiện ích
15: };
16:
17: //Hàm constructor khởi tạo mảng
18: SalesPerson::SalesPerson()
19: {
20: for (int I = 0; I < 12; I++)
21: Sales[I] = 0.0;
22: }
23:
24: //Hàm thiết lập một trong 12 hình của những hàng bán hằng
tháng
25: void SalesPerson::SetSales(int Month, double Amount)
26: {
27: if (Month >= 1 && Month <= 12 && Amount > 0)
28: Sales[Month - 1] = Amount;
```

Chúng ta [chạy ví dụ 3.6](#) , kết quả ở hình 3.6



Hình 3.6: Kết quả của ví dụ 3.6

3.7 KHỞI ĐỘNG CÁC ĐỐI TƯỢNG CỦA LỚP : CONSTRUCTOR

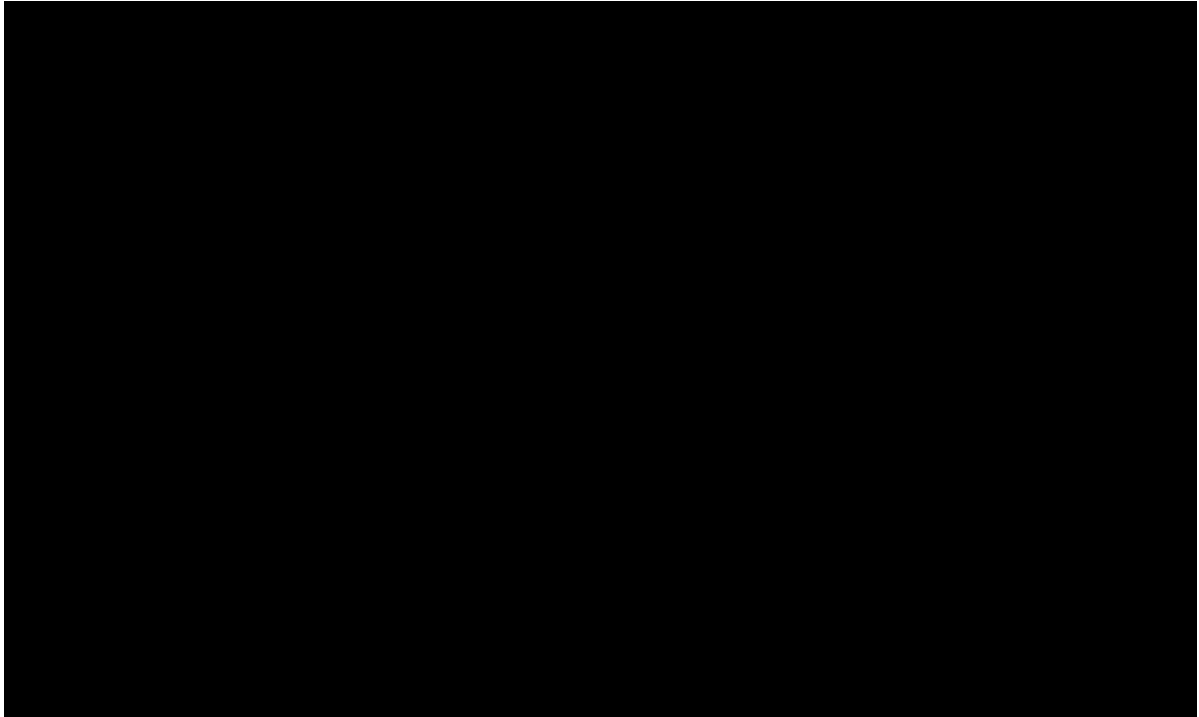
Khi một đối tượng được tạo, các thành viên của nó có thể được khởi tạo bởi một hàm constructor. Một constructor là một hàm thành viên với tên giống như tên của lớp. Lập trình viên cung cấp constructor mà được gọi tự động mỗi khi đối tượng của lớp đó được tạo. Các thành viên dữ liệu của một lớp không thể được khởi tạo trong định nghĩa của lớp. Hơn nữa, các thành viên dữ liệu phải được khởi động hoặc trong một constructor của lớp hoặc các giá trị của chúng có thể được thiết lập sau khi đối tượng được tạo. Các constructor không thể mô tả các kiểu trả về hoặc các giá trị trả về. Các constructor có thể được đa năng hóa để cung cấp sự đa dạng để khởi tạo các đối tượng của lớp.

Constructor có thể chứa các tham số mặc định. Bằng cách cung cấp các tham số mặc định cho constructor, ngay cả nếu không có các giá trị nào được cung cấp trong một constructor thì đối tượng vẫn được bảo đảm để trong một trạng thái phù hợp vì các tham số mặc định. Một constructor của lập trình viên cung cấp mà hoặc tất cả các tham số của nó có giá trị mặc định hoặc không có tham số nào được gọi là constructor mặc định (default constructor). Chỉ có thể có một constructor mặc định cho mỗi lớp.

[Ví dụ 3.7:](#) Constructor với các tham số mặc định

```
1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0); //Constructor mặc định
7: void SetTime(int, int, int);
8: void PrintMilitary();
9: void PrintStandard();
10:
11: private:
12: int Hour;
13: int Minute;
14: int Second;
15: };
16:
17: //Hàm constructor để khởi động dữ liệu private
18: //Các giá trị mặc định là 0
19: Time::Time(int Hr, int Min, int Sec)
20: {
21: SetTime(Hr, Min, Sec);
22: }
23:
24: //Thiết lập các giá trị của Hour, Minute và Second
25: //Giá trị không hợp lệ được thiết lập là 0
26: void Time::SetTime(int H, int M, int S)
27: {
28: Hour = (H >= 0 && H < 24) ? H : 0;
29: Minute = (M >= 0 && M < 60) ? M : 0;
```


Chương trình ở ví dụ 3.7 khởi tạo năm đối tượng của lớp *Time* (ở dòng 52). Đối tượng *T1* với ba tham số lấy giá trị mặc định, đối tượng *T2* với một tham số được mô tả, đối tượng *T3* với hai tham số được mô tả, đối tượng *T4* với ba tham số được mô tả và đối tượng *T5* với các tham số có giá trị không hợp lệ. Chúng ta [chạy ví dụ 3.7](#), kết quả ở hình 3.7



Hình 3.7: Kết quả của ví dụ 3.7

Nếu không có constructor nào được định nghĩa trong một lớp thì trình biên dịch tạo một constructor mặc định. Constructor này không thực hiện bất kỳ sự khởi tạo nào, vì vậy khi đối tượng được tạo, nó không bảo đảm ở trong một trạng thái phù hợp.

3.8 SỬ DỤNG DESTRUCTOR

Một destructor là một hàm thành viên đặc biệt của một lớp. Tên của destructor đối với một lớp là ký tự ngã (~) theo sau bởi tên lớp. Destructor của một lớp được gọi khi đối tượng được hủy bỏ nghĩa là khi sự thực hiện chương trình rời khỏi phạm vi mà trong đó đối tượng của lớp đó được khởi tạo. Destructor không thực sự hủy bỏ đối tượng – nó thực hiện "công việc nội trợ kết thúc" trước khi hệ thống phục hồi không gian bộ nhớ của đối tượng để nó có thể được sử dụng giữ các đối tượng mới. Một destructor không nhận các tham số và không trả về giá trị. Một lớp chỉ có duy nhất một destructor – đa năng hóa destructor là không cho phép. Nếu trong một lớp không có định nghĩa một destructor thì trình biên dịch sẽ tạo một destructor mặc định không làm gì cả.

[Ví dụ 3.8](#): Lớp có hàm destructor

```
1: #include <iostream.h>
2:
3: class Simple
4: {
5: private:
6: int *X;
7: public:
8: Simple(); //Constructor
9: ~Simple(); //Destructor
10: void SetValue(int V);
11: int GetValue();
12: };
13:
14: Simple::Simple()
15: {
16: X = new int; //Cấp phát vùng nhớ cho X
17: }
18:
19: Simple::~~Simple()
20: {
21: delete X; //Giải phóng vùng nhớ khi đối tượng bị hủy bỏ.
22: }
23:
24: void Simple::SetValue(int V)
25: {
26: *X = V;
27: }
28:
29: int Simple::GetValue()
```

Chúng ta [chạy ví dụ 3.8](#), kết quả ở hình 3.8



Hình 3.8: Kết quả của ví dụ 3.8

3.9 KHI NÀO CÁC CONSTRUCTOR VÀ DESTRUCTOR ĐƯỢC GỌI ?

Các constructor và destructor được gọi một cách tự động. Thứ tự các hàm này được gọi phụ thuộc vào thứ tự trong đó sự thực hiện vào và rời khỏi phạm vi mà các đối tượng được khởi tạo. Một cách tổng quát, các destructor được gọi theo thứ tự ngược với thứ tự của các constructor được gọi. Các constructor được gọi của các đối tượng khai báo trong phạm vi toàn cục trước bất kỳ hàm nào (bao gồm hàm **main()**) trong file mà bắt đầu thực hiện. Các destructor tương ứng được gọi khi hàm **main()** kết thúc hoặc hàm **exit()** được gọi. Các constructor của các đối tượng cục bộ tự động được gọi khi sự thực hiện đến điểm mà các đối tượng được khai báo. Các destructor tương ứng được gọi khi các đối tượng rời khỏi phạm vi (nghĩa là khối mà trong đó chúng được khai báo). Các constructor và destructor đối với các đối tượng cục bộ tự động được gọi mỗi khi các đối tượng vào và rời khỏi phạm vi. Các constructor được gọi của các đối tượng cục bộ tĩnh (static) khi sự thực hiện đến điểm mà các đối tượng được khai báo lần đầu tiên. Các destructor tương ứng được gọi khi hàm **main()** kết thúc hoặc hàm **exit()** được gọi.

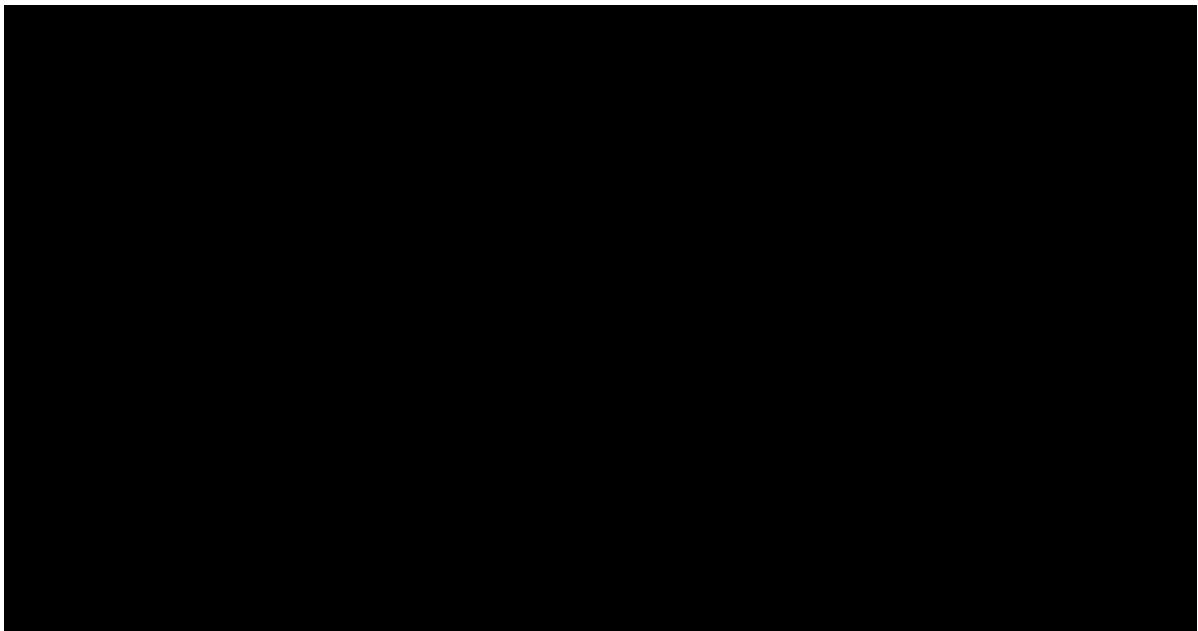
[Ví dụ 3.9](#): Chương trình sau minh



```
1: #include <iostream.h>
2:
3: class CreateAndDestroy
4: {
5: public:
6: CreateAndDestroy(int); //Constructor
7: ~CreateAndDestroy(); //Destructor
8: private:
9: int Data;
10: };
11:
12: CreateAndDestroy::CreateAndDestroy(int Value)
13: {
14: Data = Value;
15: cout << "Object " << Data << " constructor";
16: }
17:
18: CreateAndDestroy::~~CreateAndDestroy()
19: {
20: cout << "Object " << Data << " destructor " << endl;
21: }
22:
23: void Create(void); //Prototype
24: CreateAndDestroy First(1); //Đối tượng toàn cục
25:
26: int main()
27: {
28: cout << " (global created before main)" << endl;
29:
```

Chương trình khai báo *First* ở phạm vi toàn cục. Constructor của nó được gọi khi chương trình bắt đầu thực hiện và destructor của nó được gọi lúc chương trình kết thúc sau tất cả các đối tượng khác được hủy bỏ. Hàm *main()* khai báo ba đối tượng. Các đối tượng *Second* và *Fourth* là các đối tượng cục bộ tự động và đối tượng *Third* là một đối tượng cục bộ tĩnh. Các constructor của các đối tượng này được gọi khi chương trình thực hiện đến điểm mà mỗi đối tượng được khai báo. Các destructor của các đối tượng *Fourth* và *Second* được gọi theo thứ tự này khi kết thúc của **main()** đạt đến. Vì đối tượng *Third* là tĩnh, nó tồn tại cho đến khi chương trình kết thúc. Destructor của đối tượng *Third* được gọi trước destructor của *First* nhưng sau tất cả các đối tượng khác được hủy bỏ.

Hàm *Create()* khai báo ba đối tượng – *Fifth* và *Seventh* là các đối tượng cục bộ tự động và *Sixth* là một đối tượng cục bộ tĩnh. Các destructor của các đối tượng *Seventh* và *Fifth* được gọi theo thứ tự này khi kết thúc của *create()* đạt đến. Vì đối tượng *Sixth* là tĩnh, nó tồn tại cho đến khi chương trình kết thúc. Destructor của đối tượng *Sixth* được gọi trước các destructor của *Third* và *First* nhưng sau tất cả các đối tượng khác được hủy bỏ. Chúng ta [chạy ví dụ 3.9](#), kết quả ở hình 3.9



Hình 3.9: Kết quả của ví dụ 3.9

3.10 SỬ DỤNG CÁC THÀNH VIÊN DỮ LIỆU VÀ CÁC HÀM THÀNH VIÊN

Các thành viên dữ liệu **private** chỉ có thể được xử lý bởi các hàm thành viên (hay hàm **friend**) của lớp. Các lớp thường cung cấp các hàm thành viên **public** để cho phép các client của lớp để thiết lập (set) (nghĩa là "ghi") hoặc lấy (get) (nghĩa là "đọc") các giá trị của các thành viên dữ liệu **private**. Các hàm này thường không cần phải được gọi "set" hay "get", nhưng chúng thường đặt tên như vậy. Chẳng hạn, một lớp có thành viên dữ liệu **private** có tên *InterestRate*, hàm thành viên thiết lập giá trị có tên là *SetInterestRate()* và hàm thành viên lấy giá trị có tên là *GetInterestRate()*. Các hàm "Get" cũng thường được gọi là các hàm chất vấn (query functions).

Nếu một thành viên dữ liệu là **public** thì thành viên dữ liệu có thể được đọc hoặc ghi tại bất kỳ hàm nào trong chương trình. Nếu một thành viên dữ liệu là **private**, một hàm "get" public nhất định cho phép các hàm khác để đọc dữ liệu nhưng hàm get có thể điều khiển sự định dạng và hiển thị của dữ liệu. Một hàm "set" public có thể sẽ xem xét cẩn thận bất kỳ cố gắng nào để thay đổi giá trị của thành viên dữ liệu. Điều

này sẽ bảo đảm rằng giá trị mới thì tương thích đối với mục dữ liệu. Chẳng hạn, một sự cố gắng thiết lập ngày của tháng là 37 sẽ bị loại trừ.

Các lợi ích của sự toàn vẹn dữ liệu thì không tự động đơn giản bởi vì các thành viên dữ liệu được tạo là **private** – lập trình viên phải cung cấp sự kiểm tra hợp lệ. Tuy nhiên C++ cung cấp một khung làm việc trong đó các lập trình viên có thể thiết kế các chương trình tốt hơn.

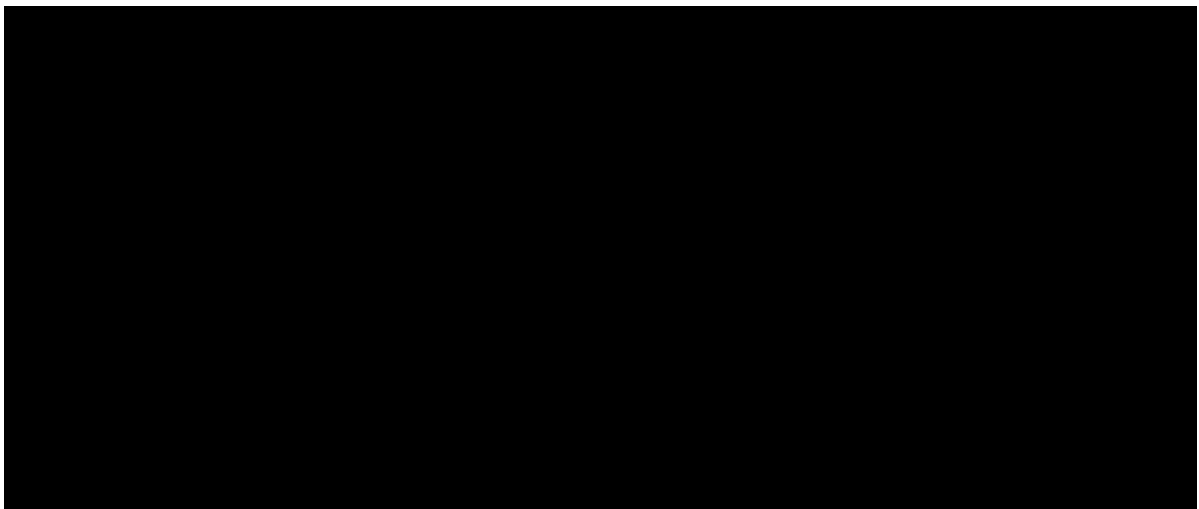
Client của lớp phải được thông báo khi một sự cố gắng được tạo ra để gán một giá trị không hợp lệ cho một thành viên dữ liệu. Chính vì lý do này, các hàm "set" của lớp thường được viết trả về các giá trị cho biết rằng một sự cố gắng đã tạo ra để gán một dữ liệu không hợp lệ cho một đối tượng của lớp. Điều này cho phép các client của lớp kiểm tra các giá trị trả về để xác định nếu đối tượng mà chúng thao tác là một đối tượng hợp lệ và để bắt giữ hoạt động thích hợp nếu đối tượng mà chúng thao tác thì không phải hợp lệ.

Ví dụ 3.10: Chương trình mở rộng lớp *Time* ở ví dụ 3.2 bao gồm hàm get và set đối với các thành viên dữ liệu private là *hour*, *minute* và *second*.

```
128: cout << endl << "Minute + 1: ";  
  
129: TT.PrintStandard();  
  
130: }  
  
131: cout << endl;  
  
132: }
```

Trong ví dụ trên chúng ta có hàm *IncrementMinutes()* là hàm dùng để tăng *Minute*. Đây là hàm không thành viên mà sử dụng các hàm thành viên get và set để tăng thành viên *Minute*.

Chúng ta chạy ví dụ .10, kết quả ở hình 3.10



Hình 3.10: Kết quả của ví dụ 3.10

3.11 TRẢ VỀ MỘT THAM CHIẾU TỚI MỘT THÀNH VIÊN DỮ LIỆU PRIVATE

Một tham chiếu tới một đối tượng là một bí danh của chính đối tượng đó và do đó có thể được sử dụng ở vế trái của phép gán. Trong khung cảnh đó, tham chiếu tạo một lvalue được chấp nhận hoàn toàn mà có thể nhận một giá trị. Một cách để sử dụng khả năng này (thật không may!) là có một hàm thành viên **public** của lớp trả về một tham chiếu không **const** tới một thành viên dữ liệu **private** của lớp đó.

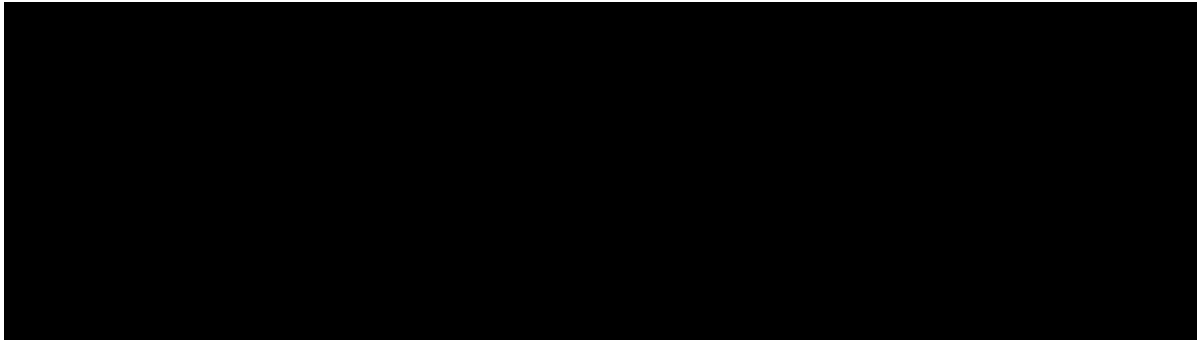
Ví dụ 3.11: Chương trình sau sử dụng một phiên bản đơn giản của lớp *Time* để minh họa trả về một tham chiếu tới một dữ liệu **private**.

```
1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0);
7: void SetTime(int, int, int);
8: int GetHour();
9: int &BadSetHour(int); //Nguy hiểm trả về tham chiếu !!!
10: private:
11: int Hour;
12: int Minute;
13: int Second;
14: };
15:
16: //Constructor khởi động dữ liệu private
17: //Gọi hàm thành viên SetTime() để thiết lập các biến
18: //Các giá trị mặc định là 0
19: Time::Time(int Hr, int Min, int Sec)
20: {
21: SetTime(Hr, Min, Sec);
22: }
23: //Thiết lập các giá trị của Hour, Minute, và Second
24: void Time::SetTime(int H, int M, int S)
25: {
26: Hour = (H >= 0 && H < 24) ? H : 0;
27: Minute = (M >= 0 && M < 60) ? M : 0;
28: Second = (S >= 0 && S < 60) ? S : 0;
29: }
```




Trong chương trình hàm *BadSetHour()* trả về một tham chiếu tới thành viên dữ liệu *Hour*.

Chúng ta [chạy ví dụ 3.11](#), kết quả ở hình 3.11



Hình 3.11: Kết quả của ví dụ 3.11

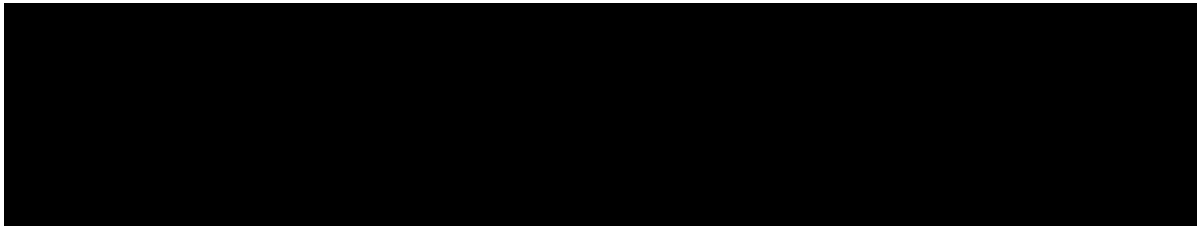
3.12 PHÉP GÁN BỞI TOÁN TỬ SAO CHÉP THÀNH VIÊN MẶC ĐỊNH

Toán tử gán (=) được sử dụng để gán một đối tượng cho một đối tượng khác của cùng một kiểu. Toán tử gán như thế bình thường được thực hiện bởi toán tử sao chép thành viên (Memberwise copy) – Mỗi thành viên của một đối tượng được sao chép riêng rẽ tới cùng thành viên ở đối tượng khác (Chú ý rằng sao chép thành viên có thể phát sinh các vấn đề nghiêm trọng khi sử dụng với một lớp mà thành viên dữ liệu chứa vùng nhớ cấp phát động). Các đối tượng có thể được truyền cho các tham số của hàm và có thể được trả về từ các hàm. Như thế việc truyền và trả về được thực hiện theo truyền giá trị – một sao chép của đối tượng được truyền hay trả về.: [Ví dụ 3.12](#): Chương trình sau minh họa toán tử sao chép thành viên mặc định



```
1: #include <iostream.h>
2:
3: //Lớp Date đơn giản
4: class Date
5: {
6: public:
7: Date(int = 1, int = 1, int = 1990); //Constructor mặc định
8: void Print();
9: private:
10: int Month;
11: int Day;
12: int Year;
13: };
14:
15: //Constructor Date đơn giản với việc không kiểm tra miền
16: Date::Date(int m, int d, int y)
17: {
18: Month = m;
19: Day = d;
20: Year = y;
21: }
22:
23: //In Date theo dạng mm-dd-yyyy
24: void Date::Print()
25: {
26: cout << Month << '-' << Day << '-' << Year;
27: }
28:
29: int main()
```

Chúng ta [chạy ví dụ 3.12](#), kết quả ở hình 3.12



Hình 3.12: Kết quả của ví dụ 3.12

3.13 CÁC ĐỐI TƯỢNG HẰNG VÀ CÁC HÀM THÀNH VIÊN CONST

Một vài đối tượng cần được thay đổi và một vài đối tượng thì không. Lập trình viên có thể sử dụng từ khóa **const** để cho biết đối tượng không thể thay đổi được, và nếu có cố gắng thay đổi đối tượng thì xảy ra lỗi.

Chẳng hạn: `const Time Noon(12,0,0);` //Khai báo một đối tượng **const**

Các trình biên dịch C++ lưu ý đến các khai báo **const** vì thế các trình biên dịch cấm hoàn toàn bất kỳ hàm thành viên nào gọi các đối tượng **const** (Một vài trình biên dịch chỉ cung cấp một cảnh báo). Điều này thì khác nghiệt bởi vì các client của đối tượng hầu như chắc chắn sẽ muốn sử dụng các hàm thành viên "get" khác nhau với đối tượng, và tất nhiên không thể thay đổi đối tượng. Để cung cấp cho điều này, lập trình viên có thể khai báo các hàm thành viên **const**; điều này chỉ có thể thao tác trên các đối tượng **const**. Dĩ nhiên các hàm thành viên **const** không thể thay đổi đối tượng - trình biên dịch cấm điều này. Một hàm được mô tả như **const** khi cả hai trong phần khai báo và trong phần định nghĩa của nó được chèn thêm từ khóa **const** sau danh sách các tham số của hàm, và trong trường hợp của định nghĩa hàm trước dấu ngoặc móc trái ({) mà bắt đầu thân hàm. Chẳng hạn, hàm thành viên của lớp *A* nào đó:

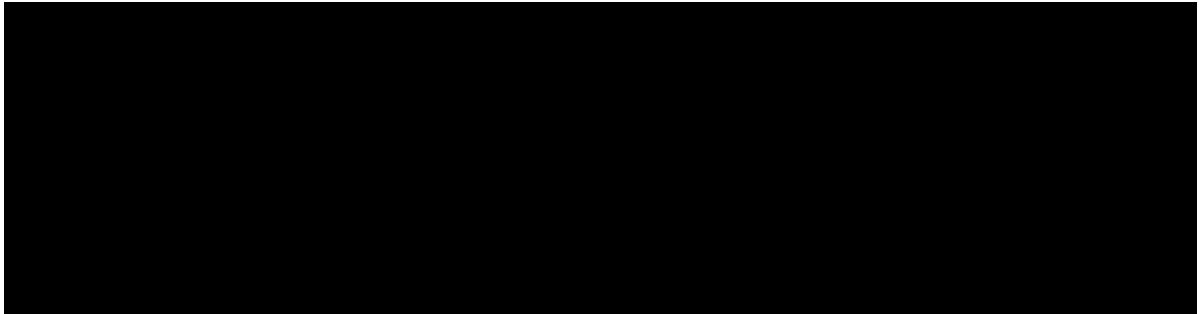
```
int A::GetValue() const
{
    return PrivateDataMember;
}
```

Nếu một hàm thành viên **const** được định nghĩa bên ngoài định nghĩa của lớp thì khai báo hàm và định nghĩa hàm phải bao gồm **const** ở mỗi phần. Một vấn đề nảy sinh ở đây đối với các constructor và destructor, mỗi hàm thường cần thay đổi đối tượng. Khai báo **const** không yêu cầu đối với các constructor và destructor của các đối tượng **const**. Một constructor phải được phép thay đổi một đối tượng mà đối tượng có thể được khởi tạo thích hợp. Một destructor phải có khả năng thực hiện vai trò "công việc kết thúc nội trợ" trước khi đối tượng được hủy.

[Ví dụ 3.13](#): Chương trình sau sử dụng một lớp *Time* với các đối tượng **const** và các hàm thành viên **const**.

```
1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0); //Constructor mặc định
7: //Các hàm set
8: void SetTime(int, int, int); //Thiết lập thời gian
9: void SetHour(int); //Thiết lập Hour
10: void SetMinute(int); //Thiết lập Minute
11: void SetSecond(int); //Thiết lập Second
12: //Các hàm get
13: int GetHour() const; //Trả về Hour
14: int GetMinute() const; //Trả về Minute
15: int GetSecond() const; //Trả về Second
16: //Các hàm in
17: void PrintMilitary() const; //In thời gian theo dạng giờ quân
    đội
18: void PrintStandard() const; //In thời gian theo dạng giờ
    chuẩn
19: private:
20: int Hour; //0 - 23
21: int Minute; //0 - 59
22: int Second; //0 - 59
23: };
24:
25: //Constructor khởi động dữ liệu private
26: //Các giá trị mặc định là 0
27: Time::Time(int hr, int min, int sec)
28: {
```

Chương trình này khai báo một đối tượng hằng của lớp *Time* và cố gắng sửa đổi đối tượng với các hàm thành viên không hằng *SetHour()*, *SetMinute()* và *SetSecond()*. Các lỗi cảnh báo được phát sinh bởi trình biên dịch (Borland C++) như hình 3.13.



Hình 3.13: Các cảnh báo của chương trình ở ví dụ 3.13

Lưu ý: Hàm thành viên **const** có thể được đa năng hóa với một phiên bản non-const. Việc lựa chọn hàm thành viên đa năng hóa nào để sử dụng được tạo một cách tự động bởi trình biên dịch dựa vào nơi mà đối tượng được khai báo **const** hay không. Một đối tượng **const** không thể được thay đổi bởi phép gán vì thế nó phải được khởi động. Khi một thành viên dữ liệu của một lớp được khai báo **const**, một bộ khởi tạo thành viên (member initializer) phải được sử dụng để cung cấp cho constructor với giá trị ban đầu của thành viên dữ liệu đối với một đối tượng của lớp.

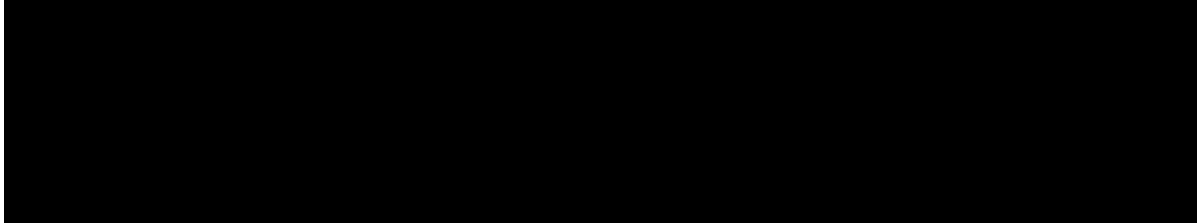
Ví dụ 3.14: Chương trình sau sử dụng một bộ khởi tạo thành viên để khởi tạo một hằng của kiểu dữ liệu có sẵn.



```
1: #include <iostream.h>
2:
3: class IncrementClass
4: {
5: public:
6: IncrementClass (int C = 0, int I = 1);
7: void AddIncrement()
8: {
9: Count += Increment;
10: }
11: void Print() const;
12: private:
13: int Count;
14: const int Increment; //Thành viên dữ liệu const
15: };
16:
17: //Constructor của lớp IncrementClass
18: //Bộ khởi tạo với thành viên const
19: IncrementClass::IncrementClass (int C, int I) : Increment(I)
20: {
21: Count = C;
22: }
23:
24: //In dữ liệu
25: void IncrementClass::Print() const
26: {
27: cout << "Count = " << Count
28: #    #    << ", Increment = " << Increment << endl;
30: }
```

Chương trình này sử dụng cú pháp bộ khởi tạo thành viên để khởi tạo thành viên dữ liệu **const** *Increment* của lớp *IncrementClass* ở dòng 19.

Chúng ta [chạy ví dụ 3.14](#), kết quả ở hình 3.14

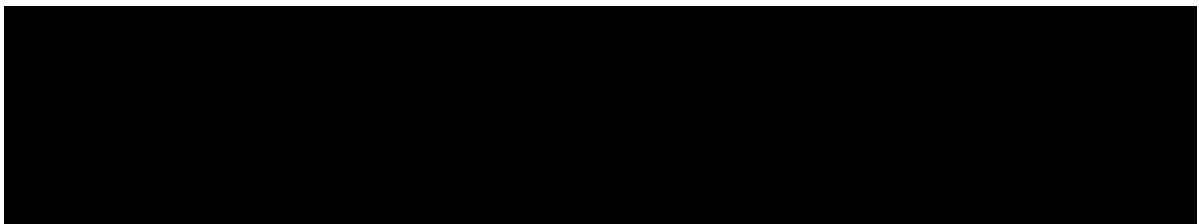


Hình 3.14: Kết quả của ví dụ 3.14

Ký hiệu : *Increment(I)* (ở dòng 19 của ví dụ 3.14) sinh ra *Increment* được khởi động với giá trị là *I*. Nếu nhiều bộ khởi tạo thành viên được cần, đơn giản bao gồm chúng trong danh sách phân cách dấu phẩy sau dấu hai chấm. Tất cả các thành viên dữ liệu có thể được khởi tạo sử dụng cú pháp bộ khởi tạo thành viên. Nếu trong [ví dụ 3.14](#) chúng ta cố gắng khởi tạo *Increment* với một lệnh gán hơn là với một bộ khởi tạo thành viên như sau:

```
IncrementClass::IncrementClass (int C, int I)
{
    Count = C;
    Increment = I;
}
```

Khi đó trình biên dịch (Borland C++) sẽ có thông báo lỗi như sau:



Hình 3.15: Thông báo lỗi khi cố gắng khởi tạo một thành viên dữ liệu **const** bằng phép gán

3.14 LỚP NHƯ LÀ CÁC THÀNH VIÊN CỦA CÁC LỚP KHÁC

Một lớp có thể có các đối tượng của các lớp khác như các thành viên. Khi một đối tượng đi vào phạm vi, constructor của nó được gọi một cách tự động, vì thế chúng ta cần mô tả các tham số được truyền như thế nào tới các constructor của đối tượng thành viên. Các đối tượng thành viên được xây dựng theo thứ tự mà trong đó chúng được khai báo (không theo thứ tự mà chúng được liệt kê trong danh sách bộ khởi tạo thành viên của constructor) và trước các đối tượng của lớp chứa đựng chúng được xây dựng.

[Ví dụ 3.15](#): Chương trình sau minh họa các đối tượng như các thành viên của các đối tượng khác.

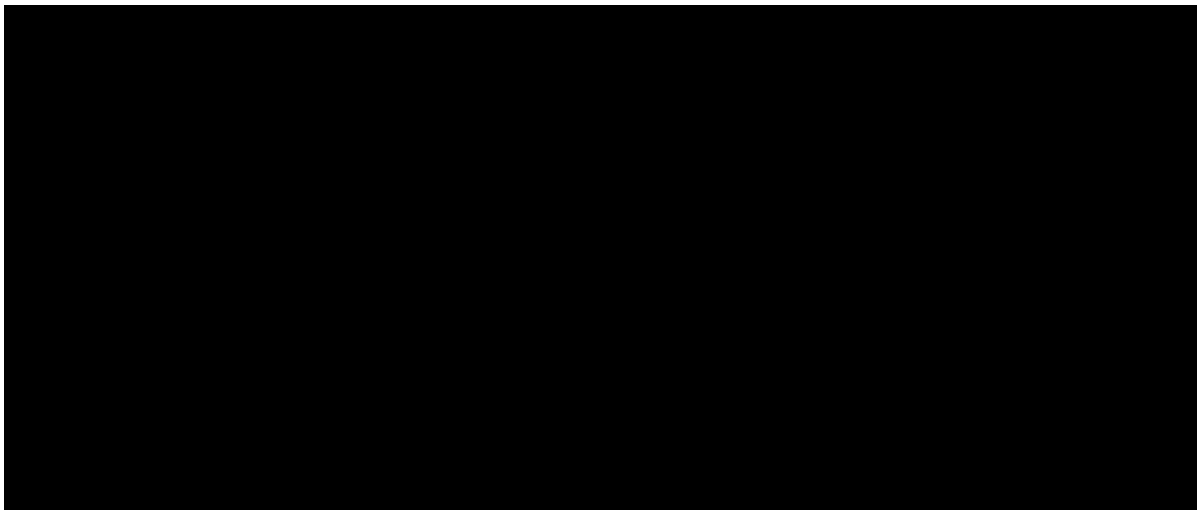



```
1: #include <iostream.h>
2: #include <string.h>
3:
4: class Date
5: {
6: public:
7: Date(int = 1, int = 1, int = 1900); //Constructor mặc định
8: void Print() const; //In ngày theo dạng Month/Day/Year
9: private:
10: int Month; //1-12
11: int Day; //1-31
12: int Year; //Năm bất kỳ
13: //Hàm tiện ích để kiểm tra Day tương thích đối với Month và Year
14: int CheckDay(int);
15: };
16:
17: class Employee
18: {
19: public:
20: Employee(char *, char *, int, int, int, int, int, int);
21: void Print() const;
22: private:
23: char LastName[25];
24: char FirstName[25];
25: Date BirthDate;
26: Date HireDate;
27: };
28:
```

Chương trình gồm lớp *Employee* chứa các thành viên dữ liệu **private** *LastName*, *FirstName*, *BirthDate* và *HireDate*. Các thành viên *BirthDate* và *HireDate* là các đối tượng của lớp *Date* mà chứa các thành viên dữ liệu **private** *Month*, *Day* và *Year*. Chương trình khởi tạo một đối tượng *Employee*, và các khởi tạo và các hiển thị các thành viên dữ liệu của nó. Chú ý về cú pháp của phần đầu trong định nghĩa constructor của lớp *Employee*:

```
Employee::Employee(char *FName, char *LName,
int BMonth, int BDay, int BYear,
int HMonth, int HDay, int HYear)
:BirthDate(BMonth, BDay, BYear), HireDate(HMonth, HDay, HYear)
```

Constructor lấy tám tham số (*FName*, *LName*, *BMonth*, *BDay*, *BYear*, *HMonth*, *HDay*, và *HYear*). Dấu hai chấm trong phần đầu phân tách các bộ khởi tạo từ danh sách các tham số. Các bộ khởi tạo định rõ các tham số truyền cho constructor của các đối tượng thành viên. Vì thế *BMonth*, *BDay* và *BYear* được truyền cho constructor của đối tượng *BirthDate*, và *HMonth*, *HDay*, và *HYear* được truyền cho constructor của đối tượng *HireDate*. Nhiều bộ khởi tạo được phân tách bởi dấu phẩy. Chúng ta [chạy ví dụ 3.15](#), kết quả ở hình 3.16



Hình 3.16: Kết quả của ví dụ 3.15

Một đối tượng thành viên không cần được khởi tạo thông qua một bộ khởi tạo thành viên. Nếu một bộ khởi tạo thành viên không được cung cấp, constructor mặc định của đối tượng thành viên sẽ được gọi một cách tự động. Các giá trị nếu có thiết lập bởi constructor mặc định thì có thể được ghi đè bởi các hàm set.

3.15 CÁC HÀM VÀ CÁC LỚP FRIEND

Một hàm **friend** của một lớp được định nghĩa bên ngoài phạm vi của lớp đó, lúc này có quyền truy cập đến các thành viên **private** hoặc **protected** của một lớp. Một hàm hay toàn bộ lớp có thể được khai báo là một **friend** của lớp khác.

Để khai báo một hàm là một **friend** của một lớp, đứng trước prototype của hàm trong định nghĩa lớp với từ khóa **friend**, như sau:

```
friend <function-declarator>;
```

Để khai báo một lớp là **friend** của lớp khác như sau:

```
friend <class-name>; Ví dụ 3.16: Chương trình sau minh họa khai báo và sử dụng hàm friend.
```

```
1: #include <iostream.h>
2:
3: class Count
4: {
5: friend void SetX(Count &, int); //Khai báo friend
6: public:
7: Count()//Constructor
8: {
9: X = 0;
10: }
11: void Print() const //Xuất
12: {
13: cout << X << endl;
14: }
15: private:
16: int X;
17: };
18:
19: //Có thể thay đổi dữ liệu private của lớp Count vì
20: //SetX() khai báo là một hàm friend của lớp Count
21: void SetX(Count &C, int Val)
22: {
23: C.X = Val; //Hợp lệ: SetX() là một hàm friend của lớp Count
24: }
25:
26: int main()
27: {
28: Count Object;
29:
```

Chúng ta [chạy ví dụ 3.16](#), kết quả ở hình 3.17



Hình 3.17: Kết quả của ví dụ 3.16

Có thể chỉ định các hàm được đa năng hóa là các **friend** của lớp. Mỗi hàm được đa năng hóa phải được khai báo tường minh trong định nghĩa lớp như là một **friend** của lớp.

3.16 CON TRỎ THIS

Khi một hàm thành viên tham chiếu thành viên khác của lớp cho đối tượng cụ thể của lớp đó, làm thế nào C++ bảo đảm rằng đối tượng thích hợp được tham chiếu? Câu trả lời là mỗi đối tượng duy trì một con trỏ trỏ tới chính nó – gọi là con trỏ **this** – Đó là một tham số ẩn trong tất cả các tham chiếu tới các thành viên bên trong đối tượng đó. Con trỏ **this** cũng có thể được sử dụng tường minh. Mỗi đối tượng có thể xác định địa chỉ của chính mình bằng cách sử dụng từ khóa **this**.

Con trỏ **this** được sử dụng để tham chiếu cả các thành viên dữ liệu và hàm thành viên của một đối tượng. Kiểu của con trỏ **this** phụ thuộc vào kiểu của đối tượng và trong hàm thành viên con trỏ **this** được sử dụng là khai báo **const**. Chẳng hạn, một hàm thành viên không hằng của lớp *Employee* con trỏ **this** có kiểu là:

`Employee * const //Con trỏ hằng trỏ tới đối tượng Employee`

Đối với một hàm thành viên hằng của lớp *Employee* con trỏ **this** có kiểu là:

`const Employee * const //Con trỏ hằng trỏ tới đối tượng Employee mà là một hằng`

Ví dụ 3.17: Chương trình sau minh họa sử dụng tường minh của con trỏ **this** để cho phép một hàm thành viên của lớp *Test* in dữ liệu *X* của một đối tượng *Test*.



```
1: #include <iostream.h>
2:
3: class Test
4: {
5: public:
6: Test(int = 0); // Constructor mặc định
7: void Print() const;
8: private:
9: int X;
10: };
11:
12: Test::Test(int A)
13: {
14: X = A;
15: }
16:
17: void Test::Print() const
18: {
19: cout << " X = " << X << endl
20:      << " this->X = " << this->X << endl
21:      << " (*this).X = " << (*this).X << endl;
22: }
23:
24: int main()
25: {
26: Test A(12);
27:
28: A.Print();
29: return 0;
```

Chúng ta [chạy ví dụ 3.17](#), kết quả ở hình 3.18



Hình 3.18: Kết quả của ví dụ 3.17

Một cách khác sử dụng con trỏ **this** là cho phép móc vào nhau các lời gọi hàm thành viên.

[Ví dụ 3.18](#): Chương trình sau minh họa trả về một tham chiếu tới một đối tượng *Time* để cho phép các lời gọi hàm thành viên của lớp *Time* được móc nối vào nhau.



```

1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0); // Constructor mặc định
7: // Các hàm set
8: Time &SetTime(int, int, int); // Thiết lập Hour, Minute và
   Second
9: Time &SetHour(int); // Thiết lập Hour
10: Time &SetMinute(int); // Thiết lập Minute
11: Time &SetSecond(int); // Thiết lập Second
12: // Các hàm get
13: int GetHour() const; // Trả về Hour
14: int GetMinute() const; // Trả về Minute
15: int GetSecond() const; // Trả về Second
16: // Các hàm in
17: void PrintMilitary() const; // In thời gian theo dạng giờ
   quân đội
18: void PrintStandard() const; // In thời gian theo dạng giờ
   chuẩn
19: private:
20: int Hour; // 0 - 23
21: int Minute; // 0 - 59
22: int Second; // 0 - 59
23: };
24:
25: // Constructor khởi động dữ liệu private
26: // Gọi hàm thành viên SetTime() để thiết lập các biến
27: // Các giá trị mặc định là 0

```

Các hàm thành viên *SetTime()*, *SetHour()*, *SetMinute()* và *SetSecond()* mỗi hàm đều trả về ***this** với kiểu trả về là *Time &*. Toán tử chấm liên kết từ trái sang phải, vì vậy biểu thức:

```
T.SetHour(18).SetMinute(30).SetSecond(22);
```

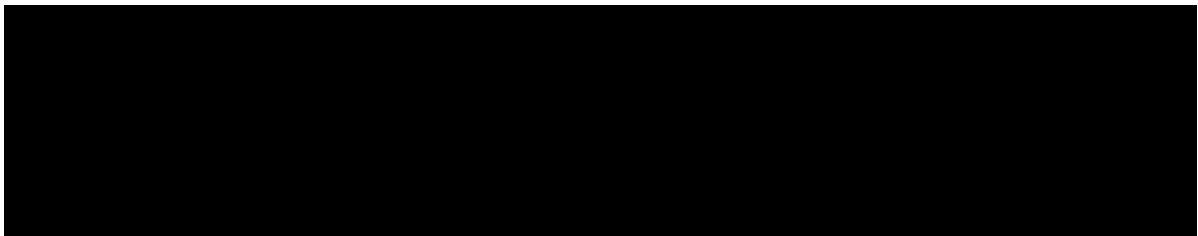
Đầu tiên gọi *T.SetHour(18)* thì trả về một tham chiếu tới đối tượng *T* là giá trị của lời gọi hàm này. Phần còn lại của biểu thức được hiểu như sau:

```
T.SetMinute(30).SetSecond(22);
```

T.SetMinute(30) gọi thực hiện và trả về tương đương của *T*. Phần còn lại của biểu thức là:

```
T.SetSecond(22);
```

Chúng ta [chạy ví dụ 3.18](#), kết quả ở hình 3.19



Hình 3.19: Kết quả của ví dụ 3.18

3.17 CÁC ĐỐI TƯỢNG ĐƯỢC CẤP PHÁT ĐỘNG

Các đối tượng có thể được cấp phát động giống như các dữ liệu khác bằng toán tử **new** và **delete**. Chẳng hạn:

```
Time *TimePtr = new Time(1,20,26);
```

```
.....
```

```
delete TimePtr;
```

Toán tử **new** tự động gọi hàm constructor ,và toán tử **delete** tự động gọi destructor.

3.18 CÁC THÀNH VIÊN TÍNH CỦA LỚP

Bình thường, mỗi đối tượng của một lớp có bản sao chép của chính nó của tất cả các thành viên dữ liệu của lớp. Trong các trường hợp nhất định chỉ có duy nhất một bản chép thành viên dữ liệu đặc biệt cần phải dùng chung bởi tất cả các đối tượng của một lớp. Một thành viên dữ liệu tĩnh được sử dụng cho những điều đó và các lý do khác. Một thành viên dữ liệu tĩnh biểu diễn thông tin toàn lớp (class-wide). Khai báo một thành viên tĩnh bắt đầu với từ khóa **static**.

Mặc dù các thành viên dữ liệu tĩnh có thể giống như các biến toàn cục, các thành viên dữ liệu tĩnh có phạm vi lớp. Các thành viên tĩnh có thể là **public**, **private** hoặc **protected**. Các thành viên dữ liệu tĩnh phải được khởi tạo một lần (và chỉ một lần) tại phạm vi file. Các thành viên lớp tĩnh **public** có thể được truy cập thông qua bất kỳ đối tượng nào của lớp đó, hoặc chúng có thể được truy cập thông qua tên lớp sử dụng toán tử định phạm vi. Các thành viên lớp tĩnh **private** và **protected** phải được truy cập thông qua các hàm thành viên **public** của lớp hoặc thông qua các **friend** của lớp. Các thành viên lớp tĩnh tồn tại ngay cả khi đối tượng của lớp đó không tồn tại. Để truy cập một thành viên lớp tĩnh **public** khi các đối tượng của lớp không tồn tại, đơn giản thêm vào đầu tên lớp và toán tử định phạm vi cho thành viên dữ liệu. Để truy cập một thành viên lớp tĩnh **private** hoặc **protected** khi các đối tượng của lớp không tồn tại, một hàm thành viên **public** phải được cung cấp và hàm phải được gọi bởi thêm vào đầu tên của nó với tên lớp và toán tử định phạm vi.

Ví dụ 3.19: Chương trình sau minh họa việc sử dụng thành viên dữ liệu tĩnh **private** và một hàm thành viên tĩnh **public**.

```
1: #include <iostream.h>
2: #include <string.h>
3: #include <assert.h>
4:
5: class Employee
6: {
7: public:
8: Employee(const char*, const char*); // Constructor
9: ~Employee(); // Destructor
10: char *GetFirstName() const; // Trả về first name
11: char *GetLastName() const; // Trả về last name
12: // Hàm thành viên tĩnh
13: static int GetCount(); // Trả về số đối tượng khởi tạo
14: private:
15: char *FirstName;
16: char *LastName;
17: // static data member
18: static int Count; // Số đối tượng khởi tạo
19: };
20:
21: // Khởi tạo thành viên dữ liệu tĩnh
22: int Employee::Count = 0;
23:
24: // Định nghĩa hàm thành viên tĩnh mà trả về số đối tượng khởi
   tạo
25: int Employee::GetCount()
26: {
27: return Count;
28: }
```

Thành viên dữ liệu *Count* được khởi tạo là zero ở phạm vi file với lệnh:

```
int Employee::Count = 0;
```

Thành viên dữ liệu *Count* duy trì số các đối tượng của lớp *Employee* đã được khởi tạo. Khi đối tượng của lớp *Employee* tồn tại, thành viên *Count* có thể được tham chiếu thông qua bất kỳ hàm thành viên nào của một đối tượng *Employee* – trong ví dụ này, *Count* được tham chiếu bởi cả constructor lẫn destructor. Khi các đối tượng của lớp *Employee* không tồn tại, thành viên *Count* có thể vẫn được tham chiếu nhưng chỉ thông qua một lời gọi hàm thành viên tĩnh public *GetCount()* như sau:

```
Employee::GetCount()
```

Hàm *GetCount()* được sử dụng để xác định số các đối tượng của *Employee* khởi tạo hiện hành. Chú ý rằng khi không có các đối tượng trong chương trình, lời gọi hàm *Employee::GetCount()* được đưa ra. Tuy nhiên khi có các đối tượng khởi động hàm *GetCount()* có thể được gọi thông qua một trong các đối tượng như sau:

```
E1Ptr->GetCount()
```

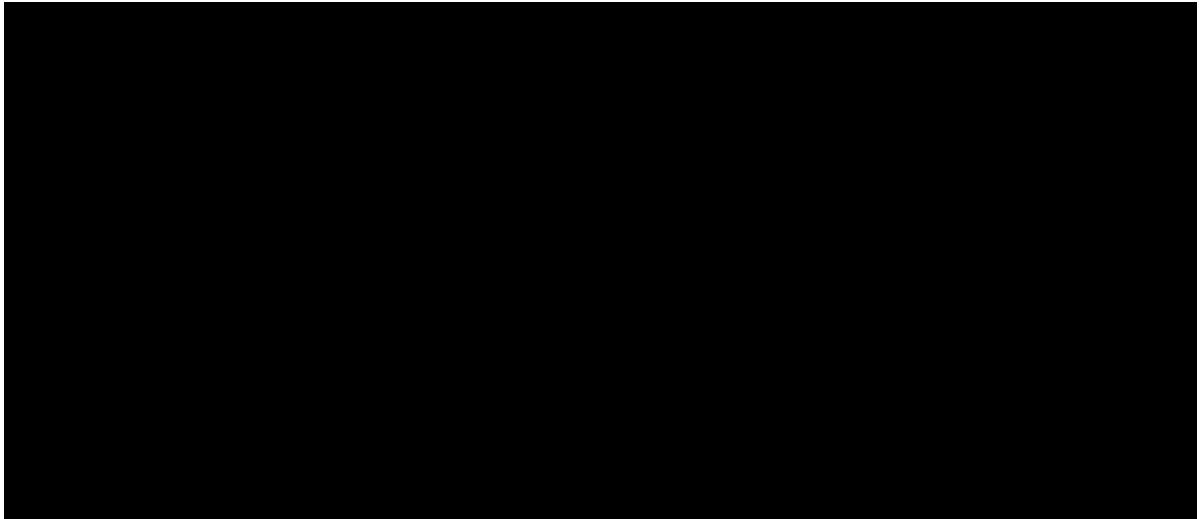
Trong chương trình ở các dòng 34, 37, 58 và 67 sử dụng hàm **assert()** (định nghĩa trong *assert.h*). Hàm này kiểm tra giá trị của biểu thức. Nếu giá trị của biểu thức là 0 (false), hàm **assert()** in một thông báo lỗi và gọi hàm **abort()** (định nghĩa trong *stdlib.h*) để kết thúc chương trình thực thi. Nếu biểu thức có giá trị khác 0 (true) thì chương trình tiếp tục. Điều này rất có ích cho công cụ debug đối với việc kiểm tra nếu một biến có giá trị đúng. Chẳng hạn hàm ở dòng 34 hàm **assert()** kiểm tra con trỏ *FirstName* để xác định nếu nó không bằng 0 (null). Nếu điều kiện trong khẳng định (assertion) cho trước là đúng, chương trình tiếp tục mà không ngắt. Nếu điều kiện trong khẳng định cho trước là sai, một thông báo lỗi chứa số dòng, điều kiện được kiểm tra, và tên file trong đó sự khẳng định xuất hiện được in, và chương trình kết thúc. Khi đó lập trình viên có thể tập trung vào vùng này của đoạn mã để tìm lỗi.

Các khẳng định không phải xóa từ chương trình khi debug xong. Khi các khẳng định không còn cần thiết cho mục đích debug trong một chương trình, dòng sau:

```
#define NDEBUG
```

được thêm vào ở đầu file chương trình. Điều này phát sinh tiền xử lý bỏ qua tất cả các khẳng định thay thế cho lập trình viên xóa mỗi khẳng định bằng tay.

Chúng ta [chạy ví dụ 3.19](#), kết quả ở hình 3.20



Hình 3.20: Kết quả của ví dụ 3.19

Một hàm thành viên có thể được khai báo là **static** nếu nó không truy cập đến các thành viên không tĩnh. Không giống như các thành viên không tĩnh, một hàm thành viên tĩnh không có con trỏ **this** bởi vì các thành viên dữ liệu tĩnh và các hàm thành viên tĩnh tồn tại độc lập với bất kỳ đối tượng nào của lớp.

Chú ý: Hàm thành viên dữ liệu tĩnh không được là **const**.

4.1 DẪN NHẬP

Trong chương 3, chúng ta đã tìm hiểu các điều cơ bản của các lớp C++ và khái niệm kiểu dữ liệu trừu tượng (ADTs). Các thao tác trên các đối tượng của lớp (nghĩa là các thực thể của ADTs) được thực hiện bởi gọi các thông điệp (dưới dạng các lời gọi hàm thành viên) tới các đối tượng. Ký pháp gọi hàm này thì công kênh cho các loại lớp nhất định, đặc biệt là các lớp toán học. Đối với các loại lớp này sẽ là đẹp để sử dụng tập các toán tử có sẵn phong phú của C++ để chỉ rõ các thao tác của đối tượng. Trong chương này tìm hiểu làm thế nào cho phép các toán tử của C++ làm việc với các đối tượng của lớp. Xử lý này được gọi là đa năng hóa toán tử (operator overloading).

Toán tử << được sử dụng nhiều mục đích trong C++ đó là toán tử chèn dòng (stream-insertion) và toán tử dịch chuyển trái. Đây là một ví dụ của đa năng hóa toán tử. Tương tự >> cũng được đa năng hóa. Nó được sử dụng vừa toán tử trích dòng (stream-extraction) và toán tử dịch chuyển phải.

C++ cho phép các lập trình viên đa năng hóa hầu hết các toán tử để biểu thị ngữ cảnh mà trong đó chúng được sử dụng. Trình biên dịch phát sinh đoạn mã thích hợp dựa trên kiểu mà trong đó toán tử được sử dụng. Một vài toán tử được đa năng hóa thường xuyên, đặc biệt là toán tử gán và các toán tử số học như + và -. Công việc thực hiện bởi đa năng hóa các toán tử cũng có thể được thực hiện bởi các lời gọi hàm tường minh, nhưng ký pháp thường sử dụng dễ dàng để đọc.

4.2 CÁC NGUYÊN TẮC CƠ BẢN CỦA ĐA NĂNG HÓA TOÁN TỬ

Lập trình viên có thể sử dụng các kiểu có sẵn và có thể định nghĩa các kiểu mới. Các kiểu có thể được sử dụng với tập các toán tử phong phú. Các toán tử cung cấp cho các lập trình viên với ký pháp ngắn gọn cho việc biểu thị các thao tác của đối tượng của các kiểu có sẵn.

Các lập trình viên có thể sử dụng các toán tử với các kiểu do người dùng định nghĩa. Mặc dù C++ không cho phép các toán tử mới được tạo, nó cho phép các toán tử đã tồn tại được đa năng hóa sao cho khi các toán tử này được sử dụng với các đối tượng của lớp, các toán tử có ý nghĩa thích hợp các kiểu mới. Đây chính là một đặc điểm mạnh của C++.

Các toán tử được đa năng hóa bằng cách viết một định nghĩa hàm (bao gồm phần đầu và thân) như khi chúng ta viết một hàm bình thường, ngoại trừ tên hàm bây giờ trở thành từ khóa **operator** theo sau bởi ký hiệu của toán tử được đa năng hóa. Prototype của nó có dạng như sau:

```
type operator operator_symbol ( parameter_list );
```

Để sử dụng một toán tử một các đối tượng của lớp, toán tử phải được đa năng hóa ngoại trừ hai điều. Điều thứ nhất toán tử gán có thể sử dụng với mọi lớp mà không cần đa năng hóa. Cách cư xử mặc định của toán tử gán là một phép gán thành viên của các thành viên dữ liệu của lớp. Chúng ta nhận thấy rằng sao chép thành viên mặc định thì nguy hiểm đối với các lớp với các thành viên mà được cấp phát động. Chúng ta sẽ đa năng hóa một cách tường minh toán tử gán đối với các lớp như thế. Điều thứ hai toán tử địa chỉ (&) cũng có thể được sử dụng với các đối tượng của bất kỳ lớp nào mà không cần đa năng hóa; Nó trả về địa chỉ của đối tượng trong bộ nhớ. Toán tử địa chỉ cũng có thể được đa năng hóa.

4.3 CÁC GIỚI HẠN CỦA ĐA NĂNG HÓA TOÁN TỬ

Phần lớn các toán tử của C++ có thể được đa năng hóa. Hình 4.1 cho thấy các toán tử có thể được đa năng hóa và hình 4.1 là các toán tử không thể đa năng hóa.

+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->		()	new	delete

Hình 4.1: Các toán tử có thể được đa năng hóa

.	.*	::	?:	sizeof

Hình 4.2: Các toán tử không thể đa năng hóa

Chú ý rằng toán tử ngoặc tròn () trong bảng 4.1 là toán tử gọi hàm. Vì toán tử này đứng sau tên hàm có thể chứa trong nó nhiều tham số do đó toán tử ngoặc tròn là một toán tử nhiều ngôi.

Thứ tự ưu tiên của một toán tử không thể được thay đổi bởi đa năng hóa. Điều này có thể dẫn tới các tình trạng bất tiện trong đó một toán tử được đa năng hóa theo một cách đối với mức độ ưu tiên cố định của nó thì không thích hợp. Tuy nhiên, các dấu ngoặc đơn có thể được sử dụng để đặt thứ tự ước lượng của các toán tử đã đa năng hóa trong một biểu thức.

Tính kết hợp của một toán tử không thể được thay đổi bởi đa năng hóa. Các tham số mặc định không thể sử dụng với một toán tử đa năng hóa.

Không thể thay đổi số các toán hạng mà một toán tử yêu cầu: Đa năng hóa các toán tử một ngôi vẫn là các toán tử một ngôi; đa năng hóa các toán tử hai ngôi vẫn là các toán tử hai ngôi. Toán tử ba ngôi duy nhất (?) của C++ không thể đa năng hóa. Các toán tử **&**, *****, **+** và **-** mỗi toán tử có các phiên bản một và hai ngôi.; Các phiên bản một và hai ngôi này có thể được đa năng hóa riêng biệt.

Ý nghĩa của làm sao một toán tử làm việc trên các đối tượng của các kiểu có sẵn không thể thay đổi bởi việc đa năng hóa toán tử. Chẳng hạn, lập trình viên không thể thay đổi ý nghĩa của làm sao toán tử (+) cộng hai số nguyên. Việc đa năng hóa toán tử chỉ làm việc với các đối tượng của các kiểu do người dùng định nghĩa hoặc với một sự pha trộn của một đối tượng của kiểu do người dùng định nghĩa và một đối tượng của một kiểu có sẵn.

Đa năng hóa một toán tử gán và một toán tử cộng để cho phép các lệnh như là:

```
object2 = object2 + object1
```

không bao hàm toán tử += cũng được đa năng hóa để phép các lệnh như là:

```
object2 += object1
```

Hành vi như thế có thể được thực hiện bởi việc đa năng hóa rõ ràng toán tử += cho lớp đó.

4.4 CÁC HÀM TOÁN TỬ CÓ THỂ LÀ CÁC THÀNH VIÊN CỦA LỚP HOẶC KHÔNG LÀ CÁC THÀNH VIÊN

Các hàm toán tử có thể là các hàm thành viên hoặc hàm không thành viên; hàm không thành viên thường là các hàm **friend**. Các hàm thành viên sử dụng ngầm con trỏ **this** để chứa một trong các tham số đối tượng lớp của chúng. Tham số lớp đó phải được liệt kê một cách tường minh trong lời gọi hàm không thành viên.

Khi đa năng hóa (), [], -> hoặc =, hàm đa năng hóa toán tử phải được khai báo như một thành viên lớp. Đối với các toán tử khác, các hàm đa năng hóa toán tử có thể là các hàm không thành viên (thường là các hàm **friend**).

Liệu có phải một hàm toán tử được cài đặt như một hàm thành viên hoặc như hàm không thành viên, toán tử vẫn còn được sử dụng cùng cách trong biểu thức. Như vậy cách là cách cài đặt nào tốt nhất?

Khi một hàm toán tử được cài đặt như một hàm thành viên, toán hạng cực trái phải là một đối tượng lớp của toán tử. Nếu toán hạng bên trái phải là một đối tượng của lớp khác hoặc một kiểu có sẵn thì hàm toán tử này phải được cài đặt như hàm không thành viên. Một hàm toán tử cài đặt như hàm không thành viên cần là một **friend** nếu hàm phải truy cập đến các thành viên **private** hoặc **protected**.

Các hàm thành viên chỉ được gọi khi toán hạng trái của một toán tử hai ngôi là một đối tượng cụ thể của lớp đó, hoặc khi toán hạng đơn của một toán tử một ngôi là một đối tượng của lớp đó.

Ví dụ 4.1: Chúng ta xây dựng lớp số phức với tên lớp là *Complex* và đa năng hóa toán tử + trên lớp này.

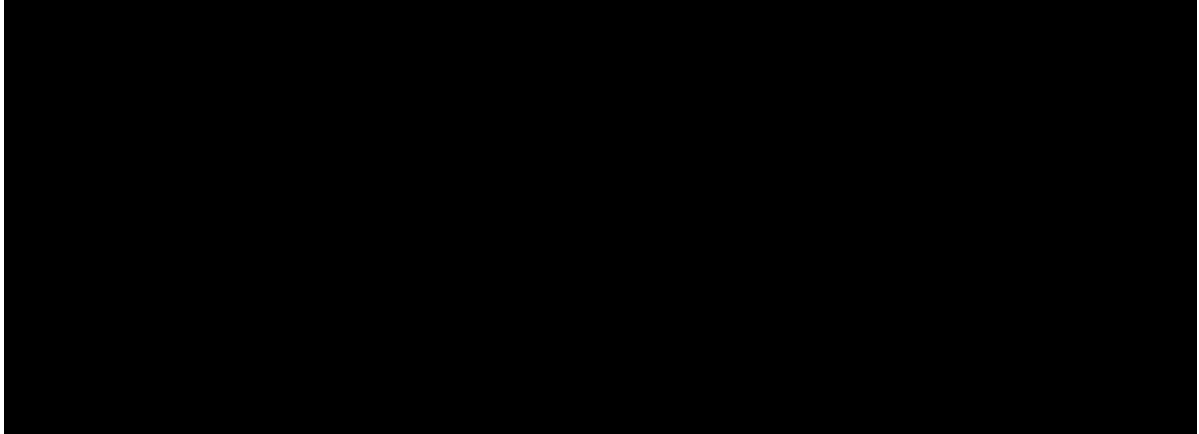
```

1: #include <iostream.h>
2:
3: class Complex
4: {
5: private:
6: double Real, Imaginary;
7: public:
8: Complex(double R=0.0,double I=0.0); // Constructor mặc định
9: void Print(); // Hiển thị số phức
10: Complex operator+(Complex Z); // Phép cộng giữa hai số phức
11: Complex operator+(double R); // Phép cộng một số phức với một
    số thực
12: };
13:
14: Complex::Complex(double R,double I)
15: {
16: Real = R;
17: Imaginary = I;
18: }
19:
20: void Complex::Print()
21: {
22: cout<<'('<<Real<<','<<Imaginary<<')';
23: }
24:
25: Complex Complex::operator + (Complex Z)
26: {
27: Complex Tmp;
28: Tmp.Real = Real + Z.Real;

```

Hàm thành viên toán tử **operator + ()** (từ dòng 25 đến 31 và từ dòng 33 đến 39) trả về một đối tượng có kiểu *Complex* là tổng của hai số phức hoặc tổng của một số phức với một số thực. Chú ý rằng đối tượng tạm thời *Tmp* được dùng bên trong hàm **operator + ()** để giữ kết quả, và đó là đối tượng được trả về.

Chúng ta [chạy ví dụ 4.1](#), kết quả ở hình 4.3



Hình 4.3: Kết quả của ví dụ 4.1

Do đa năng hóa toán tử **+** trên lớp *Complex* ở ví dụ 4.1, chúng ta có thể viết:

```
X = Y + Z;
```

Câu lệnh này được trình biên dịch hiểu:

```
X = Y.operator + (Z);
```

Như vậy, trong biểu thức $Y + Z$ đối tượng bên trái toán tử **+** (là đối tượng Y) là đối tượng mà qua đó, hàm thành viên toán tử **operator + ()** được gọi. Do đó hàm thành viên toán tử **+** chỉ nhận một tham số là đối tượng bên phải toán tử và đối tượng bên trái toán tử là đối tượng tạo lời gọi cho hàm toán tử và được truyền bởi con trỏ **this**.

Hàm **operator + ()** trả về một đối tượng *Complex*. Do vậy chúng ta có thể viết:

```
(Y + Z).Print();
```

để in trên màn hình số phức của đối tượng được trả về. Đối tượng do $Y + Z$ sinh ra như vậy là một đối tượng tạm thời. Nó sẽ không tồn tại khi hàm thành *Print()* kết thúc.

Hơn nữa khi trả về một đối tượng, toán tử **+** cho phép một chuỗi phép cộng. Nên chúng ta cũng có thể viết:

```
X = X + Y + Z;
```

Tuy nhiên chúng ta không thể nào viết được câu lệnh sau:

```
X = 3.5 + Y; // Lỗi !!!
```


Chính vì lý do này chúng ta chọn một hàm không thành viên để đa năng hóa một toán tử để cho phép toán tử được giao hoán. Chú ý rằng hàm không thành viên không cần thiết phải là hàm **friend** nếu các hàm set và get thích hợp tồn tại trong phần giao diện public, và đặt biệt nhất nếu các hàm set và get là các hàm **inline**.

Để đa năng hóa toán tử << phải có một toán hạng trái của kiểu **ostream &** (như là **cout** trong biểu thức `cout<<X`), vì thế nó phải là hàm không thành viên. Tương tự, đa năng hóa toán tử >> phải có một toán hạng trái của kiểu **istream &** (như là **cin** trong biểu thức `cin>>X`), vì thế nó cũng phải là hàm không thành viên.

Ngoại trừ đa năng hóa toán tử >> và << liên quan đến dòng nhập/xuất dữ liệu chúng ta có hình 4.4 về cách đa năng hóa toán tử như sau:

Biểu thức	Hàm thành viên	Hàm không thành viên
a#b	a.operator#(b)	operator#(a,b)
#a	a.operator()	operator#(a)
a=b	a.operator=(b)	
a[b]	a.operator[](b)	
a(b)	a.operator()(b)	
a->	a.operator->()	
a++	a.operator++(0)	operator++(a,0)
a--	a.operator--(0)	operator--(a,0)

Hình 4.4: Việc cài đặt các hàm toán tử

4.5 ĐA NĂNG HOÁ CÁC TOÁN TỬ HAI NGÔI

Các toán tử hai ngôi được đa năng hóa trong hình 4.5 sau:

Toán tử	Ví dụ	Toán tử	Ví dụ	Toán tử	Ví dụ
+	a+b	+=	a+=b	<<=	a<<=b
-	a-b	-=	a-=b	==	a==b
*	a*b	*=	a*=b	!=	a!=b
/	a/b	/=	a/=b	<=	a<=b
%	a%b	%=	a%=b	>=	a>=b
^	a^b	^=	a^=b	&&	a&& b

&	a&b	&=	a&=b		a b
	a b	=	a =b	,	a,b
=	a=b	<<	a<<b	[]	a[b]
<	a>	a>>b	->*	a->*b
>	a>b	>>=	a>>=b		

Hình 4.5: Các toán tử hai ngôi được đa năng hóa

Một toán tử hai ngôi có thể được đa năng hóa như là hàm thành viên không tĩnh với một tham số hoặc như một hàm không thành viên với hai tham số (một trong các tham số này phải là hoặc là một đối tượng lớp hoặc là một tham chiếu đến đối tượng lớp).

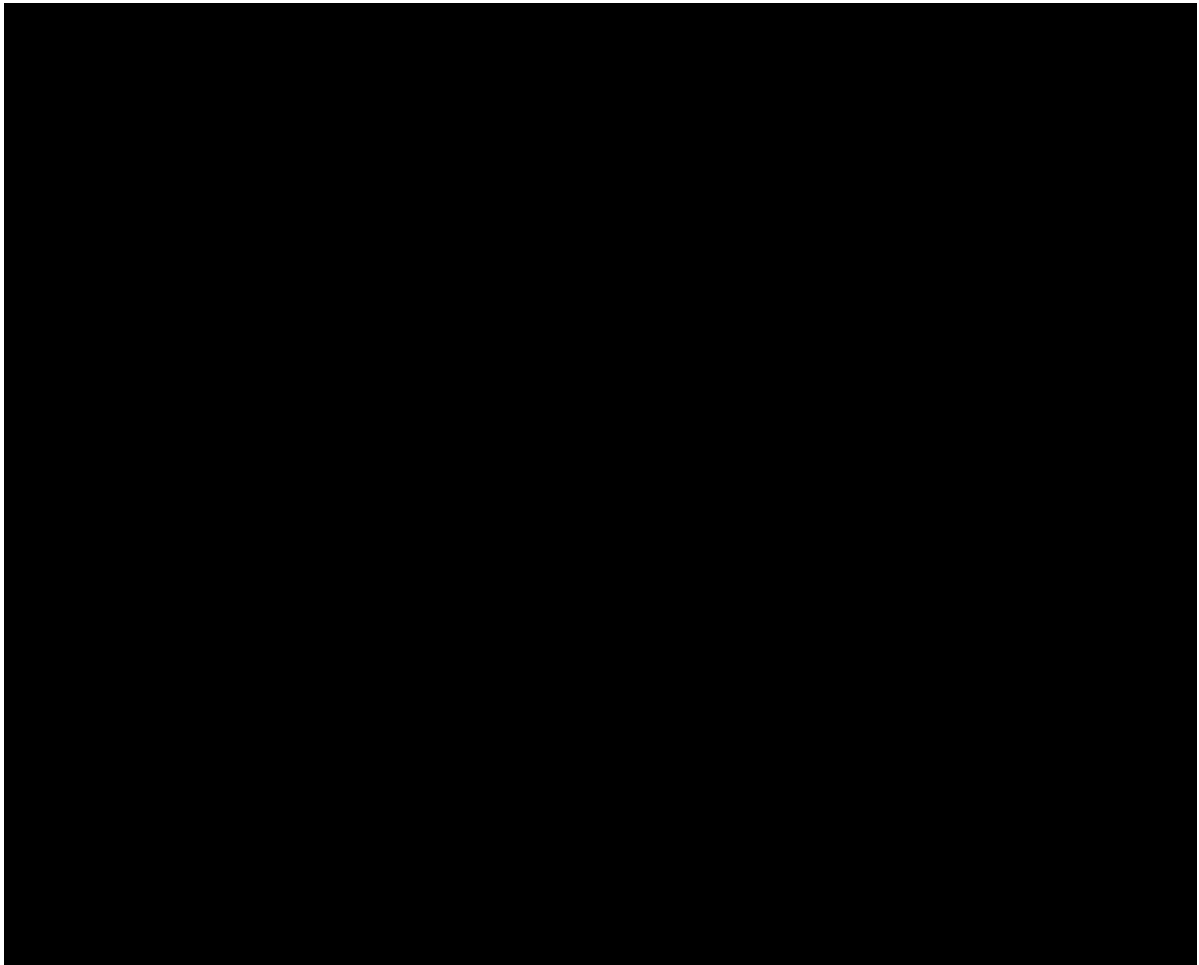
Ví dụ 4.2: Chúng ta xây dựng lớp số phức với tên lớp là *Complex* và đa năng hóa các toán tử tính toán + - += -= và các toán tử so sánh == != > >= < <= với các hàm toán tử là các hàm thành viên.

```

130: cout<<"X: ";
131: X.Print();
132: cout<<endl<<"Y: ";
133: Y.Print();
134: cout<<endl<<"Z: ";
135: Z.Print();
136: cout<<endl<<"T: ";
137: T.Print();
138: T=5.3;// Gọi constructor chuyển kiểu
139: cout<<endl<<endl<<"T = 5.3"<<endl;
140: cout<<"T: ";
141: T.Print();
142: X = Y + Z;
143: cout<<endl<<endl<<"X = Y + Z: ";
144: X.Print();
145: cout<<" = ";
146: Y.Print();
147: cout<<" + ";
148: Z.Print();
149: X = Y - Z;
150: cout<<endl<<"X = Y - Z: ";
151: X.Print();
152: cout<<" = ";
153: Y.Print();
154: cout<<" - ";
155: Z.Print();
156: cout<<endl<<endl<<"Y += T i.e ";
157: Y.Print();
158: cout<<" += ";

```

Chúng ta [chạy ví dụ 4.2](#), kết quả ở hình 4.6



Hình 4.6: Kết quả của ví dụ 4.2

Dòng thứ 11 của chương trình ở ví dụ 4.2:

```
Complex(const Complex &Z);
```

là một constructor sao chép (copy constructor). Nó khởi động một đối tượng lớp bằng cách tạo một sao chép của một đối tượng lớp đó. Constructor sao chép thực hiện công việc giống như toán tử sao chép nhưng nó có một vai trò đặc biệt. Constructor sao chép chỉ nhận tham số là một tham chiếu chỉ đến đối tượng thuộc chính lớp mà nó được định nghĩa. Các constructor sao chép được dùng mỗi khi một sự sao chép của một đối tượng cần thiết như khi có sự truyền tham số bằng trị, khi trả về một đối tượng từ hàm, hoặc khi khởi động một đối tượng mà được sao chép từ đối tượng khác của cùng lớp. Chẳng hạn:

```
Complex A(3.5, 4.5);
```

```
Complex B(A); // Gọi constructor sao chép
```

```
Complex C = B; // Gọi constructor sao chép
```

.....
Complex MyFunc(Complex Z) // Gọi constructor sao chép

Z; // Gọi constructor sao chép

Chúng ta chú ý rằng, dấu = trong câu lệnh trên ứng với constructor sao chép chứ không phải là toán tử gán . Nếu chúng ta không định nghĩa constructor sao chép, trình biên dịch tạo ra một constructor sao chép mặc định sẽ sao chép từng thành viên một.

Ở dòng 12 của chương trình ở ví dụ 4.2:

Complex(double R);

là một constructor chuyển đổi (conversion constructor). Constructor này lấy một tham số **double** và khởi tạo đối tượng *Complex* mà phần thực bằng giá trị tham số truyền vào và phần ảo bằng 0.0 (từ dòng 48 đến 52). Bất kỳ một constructor nào có tham số đơn có thể được nghĩ như một constructor chuyển đổi. Constructor chuyển đổi sẽ đổi một số thực thành một đối tượng *Complex* rồi gán cho đối tượng đích *Complex*. Chẳng hạn:

T = 3.5; // Ngầm định: T = Complex(3.5)

Trình biên dịch tự động dùng constructor chuyển đổi để tạo một đối tượng tạm thời *Complex*, rồi dùng toán tử gán để gán đối tượng tạm thời này cho đối tượng khác của *Complex*. Chẳng hạn câu lệnh sau vẫn đúng:

X = Y + 3.5; // Ngầm định: X = Y + Complex(3.5);

Như vậy một constructor chuyển đổi được sử dụng để thực hiện một sự chuyển đổi ngầm định.

Ví dụ 4.3: Lấy lại ví dụ 4.2 nhưng các hàm toán tử +, - và các hàm toán tử so sánh là hàm không thành viên.

```

129:

130: cout<<"X: ";

131: X.Print();

132: cout<<endl<<"Y: ";

133: Y.Print();

134: cout<<endl<<"Z: ";

135: Z.Print();

136: X = Y + 3.6;

137: cout<<endl<<endl<<"X = Y + 3.6: ";

138: X.Print();

139: cout<<" = ";

140: Y.Print();

141: cout<<" + 3.6 ";

142: X = 3.6 + Y;

143: cout<<endl<<"X = 3.6 + Y: ";

144: X.Print();

145: cout<<" = 3.6 + ";

146: Y.Print();

147: X = 3.8 - Z;

148: cout<<endl<<"X = 3.8 - Z: ";

149: X.Print();

150: cout<<" = 3.8 - ";

151: Z.Print();

152: X = Z - 3.8;

153: cout<<endl<<"X = Z - 3.8: ";

154: X.Print();

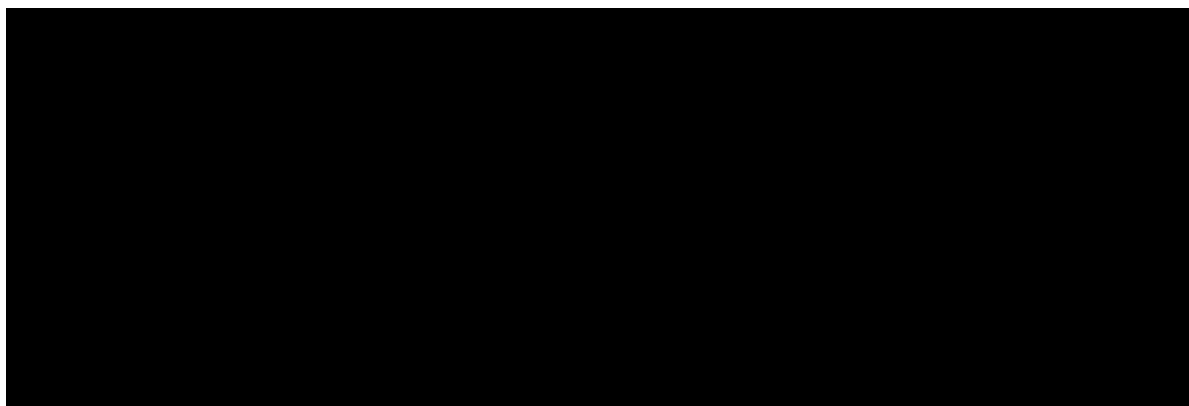
155: cout<<" = ";

156: Z.Print();

157: cout<<" - 3.8 ";

```

Chúng ta [chạy ví dụ 4.3](#), kết quả ở hình 4.7



Hình 4.7: Kết quả của ví dụ 4.3

4.6 ĐA NĂNG HÓA CÁC TOÁN TỬ MỘT NGÔI

Các toán tử một ngôi được đa năng hóa trong hình 4.8 sau:

Toán tử	Ví dụ	Toán tử	Ví dụ
+	+c	~	~c
-	-c	!	!a
*	*c	++	++c, c++
&	&c	--	--c, c--
->	c->		

Hình 4.8: Các toán tử một ngôi được đa năng hóa

Một toán tử một ngôi của lớp được đa năng hóa như một hàm thành viên không tĩnh với không có tham số hoặc như một hàm không thành viên với một tham số; Tham số đó phải hoặc là một đối tượng lớp hoặc là một tham chiếu đến đối tượng lớp.

[Ví dụ 4.4:](#) Lấy lại ví dụ 4.3 và thêm toán tử dấu trừ một ngôi.

```

129: return Z1.Abs() <= Z2.Abs();

130: }

131:

132: double Complex::Abs()

133: {

134: return sqrt(Real*Real+Imaginary*Imaginary);

135: }

136:

137: int main()

138: {

139: Complex X, Y(4.3,8.2), Z(3.3,1.1);

140:

141: cout<<"X: ";

142: X.Print();

143: cout<<endl<<"Y: ";

144: Y.Print();

145: cout<<endl<<"Z: ";

146: Z.Print();

147: X = -Y + 3.6;

148: cout<<endl<<endl<<"X = -Y + 3.6: ";

149: X.Print();

150: cout<<" = ";

151: (-Y).Print();

152: cout<<" + 3.6 ";

153: X = -Y + -Z;

154: cout<<endl<<"X = -Y + -Z: ";

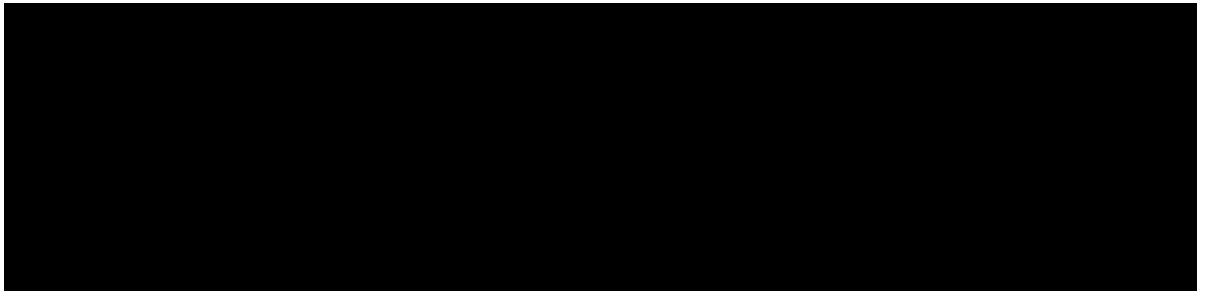
155: X.Print();

156: cout<<" = ";

157: (-Y).Print();

```


Chúng ta [chạy ví dụ 4.4](#), kết quả ở hình 4.9



Hình 4.9: Kết quả của ví dụ 4.4

4.7 ĐA NĂNG HÓA MỘT SỐ TOÁN TỬ ĐẶC BIỆT

Trong phần này chúng ta sẽ tìm hiểu cách cài đặt một vài toán tử đặc biệt như `() [] ++ -- , = ->`

4.7.1 Toán tử `[]`:

Khi cài đặt các lớp vector hoặc chuỗi ký tự, chúng ta cần phải truy cập đến từng phần tử của chúng, trong ngôn ngữ C/C++ đã có toán tử `[]` để truy cập đến một phần tử của mảng. Đây là toán tử hai ngôi, có dạng `a[b]` và khi đa năng toán tử này thì hàm toán tử tương ứng phải là thành viên của một lớp.

[Ví dụ 4.5](#): Đa năng hóa toán tử `[]` để truy cập đến một phần tử của vector.

```
1: #include <iostream.h>
2:
3: class Vector
4: {
5: private:
6: int Size;
7: int *Data;
8: public:
9: Vector(int S=2,int V=0);
10: ~Vector();
11: void Print() const;
12: int & operator [] (int I);
13: };
14:
15: Vector::Vector(int S,int V)
16: {
17: Size = S;
18: Data=new int[Size];
19: for(int I=0;I<Size;++I)
20: Data[I]=V;
21: }
22:
23: Vector::~~Vector()
24: {
25: delete []Data;
26: }
27: void Vector::Print() const
28: {
29: cout<<"Vector: (";
```

Chúng ta [chạy ví dụ 4.5](#), kết quả ở hình 4.10



Hình 4.10: Kết quả của ví dụ 4.5

Trong chương trình ở ví dụ 4.5, hàm toán tử của toán tử `[]` ở lớp *Vector* trả về một tham chiếu vì toán tử này có thể dùng ở vế trái của phép gán.

4.7.2 Toán tử `()`

Toán tử `()` được dùng để gọi hàm, toán tử này gồm hai toán hạng: toán hạng đầu tiên là tên hàm, toán hạng thứ hai là danh sách các tham số của hàm. Toán tử này có dạng giống như toán tử `[]` và khi đa năng toán tử này thì hàm toán tử tương ứng phải là thành viên của một lớp.

[Ví dụ 4.6](#): Lấy lại ví dụ 4.5 nhưng đa năng hóa toán tử `()` để truy cập đến một phần tử của vector.



```
1: #include <iostream.h>
2:
3: class Vector
4: {
5: private:
6: int Size;
7: int *Data;
8: public:
9: Vector(int S=2,int V=0);
10: ~Vector();
11: void Print() const;
12: int & operator () (int I);
13: };
14:
15: Vector::Vector(int S,int V)
16: {
17: Size = S;
18: Data=new int[Size];
19: for(int I=0;I<Size;++I)
20: Data[I]=V;
21: }
22:
23: Vector::~~Vector()
24: {
25: delete []Data;
26: }
27: void Vector::Print() const
28: {
29: cout<<"Vector: (";
```

Chúng ta [chạy ví dụ 4.6](#), kết quả ở hình 4.11



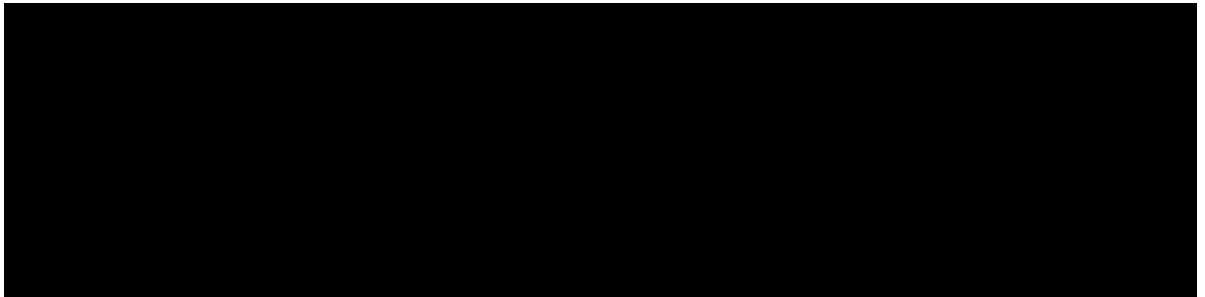
Hình 4.11: Kết quả của ví dụ 4.6

[Ví dụ 4.7](#): Đa năng hóa toán tử () để truy cập đến phần tử của ma trận.



```
1: #include <iostream.h>
2:
3: class Matrix
4: {
5: private:
6: int Rows, Cols;
7: int **Data;
8: public:
9: Matrix(int R=2, int C=2, int V=0);
10: ~Matrix();
11: void Print() const;
12: int & operator () (int R, int C);
13: };
14:
15: Matrix::Matrix(int R, int C, int V)
16: {
17: int I, J;
18: Rows=R;
19: Cols=C;
20: Data = new int *[Rows];
21: int *Temp=new int[Rows*Cols];
22: for(I=0; I<Rows; ++I)
23: {
24: Data[I]=Temp;
25: Temp+=Cols;
26: }
27: for(I=0; I<Rows; ++I)
28: for(J=0; J<Cols; ++J)
29: Data[I][J]=V;
```

Chúng ta [chạy ví dụ 4.7](#), kết quả ở hình 4.12



Hình 4.12: Kết quả của ví dụ 4.7

4.7.3 Toán tử ++ và --

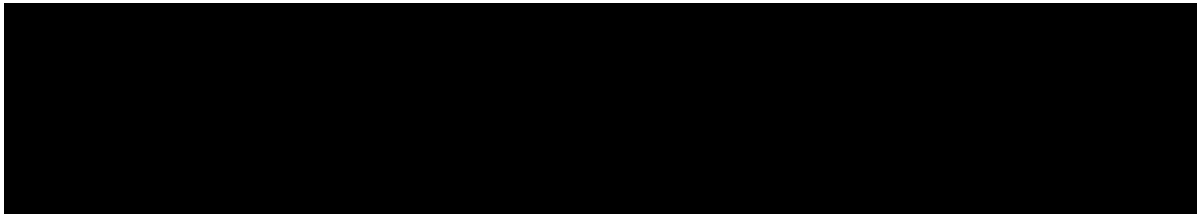
Cài đặt các phép toán ++ và -- cũng tương tự như các toán tử khác.

[Ví dụ 4.8](#): Đa năng hóa toán tử ++ và --



```
1: #include <iostream.h>
2:
3: class Point
4: {
5: private:
6: int X,Y;
7: public:
8: Point(int A=0,int B=0);
9: Point operator ++();
10: Point operator --();
11: void Print() const;
12: };
13:
14: Point::Point(int A,int B)
15: {
16: X = A;
17: Y = B;
18: }
19:
20: Point Point::operator++()
21: {
22: ++X;
23: ++Y;
24: return *this;
25: }
26:
27: Point Point::operator--()
28: {
29: --X;
```


Chúng ta [chạy ví dụ 4.8](#), kết quả ở hình 4.13



Hình 4.13: Kết quả của ví dụ 4.8

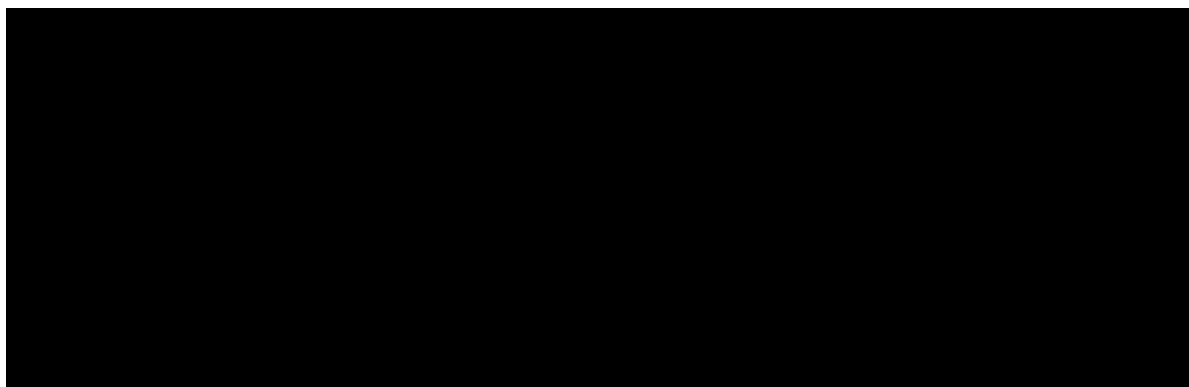
Tuy nhiên ở ví dụ 4.8, không phân biệt được các toán tử ++ và -- khi ở dạng tiền tố và hậu tố. Để có thể phân biệt được tiền tố và hậu tố khi đa năng hóa, toán tử ++a (hoặc --a) là toán tử một ngôi, còn toán tử a++ (hoặc a--) được xem là toán tử hai ngôi trong đó toán hạng thứ hai kiểu **int**.

[Ví dụ 4.9](#): Đa năng hóa toán tử ++ và – có phân biệt tiền tố và hậu tố.



```
1: #include <iostream.h>
2:
3: class Point
4: {
5: private:
6: int X,Y;
7: public:
8: Point(int A=0,int B=0);
9: Point operator ++();
10: Point operator ++(int); //Hậu tố
11: Point operator --(); //Tiền tố
12: Point operator --(int); //Hậu tố
13: void Print() const;
14: };
15:
16: Point::Point(int A,int B)
17: {
18: X = A;
19: Y = B;
20: }
21:
22: Point Point::operator++()
23: {
24: ++X;
25: ++Y;
26: return *this;
27: }
28:
29: Point Point::operator++(int)
```

Chúng ta [chạy ví dụ 4.9](#), kết quả ở hình 4.14



Hình 4.14: Kết quả của ví dụ 4.9

4.7.4 Toán tử dấu phẩy

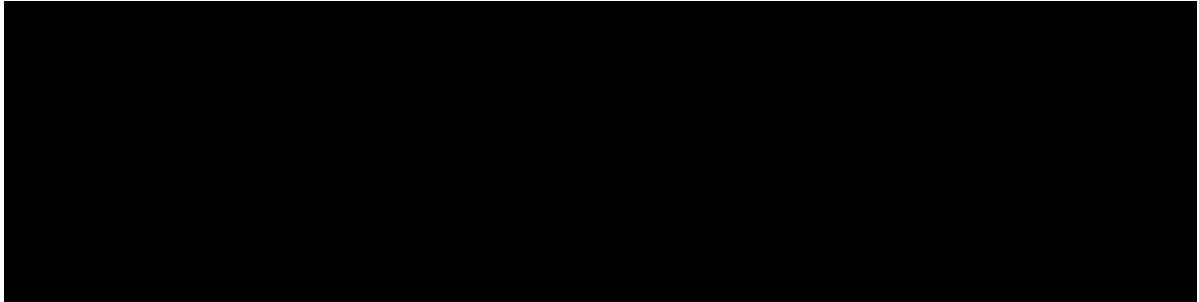
Toán tử dấu phẩy là toán tử hai ngôi. Chúng ta có thể đa năng hóa toán tử dấu phẩy có ý nghĩa tùy thích. Tuy nhiên, nếu muốn đa năng hóa toán tử phẩy có ý nghĩa tương tự như toán tử phẩy bình thường, chúng ta cần bỏ qua toán hạng bên trái dấu phẩy (nghĩa là đối tượng mà qua đó hàm toán tử tương ứng được gọi) và trả về đối tượng bên phải dấu phẩy như là kết quả của toán tử. Do vậy trong một danh sách gồm các biểu thức được ngăn cách bởi các dấu phẩy, mọi toán hạng trừ toán hạng ngoài cùng bên phải đều bị bỏ qua.

[Ví dụ 4.10](#): Đa năng hóa toán tử dấu phẩy.



```
1: #include <iostream.h>
2:
3: class Point
4: {
5: private:
6: int X,Y;
7: public:
8: Point(int A=0,int B=0);
9: Point operator +(Point P);
10: Point operator ,(Point P);
11: void Print() const;
12: };
13:
14: Point::Point(int A,int B)
15: {
16: X = A;
17: Y = B;
18: }
19:
20: Point Point::operator+(Point P)
21: {
22: Point Tmp;
23: Tmp.X=X+P.X;
24: Tmp.Y=Y+P.Y;
25: return Tmp;
26: }
27:
28: Point Point::operator,(Point P)
29: {
```

Chúng ta [chạy ví dụ 4.10](#), kết quả ở hình 4.15



Hình 4.15: Kết quả của ví dụ 4.10

4.7.5 Toán tử ->

Toán tử dấu mũi tên -> (hoặc toán tử con trỏ cấu trúc) cho phép chúng ta truy cập thành viên của đối tượng thuộc class, struct và union khi được áp dụng vào con trỏ chỉ đến đối tượng đó. Chúng ta có thể đa năng hóa toán tử này, hàm toán tử của toán tử -> phải là hàm thành viên của lớp. Nếu toán tử -> có dạng a-> thì hàm toán tử là a.operator->()

[Ví dụ 4.11](#): Đa năng hóa toán tử ->.



```

1: #include <iostream.h>
2:
3: class MyClass
4: {
5: public:
6: int Data;
7: MyClass * operator ->()
8: {
9: return this;
10: }
11: };
12:
13: int main()
14: {
15: MyClass M;
16: M->Data = 10;
17: cout<<M.Data<<" "<<M->Data;
18: return 0;
19: }

```

Chúng ta [chạy ví dụ 4.11](#), kết quả ở hình 4.16



Hình 4.16: Kết quả của ví dụ 4.11

Khi đa năng hóa toán tử $\rightarrow$, C++ quy định hàm toán tử của nó phải trả về:

Hoặc là con trỏ trỏ đến đối tượng thuộc lớp chứa thành viên dữ liệu cần truy cập (lớp mà trong đó hàm toán tử được định nghĩa). Một toán tử $\rightarrow$ bình thường sẽ được trình biên dịch tự động áp dụng vào con trỏ ấy để truy cập thành viên dữ liệu đó của đối tượng.

Hoặc là đối tượng thuộc lớp khác. Lớp này phải chứa định nghĩa hàm toán tử. Trình biên dịch sẽ tự động áp dụng toán tử $\rightarrow$ trên đối tượng được trả về. Kết quả thu được có thể là một con trỏ trỏ đến đối tượng hoặc một đối tượng. Cụ thể, chúng ta có sự diễn dịch theo lối đệ quy cho tới khi nào kết quả là một con trỏ trỏ đến đối tượng thuộc lớp chứa thành viên dữ liệu cần truy cập.

Ví dụ 4.12: Minh họa quá trình diễn dịch đệ quy khi đa năng hóa toán tử $\rightarrow$.



```
1: #include <iostream.h>
2: #include <string.h>
3: #include <stdlib.h>
4:
5: class String
6: {
7: public:
8: char *St;
9: String(char *S)
10: {
11: St=strdup(S);
12: }
13: ~String()
14: {
15: free(St);
16: }
17: };
18:
19: class StringPointer
20: {
21: private:
22: String *M;
23: public:
24: StringPointer(String &S):M(&S)
25: {
26: }
27: String * operator ->()
28: {
29: return M;
```


Chúng ta [chạy ví dụ 4.12](#), kết quả ở hình 4.17



Hình 4.17: Kết quả của ví dụ 4.12

4.7.6 Toán tử gán

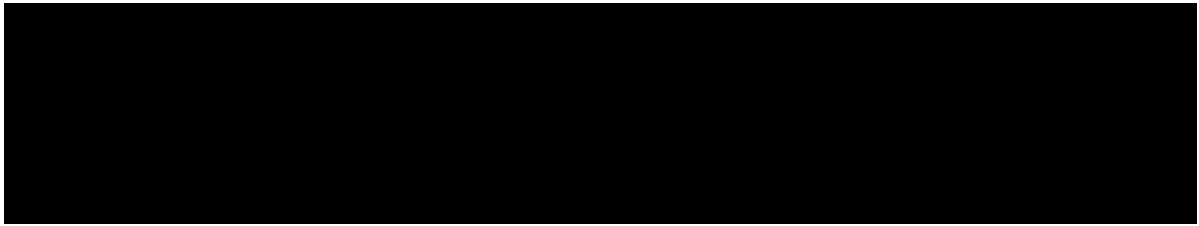
Chúng ta có thể đa năng hóa toán tử gán ($=$) nhưng hàm toán tử của nó phải là hàm thành viên của lớp. Nếu toán tử gán có dạng $a=b$ thì hàm toán tử có dạng $a.operator(b)$.

[Ví dụ 4.13](#): Đa năng hóa toán tử gán.



```
1: #include <iostream.h>
2: #include <string.h>
3:
4: class String
5: {
6: private:
7: char *St;
8: int Len;
9: public:
10: String(char *S);
11: ~String();
12: char *GetStr();
13: String & operator=(String &Obj);
14: };
15:
16: String::String(char *S)
17: {
18: Len=strlen(S);
19: St=new char[Len+1];
20: strcpy(St,S);
21: }
22:
23: String::~~String()
24: {
25: delete []St;
26: }
```

Chúng ta [chạy ví dụ 4.13](#), kết quả ở hình 4.18



Hình 4.18: Kết quả của ví dụ 4.13

4.8 TOÁN TỬ CHUYỂN ĐỔI KIỂU

Phần lớn các chương trình xử lý thông tin sự đa dạng của các kiểu. Đôi khi tất cả các thao tác "dừng lại bên trong một kiểu". Chẳng hạn, một số nguyên với một số nguyên tạo thành một số nguyên (miễn là kết quả không quá lớn để được biểu diễn như một số nguyên). Nhưng thật cần thiết để chuyển đổi dữ liệu của một kiểu tới dữ liệu của kiểu khác. Điều này có thể xảy ra trong các phép gán, các kết quả tính toán, trong việc chuyển các giá trị tới hàm, và trong việc trả về từ hàm. Trình biên dịch biết làm thế nào để thực hiện các chuyển đổi nào đó trong số các kiểu có sẵn. Các lập trình viên có thể ép buộc các chuyển đổi trong số các kiểu có sẵn bởi ép kiểu.

Nhưng đối với các kiểu do người dùng định nghĩa thì trình biên dịch không thể tự động biết làm thế nào chuyển đổi trong số các kiểu dữ liệu do người dùng định nghĩa và các kiểu có sẵn. Lập trình viên phải chỉ rõ làm sao các chuyển đổi như vậy sẽ xuất hiện. Các chuyển đổi như thế có thể được thực hiện với constructor chuyển đổi.

Một toán tử chuyển đổi kiểu có thể được sử dụng để chuyển đổi một đối tượng của một lớp thành đối tượng của một lớp khác hoặc thành một đối tượng của một kiểu có sẵn. Toán tử chuyển đổi kiểu như thế phải là hàm thành viên không tĩnh và không là hàm **friend**. Prototype của hàm thành viên này có cú pháp:

operator <data type> ();

[Ví dụ 4.14](#): Toán tử chuyển đổi kiểu



```
1: #include <iostream.h>
2:
3: class Number
4: {
5: private:
6: float Data;
7: public:
8: Number(float F=0.0)
9: {
10: Data=F;
11: }
12: operator float()
13: {
14: return Data;
15: }
16: operator int()
17: {
18: return (int)Data;
19: }
20: };
21:
22: int main()
23: {
24: Number N1(9.7), N2(2.6);
25: float X=(float)N1; //Gọi operator float()
26: cout<<X<<endl;
27: int Y=(int)N2; //Gọi operator int()
28: cout<<Y<<endl;
29: return 0;
```

Chúng ta [chạy ví dụ 4.14](#), kết quả ở hình 4.19



Hình 4.19: Kết quả của ví dụ 4.14

4.9 TOÁN TỬ NEW VÀ DELETE

Các toán tử **new** và **delete** toàn cục có thể được đa năng hóa. Điều này cho phép các lập trình viên C++ có khả năng xây dựng một hệ thống cấp phát bộ nhớ theo ý người dùng, cói cùng giao tiếp như hệ thống cấp phát mặc định.

Có hai cách đa năng hóa các toán tử **new** và **delete**:

Chúng ta có thể đa năng hóa một cách toàn cục nghĩa là thay thế hẳn các toán tử **new** và **delete** mặc định.

Chúng ta năng hóa các toán tử **new** và **delete** với tư cách là hàm thành viên của lớp nếu muốn các toán tử **new** và **delete** áp dụng đối với lớp đó. Khi chúng ta dùng **new** và **delete** đối với lớp nào đó, trình biên dịch sẽ kiểm tra xem **new** và **delete** có được định nghĩa riêng cho lớp đó hay không; nếu không thì dùng **new** và **delete** toàn cục (có thể đã được đa năng hóa).

Hàm toán tử của toán tử **new** và **delete** có prototype như sau:

```
void * operator new(size_t size);
```

```
void operator delete(void * ptr);
```

Trong đó tham số kiểu **size_t** được trình biên dịch hiểu là kích thước của kiểu dữ liệu được trao cho toán tử **new**.

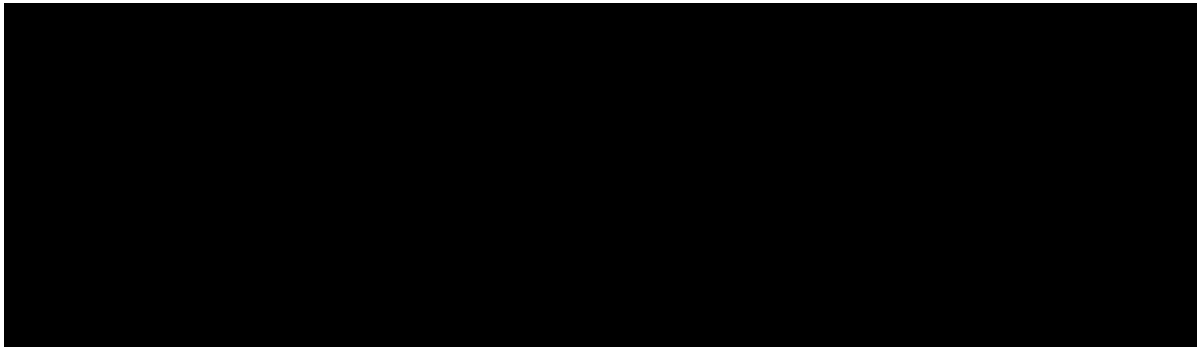
4.9.1 Đa năng hóa toán tử new và delete toàn cục:

Ví dụ 4.15: Đa năng hóa toán tử **new** và **delete** toàn cục đồng thời chứng tỏ rằng toán tử **new** và **delete** do đa năng hóa thay thế toán tử **new** và **delete** mặc định.



```
1: #include <iostream.h>
2: #include <stdlib.h>
3:
4: class Point
5: {
6: private:
7: int X, Y;
8: public:
9: Point(int A=0,int B=0)
10: {
11: X=A;
12: Y=B;
13: cout<<"Constructor!"<<endl;
14: }
15: ~Point()
16: {
17: cout<<"Destructor!"<<endl;
18: }
19: void Print() const
20: {
21: cout<<"X="<<X<<","<<"Y="<<Y<<endl;
22: }
23: };
24:
25: void * operator new(size_t Size)
26: {
27: return malloc(Size);
28: }
29:
```

Chúng ta [chạy ví dụ 4.15](#), kết quả ở hình 4.20



Hình 4.20: Kết quả của ví dụ 4.15

4.9.2 Đa năng hóa toán tử **new** và **delete** cho một lớp:

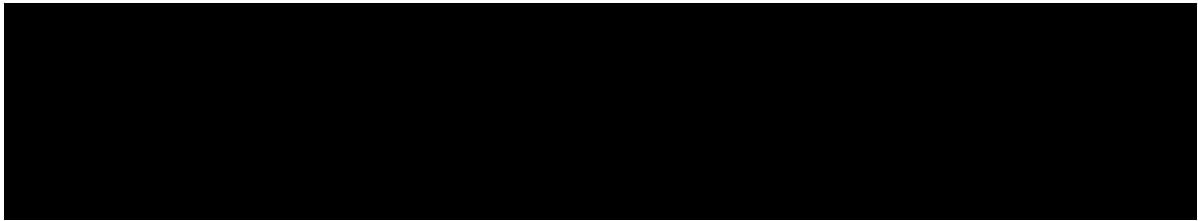
Nếu muốn toán tử **new** và **delete** có tính chất đặc biệt chỉ khi áp dụng cho đối tượng của lớp nào đó, chúng ta có thể đa năng hóa toán tử **new** và **delete** với tư cách là hàm thành viên của lớp đó. Việc này không khác lắm so với cách đa năng hóa toán tử **new** và **delete** một cách toàn cục.

[Ví dụ 4.16](#): Đa năng hóa toán tử **new** và **delete** cho một lớp.



```
1: #include <iostream.h>
2: #include <stdlib.h>
3: class Number
4: {
5: private:
6: int Data;
7: public:
8: Number(int X=0)
9: {
10: Data=X;
11: }
12:
13: void * operator new(size_t Size)
14: {
15: cout<<"Toan tu new cua lop!"<<endl;
16: return ::new unsigned char[Size];
17: }
18:
19: void operator delete(void *Ptr)
20: {
21: cout<<"Toan tu delete cua lop!"<<endl;
22: ::delete Ptr;
23: }
24:
25: void Print() const
26: {
27: cout<<"Data:"<<Data<<endl;
28: }
29:
```


Chúng ta [chạy ví dụ 4.16](#), kết quả ở hình 4.21



Hình 4.21: Kết quả của ví dụ 4.16

4.10 ĐA NĂNG HÓA CÁC TOÁN TỬ CHÈN DÒNG << VÀ TRÍCH DÒNG >>

Chúng ta có thể đa năng hóa các toán tử chèn dòng << (stream insertion) và trích dòng >> (stream extraction). Hàm toán tử của toán tử << được đa năng hóa có prototype như sau:

ostream & operator << (**ostream & stream**, *ClassName Object*);

Hàm toán tử << trả về tham chiếu chỉ đến dòng xuất **ostream**. Tham số thứ nhất của hàm toán tử << là một tham chiếu chỉ đến dòng xuất **ostream**, tham số thứ hai là đối tượng được chèn vào dòng. Khi sử dụng, dòng trao cho toán tử << (tham số thứ nhất) là toán hạng bên trái và đối tượng được đưa vào dòng (tham số thứ hai) là toán hạng bên phải. Để bảo đảm cách dùng toán tử << luôn nhất quán, chúng ta không thể định nghĩa hàm toán tử << như là hàm thành viên của lớp đang xét, thông thường nó chính là hàm **friend**.

Còn hàm toán tử của toán tử >> được đa năng hóa có prototype như sau:

istream & operator >> (**istream & stream**, *ClassName Object*);

Hàm toán tử >> trả về tham chiếu chỉ đến dòng nhập **istream**. Tham số thứ nhất của hàm toán tử này là một tham chiếu chỉ đến dòng nhập **istream**, tham số thứ hai là đối tượng của lớp đang xét mà chúng ta muốn tạo dựng nhờ vào dữ liệu lấy từ dòng nhập. Khi sử dụng, dòng nhập đóng vai toán hạng bên trái, đối tượng nhận dữ liệu đóng vai toán hạng bên phải. Cũng như trường hợp toán tử <<, hàm toán tử >> không là hàm thành viên của lớp, thông thường nó chính là hàm **friend**.

[Ví dụ 4.17:](#)



```
1: #include <iostream.h>
2:
3: class Point
4: {
5: private:
6: int X,Y;
7: public:
8: Point();
9: friend ostream & operator << (ostream & Out,Point & P);
10: friend istream & operator >> (istream & In,Point & P);
11: };
12:
13: Point::Point()
14: {
15: X=Y=0;
16: }
17:
18: ostream & operator << (ostream & Out,Point & P)
19: {
20: Out<<"X="<<P.X<<"",Y="<<P.Y<<endl;
21: return Out; //Cho phép cout<<a<<b<<c;
22: }
23:
24: istream & operator >> (istream &In,Point & P)
25: {
26: cout<<"X:";
27: In>>P.X;
28: cout<<"Y:";
29: In>>P.Y;
```

Chúng ta [chạy ví dụ 4.17](#), kết quả ở hình 4.22



Hình 4.22: Kết quả của ví dụ 4.17

4.11 MỘT SỐ VÍ DỤ

4.11.1 Lớp String

Ví dụ 4.18: Chúng ta sẽ xây dựng một lớp xử lý việc tạo và thao tác trên các chuỗi (string). C++ không cài sẵn kiểu dữ liệu chuỗi. Nhưng C++ cho phép chúng ta thêm kiểu chuỗi như một lớp thông qua cơ chế đa năng hóa.



```

130: assert(SubPtr != 0);

131: if ((SubLength == 0) || (Index + SubLength > Length))

132: SubPtr->Length = Length - Index + 1;

133: else

134: SubPtr->Length = SubLength + 1;

135: delete SubPtr->Ptr;

136: SubPtr->Ptr = new char[SubPtr->Length];

137: assert(SubPtr->Ptr != 0);

138: strncpy(SubPtr->Ptr, &Ptr[Index], SubPtr->Length);

139: SubPtr->Ptr[SubPtr->Length] = '\\0';

140: return *SubPtr;

141: }

142:

143: int String::GetLength() const

144: {

145: return Length;

146: }

147: ostream &operator<<(ostream &Output, const String &S)

148: {

149: Output << S.Ptr;

150: return Output;

151: }

152:

153: istream &operator>>(istream &Input, String &S)

154: {

155: char Temp[100];

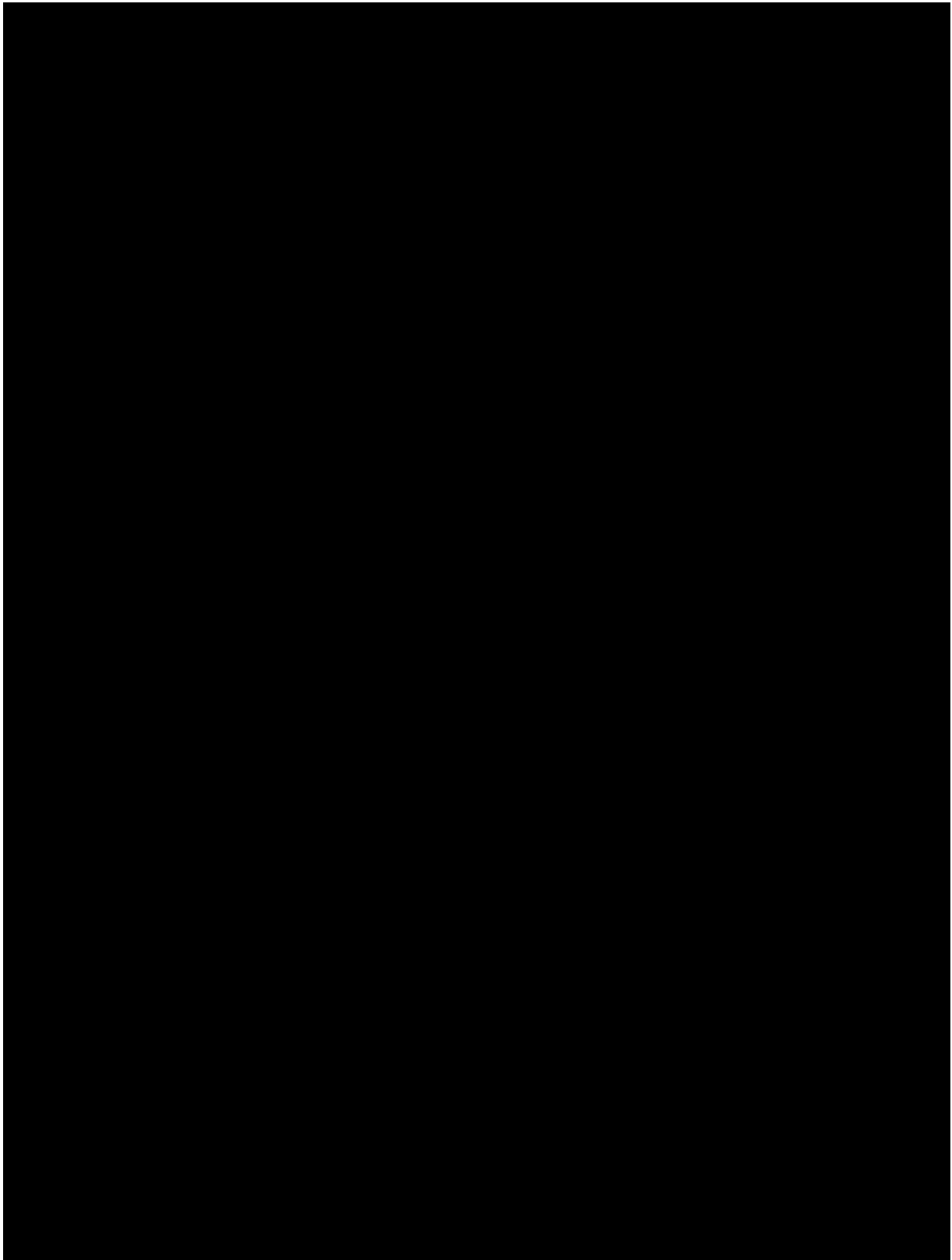
156: Input >> setw(100) >> Temp;

157: S = Temp;

158: return Input;

```

Chúng ta [chạy ví dụ 4.18](#), kết quả ở hình 4.23



Hình 4.23: Kết quả của ví dụ 4.18

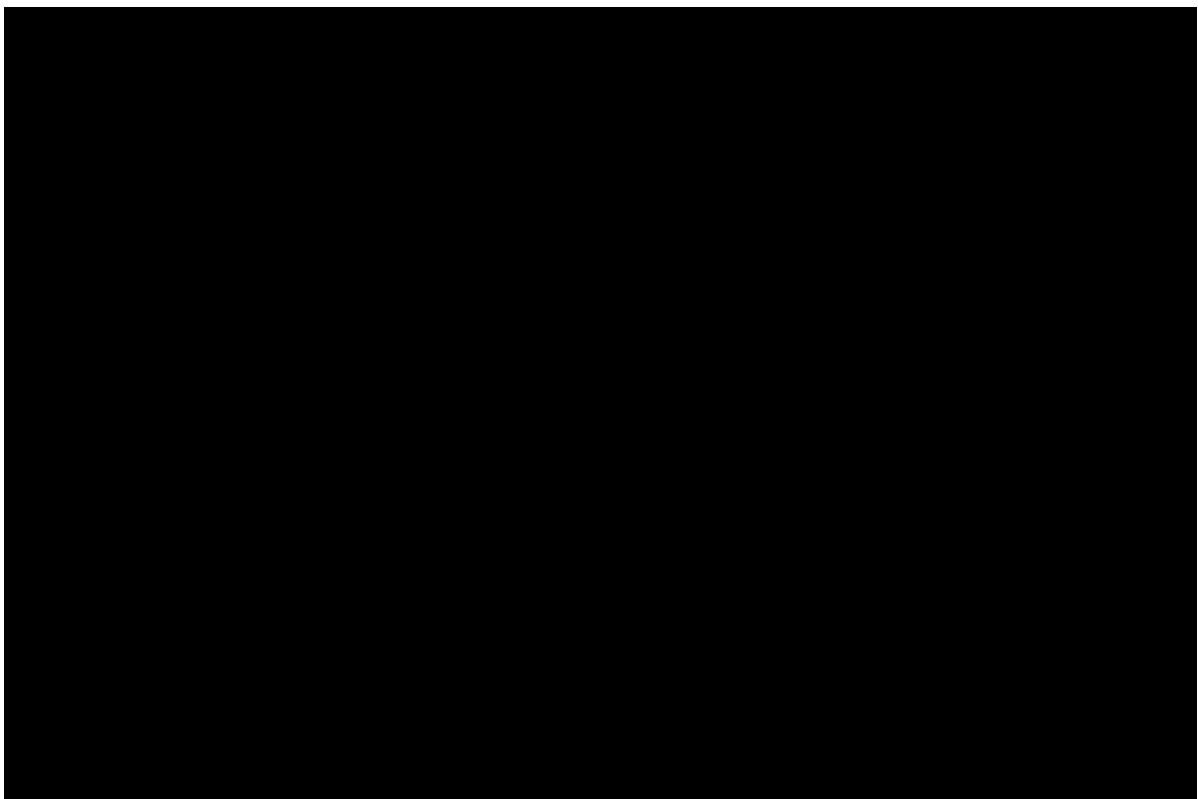
4.11.1 Lớp Date

[Ví dụ 4.19:](#)

```
119: return 0;
```

```
120: }
```

Chúng ta [chạy ví dụ 4.19](#), kết quả ở hình 4.24



Hình 4.24: Kết quả của ví dụ 4.19

5.1 DẪN NHẬP

Trong chương này và chương kế, chúng ta tìm hiểu hai khả năng mà lập trình hướng đối tượng cung cấp là tính kế thừa (inheritance) và tính đa hình (polymorphism). Tính kế thừa là một hình thức của việc sử dụng lại phần mềm trong đó các lớp mới được tạo từ các lớp đã có bằng cách "hút" các thuộc tính và hành vi của chúng và tô điểm thêm với các khả năng mà các lớp mới đòi hỏi. Việc sử dụng lại phần mềm tiết kiệm thời gian trong việc phát triển chương trình. Nó khuyến khích sử dụng lại phần mềm chất lượng cao đã thử thách và gỡ lỗi, vì thế giảm thiểu các vấn đề sau khi một hệ trở thành chính thức. Tính đa hình cho phép chúng ta viết các chương trình trong một kiểu cách chung để xử lý các lớp có liên hệ nhau. Tính kế thừa và tính đa hình các kỹ thuật có hiệu lực đối với sự chia sẻ sự phức tạp của phần mềm.

Khi tạo một lớp mới, thay vì viết các thành viên dữ liệu và các hàm thành viên, lập trình viên có thể thiết kế mà lớp mới được kế thừa các thành viên dữ liệu và các hàm thành viên của lớp trước định nghĩa là lớp cơ sở (base class). Lớp mới được tham chiếu là lớp dẫn xuất (derived class). Mỗi lớp dẫn xuất tự nó trở thành một ứng cử là một lớp cơ sở cho lớp dẫn xuất tương lai nào đó.

Bình thường một lớp dẫn xuất thêm các thành viên dữ liệu và các hàm thành viên, vì thế một lớp dẫn xuất thông thường rộng hơn lớp cơ sở của nó. Một lớp dẫn xuất được chỉ định hơn một lớp cơ sở và biểu diễn một nhóm của các đối tượng nhỏ hơn. Với đối tượng đơn, lớp dẫn xuất, lớp dẫn xuất bắt đầu bên ngoài thực chất giống như lớp cơ sở. Sức mạnh thực sự của sự kế thừa là khả năng định nghĩa trong lớp dẫn xuất các phần thêm, thay thế hoặc tinh lọc các đặc tính kế thừa từ lớp cơ sở.

Mỗi đối tượng của một lớp dẫn xuất cũng là một đối tượng của lớp cơ sở của lớp dẫn xuất đó. Tuy nhiên điều ngược lại không đúng, các đối tượng lớp cơ sở không là các đối tượng của các lớp dẫn xuất của lớp cơ sở đó. Chúng ta sẽ lấy mỗi quan hệ "đối tượng lớp dẫn xuất là một đối tượng lớp cơ sở" để thực hiện các thao tác quan trọng nào đó. Chẳng hạn, chúng ta có thể luận một sự đa dạng của các đối tượng khác nhau có liên quan thông qua sự kế thừa thành danh sách liên kết của các đối tượng lớp cơ sở. Điều này cho phép sự đa dạng của các đối tượng để xử lý một cách tổng quát.

Chúng ta phân biệt giữa "là một" (is a) quan hệ và "có một" (has a) quan hệ. "là một" là sự kế thừa. Trong một "là một" quan hệ, một đối tượng của kiểu lớp dẫn xuất cũng có thể được xử lý như một đối tượng của kiểu lớp cơ sở. "có một" là sự phức hợp (composition). Trong một "có một" quan hệ, một đối tượng lớp có một hay nhiều đối tượng của các lớp khác như là các thành viên, do đó lớp bao các đối tượng này gọi là lớp phức hợp (composed class).

5.2 KẾ THỪA ĐƠN

5.2.1 Các lớp cơ sở và các lớp dẫn xuất

Thường một đối tượng của một lớp thật sự là một đối tượng của lớp khác cũng được. Một hình chữ nhật là một tứ giác, vì thế lớp *Rectangle* có thể kế thừa từ lớp *Quadrilateral*. Trong khung cảnh này, lớp *Quadrilateral* gọi là một lớp cơ sở và lớp *Rectangle* gọi là một lớp dẫn xuất. Hình 5.1 cho chúng ta một vài ví dụ về kế thừa đơn.

Các ngôn ngữ lập trình hướng đối tượng như SMALLTALK sử dụng thuật ngữ khác: Trong kế thừa, lớp cơ sở được gọi là lớp cha (superclass), lớp dẫn xuất được gọi là lớp con (subclass).

Lớp cơ sở	Lớp dẫn xuất
Student	GraduateStudent
	UndergraduateStudent
Shape	Circle
	Triangle
	Rectangle
Loan	CarLoan
	HomeImprovementLoan
	MortgageLoan
Employee	FacultyMember
	StaffMember
Account	CheckingAccount
	SavingsAccount

Hình 5.1: Một vài kế thừa đơn.

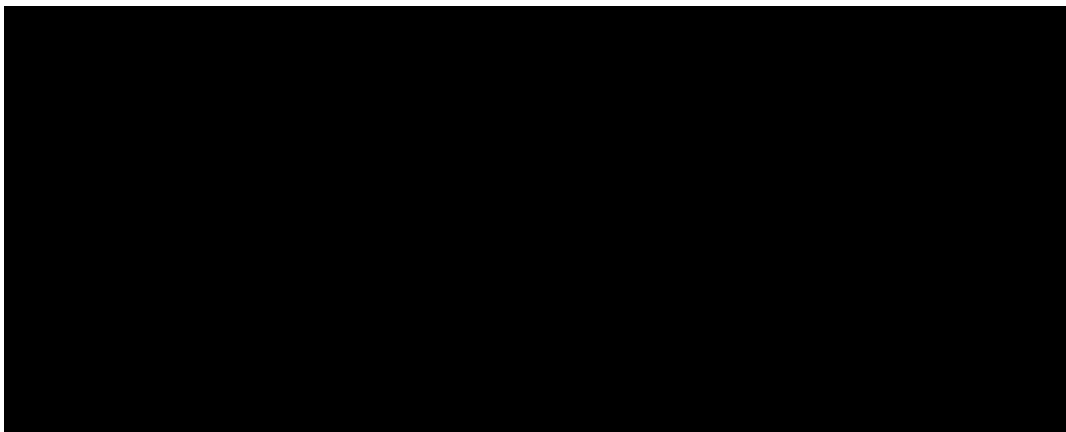
Sự kế thừa hình thành các cấu trúc phân cấp giống cây (còn gọi là cây phả hệ). Một lớp cơ sở tồn tại trong một phân cấp quan hệ với lớp dẫn xuất của nó. Một lớp có thể tồn tại chắc chắn bởi chính nó, nhưng khi một lớp được sử dụng với cơ chế của sự kế thừa thì lớp trở thành hoặc là một lớp cơ sở mà cung cấp các thuộc tính và các hành vi cho các lớp khác, hoặc là lớp trở thành một lớp dẫn xuất mà kế thừa các thuộc tính và các hành vi.

Chúng ta phát triển một phân cấp kế thừa đơn. Một trường đại học cộng đồng đặc thù có hàng ngàn người mà là các thành viên cộng đồng. Những người này gồm các người làm công và các sinh viên. Những người làm công hoặc là các thành viên khoa hoặc các thành viên nhân viên. Các thành viên khoa hoặc là các nhà quản lý hoặc giảng viên. Điều này trở thành phân cấp kế thừa như hình 5.2



Hình 5.2: Một phân cấp kế thừa cho các thành viên của trường đại học cộng đồng.

Phân cấp kế thừa quan trọng khác là phân cấp *Shape* ở hình 5.3.



Hình 5.3: Phân cấp lớp Shape

Để chỉ định lớp *CommissionWorker* được dẫn xuất từ lớp *Employee*, lớp *CommissionWorker* được định nghĩa như sau:

$\};$

kế thừa **private** và kế thừa **protected**. Với kế thừa **public**, các thành viên **public** và **protected** của lớp cơ sở được kế thừa như là các thành viên **public** và **protected** của lớp dẫn xuất tương ứng. Nên nhớ rằng các thành viên **private** của lớp cơ sở không thể truy cập từ các lớp dẫn xuất của lớp đó.

bằng các thuộc tính và các hành vi của lớp cơ sở. Các đối tượng của bất kỳ lớp nào dẫn xuất từ một lớp cơ sở chung có thể tất cả được xử lý như các đối tượng của lớp cơ sở đó.

5.2.2 Các thành viên protected:

Các thành viên **public** của một lớp cơ sở được truy cập bởi tất cả các hàm trong chương trình. Các thành viên **private** của một lớp cơ sở chỉ được truy cập bởi các hàm thành viên và các hàm **friend** của lớp cơ sở.

Truy cập **protected** phục vụ như một mức trung gian của sự bảo vệ giữa truy cập **public** và truy cập **private**. Các thành viên **protected** của một lớp cơ sở có thể chỉ được truy cập bởi các hàm thành viên và các hàm **friend** của lớp cơ sở và bởi các hàm thành viên và các hàm **friend** của lớp dẫn xuất.

Các thành viên lớp dẫn xuất kế thừa **public** có thể tham khảo tới các thành viên **public** và **protected** bằng cách sử dụng các tên thành viên.

5.2.3 Ép kiểu các con trỏ lớp cơ sở tới các con trỏ lớp dẫn xuất

Một đối tượng của một lớp dẫn xuất kế thừa **public** cũng có thể được xử lý như một đối tượng của lớp cơ sở của nó tương ứng. Nhưng ngược lại không đúng: một đối tượng lớp cơ sở cũng không tự động là một đối tượng lớp dẫn xuất.

Tuy nhiên, lập trình viên sử dụng ép kiểu để chuyển đổi một con trỏ lớp cơ sở thành một con trỏ lớp dẫn xuất.

Ví dụ 5.1: Chương trình sau sẽ được chia thành nhiều file (gồm các file .H và .CPP) và tạo một project có tên là CT5_1.PRJ gồm các file .cpp

File POINT.H:

```
1: //POINT.H
2: //Định nghĩa lớp Point
3: #ifndef POINT_H
4: #define POINT_H
5:
6: class Point
7: {
8: protected:
9: float X,Y;
10: public:
11: Point(float A= 0, float B= 0);
12: void SetPoint(float A, float B);
13: float GetX() const
14: {
15: return X;
16: }
17: float GetY() const
18: {
19: return Y;
20: }
21: friend ostream & operator <<(ostream &Output, const Point
&P);
22: };
23:
24: #endif
```

POINT.CPP

```
1: //POINT.CPP

2: //Định nghĩa các hàm thành viên của lớp Point

3: #include <iostream.h>

4: #include "point.h"

5:

6: Point::Point(float A, float B)

7: {

8:     SetPoint(A, B);

9: }

10:

11: void Point::SetPoint(float A, float B)

12: {

13:     X = A;

14:     Y = B;

15: }

16:

17: ostream & operator <<(ostream &Output, const Point &P)

18: {

19:     Output << '[' << P.X << ", " << P.Y << ']'<<endl;

20:     return Output;

21: }
```

File CIRCLE.H

CIRCLE.H

```
1: //CIRCLE.H
2: //Định nghĩa lớp Circle
3: #ifndef CIRCLE_H
4: #define CIRCLE_H
5:
6: #include "point.h"
7: class Circle : public Point
8: {
9: protected:
10: float Radius;
11: public:
12: Circle(float R = 0.0, float A = 0, float B = 0);
13: void SetRadius(float R);
14: float GetRadius() const;
15: float Area() const;
16: friend ostream & operator <<(ostream &Output, const Circle
    &C);
17: };
18:
19: #endif
```

File CIRCLE.CPP

```
1: //CIRCLE.CPP

2: //Định nghĩa các hàm thành viên của lớp Circle

3: #include <iostream.h>

4: #include <iomanip.h>

5: #include "circle.h"

6:

7: Circle::Circle(float R, float A, float B): Point(A, B)

8: {

9: Radius = R;

10: }

11:

12: void Circle::SetRadius(float R)

13: {

14: Radius = R;

15: }

16:

17: float Circle::GetRadius() const

18: {

19: return Radius;

20: }

21:

22: float Circle::Area() const

23: {

24: return 3.14159 * Radius * Radius;

25: }

26:

27: //Xuất một Circle theo dạng: Center = [x, y]; Radius = #.##

28: ostream & operator <<(ostream &Output, const Circle &C)

29: {
```

[REDACTED]

■ **File CT5_1.CPP:**

[REDACTED]

```

1: //CT5_1.CPP

2: //Chương trình 5.1: Ép các con trỏ lớp cơ sở tới các con trỏ
   lớp dẫn xuất

3: #include <iostream.h>

4: #include <iomanip.h>

5: #include "point.h"

6: #include "circle.h"

7:

8: int main()

9: {

10: Point *PointPtr, P(3.5, 5.3);

11: Circle *CirclePtr, C(2.7, 1.2, 8.9);

12: cout << "Point P: "<<P<<endl<<"Circle C: "<<C<< endl;

13 //Xử lý một Circle như một Point (chỉ xem một phần lớp cơ sở)

14: PointPtr = &C;

15: cout << endl << "Circle C (via *PointPtr):
   "<<*PointPtr<<endl;

16 //Xử lý một Circle như một Circle

17: PointPtr = &C;

18: CirclePtr = (Circle *) PointPtr;

19: cout << endl << "Circle C (via *CirclePtr): " << endl

20:          <<*CirclePtr<< endl << "Area of C (via CirclePtr):
   "

21:          << CirclePtr->Area() << endl;

22: //Nguy hiểm: Xem một Point như một Circle

23: PointPtr = &P;

24: CirclePtr = (Circle *) PointPtr;

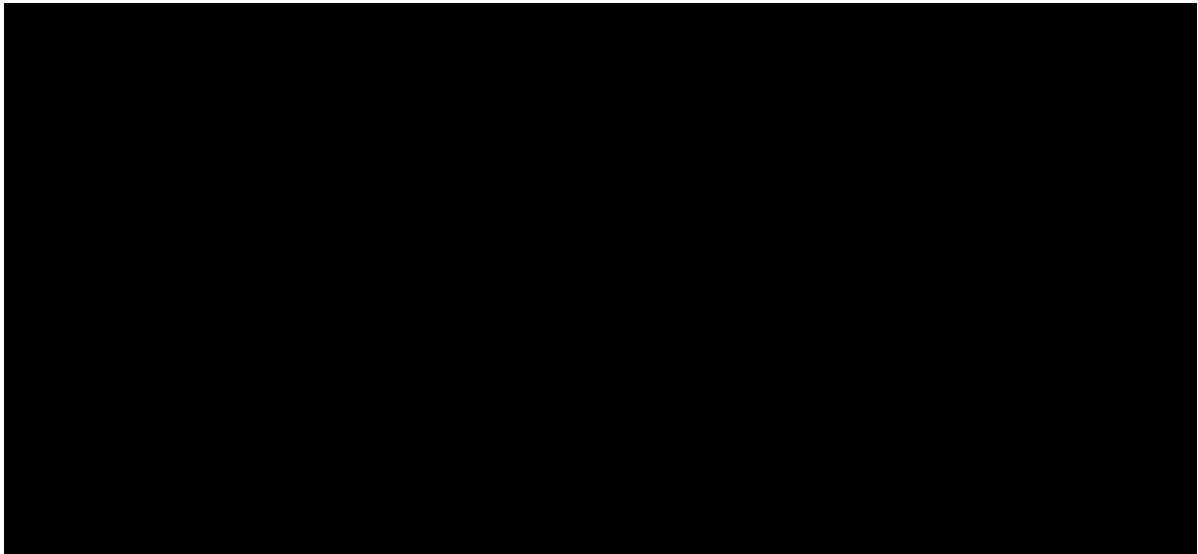
25: cout << endl << "Point P (via *CirclePtr): "<< endl

26:          <<*CirclePtr<< endl << "Area of object CirclePtr
   points to: "

27:          <<CirclePtr->Area() << endl;

```

Chúng ta [chạy ví dụ 5.1](#), kết quả ở hình 5.4



Hình 5.4: Kết quả của ví dụ 5.1

Trong định nghĩa lớp *Point*, các thành viên dữ liệu *X* và *Y* được chỉ định là **protected**, điều này cho phép các lớp dẫn xuất từ lớp *Point* truy cập trực tiếp các thành viên dữ liệu kế thừa. Nếu các thành viên dữ liệu này được chỉ định là **private**, các hàm thành viên **public** của *Point* phải được sử dụng để truy cập dữ liệu, ngay cả bởi các lớp dẫn xuất.

Lớp *Circle* được kế thừa từ lớp *Point* với kế thừa **public** (ở dòng 7 file CIRCLE.H), tất cả các thành viên của lớp *Point* được kế thừa thành lớp *Circle*. Điều này có nghĩa là giao diện **public** bao gồm các hàm thành viên **public** của *Point* cũng như các hàm thành viên *Area()*, *SetRadius()* và *GetRadius()*.

Constructor lớp *Circle* phải bao gồm constructor lớp *Point* để khởi động phần lớp cơ sở của đối tượng lớp *Circle* ở dòng 7 file CIRCLE.CPP, dòng này có thể được viết lại như sau:

```
Circle::Circle(float R, float A, float B)
```

```
    : Point(A, B) //Gọi constructor của lớp cơ sở
```

Các giá trị *A* và *B* được chuyển từ constructor lớp *Circle* tới constructor lớp *Point* để khởi động các thành viên *X* và *Y* của lớp cơ sở. Nếu constructor lớp *Circle* không bao gồm constructor lớp *Point* thì constructor lớp *Point* gọi với các giá trị mặc định cho *X* và *Y* (nghĩa là 0 và 0). Nếu lớp *Point* không cung cấp một constructor mặc định thì trình biên dịch phát sinh lỗi.

Trong chương trình chính (file CT5_1.CPP) gán một con trỏ lớp dẫn xuất (địa chỉ của đối tượng *C*) cho con trỏ lớp cơ sở *PointPtr* và xuất đối tượng *C* của *Circle* bằng toán tử chèn dòng của lớp *Point* (ở dòng 14 và 15). Chú ý rằng chỉ phần *Point* của đối tượng *C* của *Circle* được hiển thị. Nó luôn luôn đúng để gán một con trỏ lớp dẫn xuất cho con trỏ lớp cơ sở bởi vì một đối tượng lớp dẫn xuất là một đối tượng lớp cơ sở. Con trỏ lớp cơ sở chỉ trông thấy phần lớp cơ sở của đối tượng lớp dẫn xuất. Trình biên dịch thực hiện một chuyển đổi ngầm của con trỏ lớp dẫn xuất cho một con trỏ lớp cơ sở.

Sau đó chương trình gán một con trỏ lớp dẫn xuất (địa chỉ của đối tượng *C*) cho con trỏ lớp cơ sở *PointPtr* và ép *PointPtr* trở về kiểu *Circle **. Kết quả của ép kiểu được gán cho *CirclePtr*. Đối tượng *C* của *Circle* được xuất bằng cách sử dụng toán tử chèn dòng của *Circle*. Diện tích của đối tượng *C* được xuất thông qua *CirclePtr*. Các kết quả này là giá trị diện tích đúng bởi vì các con trỏ luôn luôn được trỏ tới một đối tượng *Circle* (từ dòng 17 đến 22).

Kế tiếp, chương trình gán một con trỏ lớp cơ sở (địa chỉ của đối tượng *P*) cho con trỏ lớp cơ sở *PointPtr* và ép *PointPtr* trở về kiểu *Circle **. Kết quả của ép kiểu được gán cho *CirclePtr*. Đối tượng *P* được xuất sử dụng toán tử chèn dòng của lớp *Circle*. Chú ý rằng giá trị xuất của thành viên *Radius* "kỳ lạ". Việc xuất một *Point* như một *Circle* đưa đến một giá trị không hợp lệ cho *Radius* bởi vì các con trỏ luôn được trỏ đến một đối tượng *Point*. Một đối tượng *Point* không có một thành viên *Radius*. Vì thế, chương trình xuất giá trị "rác" đối với thành viên dữ liệu *Radius*. Chú ý rằng giá trị của diện tích là 0.0 bởi vì tính toán này dựa trên giá trị không tồn tại của *Radius* (từ dòng 23 đến 27). Rõ ràng, truy cập các thành viên dữ liệu mà không phải ở đó thì nguy hiểm. Gọi các hàm thành viên mà không tồn tại có thể phá hủy chương trình.

5.2.4 Định nghĩa lại các thành viên lớp cơ sở trong một lớp dẫn xuất:

Một lớp dẫn xuất có thể định nghĩa lại một hàm thành viên lớp cơ sở. Điều này được gọi là overriding. Khi hàm đó được đề cập bởi tên trong lớp dẫn xuất, phiên bản của lớp dẫn xuất được chọn một cách tự động. Toán tử định phạm vi có thể sử dụng để truy cập phiên bản của lớp cơ sở từ lớp dẫn xuất.

Ví dụ 5.2: Định nghĩa một hàm thành viên lớp cơ sở trong lớp dẫn xuất và project có tên là CT5_2.PRJ

■ File EMPLOY.H

EMPLOY.H

```
1: //EMPLOY.H
2: //Định nghĩa lớp Employee
3: #ifndef EMPLOY_H
4: #define EMPLOY_H
5:
6: class Employee
7: {
8: private:
9: char *FirstName;
10: char *LastName;
11: public:
12: Employee(const char *First, const char *Last);
13: void Print() const;
14: ~Employee();
15: };
16:
17: #endif
```

File EMPLOY.CPP

```
1: //EMPLOY.CPP

2: //Định nghĩa các hàm thành viên của lớp Employee

3: #include <string.h>

4: #include <iostream.h>

5: #include <assert.h>

6: #include "employ.h"

7:

8: Employee::Employee(const char *First, const char *Last)

9: {

10:  FirstName = new char[ strlen(First) + 1 ];

11:  assert(FirstName != 0);

12:  strcpy(FirstName, First);

13:  LastName = new char[ strlen(Last) + 1 ];

14:  assert(LastName != 0);

15:  strcpy(LastName, Last);

16: }

17:

18: void Employee::Print() const

19: {

20:  cout << FirstName << ' ' << LastName;

21: }

22:

23: Employee::~~Employee()

24: {

25:  delete [] FirstName;

26:  delete [] LastName;

27: }
```

File HOURLY.H

```
1: //HOURLY.H
2: //Định nghĩa lớp HourlyWorker
3: #ifndef HOURLY_H
4: #define HOURLY_H
5:
6: #include "employ.h"
7:
8: class HourlyWorker : public Employee
9: {
10: private:
11: float Wage; //Tiền lương cho mỗi giờ
12: float Hours; //Số giờ làm việc cho một tuần
13: public:
14: HourlyWorker(const char *First, const char *Last,
15:             float InitHours, float InitWage);
16: float GetPay() const; //Tính toán và trả về lương
17: void Print() const; //Định nghĩa lại Print() của lớp cơ sở
18: };
19:
20: #endif
```

File HOURLY.CPP

```
1: //HOURLY.CPP

2: //Định nghĩa các hàm thành viên của lớp HourlyWorker

3: #include <iostream.h>

4: #include <iomanip.h>

5: #include "hourly.h"

6:

7: HourlyWorker::HourlyWorker(const char *First, const char
*Last,

8:         float InitHours, float InitWage)

9: : Employee(First, Last)

10: {

11:     Hours = InitHours;

12:     Wage = InitWage;

13: }

14:

15: float HourlyWorker::GetPay() const

16: {

17:     return Wage * Hours;

18: }

19:

20: void HourlyWorker::Print() const

21: {

22:     cout << "HourlyWorker::Print()" << endl;

23:     Employee::Print(); //Gọi hàm Print() của lớp cơ sở

24:     cout << " is an hourly worker with pay of"

25:         << " $" << setiosflags(ios::showpoint)

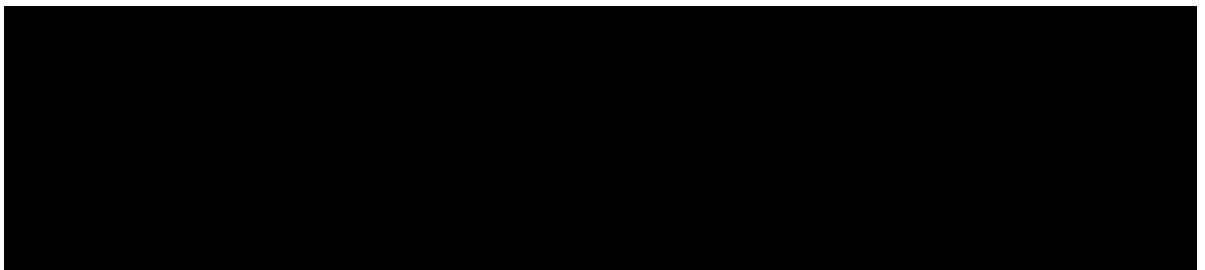
26:         << setprecision(2) << GetPay() << endl;

27: }
```

File CT5_2.CPP

```
1: //CT5_2.CPP
2: #include <iostream.h>
3: #include "hourly.h"
4:
5: int main()
6: {
7:     HourlyWorker H("Bob", "Smith", 40.0, 7.50);
8:     H.Print(); //Gọi hàm HourlyWorker::Print()
9:     cout<<endl<<"Employee::Print()<<endl;
10:    H.Employee::Print();
11:    return 0;
12: }
```

Chúng ta [chạy ví dụ 5.2](#), kết quả ở hình 5.5

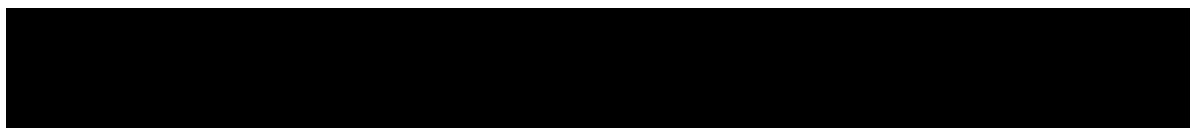


Hình 5.5: Kết quả của ví dụ 5.2

[Ví dụ 5.3](#): Định nghĩa một thành viên dữ liệu của lớp cơ sở trong lớp dẫn xuất.

```
1: //Chương trình 5.3
2: #include <iostream.h>
3: class Base
4: {
5: protected:
6: int Value;
7: public:
8: Base(int X)
9: {
10: Value = X;
11: }
12: };
13:
14: class Derived : public Base
15: {
16: private:
17: int Value; //Định nghĩa lại thành viên dữ liệu
18: public:
19: Derived(int X):Base(X-1)
20: {
21: Value=X;
22: }
23: void Print() const
24: {
25: cout<<"Base::Value="<<Base::Value<<endl;
26: cout<<"Derived::Value="<<Value<<endl;
27: }
28: };
29:
```

Chúng ta [chạy ví dụ 5.3](#), kết quả ở hình 5.6



Hình 5.6: Kết quả của ví dụ 5.3

5.2.5 Các lớp cơ sở **public**, **protected** và **private**

Khi dẫn xuất một lớp từ một lớp cơ sở, lớp cơ sở có thể được kế thừa là **public**, **protected** và **private**.

```
class <drived_class_name> : <type_of_inheritance> <base_class_name>
{
    .....
};
```

Trong đó *type_of_inheritance* là **public**, **protected** hoặc **private**. Mặc định là **private**.

Khi dẫn xuất một lớp từ một lớp cơ sở **public**, các thành viên **public** của lớp cơ sở trở thành các thành viên **public** của lớp dẫn xuất, và các thành viên **protected** của lớp cơ sở trở thành các thành viên **protected** của lớp dẫn xuất. Các thành viên **private** của lớp cơ sở không bao giờ được truy cập trực tiếp từ một lớp dẫn xuất.

Khi dẫn xuất một lớp từ một lớp cơ sở **protected**, các thành viên **public** và **protected** của lớp cơ sở trở thành các thành viên **protected** của lớp dẫn xuất. Khi dẫn xuất một lớp từ một lớp cơ sở **private**, các thành viên **public** và **protected** của lớp cơ sở trở thành các thành viên **private** của lớp dẫn xuất.

Bảng sau (hình 5.6) tổng kết khả năng truy cập các thành viên lớp cơ sở trong một lớp dẫn xuất dựa trên thuộc tính xác định truy cập thành viên của các thành viên trong lớp cơ sở và kiểu kế thừa.

Kiểu kế thừa			
Kế thừa public	Kế thừa protected	Kế thừa private	
public	public trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành	protected trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành	private trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các

	viên không tĩnh, các hàm friend và các hàm không thành viên.	viên không tĩnh, các hàm friend .	hàm thành viên không tĩnh, các hàm friend .
protected	protected trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm friend .	protected trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm friend .	private trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm friend .
private	Dấu trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm friend thông qua các hàm thành viên public và protected của lớp cơ sở.	Dấu trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm friend thông qua các hàm thành viên public và protected của lớp cơ sở.	Dấu trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm friend thông qua các hàm thành viên public và protected của lớp cơ sở.

Hình 5.7: Tổng kết khả năng truy cập thành viên lớp cơ sở trong lớp dẫn xuất.

5.2.6 Các constructor và destructor lớp dẫn xuất

Bởi vì một lớp dẫn xuất kế thừa các thành viên lớp cơ sở của nó (ngoại trừ constructor và destructor), khi một đối tượng của lớp dẫn xuất được khởi động, constructor lớp cơ sở phải được gọi để khởi động các thành viên lớp cơ sở của đối tượng lớp dẫn xuất. Một bộ khởi tạo lớp cơ sở (sử dụng cú pháp giống như bộ khởi tạo thành viên) có thể được cung cấp trong constructor lớp dẫn xuất để gọi tường minh constructor lớp cơ sở, mặt khác constructor lớp dẫn xuất sẽ gọi constructor mặc định lớp cơ sở.

Các constructor lớp cơ sở và các toán tử gán lớp cơ sở không được kế thừa bởi lớp dẫn xuất. Tuy nhiên, các constructor và các toán tử gán lớp dẫn xuất có thể gọi các constructor và các toán tử gán lớp cơ sở.

Một constructor lớp dẫn xuất luôn gọi constructor lớp cơ sở của nó đầu tiên để khởi tạo các thành viên lớp cơ sở của lớp dẫn xuất. Nếu constructor lớp dẫn bị bỏ qua, constructor mặc định lớp dẫn gọi constructor lớp cơ sở. Các destructor được gọi theo thứ tự ngược lại thứ tự gọi các constructor, vì thế destructor lớp dẫn xuất được gọi trước destructor lớp cơ sở của nó.

Ví dụ 5.4: Minh họa thứ tự các constructor và destructor lớp cơ sở và lớp dẫn xuất được gọi và project có tên là CT5_4.PRJ

■ File POINT.H

```
1: //POINT.H
2: //Định nghĩa lớp Point
3: #ifndef POINT_H
4: #define POINT_H
5:
6: class Point
7: {
8: public:
9: Point(float A= 0.0, float B= 0.0);
10: ~Point();
11: protected:
12: float X, Y;
13: };
14:
15: #endif
```

■ File POINT.CPP

POINT.CPP

```
1: //POINT.CPP

2: //Định nghĩa các hàm thành viên lớp Point

3: #include <iostream.h>

4: #include "point.h"

5:

6: Point::Point(float A, float B)

7: {

8:     X = A;

9:     Y = B;

10: cout << "Point constructor: "

11:         << '[' << X << ", " << Y << ']' << endl;

12: }

13:

14: Point::~~Point()

15: {

16: cout << "Point destructor: "

17:         << '[' << X << ", " << Y << ']' << endl;

18: }
```

File CIRCLE.H

CIRCLE.H

```
1: //CIRCLE.H
2: //Định nghĩa lớp Circle
3: #ifndef CIRCLE_H
4: #define CIRCLE_H
5:
6: #include "point.h"
7: #include <iomanip>
8:
9: class Circle : public Point
10: {
11: public:
12: Circle(float R = 0.0, float A = 0, float B = 0);
13: ~Circle();
14: private:
15: float Radius;
16: };
17:
18: #endif
```

File CIRCLE.CPP

CIRCLE.CPP

```
1: //CIRCLE.CPP

2: //Định nghĩa các hàm thành viên lớp Circle

3: #include "circle.h"

4:

5: Circle::Circle(float R, float A, float B): Point(A, B)

6: {

7:     Radius = R;

8:     cout << "Circle constructor: Radius is "

9:         << Radius << " [" << A << ", " << B << "]" << endl;

10: }

11:

12: Circle::~~Circle()

13: {

14:     cout << "Circle destructor: Radius is "

15:         << Radius << " [" << X << ", " << Y << "]" <<

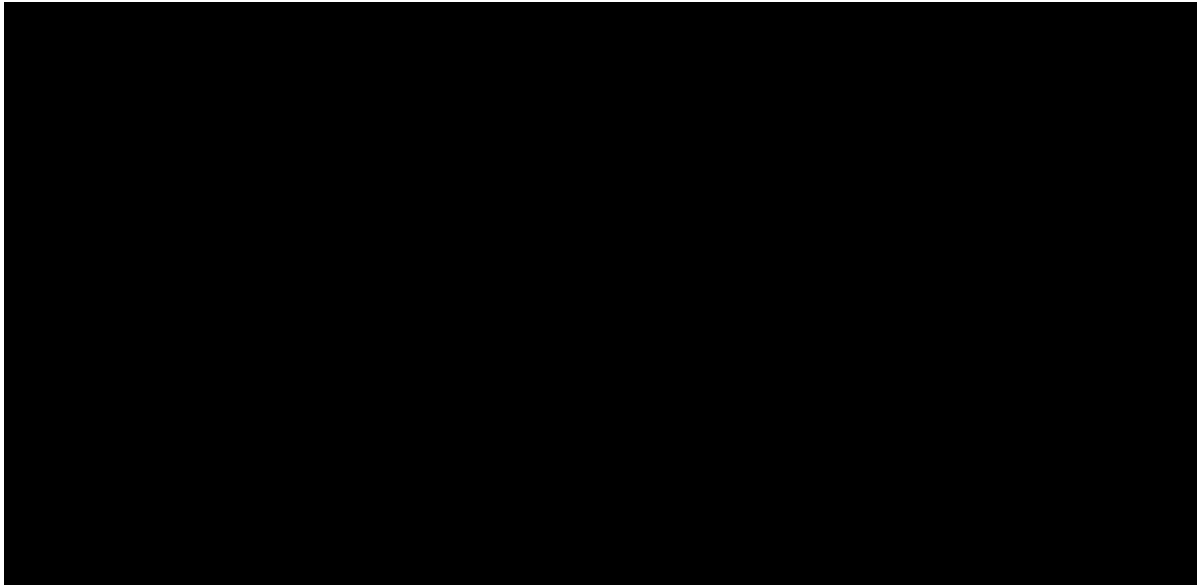
endl;

16: }
```

File CT5_4.CPP

```
1: //CT5_4.CPP
2: //Chương trình 5.4
3: #include <iostream.h>
4: #include "point.h"
5: #include "circle.h"
6: int main()
7: {
8: {
9: Point P(1.1, 2.2);
10: }
11: cout << endl;
12: Circle C1(4.5, 7.2, 2.9);
13: cout << endl;
14: Circle C2(10, 5, 5);
15: cout << endl;
16: return 0;
17: }
```

Chúng ta [chạy ví dụ 5.4](#), kết quả ở hình 5.8



Hình 5.8: Kết quả của ví dụ 5.4

5.2.7 Chuyển đổi ngầm định đối tượng lớp dẫn xuất sang đối tượng lớp cơ sở

Mặc dù một đối tượng lớp dẫn xuất cũng là một đối tượng lớp cơ sở, kiểu lớp dẫn xuất và kiểu lớp cơ sở thì khác nhau. Các đối tượng lớp dẫn xuất có thể được xử lý như các đối tượng lớp cơ sở. Điều này có ý nghĩa bởi vì lớp dẫn xuất có các thành viên tương ứng với mỗi thành viên của lớp cơ sở. Phép gán theo chiều hướng ngược lại là không cho phép bởi vì gán một đối tượng lớp cơ sở cho đối tượng lớp dẫn xuất sẽ cho phép thêm các thành viên lớp dẫn xuất không xác định.

Một con trỏ trỏ tới một đối tượng lớp dẫn xuất có thể được chuyển đổi ngầm định thành một con trỏ trỏ tới một đối tượng lớp cơ sở bởi vì một đối tượng lớp dẫn xuất là một đối tượng lớp cơ sở.

Có bốn cách để trộn và đối sánh các con trỏ lớp cơ sở và các con trỏ lớp dẫn xuất với các đối tượng lớp cơ sở và các đối tượng lớp dẫn xuất:

Tham chiếu tới một đối tượng lớp cơ sở với một con trỏ lớp cơ sở thì không phức tạp.

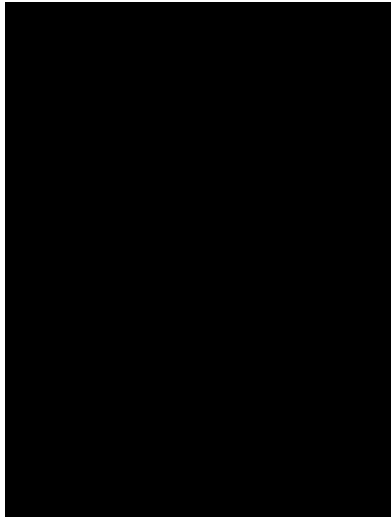
Tham chiếu tới một đối tượng lớp dẫn xuất với một con trỏ lớp dẫn xuất thì không phức tạp.

Tham chiếu tới đối tượng lớp dẫn xuất với một con trỏ lớp cơ sở thì an toàn bởi vì đối tượng lớp dẫn xuất cũng là một đối tượng lớp cơ sở của nó. Như vậy đoạn mã chỉ có thể tham chiếu tới các thành viên lớp cơ sở. Nếu đoạn mã tham chiếu tới các thành viên lớp dẫn xuất thông qua con trỏ lớp cơ sở, trình biên dịch sẽ báo một lỗi về cú pháp.

Tham chiếu tới một đối tượng lớp cơ sở với một con trỏ lớp dẫn xuất thì có lỗi cú pháp. Đầu tiên con trỏ lớp dẫn xuất phải được ép sang con trỏ lớp cơ sở.

5.2.8 Một số ví dụ:

Chúng ta sẽ xem xét một phân cấp các lớp Point, Circle, Cylinder như hình 5.9



Hình 5.9

Ví dụ 5.5: Lớp Point và project có tên là CT5_5.PRJ (gồm các file POINT.CPP, CT5_5.CPP).

■ File POINT.H:


```
1: //POINT.H
2: //Định nghĩa lớp Point
3: #ifndef POINT_H
4: #define POINT_H
5:
6: class Point
7: {
8: protected:
9: float X,Y;
10: public:
11: Point(float A = 0, float B = 0);
12: void SetPoint(float A, float B);
13: float GetX() const
14: {
15: return X;
16: }
17: float GetY() const
18: {
19: return Y;
20: }
21: friend ostream & operator <<(ostream &Output, const Point
&P);
22: };
23:
24: #endif
```

```
1: //POINT.CPP

2: //Định nghĩa các hàm thành viên của lớp Point

3: #include <iostream.h>

4: #include "point.h"

5:

6: Point::Point(float A, float B)

7: {

8:     SetPoint(A, B);

9: }

10:

11: void Point::SetPoint(float A, float B)

12: {

13:     X = A;

14:     Y = B;

15: }

16:

17: ostream & operator <<(ostream &Output, const Point &P)

18: {

19:     Output << '[' << P.X << ", " << P.Y << ']'<<endl;

20:     return Output;

21: }
```

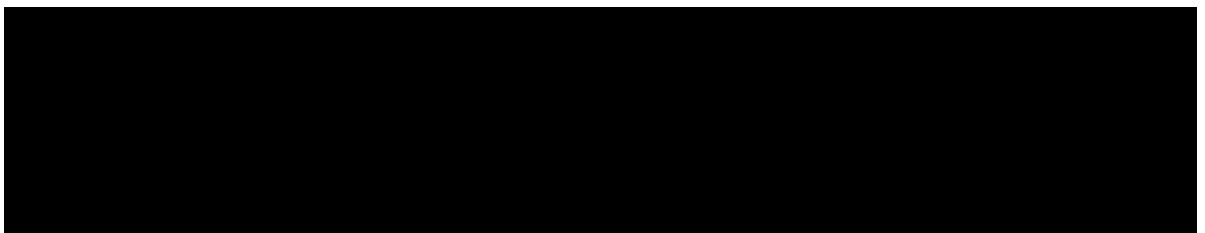
File CT5_5.CPP:

```

1: //CT5_5.CPP
2: //Chương trình 5.5
3: #include <iostream.h>
4: #include "point.h"
5:
6: int main()
7: {
8:     Point P(7.2, 11.5);
9:     cout << "X coordinate is " << P.GetX()
10:          << endl << "Y coordinate is " << P.GetY();
11:     P.SetPoint(10, 10);
12:     cout << endl << endl << "The new location of P is " << P <<
endl;
14:     return 0;
15: }

```

Chúng ta [chạy ví dụ 5.5](#), kết quả ở hình 5.10



Hình 5.10: Kết quả của ví dụ 5.5

[Ví dụ 5.6](#): Lớp Circle và project có tên là CT5_6.PRJ (gồm các file POINT.CPP, CIRCLE.CPP, CT5_6.CPP).

File CIRCLE.H:

```
1: //CIRCLE.H
2: //Định nghĩa lớp Circle
3: #ifndef CIRCLE_H
4: #define CIRCLE_H
5:
6: #include "point.h"
7:
8: class Circle : public Point
9: {
10: protected:
11: float Radius;
12: public:
13: Circle(float R = 0.0, float A = 0, float B = 0);
14: void SetRadius(float R);
15: float GetRadius() const;
16: float Area() const;
17: friend ostream & operator <<(ostream &Output, const Circle
    &C);
18: };
19:
20: #endif
```

File CIRCLE.CPP

```
1: //CIRCLE.CPP

2: //Định nghĩa các hàm thành viên của lớp Circle

3: #include <iostream.h>

4: #include <iomanip.h>

5: #include "circle.h"

6:

7: Circle::Circle(float R, float A, float B) : Point(A, B)

8: {

9: Radius = R;

10: }

11:

12: void Circle::SetRadius(float R)

13: {

14: Radius = R;

15: }

16:

17: float Circle::GetRadius() const

18: {

19: return Radius;

20: }

21:

22: float Circle::Area() const

23: {

24: return 3.14159 * Radius * Radius;

25: }

26:

27: //Xuất một Circle theo dạng: Center = [X, Y]; Radius = #.##

28: ostream & operator <<(ostream &Output, const Circle &C)

29: {
```

File CT5_6.CPP

```
1: //CT5_6.CPP
2: //Chương trình 5.6
3: #include <iostream.h>
4: #include "point.h"
5: #include "circle.h"
6:
7: int main()
8: {
9: Circle C(2.5, 3.7, 4.3);
10: cout << "X coordinate is " << C.GetX()
11:         << endl << "Y coordinate is " << C.GetY()
12:         << endl << "Radius is " << C.GetRadius();
13: C.SetRadius(4.25);
14: C.SetPoint(2, 2);
15: cout << endl << endl << "The new location and radius of C
are"
16:         << endl << C << endl << "Area " << C.Area() <<
endl;
17: Point &PRef = C;
18: cout << endl << "Circle printed as a Point is: "
19:         << PRef << endl;
20: return 0;
21: }
```

Chúng ta [chạy ví dụ 5.6](#), kết quả ở hình 5.11



Hình 5.11: Kết quả của ví dụ 5.6

[Ví dụ 5.7:](#) Lớp Cylinder và project có tên là CT5_7.PRJ (gồm các file POINT.CPP, CIRCLE.CPP, CYLINDR.CPP, CT5_7.CPP).

■ File CYLINDR.H

CYLINDR.H

```
1: //CYLINDR.H
2: //Định nghĩa lớp Cylinder
3: #ifndef CYLINDR_H
4: #define CYLINDR_H
5:
6: #include "circle.h"
7:
8: class Cylinder : public Circle
9: {
10: protected:
11: float Height;
12: public:
13: Cylinder(float H = 0.0, float R = 0.0, float A = 0.0, float B
= 0.0);
14: void SetHeight(float);
15: float GetHeight() const;
16: float Area() const;
17: float Volume() const;
18: friend ostream & operator <<(ostream &Output, const Cylinder
&C);
19: };
20:
21: #endif
```

File CYLINDR.CPP


```

1: //CYLINDR.CPP

2: //Định nghĩa các hàm thành viên lớp Cylinder

3: #include <iostream.h>

4: #include <iomanip.h>

5: #include "cylindr.h"

6:

7: Cylinder::Cylinder(float H, float R, float A, float B) :
Circle(R, A, B)

8: {

9: Height = H;

10: }

11:

12: void Cylinder::SetHeight(float H)

13: {

14: Height = H;

15: }

16:

17: float Cylinder::GetHeight() const

18: {

19: return Height;

20: }

21:

22: //Tính diện tích của Cylinder (diện tích mặt)

23: float Cylinder::Area() const

24: {

25: return 2*Circle::Area()+2 * 3.14159*Radius*Height;

26: }

27:

28: float Cylinder::Volume() const

```

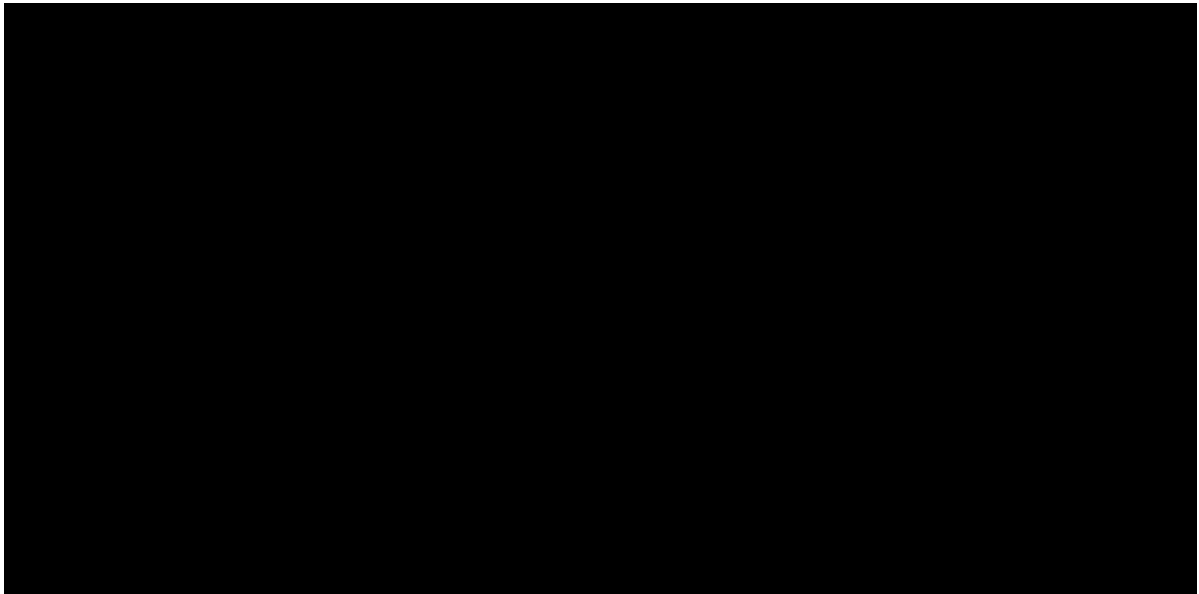
[REDACTED]

■ **File CT5_7.CPP**

[REDACTED]

```
1: //CT5_7.CPP
2: //Chương trình 5.7
3: #include <iostream.h>
4: #include <iomanip.h>
5: #include "point.h"
6: #include "circle.h"
7: #include "cylindr.h"
8: int main()
9: {
10: Cylinder Cyl(5.7, 2.5, 1.2, 2.3);
11: cout << "X coordinate is " << Cyl.GetX() << endl
12:      << "Ycoordinate is " << Cyl.GetY() << endl
13:      << "Radius is " << Cyl.GetRadius() << endl
14:      << "Height is " << Cyl.GetHeight() << endl << endl
15:      ;
16: Cyl.SetHeight(10);
17: Cyl.SetRadius(4.25);
18: Cyl.SetPoint(2, 2);
19: cout << "The new location, radius, "
20:      << "and height of Cyl are:" << endl << Cyl << endl;
21: Point &PRef = Cyl;
22: cout << endl << "Cylinder printed as a Point is: "
23:      << PRef << endl << endl;
24: Circle &cRef = Cyl;
25: cout << "Cylinder printed as a Circle is:" << endl << cRef
26:      << endl << "Area: " << cRef.Area() << endl;
27: return 0;
28: }
```

Chúng ta [chạy ví dụ 5.7](#), kết quả ở hình 5.12



Hình 5.12: Kết quả của ví dụ 5.7

5.3 ĐA KẾ THỪA (MULTIPLE INHERITANCE)

Một lớp có thể được dẫn xuất từ nhiều lớp cơ sở, sự dẫn xuất như vậy được gọi là đa kế thừa. Đa kế thừa có nghĩa là một lớp dẫn xuất kế thừa các thành viên của các lớp cơ sở khác nhau. Khả năng mạnh này khuyến khích các dạng quan trọng của việc sử dụng lại phần mềm, nhưng có thể sinh ra các vấn đề nhập nhằng.

[Ví dụ 5.7](#): Lớp Circle và project có tên là CT5_8.PRJ (gồm các file DIRIVED.CPP, CT5_8.CPP).

■ File BASE1.H

BASE1.H

```
1: //BASE1.H
2: //Định nghĩa lớp Base1
3: #ifndef BASE1_H
4: #define BASE1_H
5:
6: class Base1
7: {
8: protected:
9: int Value;
10: public:
11: Base1(int X)
12: {
13: Value = X;
14: }
15: int GetData() const
16: {
17: return Value;
18: }
19: };
20:
21: #endif
```

File BASE2.H

BASE2.H

```
1: //BASE2.H
2: //Định nghĩa lớp Base2
3: #ifndef BASE2_H
4: #define BASE2_H
5:
6: class Base2
7: {
8: protected:
9: char Letter;
10: public:
11: Base2(char C)
12: {
13: Letter = C;
14: }
15: char GetData() const
16: {
17: return Letter;
18: }
19: };
20:
21: #endif
```

File DERIVED.H

DERIVED.H

```
1: //DERIVED.H

2: //Định nghĩa lớp Derived mà kế thừa từ nhiều lớp cơ sở (Base1
  & Base2)

3: #ifndef DERIVED_H

4: #define DERIVED_H

5:

6: #include "base1.h"

7: #include "base2.h"

8:

9: class Derived : public Base1, public Base2

10: {

11: private:

12: float Real;

13: public:

14: Derived(int, char, float);

15: float GetReal() const;

16: friend ostream & operator <<(ostream &Output, const Derived
  &D);

17: };

18:

19: #endif
```

File DERIVED.CPP

DERIVED.CPP

```
1: //DERIVED.CPP

2: //Định nghĩa các hàm thành viên của lớp Derived

3: #include <iostream.h>

4: #include "derived.h"

5:

6: Derived::Derived(int I, char C, float F)

7: : Base1(I), Base2(C) //Gọi các constructor lớp cơ sở

8: {

9: Real = F;

10: }

11:

12: float Derived::GetReal() const

13: {

14: return Real;

15: }

16:

17: ostream & operator <<(ostream &Output, const Derived &D)

18: {

19: Output << " Integer: " << D.Value << endl

20: << " Character: " << D.Letter << endl

21: << "Real number: " << D.Real;

22: return Output;

23: }
```

File CT5_8.CPP


```

1: //CT5_8.CPP

2: //Chương trình 5.8

3: #include <iostream.h>

4: #include "base1.h"

5: #include "base2.h"

6: #include "derived.h"

7:

8: int main()

9: {

10: Base1 B1(10), *Base1Ptr;

11: Base2 B2('Z'), *Base2Ptr;

12: Derived D(7, 'A', 3.5);

13: cout << "Object B1 contains integer "

14:         << B1.GetData() << endl

15:         << "Object B2 contains character "

16:         << B2.GetData() << endl

17:         << "Object D contains:" << endl << D << endl <<
endl;

18: cout << "Data members of Derived can be"

19:         << " accessed individually:" << endl

20:         << " Integer: " << D.Base1::GetData() << endl

21:         << " Character: " << D.Base2::GetData() << endl

22:         << "Real number: " << D.GetReal() << endl << endl;

23: cout << "Derived can be treated as an "

24:         << "object of either base class:" << endl;

25: Base1Ptr = &D;

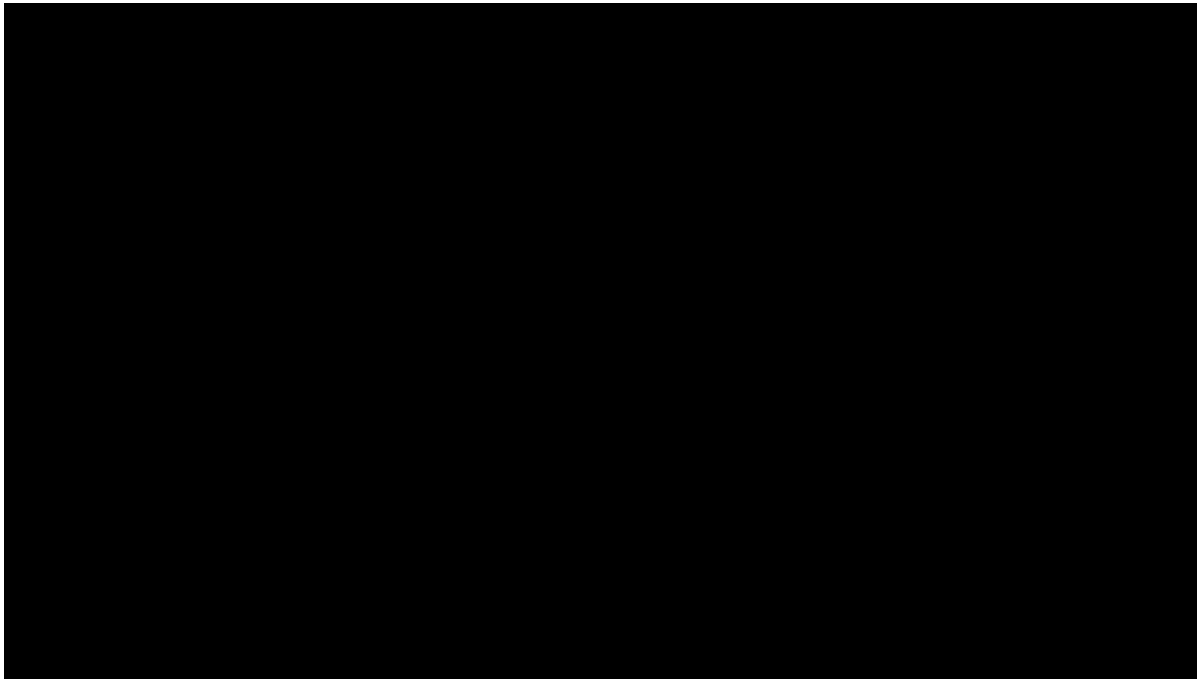
26: cout << "Base1Ptr->GetData() yields "

27:         << Base1Ptr->GetData() << endl ;

28: Base2Ptr = &D;

```

Chúng ta [chạy ví dụ 5.8](#), kết quả ở hình 5.13



Hình 5.13: Kết quả của ví dụ 5.8

Việc kế thừa nhiều lớp cơ sở tạo ra một loạt các điểm nhập nhằng trong các chương trình C++. Chẳng hạn, trong các chương trình của ví dụ 5.8, nếu thực hiện lệnh:

```
D.GetData();
```

thì trình biên dịch (Borland C++ 3.1) sẽ báo lỗi:

Member is ambiguous: 'Base1::GetData' and 'Base2::GetData'

Bởi vì lớp *Derived* kế thừa hai hàm khác nhau có cùng tên là *GetData()* từ hai lớp cơ sở của nó. *Base1::GetData()* là hàm thành viên **public** của lớp cơ sở **public**, và nó trở thành một hàm thành viên **public** của *Derived*. *Base2::GetData()* là hàm thành viên **public** của lớp cơ sở **public**, và nó trở thành một hàm thành viên **public** của *Derived*. Do đó trình biên dịch không thể xác định hàm thành viên *GetData()* của lớp cơ sở nào để gọi thực hiện. Vì vậy, chúng ta phải sử dụng tên lớp cơ sở và toán tử định phạm vi để xác định hàm thành viên của lớp cơ sở lúc gọi hàm *GetData()*.

Cú pháp của một lớp kế thừa nhiều lớp cơ sở:

```
class <drived_class_name> : <type_of_inheritance> <base_class_name1> ,  
                                <type_of_inheritance>  
                                <base_class_name2>, ...  
{
```

```
.....  
};
```

Trình tự thực hiện constructor trong đa kế thừa: constructor lớp cơ sở xuất hiện trước sẽ thực hiện trước và cuối cùng mới tới constructor lớp dẫn xuất. Đối với destructor có trình tự thực hiện theo thứ tự ngược lại.

5.4 CÁC LỚP CƠ SỞ ẢO (VIRTUAL BASE CLASSES)

Chúng ta không thể khai báo hai lần cùng một lớp trong danh sách của các lớp cơ sở cho một lớp dẫn xuất. Tuy nhiên vẫn có thể có trường hợp cùng một lớp cơ sở được đề cập nhiều hơn một lần trong các lớp tổ tiên của một lớp dẫn xuất. Điều này phát sinh lỗi vì không có cách nào để phân biệt hai lớp cơ sở gốc.

■ [Ví dụ 5.9:](#)

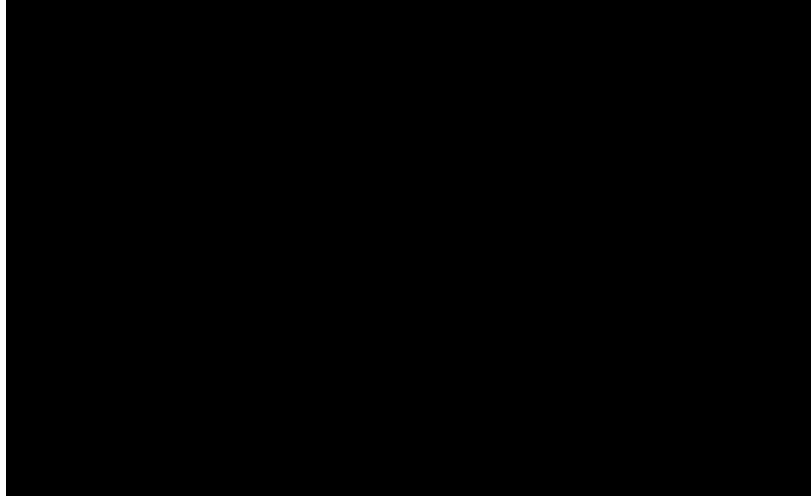
```
.....
```

```
1: //Chương trình 5.9
2: #include <iostream.h>
3: class A
4: {
5: public:
6: int X1;
7: };
8:
9: class B : public A
10: {
11: public:
12: float X2;
13: };
14:
15: class C : public A
16: {
17: public:
18: double X3;
19: };
20:
21: class D : public B,public C
22: {
23: public:
24: char X4;
25: };
26:
27: int main()
28: {
29: D Obj;
```



Khi biên dịch chương trình ở ví dụ 5.9, trình biên dịch sẽ báo lỗi ở dòng 31 và 34:

Member is ambiguous: 'A::XI' and 'A::XI'



Hình 5.14

Ở đây chúng ta thấy có hai lớp cơ sở A cho lớp D , và trình biên dịch không thể nào nhận biết được việc truy cập XI được kế thừa thông qua B hoặc truy cập XI được kế thừa thông qua C . Để khắc phục điều này, chúng ta chỉ định một cách tường minh trong lớp D như sau:

`Obj.C::XI=0;`

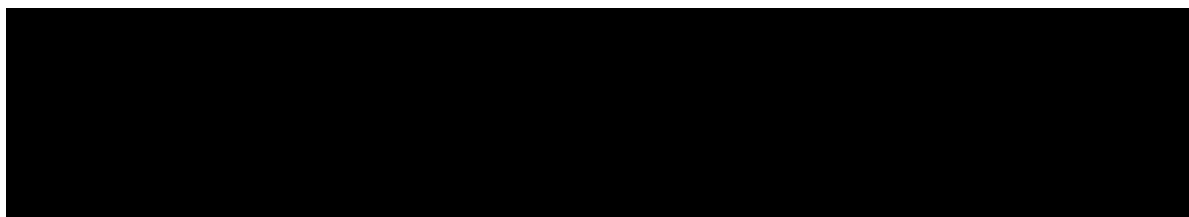
Tuy nhiên, đây cũng chỉ là giải pháp có tính chấp vá, bởi thực chất XI nào trong trường hợp nào cũng được. Giải pháp cho vấn đề này là khai báo A như lớp cơ sở kiểu **virtual** cho cả B và C . Khi đó chương trình ở ví dụ 5.9 được viết lại như sau:

■ Ví dụ 5.10:



```
1: //Chương trình 5.10
2: #include <iostream.h>
3: class A
4: {
5: public:
6: int X1;
7: };
8:
9: class B : virtual public A
10: {
11: public:
12: float X2;
13: };
14:
15: class C : virtual public A
16: {
17: public:
18: double X3;
19: };
20:
21: class D : public B,public C
22: {
23: public:
24: char X4;
25: };
26:
27: int main()
28: {
29: D Obj;
```

Chúng ta chạy ví dụ 5.10, kết quả ở hình 5.15



Hình 5.15: Kết quả của ví dụ 5.10

Các lớp cơ sở kiểu **virtual**, có cùng một kiểu lớp, sẽ được kết hợp để tạo một lớp cơ sở duy nhất có kiểu đó cho bất kỳ lớp dẫn xuất nào kế thừa chúng. Hai lớp cơ sở *A* ở trên bất giờ sẽ trở thành một lớp cơ sở *A* duy nhất cho bất kỳ lớp dẫn xuất nào từ *B* và *C*. Điều này có nghĩa là *D* chỉ có một cơ sở của lớp *A*, vì vậy tránh được sự nhập nhằng.

6.1 DẪN NHẬP

Tính đa hình (polymorphism) là khả năng thiết kế và cài đặt các hệ thống mà có thể mở rộng dễ dàng hơn. Các chương trình có thể được viết để xử lý tổng quát – như các đối tượng lớp cơ sở – các đối tượng của tất cả các lớp tồn tại trong một phân cấp. Khả năng cho phép một chương trình sau khi đã biên dịch có thể có nhiều diễn biến xảy ra là một trong những thể hiện của tính đa hình – tính muôn màu muôn vẻ – của chương trình hướng đối tượng, một thông điệp được gởi đi (gởi đến đối tượng) mà không cần biết đối tượng nhận thuộc lớp nào. Để thực hiện được tính đa hình, các nhà thiết kế C++ cho chúng ta dùng cơ chế kết nối động (dynamic binding) thay cho cơ chế kết nối tĩnh (static binding) ngay khi chương trình biên dịch được dùng trong các ngôn ngữ cổ điển như C, Pascal, ...

6.2 PHƯƠNG THỨC ẢO (VIRTUAL FUNCTION)

Khi xây dựng các lớp của một chương trình hướng đối tượng để tạo nên một cấu trúc phân cấp hoặc cây phả hệ, người lập trình phải chuẩn bị các hành vi giao tiếp chung của các lớp đó. Hành vi giao tiếp chung sẽ được dùng để thể hiện cùng một hành vi, nhưng có các hành động khác nhau, đó chính là phương thức ảo. Đây là một phương thức tồn tại để có hiệu lực nhưng không có thực trong lớp cơ sở, còn trong các lớp dẫn xuất. Như vậy phương thức ảo chỉ được xây dựng khi có một hệ thống cây phả hệ. Phương thức này sẽ được gọi thực hiện từ thực thể của lớp dẫn xuất nhưng mô tả về chúng trong lớp cơ sở.

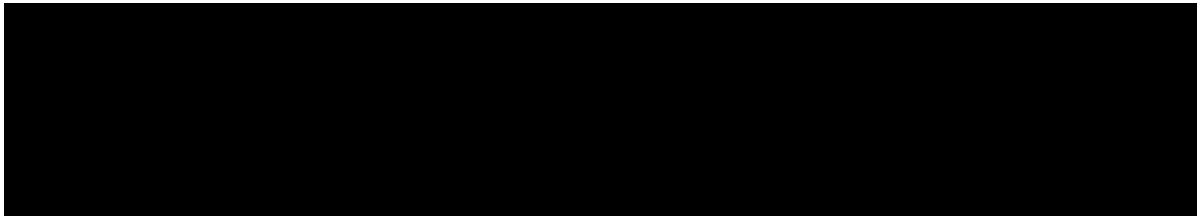
Chúng ta khai báo phương thức ảo bằng thêm từ khóa **virtual** ở phía trước. Khi đó các phương thức có cùng tên với phương thức này trong các lớp dẫn xuất cũng là phương thức ảo.

■ Ví dụ 6.1:



```
1: //Chương trình 6.1
2: #include <iostream.h>
3:
4: class Base
5: {
6: public:
7: virtual void Display()
8: {
9: cout<<"class Base"<<endl;
10: }
11: };
12:
13: class Derived : public Base
14: {
15: public:
16: virtual void Display()
17: {
18: cout<<"class Derived"<<endl;
19: }
20: };
21:
21: void Show(Base *B)
22: {
23: B->Display(); //Con trỏ B chỉ đến phương thức Display() nào
(của lớp Base
24 //hoặc lớp Derived) tùy vào lúc chạy chương trình.
25: }
26: int main()
27: {
```


Chúng ta [chạy ví dụ 6.1](#), kết quả ở hình 6.1



Hình 6.1: Kết quả của ví dụ 6.1

Trong ví dụ 6.1, lớp cơ sở *Base* có phương thức *Display()* được khai báo là phương thức ảo. Phương thức này trong lớp dẫn xuất *Derived* được định nghĩa lại nhưng cũng là một phương thức ảo. Thật ra, không ra không có khai báo **virtual** cho phương thức *Display()* của lớp *Derived* cũng chẳng sao, trình biên dịch vẫn hiểu đó là phương thức ảo. Tuy nhiên, khai báo **virtual** rõ ràng ở các lớp dẫn xuất làm cho chương trình trong sáng, dễ hiểu hơn. Hai dòng 30 và 31, chúng ta biết chắc phương thức *Display()* của lớp nào được gọi (của lớp *Base* hoặc lớp *Derived*). Nhưng hai dòng 32 và 33, nếu không có cơ chế kết nối động, chúng ta đoán rằng việc gọi hàm *Show()* sẽ luôn luôn kéo theo phương thức *Base::Display()*. Quả vậy, bỏ đi khai báo **virtual** cho phương thức *Base::Display()*, khi đó dòng lệnh:

```
Show(D);
```

gọi đến *Base::Display()* vì đối tượng lớp dẫn xuất cũng là đối tượng lớp cơ sở (nghĩa là tdb tự động chuyển đổi kiểu: đối tượng *D* kiểu *Derived* chuyển thành kiểu *Base*).

Nhờ khai báo **virtual** cho phương thức *Base::Display()* nên sẽ không thực hiện gọi phương thức *Base::Display()* một cách cứng nhắc trong hàm *Show()* mà chuẩn bị một cơ chế mềm dẻo cho việc gọi phương thức *Display()* tùy thuộc vào sự xác định kiểu của tham số vào lúc chạy chương trình.

Cơ chế đó ra sao? Khi nhận thấy có khai báo **virtual** trong lớp cơ sở, trình biên dịch sẽ thêm vào mỗi đối tượng của lớp cơ sở và các lớp dẫn xuất của nó một con trỏ chỉ đến bảng phương thức ảo (virtual function table). Con trỏ đó có tên là *vptr* (virtual pointer). Bảng phương thức ảo là nơi chứa các con trỏ chỉ đến đoạn chương trình đã biên dịch ứng với các phương thức ảo. Mỗi lớp có một bảng phương thức ảo. Trình biên dịch chỉ lập bảng phương thức ảo khi bắt đầu có việc tạo đối tượng của lớp. Đến khi chương trình chạy, phương thức ảo của đối tượng mới được nối kết và thi hành thông qua con trỏ *vptr*.

Trong ví dụ 6.1, lệnh gọi hàm:

```
Show(D);
```

đối tượng *D* thuộc lớp *Derived* tuy bị chuyển đổi kiểu thành một đối tượng thuộc lớp *Base* nhưng nó không hoàn toàn giống một đối tượng của *Base* chính cống như *B*. Nếu như con trỏ *vptr* trong *B* chỉ đến vị trí trên bảng phương thức ảo ứng với phương thức *Base::Display()*, thì con trỏ *vptr* trong *D* vẫn còn chỉ đến phương thức *Derived::Display()* cho dù *D* bị chuyển kiểu thành *Base*. Đó là lý do tại sao lệnh:

```
Show(D);
```

gọi đến phương thức *Derived::Display()*.

Các đặc trưng của phương thức ảo:

- Phương thức ảo không thể là các hàm thành viên tĩnh.
- Một phương thức ảo có thể được khai báo là **friend** trong một lớp khác nhưng các hàm **friend** của lớp thì không thể là phương thức ảo.
- Không cần thiết phải ghi rõ từ khóa **virtual** khi định nghĩa một phương thức ảo trong lớp dẫn xuất (để cũng chẳng ảnh hưởng gì).
- Để sự kết nối động được thực hiện thích hợp cho từng lớp dọc theo cây phả hệ, một khi phương thức nào đó đã được xác định là ảo, từ lớp cơ sở đến các lớp dẫn xuất đều phải định nghĩa thống nhất về tên, kiểu trả về và danh sách các tham số. Nếu đối với phương thức ảo ở lớp dẫn xuất, chúng ta lại sơ suất định nghĩa các tham số khác đi một chút thì trình biên dịch sẽ xem đó là phương thức khác. Đây chính là điều kiện để kết nối động.

■ Ví dụ 6.2:

```
1: //Chương trình 6.2
2: #include <iostream.h>
3:
4: class Base
5: {
6: public:
7: virtual void Print(int A,int B);
8: };
9:
10: class Derived : public Base
11: {
12: public:
13: virtual void Print(int A,double D);
14: };
15:
16: void Base::Print(int A,int B)
17: {
18: cout<<"A="<<A<<" ,B="<<B<<endl;
19: }
20:
21: void Derived::Print(int A,double D)
22: {
23: cout<<"A="<<A<<" ,D="<<D<<endl;
24: }
25:
26: void Show(Base *B)
27: {
28: B->Print(3,5);
29: }
```

Chúng ta [chạy ví dụ 6.2](#), kết quả ở hình 6.2



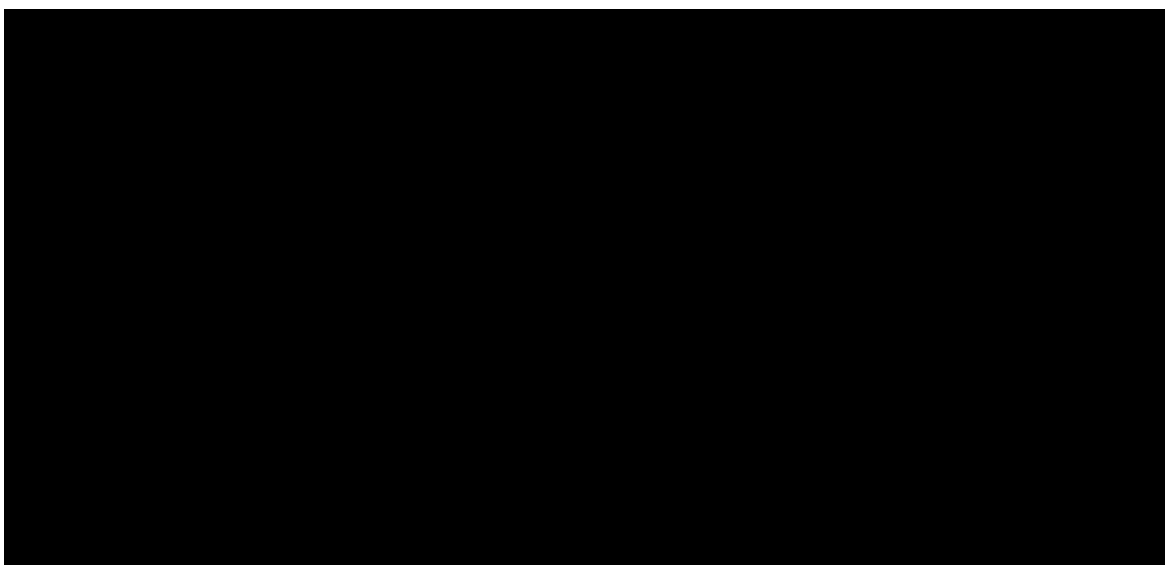
Hình 6.2: Kết quả của ví dụ 6.2

Trong ví dụ 6.2, trong lớp cơ sở *Base* và lớp dẫn xuất *Derived* đều có phương thức ảo *Print()*. Nhưng quan sát kỹ chúng ta, phương thức *Print()* trong lớp *Derived* có tham số thứ hai khác kiểu với phương thức *Print()* trong lớp *Base*. Vì thế, chúng ta không thể chờ đợi lệnh ở dòng 36 sẽ gọi đến phương thức *Derived::Print(int,double)*. Phương thức *Derived::Print(int,double)* nằm ngoài đường dây phương thức ảo nên hàm *Show()* chỉ luôn gọi đến phương thức *Derived::Print(int,int)* mà thôi. Do có khai báo *virtual* đối với phương thức *Derived::Print(int,double)*, chúng ta có thể nói phương thức này sẽ mở đầu cho một đường dây phương thức ảo *Print(int,double)* mới nếu sau lớp *Derived* còn có các lớp dẫn xuất của nó.

6.3 LỚP TRỪU TƯỢNG (ABSTRACT CLASS)

Trong quá trình thiết kế chương trình theo hướng đối tượng, để tạo nên một hệ thống phả hệ mang tính kế thừa cao, người lập trình phải đoán trước sự phát triển của cấu trúc, từ đó chọn lựa những thành viên phù hợp cho các lớp ở trên cùng. Rõ ràng đây là một công việc vô cùng khó khăn. Để tránh tình trạng người lập trình xây dựng các đối tượng lãng phí bộ nhớ, ngôn ngữ C++ cho phép chúng ta thiết kế các lớp có các không phương thức ảo không làm gì cả, và cũng không thể tạo ra đối tượng thuộc lớp đó. Những lớp như vậy gọi là lớp trừu tượng.

Trong cấu trúc trên hình 6.3, không phải lớp nào cũng thực sự cần đến phương thức *Print()*, nhưng nó có mặt khắp nơi để tạo ra bộ mặt chung cho mọi lớp trong cấu trúc cây. Phương thức của lớp trên cùng như *A::Print()* thường là phương thức ảo để có được tính đa hình.



Hình 6.3

Nhờ đó, với hàm sau:

```
void Show(A* a)
{
    a->Print();
}
```

chúng ta có thể truyền đối tượng đủ kiểu cho nó (A , B , C , D hoặc E) mà vẫn gọi đến đúng phương thức `Print()` phù hợp đủ kiểu của đối tượng lúc biên dịch vẫn còn chưa biết. Với vai trò "lót đường" như vậy, phương thức $A::Print()$ có thể chẳng có nội dung gì cả

```
class A
{
    public:
        virtual void Print()
        {
        }
};
```

Khi đó người ta gọi phương thức $A::Print()$ là phương thức ảo rỗng (null virtual function), nó chẳng làm gì hết. Tuy nhiên lớp A vẫn là một lớp bình thường, chúng ta có thể tạo ra một đối tượng thuộc nó, có thể truy cập tới phương thức $A::Print()$. Để tránh tình trạng vô tình tạo ra đối tượng thuộc lớp này, người ta thường xây dựng lớp trừu tượng, trình biên dịch cho phép tạo ra lớp có các phương thức thuần ảo (pure virtual function) như sau:

```
class A
{
    public:
        virtual void Print() = 0;
};
```

Phương thức ảo $Print()$ bây giờ là phương thức thuần ảo – phương thức có tên được gán bởi giá trị zero. Lớp A chứa phương thức thuần ảo được gọi là lớp trừu tượng.

■ Ví dụ 6.3:

```
1: //Chương trình 6.3
2: #include <iostream.h>
3:
4: class A
5: {
6: public:
7: virtual void Print()=0; //Phương thức thuần ảo
8: };
9:
10: class B : public A
11: {
12: public:
13: virtual void Print()
14: {
15: cout<<"Class B"<<endl;
16: }
17: };
18:
19: class C : public B
20: {
21: public:
22: virtual void Print()
23: {
24: cout<<"Class C"<<endl;
25: }
26: };
27:
28: void Show(A *a)
29: {
```

Chúng ta [chạy ví dụ 6.3](#), kết quả ở hình 6.4



Hình 6.4: Kết quả của ví dụ 6.3

Lớp *A* được tạo ra để làm cơ sở cho việc hình thành các lớp con cháu (*B* và *C*). Nhờ có Phương thức ảo *Print()* từ lớp *A* cho tới lớp *C*, tính đa hình được thể hiện.

Lưu ý:

- Chúng ta không thể tạo ra một đối tượng của lớp trừu tượng, nhưng hoàn toàn có thể tạo ra một con trỏ trỏ đến lớp này (vì con trỏ không phải là đối tượng thuộc lớp) hoặc là một tham chiếu.
- Nếu trong lớp kế thừa từ lớp trừu tượng chúng ta không định nghĩa phương thức thuần ảo, do tính kế thừa nó sẽ bao hàm phương thức thuần ảo của lớp cơ sở, nên lớp dẫn xuất này sẽ trở thành lớp trừu tượng.
- Theo định nghĩa lớp trừu tượng, nếu trong lớp dẫn xuất (từ lớp cơ sở trừu tượng) chúng ta định nghĩa thêm một phương thức thuần ảo khác, lớp này cũng sẽ trở thành lớp trừu tượng.

6.4 CÁC THÀNH VIÊN ẢO CỦA MỘT LỚP

6.4.1 Toán tử ảo

Toán tử thực chất cũng là một hàm nên chúng ta có thể tạo ra các toán tử ảo trong một lớp. Tuy nhiên do đa năng hóa khi tạo một toán tử cần chú ý đến các kiểu của các toán hạng phải sử dụng kiểu của lớp cơ sở gốc có toán tử ảo.

[Ví dụ 6.4:](#) Đa năng hóa toán tử với hàm toán tử là phương thức ảo.



```

130: AddObject(a,b);

131: a.Print("a");

132: AddObject(b,c);

133: b.Print("b");

134: AddObject(c,a);

135: c.Print("c");

136: a=b+c;

137: a.Print("a");

138: c=c+a;

139: c.Print("c");

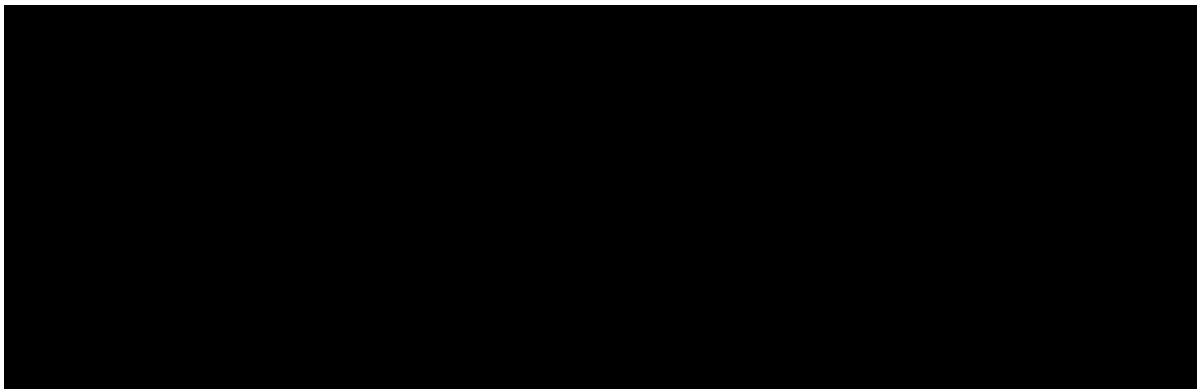
140: return 0;

141: }

```



Chúng ta [chạy ví dụ 6.4](#), kết quả ở hình 6.5



Hình 6.5: Kết quả của ví dụ 6.4

6.4.2 Có constructor và destructor ảo hay không?:

Khi một đối tượng thuộc lớp có phương thức ảo, để thực hiện cơ chế kết nối động, trình biên dịch sẽ tạo thêm một con trỏ vptr như một thành viên của lớp, con trỏ này có nhiệm vụ quản lý địa chỉ của phương thức ảo. Một lớp chỉ có một bảng phương thức ảo, trong khi đó có thể có nhiều đối tượng thuộc lớp, nên khi một đối tượng khác thuộc cùng lớp tạo ra thì con trỏ vptr đã còn tại. Chính vì vậy bảng phương thức ảo phải được tạo ra trước khi gọi thực hiện constructor, nên constructor không thể là phương thức ảo. Ngược lại do một lớp chỉ có một bảng phương thức ảo nên khi một đối tượng thuộc lớp bị hủy bỏ, bảng phương thức ảo vẫn còn đó, và con trỏ vptr vẫn còn đó. Hơn nữa, destructor được gọi thực hiện trước khi vùng nhớ dành cho đối tượng bị thu hồi, do đó destructor có

thể là phương thức ảo. Tuy nhiên, constructor của một lớp có thể gọi phương thức ảo khác. Điều này hoàn toàn không có gì mâu thuẫn với cơ chế kết nối động.

Ví dụ 6.5:



```
1: //Chương trình 6.5: Destructor ảo
2: #include <iostream.h>
3:
4: class Base
5: {
6: public:
7: virtual ~Base()
8: {
9: cout<<"~Base"<<endl;
10: }
11: };
12:
13: class Derived:public Base
14: {
15: public:
16: virtual ~Derived()
17: {
18: cout<<"~Derived"<<endl;
18: }
19: };
20:
21: int main()
22: {
23: Base *B;
24: B = new Derived;
25: delete B;
26: return 0;
27: }
```

Chúng ta [chạy ví dụ 6.5](#), kết quả ở hình 6.6



Hình 6.6: Kết quả của ví dụ 6.5

Nếu destructor không là phương thức ảo thì khi giải phóng đối tượng B chỉ có destructor của lớp cơ sở được gọi mà thôi nhưng khi destructor là phương thức ảo thì khi giải phóng đối tượng B (ở dòng 25) destructor của lớp dẫn xuất được gọi thực hiện rồi đến destructor của lớp cơ sở.

6.5 MỘT SỐ VÍ DỤ

: Chúng ta sử dụng các phương thức ảo và tính đa hình để thực hiện bảng lương dựa trên loại người lao động. Chúng ta sử dụng lớp cơ sở *Employ*. Các lớp dẫn xuất của *Employ* là *Boss* mà người được trả tiền lương hàng tuần cố định bất chấp số giờ làm việc, *CommissionWorker* mà người có tiền lương cơ sở cộng thêm phần trăm của hàng bán, *PieceWorker* mà người được trả bởi số các mục sản xuất, và *HourlyWorker* mà người được trả bởi theo giờ và nhận ngoài giờ.

[Ví dụ 6.6:](#)

■ File EMPLOY.H



```
1: //EMPLOY.H
2: //Lớp cơ sở trừu tượng Employee
3: #ifndef EMPLOY_H
4: #define EMPLOY_H
5: class Employee
6: {
7: private:
8: char *FirstName;
9: char *LastName;
10: public:
11: Employee(const char *First, const char *Last);
12: ~Employee();
13: const char *GetFirstName() const;
14: const char *GetLastName() const;
15: virtual float Earnings() const = 0;
16: virtual void Print() const = 0;
17: };
18:
19: #endif
```

File EMPLOY.CPP

```
1: //EMPLOY.CPP

2: //Định nghĩa hàm thành viên của lớp cơ sở trừu tượng

3: #include <iostream.h>

4: #include <string.h>

5: #include <assert.h>

6: #include "employ.h"

7:

8: Employee::Employee(const char *First, const char *Last)

9: {

10:  FirstName = new char[ strlen(First) + 1 ];

11:  assert(FirstName != 0);

12:  strcpy(FirstName, First);

13:  LastName = new char[ strlen(Last) + 1 ];

14:  assert(LastName != 0);

15:  strcpy(LastName, Last);

16: }

17:

18: Employee::~~Employee()

19: {

20:  delete [] FirstName;

21:  delete [] LastName;

22: }

23:

24: const char *Employee::GetFirstName() const

25: {

26:  return FirstName; //Nơi gọi phải xóa vùng nhớ

27: }

28:

29: const char *Employee::GetLastName() const
```

File BOSS.H

```
1: //BOSS.H
2: //Lớp Boss dẫn xuất từ Employee
3: #ifndef BOSS_H
4: #define BOSS_H
5: #include "employ.h"
6:
7: class Boss : public Employee
8: {
9: private:
10: float WeeklySalary;
11: public:
12: Boss(const char * First, const char *Last, float S = 0.0);
13: void SetWeeklySalary(float S);
14: virtual float Earnings() const;
15: virtual void Print() const;
16: };
17:
18: #endif
```

File BOSS.CPP

```
1: //BOSS.CPP

2: //Định nghĩa hàm thành viên của lớp Boss

3: #include <iostream.h>

4: #include "boss.h"

5:

6: Boss::Boss(const char *First, const char *Last, float S) :
Employee(First, Last)

7: {

8: WeeklySalary = S > 0 ? S : 0;

9: }

10:

11: void Boss::SetWeeklySalary(float S)

12: {

13: WeeklySalary = S > 0 ? S : 0;

14: }

15:

16: float Boss::Earnings() const

17: {

18: return WeeklySalary;

19: }

20:

21: void Boss::Print() const

22: {

23: cout << endl << " Boss: " << GetFirstName()

24:          << ' ' << GetLastName();

25: }
```

COMMIS.H

```
1: //COMMIS.H

2: //Lớp CommissionWorker dẫn xuất từ Employee

3: #ifndef COMMIS_H

4: #define COMMIS_H

5: #include "employ.h"

6:

7: class CommissionWorker : public Employee

8: {

9: private:

10: float Salary; ; //Lương cơ bản mỗi tuần

11: float Commission; //Khối lượng các mục được bán

12: unsigned Quantity; //Tổng các mục bán mỗi tuần

13: public:

14: CommissionWorker(const char *First, const char *Last,

15: float S = 0.0, float C = 0.0, unsigned Q = 0);

16: void SetSalary(float S);

17: void SetCommission(float C);

18: void SetQuantity(unsigned Q);

19: virtual float Earnings() const;

20: virtual void Print() const;

21: };

22:

23: #endif
```

File COMMIS.CPP


```

1: //COMMIS.CPP

2: //Định nghĩa hàm thành viên của lớp CommissionWorker

3: #include <iostream.h>

4: #include "commis.h"

5:

6: CommissionWorker::CommissionWorker(const char *First,const
char *Last,

7: float S, float C, unsigned Q)

8: : Employee(First, Last)

9: {

10: Salary = S > 0 ? S : 0;

11: Commission = C > 0 ? C : 0;

12: Quantity = Q > 0 ? Q : 0;

13: }

14:

15: void CommissionWorker::SetSalary(float S)

16: {

17: Salary = S > 0 ? S : 0;

18: }

19:

20: void CommissionWorker::SetCommission(float C)

21: {

22: Commission = C > 0 ? C : 0;

23: }

24:

25: void CommissionWorker::SetQuantity(unsigned Q)

26: {

27: Quantity = Q > 0 ? Q : 0;

28: }

```

File HOURLY.H

```
1: //HOURLY.H
2: //Định nghĩa lớp HourlyWorker
3: #ifndef HOURLY_H
4: #define HOURLY_H
5: #include "employ.h"
6:
7: class HourlyWorker : public Employee
8: {
9: private:
10: float Wage; //Tiền lương mỗi giờ
11: float Hours; //Số giờ làm việc mỗi tuần
12: public:
13: HourlyWorker(const char *First, const char * Last,
14:              #9; #9; #9; float = 0.0, float = 0.0);
15: void SetWage(float W);
16: void SetHours(float H);
17: virtual float Earnings() const;
18: virtual void Print() const;
19: };
20:
21: #endif
```

File HOURLY.CPP

```
1: //HOURLY.CPP

2: //Định nghĩa hàm thành viên của lớp HourlyWorker

3: #include <iostream.h>

4: #include "hourly.h"

5:

6: HourlyWorker::HourlyWorker(const char *First, const char
*Last,

7:         float W, float H)

8: : Employee(First, Last)

9: {

10: Wage = W > 0 ? W : 0;

11: Hours = H >= 0 && H < 168 ? H : 0;

12: }

13:

14: void HourlyWorker::SetWage(float W)

15: {

16: Wage = W > 0 ? W : 0;

17: }

18:

19: void HourlyWorker::SetHours(float H)

20: {

21: Hours = H >= 0 && H < 168 ? H : 0;

22: }

23:

24: float HourlyWorker::Earnings() const

25: {

26: return Wage * Hours;

27: }

28:
```

File PIECE.H

```
1: //PIECE.H
2: //Lớp PieceWorker dẫn xuất từ Employee
3: #ifndef PIECE_H
4: #define PIECE_H
5: #include "employ.h"
6:
7: class PieceWorker : public Employee
8: {
9: private:
10: float WagePerPiece; //Lương cho mỗi mảnh
11: unsigned Quantity; //Đầu ra mỗi tuần
12: public:
13: PieceWorker(const char *First, const char * Last, float
W=0.0,
14:             unsigned Q=0);
15: void SetWage(float W);
16: void SetQuantity(unsigned Q);
17: virtual float Earnings() const;
18: virtual void Print() const;
19: };
20:
21: #endif
```

File PIECE.CPP

```
1: //PIECE.CPP

2: //Định nghĩa hàm thành viên của lớp PieceWorker

3: #include <iostream.h>

4: #include "piece.h"

5:

7: PieceWorker::PieceWorker(const char *First, const char *Last,

8:                          float W, unsigned Q)

9: : Employee(First, Last)

10: {

11: WagePerPiece = W > 0 ? W : 0;

12: Quantity = Q > 0 ? Q : 0;

13: }

14:

15: void PieceWorker::SetWage(float W)

16: {

17: WagePerPiece = W > 0 ? W : 0;

18: }

19:

20: void PieceWorker::SetQuantity(unsigned Q)

21: {

22: Quantity = Q > 0 ? Q : 0;

23: }

24:

25: float PieceWorker::Earnings() const

26: {

27: return Quantity * WagePerPiece;

28: }

29:

30: void PieceWorker::Print() const
```

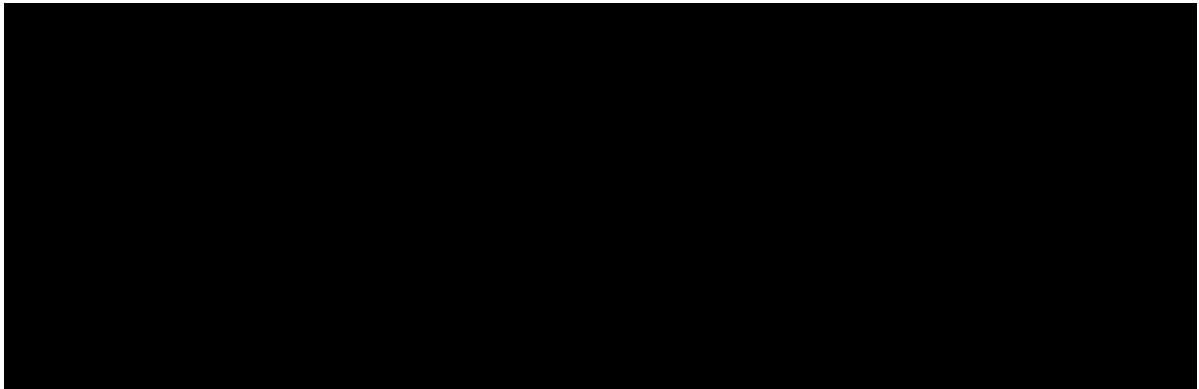
[REDACTED]

■ File CT6_6.CPP

[REDACTED]

```
1: //CT6_6.CPP
2: //Chương trình 6.6
3: #include <iostream.h>
4: #include <iomanip.h>
5: #include "employ.h"
6: #include "boss.h"
7: #include "commis.h"
8: #include "piece.h"
9: #include "hourly.h"
10:
11: int main()
12: {
13: //Ấn định dạng xuất
14: cout << setiosflags(ios::showpoint) << setprecision(2);
15: Employee *Ptr;
16: Boss B("John", "Smith", 800.00);
17: Ptr = &B;
18: Ptr->Print();
19: cout << " earned $" << Ptr->Earnings();
20: B.Print();
21: cout << " earned $" << B.Earnings();
22: CommissionWorker C("Sue", "Jones", 200.0, 3.0, 150);
23: Ptr = &C;
24: Ptr->Print();
25: cout << " earned $" << Ptr->Earnings();
26: C.Print();
27: cout << " earned $" << C.Earnings();
28: PieceWorker P("Bob", "Lewis", 2.5, 200);
29: Ptr = &P;
```

[REDACTED]





SHAPE.H

```
1: //SHAPE.H
2: //Định nghĩa lớp cơ sở trừu tượng Shape
3: #ifndef SHAPE_H
4: #define SHAPE_H
5:
6: class Shape
7: {
8: public:
9: virtual float Area() const
10: {
11: return 0.0;
12: }
13: virtual float Volume() const
14: {
15: return 0.0;
16: }
17: virtual void PrintShapeName() const = 0;
18: virtual void Print() const = 0;
19: };
20:
21: #endif
```

File POINT.H

```
1: //POINT.H
2: //Định nghĩa lớp Point
3: #ifndef POINT_H
4: #define POINT_H
5: #include <iostream.h>
6: #include "shape.h"
7:
8: class Point : public Shape
9: {
10: private:
11: float X, Y;
12: public:
13: Point(float A = 0, float B = 0);
14: void SetPoint(float A, float B);
15: float GetX() const
16: {
17: return X;
18: }
19: float GetY() const
20: {
21: return Y;
22: }
23: virtual void PrintShapeName() const
24: {
25: cout << "Point: ";
26: }
27: virtual void Print() const;
28: friend ostream &operator <<(ostream &Ouput, const Point &P);
29: };
```

[REDACTED]

■ **File POINT.CPP**

[REDACTED]

```
1: //POINT.CPP

2: //Định nghĩa hàm thành viên cho lớp Point

3: #include <iostream.h>

4: #include "point.h"

5:

6: Point::Point(float A, float B)

7: {

8:     SetPoint(A, B);

9: }

10:

11: void Point::SetPoint(float A, float B)

12: {

13:     X = A;

14:     Y = B;

15: }

16:

17: void Point::Print() const

18: {

19:     cout << '[' << X << ", " << Y << ']'<

20: }

21:

22: ostream &operator<<(ostream &Output, const Point &P)

23: {

24:     P.Print();

25:     return Output;

26: }
```

File CIRCLE.H

A.CPP

```
1: //CIRCLE.H
2: //Định nghĩa lớp Circle
3: #ifndef CIRCLE_H
4: #define CIRCLE_H
5: #include "point.h"
6:
7: class Circle : public Point
8: {
9: private:
10: float Radius;
11: public:
12: Circle(float R = 0.0, float A = 0.0, float B = 0.0);
13: void SetRadius(float R);
14: float GetRadius() const;
15: virtual float Area() const;
16: virtual void PrintShapeName() const
17: {
18: cout << "Circle: ";
19: }
20: virtual void Print() const;
21: friend ostream &operator<<(ostream &Output, const Circle &C);
22: };
23:
24: #endif
```

```
1: //CIRCLE.CPP

2: //Định nghĩa hàm thành viên cho lớp Circle

3: #include <iostream.h>

4: #include <iomanip.h>

5: #include "circle.h"

6:

7: Circle::Circle(float R, float A, float B): Point(A, B)

8: {

9: Radius = R > 0 ? R : 0;

10: }

11:

12: void Circle::SetRadius(float R)

13: {

14: Radius = R > 0 ? R : 0;

15: }

16:

17: float Circle::GetRadius() const

18: {

19: return Radius;

20: }

21:

22: float Circle::Area() const

23: {

24: return 3.14159 * Radius * Radius;

25: }

26:

27: void Circle::Print() const

28: {

29: cout << '[' << GetX() << ", " << GetY()
```

[REDACTED]

■ **File CYLINDR.H**

[REDACTED]


```
1: //CYLINDR.H
2: //Định nghĩa lớp Cylinder
3: #ifndef CYLINDR_H
4: #define CYLINDR_H
5: #include "circle.h"
6:
7: class Cylinder : public Circle
8: {
9: private:
10: float Height;
11: public:
12: Cylinder(float H = 0.0, float R = 0.0, float A = 0.0, float B
= 0.0);
13: void setHeight(float);
14: virtual float Area() const;
15: virtual float Volume() const;
16: virtual void PrintShapeName() const
17: {
18: cout << "Cylinder: ";
19: }
20: virtual void Print() const;
21: friend ostream &operator<<(ostream &Output, const Cylinder
&C);
22: };
23:
24: #endif
```



```
1: //CYLINDR.CPP

2: //Định nghĩa hàm thành viên cho lớp Cylinder

3: #include <iostream.h>

4: #include <iomanip.h>

5: #include "cylindr.h"

6:

7: Cylinder::Cylinder(float H, float R, float A, float
B):Circle(R, A, B)

8: {

9: Height = H > 0 ? H : 0;

10: }

11:

12: void Cylinder::setHeight(float H)

13: {

14: Height = H > 0 ? H : 0;

15: }

16:

17: float Cylinder::Area() const

18: {

19: return 2 * Circle::Area() + 2 * 3.14159 * Circle::GetRadius()
* Height;

20: }

21:

22: float Cylinder::Volume() const

23: {

24: return Circle::Area() * Height;

25: }

26:

27: void Cylinder::Print() const

28: {
```

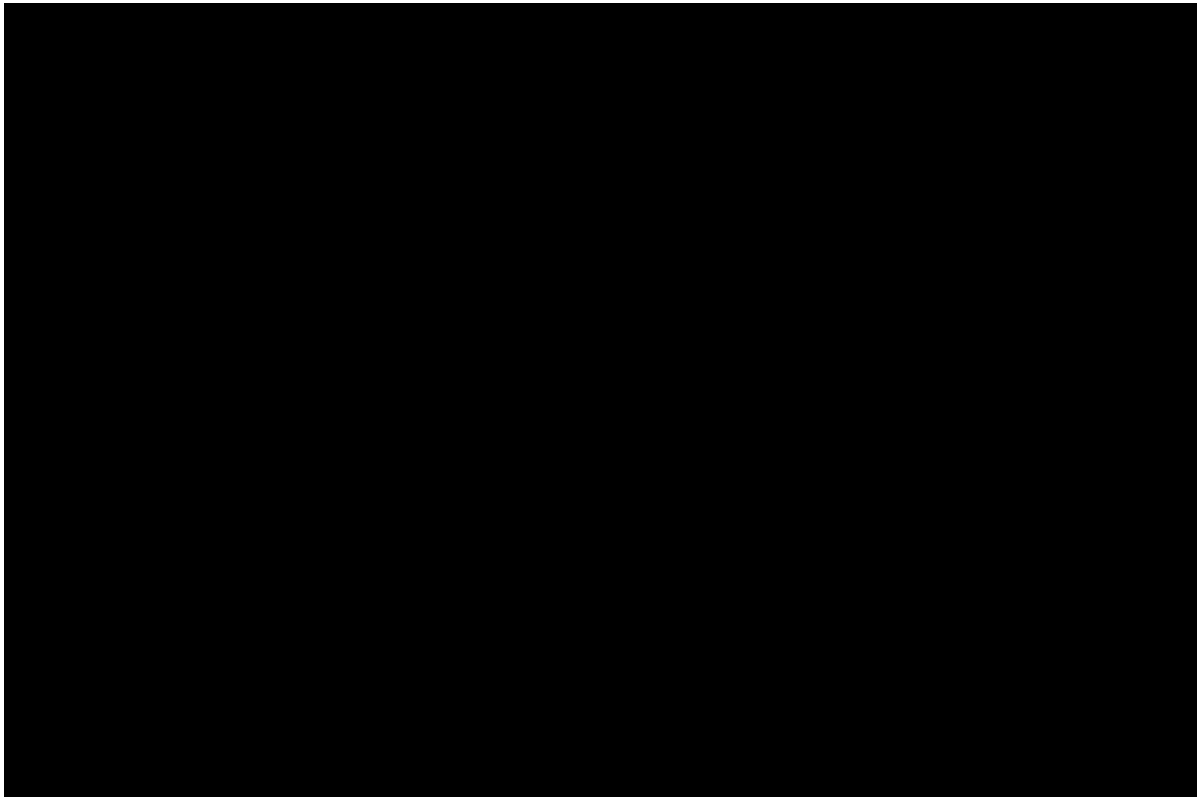
[REDACTED]

■ **File CT6_7.CPP**

[REDACTED]

```
1: //CT6_7.CPP
2: //Chương trình 6.7
3: #include <iostream.h>
4: #include <iomanip.h>
5: #include "shape.h"
6: #include "point.h"
7: #include "circle.h"
8: #include "cylindr.h"
9:
10: int main()
11: {
12:     Point P(7, 11);
13:     Circle C(3.5, 22, 8);
14:     Cylinder Cy(10, 3.3, 10, 10);
15:     P.PrintShapeName();
16:     cout << P << endl;
17:     C.PrintShapeName();
18:     cout << C << endl;
19:     Cy.PrintShapeName();
20:     cout << Cy << "\n\n";
21:     cout << setiosflags(ios::showpoint) << setprecision(2);
22:     Shape *ArrayOfShapes[3];
23:     ArrayOfShapes[0] = &P;
24:     ArrayOfShapes[1] = &C;
25:     ArrayOfShapes[2] = &Cy;
26:     for (int i = 0; i < 3; i++)
27:     {
28:         ArrayOfShapes[i]->PrintShapeName();
29:         cout << endl;
```

Chúng ta [chạy ví dụ 6.7](#), kết quả ở hình 6.8



Hình 6.8: Kết quả của ví dụ 6.7

7.1 DẪN NHẬP

Trong chương này, chúng ta tìm hiểu một ít về cách thiết kế chương trình theo hướng đối tượng, các bước cơ bản cần thiết khi bắt tay vào viết chương trình trên quan điểm thiết kế và thảo chương.

7.2 CÁC GIAI ĐOẠN PHÁT TRIỂN HỆ THỐNG

Có năm giai đoạn để phát triển hệ thống phần mềm theo hướng đối tượng:

- Phân tích yêu cầu (Requirement analysis)
- Phân tích (Analysis)
- Thiết kế (Design)
- Lập trình (Programming)
- Kiểm tra (Testing)

□ [Phân tích yêu cầu](#)

Bằng việc tìm hiểu các trường hợp sử dụng (use case) để nắm bắt các yêu cầu của khách hàng, của vấn đề cần giải quyết. Qua trường hợp sử dụng này, các nhân tố bên ngoài có tham gia vào hệ thống cũng được mô hình hóa bằng các tác nhân. Mỗi trường hợp sử dụng được mô tả bằng văn bản, đặc tả yêu cầu của khách hàng.

□ Phân tích

Từ các đặc tả yêu cầu trên, hệ thống sẽ bước đầu được mô hình hóa bởi các khái niệm lớp, đối tượng và các cơ chế để diễn tả hoạt động của hệ thống.

Trong giai đoạn phân tích chúng ta chỉ mô tả các lớp trong lĩnh vực của vấn đề cần giải quyết chứ chúng ta không đi sâu vào các chi tiết kỹ thuật.

□ Thiết kế

Trong giai đoạn thiết kế, các kết quả của quá trình phân tích được mở rộng thành một giải pháp kỹ thuật. Một số các lớp được thêm vào để cung cấp cơ sở hạ tầng kỹ thuật như lớp giao diện, lớp cơ sở dữ liệu, lớp chức năng, ...

□ Lập trình

Đây còn gọi là bước xây dựng, giai đoạn này sẽ đặc tả chi tiết kết quả của giai đoạn thiết kế. Các lớp của bước thiết kế sẽ được chuyển thành mã nguồn theo một ngôn ngữ lập trình theo hướng đối tượng nào đó.

□ Kiểm tra

Trong giai đoạn kiểm tra, có bốn hình thức kiểm tra hệ thống:

Kiểm tra từng đơn thể (unit testing) được dùng kiểm tra các lớp hoặc các nhóm đơn.

Kiểm tra tính tích hợp (integration testing), được kết hợp với các thành phần và các lớp để kiểm tra xem chúng hoạt động với nhau có đúng không.

Kiểm tra hệ thống (system testing) chỉ để kiểm tra xem hệ thống có đáp ứng được chức năng mà người dùng yêu cầu không.

Kiểm tra tính chấp nhận được (acceptance testing), việc kiểm tra này được thực hiện bởi khách hàng, việc kiểm tra cũng thực hiện giống như kiểm tra hệ thống.

7.3 CÁCH TÌM LỚP

Lớp nên được tìm từ phạm vi bài toán cần giải quyết, vì vậy tên của lớp cũng nên đặt tên các đối tượng thực mà chúng ta biểu diễn. Để tìm ra lớp cho bài toán, chúng ta cần trả lời các câu hỏi sau:

Có thông tin nào cần lưu trữ hay phân tích không? Nếu có bất kỳ thông tin nào cần phải lưu trữ, biến đổi, phân tích hoặc xử lý thì đó chính là một lớp dự định cần xây dựng.

Có hệ thống bên ngoài hay không? Hệ thống ngoài có thể được xem như các lớp mà hệ thống của chúng ta chứa hoặc tương tác với nó.

Có các mẫu thiết kế, thư viện lớp, thành phần, ... hay không? Các thành phần này đã được xây dựng từ các project trước đó, từ các đồng nghiệp hoặc các nhà sản xuất?

Có thiết bị nào mà hệ thống phải đáp ứng? Bất cứ thiết bị nào được nối với hệ thống có thể chuyển thành lớp dự tuyến.

Tác nhân đóng vai trò như thế nào trong hệ thống? Các vai diễn này nên được xem là lớp như người sử dụng, khách hàng, người điều khiển hệ thống,...

7.4 CÁC BƯỚC CẦN THIẾT ĐỂ THIẾT KẾ CHƯƠNG TRÌNH

Để thiết kế một chương trình theo hướng đối tượng, chúng ta phải trải qua bốn bước sau, từ đó chúng ta xây dựng được một cây phả hệ mang tính kế thừa và các mối quan hệ giữa các đối tượng:

Xác định các dạng đối tượng (lớp) của bài toán (định dạng các đối tượng).

Tìm kiếm các đặc tính chung (dữ liệu chung) trong các dạng đối tượng này, những gì chúng cùng nhau chia sẻ.

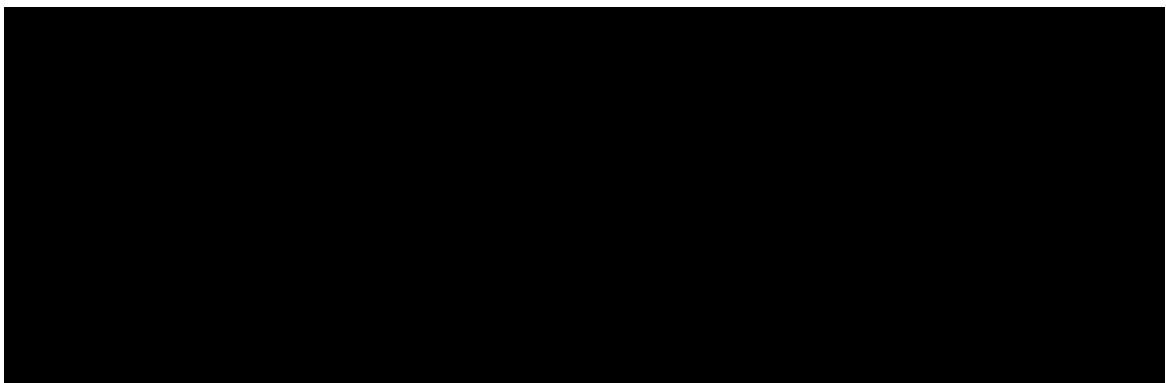
Xác định được lớp cơ sở dựa trên cơ sở các đặc tính chung của các dạng đối tượng.

Từ lớp cơ sở, sử dụng quan hệ tổng quát hóa để đặc tả trong việc đưa ra các lớp dẫn xuất chứa các thành phần, những đặc tính không chung còn lại của dạng đối tượng. Bên cạnh đó, chúng ta còn đưa ra các lớp có quan hệ với các lớp cơ sở và lớp dẫn xuất; các quan hệ này có thể là quan hệ kết hợp, quan hệ tập hợp lại, quan hệ phụ thuộc.

Với các bước trên chúng ta có được cây phả hệ và quan hệ giữa các lớp. Đối với hệ thống phức tạp hơn, chúng ta cần phải phân tích để giải quyết được vấn đề đặt ra theo trật tự sau:

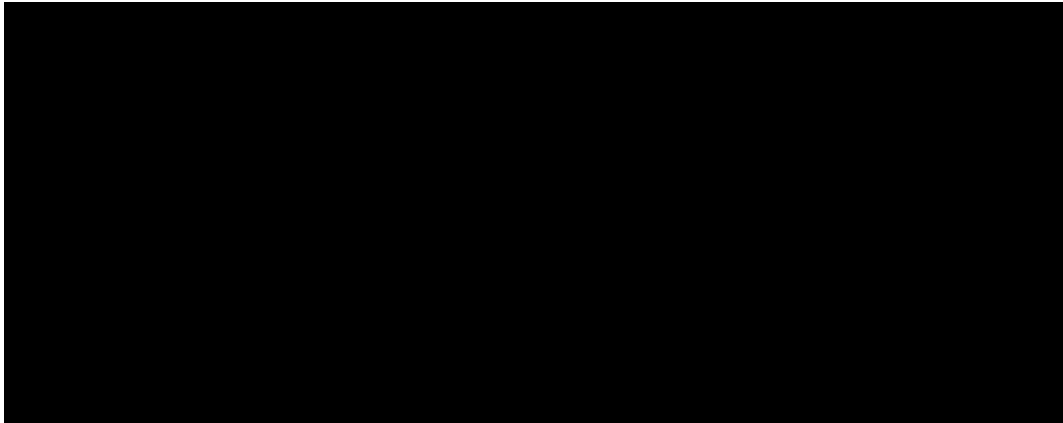
Phân tích một cách cẩn thận về các đối tượng của bài toán theo trật tự từ dưới lên (bottom up).

Tìm ra những gì tồn tại chung giữa các đối tượng, nhóm các đặc tính này lại để được các lớp cơ sở như hình 7.1



Hình 7.1

Tiếp tục theo hướng từ dưới lên, chúng ta thiết kế được các đối tượng phù hợp như hình 7.2



Hình 7.2

Bằng cách này, chúng ta tiếp tục tìm các đặc tính chung cho đến tột cùng của các đối tượng.

Sau đó cài đặt theo hướng đối tượng từ trên xuống bằng cách cài đặt lớp cơ sở chung nhất.

Tiếp tục cài đặt các lớp dẫn xuất trên cơ sở các đặc tính chung của từng nhóm đối tượng.

Cho đến khi tất cả các dạng đối tượng của hệ thống được cài đặt xong để được cây phả hệ.

7.5 CÁC VÍ DỤ

Ví dụ 7.1: Tính tiền lương của các nhân viên trong cơ quan theo các dạng khác nhau. Dạng người lao động lãnh lương từ ngân sách Nhà nước được gọi là cán bộ, công chức (dạng biên chế). Dạng người lao động lãnh lương từ ngân sách của cơ quan được gọi là người làm hợp đồng. Như vậy hệ thống chúng ta có hai đối tượng: biên chế và hợp đồng.

Hai loại đối tượng này có đặc tính chung đó là viên chức làm việc cho cơ quan. Từ đây có thể tạo nên lớp cơ sở để quản lý một viên chức (lớp *Nguyen*) bao gồm mã số, họ tên và lương.

Sau đó chúng ta xây dựng các lớp còn lại kế thừa từ lớp cơ sở trên:

Lớp dành cho cán bộ, công chức (lớp *BienChe*) gồm các thuộc tính: hệ số lương, tiền phụ cấp chức vụ.

Lớp dành cho người làm hợp đồng (lớp *HopDong*) gồm các thuộc tính: tiền công lao động, số ngày làm việc trong tháng, hệ số vượt giờ.



Hình 7.3

■ File PERSON.H



```
1: //PERSON.H
2: Định nghĩa lớp Nguoi
3: #ifndef PERSON_H
4: #define PERSON_H
5:
6: #include <iostream.h>
7:
8: #define MAX_TEN 50
9: #define MAX_MASO 5
10: #define MUC_CO_BAN 120000
11:
12: class Nguoi
13: {
14: protected:
15: char HoTen[MAX_TEN];
16: char MaSo[MAX_MASO];
17: float Luong;
18: public:
19: Nguoi();
20: virtual void TinhLuong()=0;
21: void Xuat() const;
22: virtual void Nhap();
23: };
24:
25: #endif
```



```
1: //PERSON.CPP

2: Định nghĩa hàm thành viên cho lớp Nguoi

3: #include <iomanip.h>

4: #include <string.h>

5: #include "person.h"

6:

7: Nguoi::Nguoi()

8: {

9: strcpy(HoTen, "");

10: strcpy(MaSo, "");

11: Luong=0;

12: }

13:

14: void Nguoi::Xuat() const

15: {

16: cout<<"Ma so:"<<MaSo<<" ,Ho va ten:"<<HoTen

17:      #9;

18: <<" ,Luong:"<<setiosflags(ios::fixed)<<setprecision(0)<<Luong<<endl;

19: }

20: void Nguoi::Nhap()

21: {

22: cout<<"Ma so:";

23: cin>>MaSo;

24: cin.ignore();

25: cout<<"Ho va ten:";

26: cin.getline(HoTen,MAX_TEN);

27: }
```

File STAFF.H

```
1: //STAFF.H
2 Định nghĩa lớp BienChe
3: #ifndef STAFF_H
4: #define STAFF_H
5:
6:
7: #include "person.h"
8:
9: class BienChe: public Nguoi
10: {
11: protected:
12: float HeSoLuong;
13: float HeSoPhuCap;
14: public:
15: BienChe();
16: virtual void TinhLuong();
17: virtual void Nhap();
18: };
19:
20: #endif
```

File STAFF.CPP

```
1: //STAFF.CPP

2: Định nghĩa hàm thành viên cho lớp BienChe

3: #include "staff.h"

4:

5: BienChe::BienChe()

6: {

7:     HeSoLuong=HeSoPhuCap=0;

8: }

9:

10: void BienChe::Nhap()

11: {

12:     Ngnoi::Nhap();

13:     cout<<"He so luong:";

14:     cin>>HeSoLuong;

15:     cout<<"He so phu cap chu vu:";

16:     cin>>HeSoPhuCap;

17: }

18:

19: void BienChe::TinhLuong()

20: {

21:     Luong=MUC_CO_BAN*(1.0+HeSoLuong+HeSoPhuCap);

22: }
```

File CONTRACT.H

```
1: //CONTRACT.H
2: Định nghĩa lớp HopDong
3: #ifndef CONTRACT_H
4: #define CONTRACT_H
5:
6: #include "person.h"
7:
8: class HopDong : public Nguoi
9: {
10: protected:
11: float TienCong;
12: float NgayCong;
13: float HeSoVuotGio;
14: public:
15: HopDong();
16: virtual void TinhLuong();
17: virtual void Nhap();
18: };
19:
20: #endif
```

File **CONTRACT.CPP**:


```
1: //CONTRACT.CPP

2: Định nghĩa hàm thành viên cho lớp HopDong

3: #include "contract.h"

4:

5: HopDong::HopDong()

6: {

7: TienCong=NgayCong=HeSoVuotGio=0;

8: }

9:

10: void HopDong::Nhap()

11: {

12: Nguoi::Nhap();

13: cout<<"Tien cong:";

14: cin>>TienCong;

15: cout<<"Ngay cong:";

16: cin>>NgayCong;

17: cout<<"He so vuot gio:";

18: cin>>HeSoVuotGio;

19: }

20:

21: void HopDong::TinhLuong()

22: {

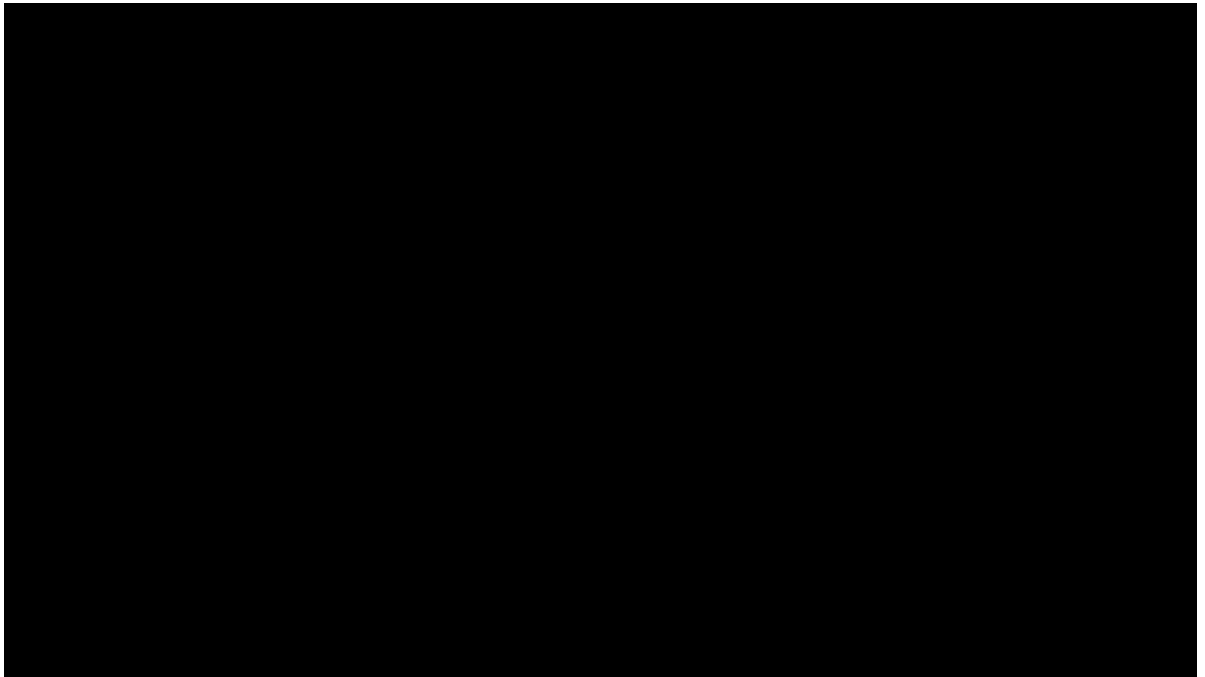
23: Luong=TienCong*NgayCong*(1+HeSoVuotGio);

24: }
```

File CT7_1.CPP:

```
1: //CT7_1.CPP
2: //Chương trình 7.1
3: #include <iostream.h>
4: #include <ctype.h>
5: #include "person.h"
6: #include "staff.h"
7: #include "contract.h"
8:
9: int main()
10: {
11:     Ngoai *Ng[100];
12:     int N=0;
13:     char Chon,Loai;
14:     do
15:     {
16:         cout<<"Bien che hay Hop dong (B/H)? ";
17:         cin>>Loai;
18:         Loai=toupper(Loai);
19:         if (Loai=='B')
20:             Ng[N]=new BienChe;
21:         else
22:             Ng[N]=new HopDong;
23:         Ng[N++]>Nhap();
24:         cout<<"Tiep tuc (C/K)? ";
25:         cin>>Chon;
26:         Chon=toupper(Chon);
27:         if ((N==100) || (Chon=='K'))
28:             break;
29:     }
```

Chúng ta [chạy ví dụ 7.1](#), kết quả ở hình 7.4



Hình 7.4: Kết quả của ví dụ 7.1

[Ví dụ 7.2](#): Giả sử cuối năm học cần trao giải thưởng cho các sinh viên xuất sắc và các giảng viên có nhiều công trình khoa học được công bố trên tạp chí. Các lớp trong cây phả hệ như hình 7.5: lớp *Ngươi* để quản lý hồ sơ cá nhân, lớp *SinhVien* quản lý về sinh viên và lớp *GiangVien* quản lý giảng viên.

Lớp *Ngươi*:

Dữ liệu họ và tên.

Phương thức kiểm tra khả năng được khen thưởng. Đây là phương thức thuần ảo.

Phương thức xuất. Đây là phương thức thuần ảo.

Lớp *SinhVien*:

Dữ liệu điểm trung bình.

Phương thức kiểm tra khả năng được khen thưởng.

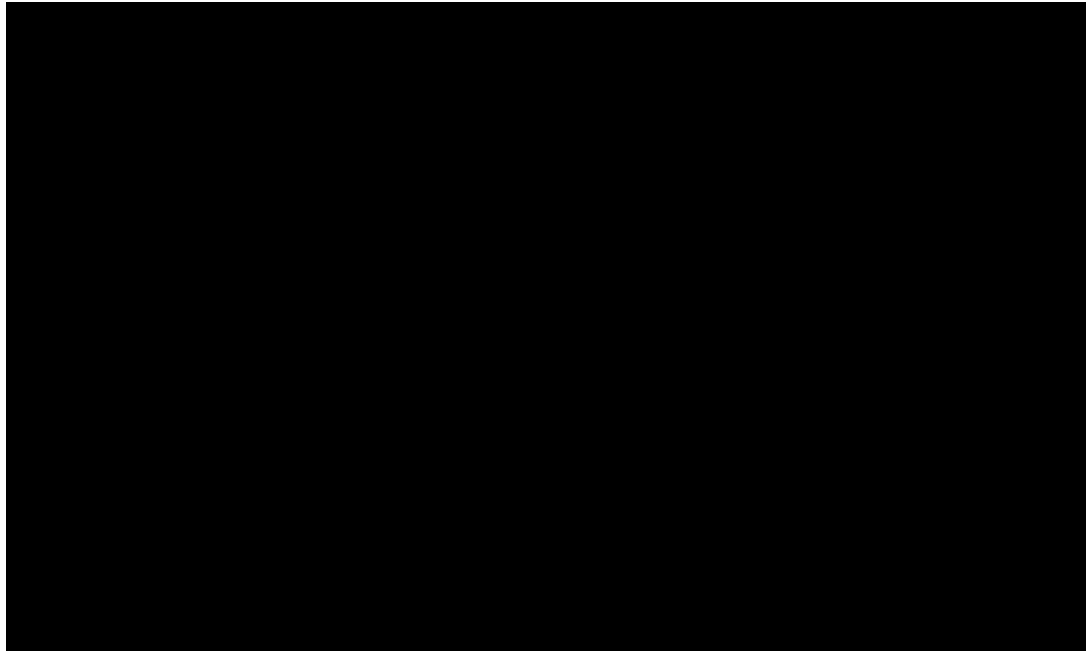
Phương thức xuất.

Lớp *GiangVien*:

Dữ liệu điểm trung bình.

Phương thức kiểm tra khả năng được khen thưởng.

Phương thức xuất.



Hình 7.5

■ File PERSON.H



```
1: //PERSON.H
2: Định nghĩa lớp Nguoi
3: #ifndef PERSON_H
4: #define PERSON_H
5:
6: #include <iostream.h>
7:
8: #define MAX_TEN 50
9:
10: class Nguoi
11: {
12: protected:
13: char HoTen[MAX_TEN];
14: public:
15: Nguoi(char *HT);
16: virtual int DuocKhenThuong() const=0;
17: virtual void Xuat() const=0;
18: };
19:
20: #endif
```

File PERSON.CPP:

PERSON.CPP

```
1: //PERSON.CPP
2: Định nghĩa hàm thành viên cho lớp Nguoi
3: #include <string.h>
4: #include "person.h"
5:
6: Nguoi::Nguoi(char *HT)
7: {
8:     strcpy(HoTen, HT);
9: }
```

File STUDENT.H:

```
1: //STUDENT.H
2: Định nghĩa lớp SinhVien
3: #ifndef STUDENT_H
4: #define STUDENT_H
5:
6: #include "person.h"
7:
8: class SinhVien : public Nguoi
9: {
10: protected:
11: float DiemTB;
12: public:
13: SinhVien(char *HT,float DTB);
14: virtual int DuocKhenThuong() const;
15: virtual void Xuat() const;
16: };
17:
18: #endif
```

File STUDENT.CPP:

```
1: //STUDENT.CPP

2: Định nghĩa hàm thành viên cho lớp SinhVien

3: #include "student.h"

4:

5: SinhVien::SinhVien(char *HT,float DTB):Nguoi(HT)

6: {

7: DiemTB=DTB;

8: }

9:

10: int SinhVien::DuocKhenThuong() const

11: {

12: return DiemTB>9.0;

13: }

14:

15: void SinhVien::Xuat() const

16: {

17: cout<<"Ho va ten cua sinh vien:"<<HoTen;

18: }
```

File TEACHER.H:

TEACHER.H

```
1: //TEACHER.H
2: Định nghĩa lớp GiangVien
3: #ifndef TEACHER_H
4: #define TEACHER_H
5:
6: #include "person.h"
7:
8: class GiangVien : public Nguoi
9: {
10: protected:
11: int SoBaiBao;
12: public:
13: GiangVien(char *HT,int SBB);
14: virtual int DuocKhenThuong() const;
15: virtual void Xuat() const;
16: };
17:
18: #endif
```

File TEACHER.CPP:

```
1: //TEACHER.CPP

2: Định nghĩa hàm thành viên cho lớp GiangVien

3: #include "teacher.h"

4:

5: GiangVien::GiangVien(char *HT,int SBB):Nguoi(HT)

6: {

7:     SoBaiBao=SBB;

8: }

9:

10: int GiangVien::DuocKhenThuong() const

11: {

12:     return SoBaiBao>5;

13: }

14:

15: void GiangVien::Xuat() const

16: {

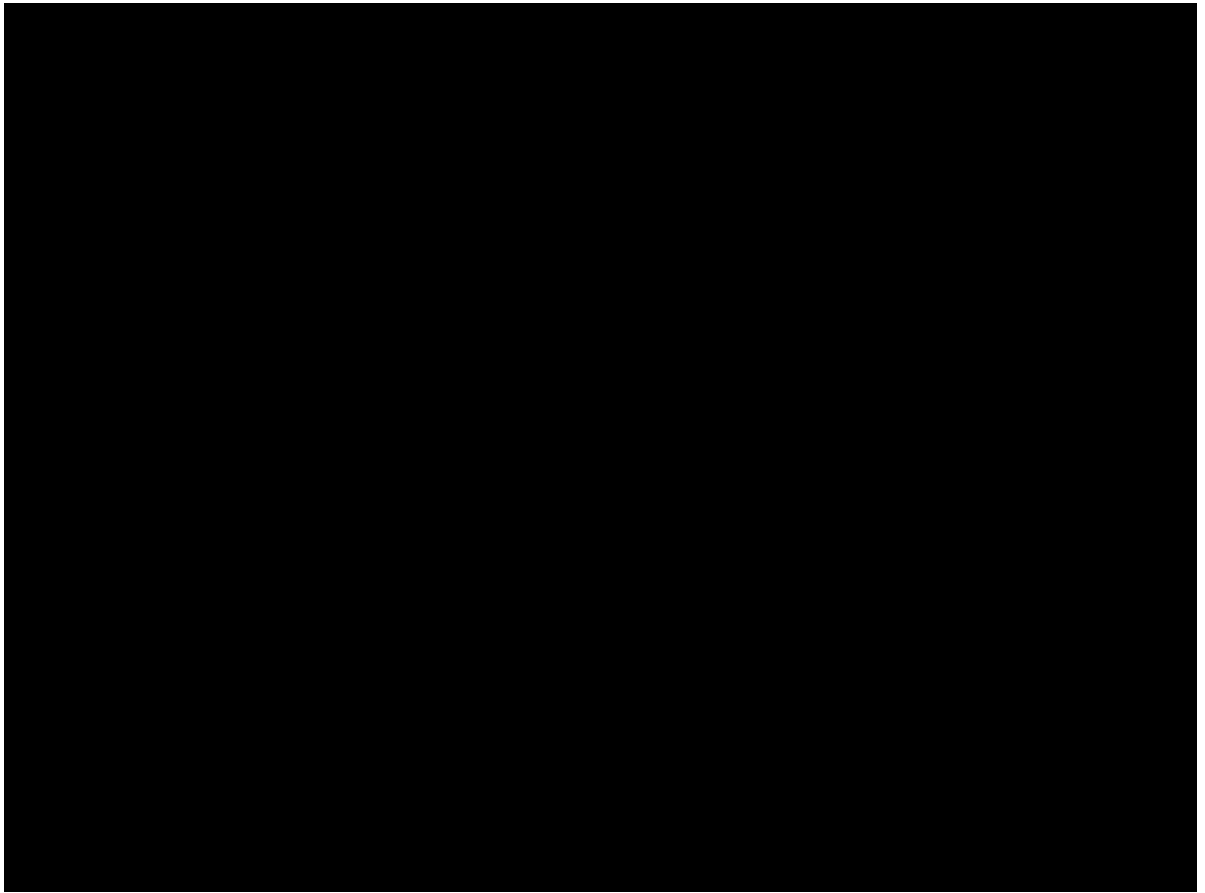
17:     cout<<"Ho va ten cua giang vien:"<<HoTen;

18: }
```

File CT7_2.CPP:

```
1: //CT7_2.CPP
2: //Chương trình 7.2
3: #include <ctype.h>
4: #include "person.h"
5: #include "student.h"
6: #include "teacher.h"
7:
8: int main()
9: {
10:     Ngnoi *Ng[100];
11:     int N=0;
12:     char Chon,Loai;
13:     char HoTen[MAX_TEN];
14:     do
15:     {
16:         cout<<"Ho va ten:";
17:         cin.getline(HoTen,MAX_TEN);
18:         cout<<"Sinh vien hay Giang vien(S/G)? ";
19:         cin>>Loai;
20:         Loai=toupper(Loai);
21:         if (Loai=='S')
22:         {
23:             float DTB;
24:             cout<<"Diem trung binh:";
25:             cin>>DTB;
26:             Ng[N++]=new SinhVien(HoTen,DTB);
27:         }
28:         else
29:         {
```

Chúng ta [chạy ví dụ 7.2](#), kết quả ở hình 7.6



Hình 7.6: Kết quả của ví dụ 7.2

[Ví dụ 7.3](#): Giả sử cần phải tạo các hình: hình tròn và hình chữ nhật được tô theo hai màu red và blue. Xây dựng một cây phả hệ để quản lý các hình này.

Trước hết chúng ta cần có lớp cơ sở *Shape* để lưu trữ thông tin chung cho các hình, sau đó là hai lớp dẫn xuất *Rectangle* về hình chữ nhật và *Circle* về hình tròn như hình 7.7

Lớp *Shape*:

Tọa độ tâm.

Màu đường biên.

Màu tô.

Phương thức thiết lập tô màu.

Phương thức vẽ hình. Đây là phương thức thuần ảo.

Lớp *Rectangle*:

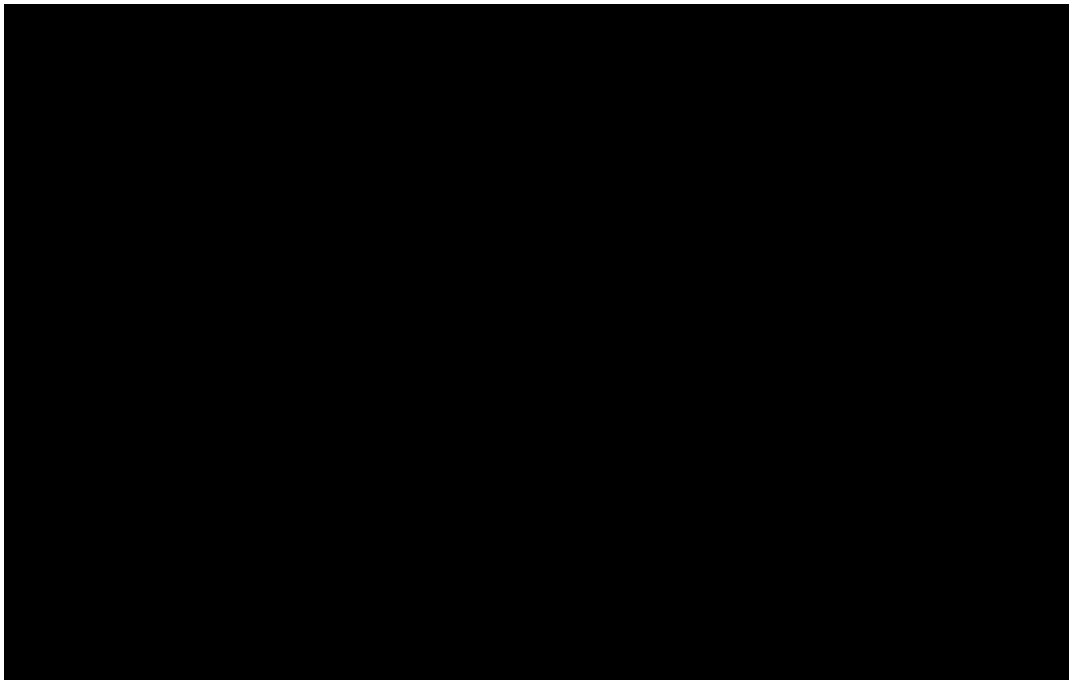
Chiều dài và chiều rộng.

Phương thức vẽ hình.

Lớp *Circle*:

Bán kính.

Phương thức vẽ hình.



Hình 7.7

■ File SHAPE.H:



```
1: //SHAPE.H
2: Định nghĩa lớp Shape
3: #ifndef SHAPE_H
4: #define SHAPE_H
5:
6: #include <iostream.h>
7: #include <graphics.h>
8:
9: class Shape
10: {
11: protected:
12: int XC,YC; //Tọa độ tâm
13: int BoundaryColor; //Màu đường biên
14: int FillColor; //Màu tô
15: void Setting() const; //Thiết lập các thông số vẽ và tô màu
16: public:
17: Shape(int X,int Y,int BC,int FC);
18: virtual void Draw() const=0;
19: };
20:
21: #endif
```

File SHAPE.CPP:

```
1: //SHAPE.CPP

2: Định nghĩa hàm thành viên cho lớp Shape

3: #include "shape.h"

4:

5: Shape::Shape(int X,int Y,int BC,int FC)

6: {

7:   XC=X;

8:   YC=Y;

9:   BoundaryColor=BC;

10:  FillColor=FC;

11: }

12:

13: void Shape::Setting() const

14: {

15:   setcolor(BoundaryColor);

16:   setlinestyle(SOLID_LINE,0,THICK_WIDTH);

17:   setfillstyle(SOLID_FILL,FillColor);

18: }
```

File RECT.H:

```
1: //Rect.H
2: Định nghĩa lớp Rectangle
3: #ifndef RECT_H
4: #define RECT_H
5:
6: #include "shape.h"
7:
8: class Rectangle : public Shape
9: {
10: protected:
11: int Width,Height;
12: public:
13: Rectangle(int X,int Y,int BC,int FC,int W,int H);
14: virtual void Draw() const;
15: };
16:
17: #endif
```

File RECT.CPP:

RECT.CPP

```
1: //RECT.CPP

2: Định nghĩa hàm thành viên cho lớp Rectangle

3: #include "rect.h"

4:

5: Rectangle::Rectangle(int X,int Y,int BC,int FC,int W,int H):
Shape(X,Y,BC,FC)

6: {

7: Width=W;

8: Height=H;

9: }

10:

11: void Rectangle::Draw() const

12: {

13: Setting();

14: rectangle(XC-Width/2,YC-Height/2,XC+Width/2,YC+Height/2);

15: floodfill(XC,YC,BoundaryColor);

16: }
```

File CIRCLE.H:

CIRCLE.H

```
1: //CIRCLE.H
2: Định nghĩa lớp Circle
3: #ifndef CIRCLE_H
4: #define CIRCLE_H
5:
6: #include "shape.h"
7:
8: class Circle : public Shape
9: {
10: protected:
11: int Radius;
12: public:
13: Circle(int X,int Y,int BC,int FC,int R);
14: virtual void Draw() const;
15: };
16:
17: #endif
```

File CIRCLE.CPP:

CIRCLE.CPP

```
1: //CIRCLE.CPP

2: Định nghĩa hàm thành viên cho lớp Circle

3: #include "circle.h"

4:

5: Circle::Circle(int X,int Y,int BC,int FC,int
R):Shape(X,Y,BC,FC)

6: {

7: Radius=R;

8: }

9:

10: void Circle::Draw() const

11: {

12: Setting();

13: circle(XC,YC,Radius);

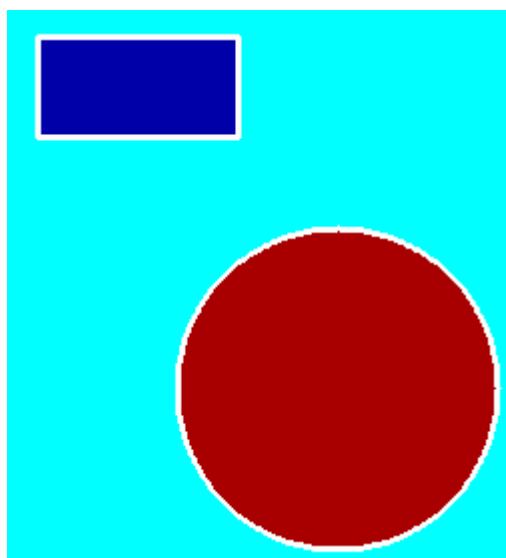
14: floodfill(XC,YC,BoundaryColor);

15: }
```

File CT7_3.CPP:

```
1: //CT7_3.CPP
2: //Chương trình 7.3
3: #include <conio.h>
4: #include <stdlib.h>
5: #include "shape.h"
6: #include "rect.h"
7: #include "circle.h"
8:
9: int main()
10: {
11:     int GDriver = DETECT, GMode, ErrorCode;
12:     ErrorCode = registerbgidriver(EGAVGA_driver);
13:     if (ErrorCode < 0)
14:     {
15:         cout<<"Graphics error:"<<grapherrormsg(ErrorCode)<<endl;
16:         cout<<"Press any key to halt:";
17:         getch();
18:         exit(1);
19:     }
20:     initgraph(&GDriver, &GMode, "");
21:     ErrorCode = graphresult();
22:     if (ErrorCode != grOk)
23:     {
24:         cout<<"Graphics error:"<<grapherrormsg(ErrorCode)<<endl;
25:         cout<<"Press any key to halt:";
26:         getch();
27:         exit(1);
28:     }
29:     Shape* S[2];
```

Chúng ta [chạy ví dụ 7.3](#), kết quả ở hình 7.8



Hình 7.8: Kết quả của ví dụ 7.3

8.1 DẪN NHẬP

Các thư viện chuẩn C++ cung cấp một tập hợp các khả năng nhập/xuất rộng lớn. Trong chương này chúng ta tìm hiểu một phạm vi của các khả năng đủ để phần lớn các thao tác nhập xuất.

Phần lớn các đặc tính nhập xuất mô tả ở đây theo hướng đối tượng. Kiểu này của nhập/xuất thi hành việc sử dụng các đặc tính khác của C++ như các tham chiếu, đa năng hóa hàm và đa năng hóa toán tử.

Như chúng ta sẽ thấy, C++ sử dụng nhập/xuất kiểu an toàn (type safe). Mỗi thao tác nhập/xuất được thực hiện một cách tự động theo lỗi nhảy cảm về kiểu dữ liệu. Mỗi thao tác nhập xuất có được định nghĩa thích hợp để xử lý một kiểu dữ liệu cụ thể thì hàm đó được gọi để xử lý kiểu dữ liệu đó. Nếu không có đối sánh giữa kiểu của dữ liệu hiện tại và một hàm cho việc xử lý kiểu dữ liệu đó, một chỉ dẫn lỗi biên dịch được thiết lập. Vì thế dữ liệu không thích hợp không thể "lách" qua hệ thống.

Các người dùng có thể chỉ định nhập/xuất của các kiểu dữ liệu do người dùng định nghĩa cũng như các kiểu dữ liệu chuẩn. Tính mở rộng này là một trong các đặc tính quan trọng của C++.

8.2 CÁC DÒNG(STREAMS)

Nhập/xuất C++ xảy ra trong các dòng của các byte. Một dòng đơn giản là một dãy tuần tự các byte. Trong các thao tác nhập, các byte chảy từ thiết bị (chẳng hạn: một bàn phím, một ổ đĩa, một kết nối mạng) tới bộ nhớ chính. Trong các thao tác xuất, các byte chảy từ bộ nhớ chính tới một thiết bị (chẳng hạn: một màn hình, một máy in, một ổ đĩa, một kết nối mạng).

Ứng dụng liên kết với các byte. Các byte có thể biểu diễn các ký tự ASCII, bên trong định dạng dữ liệu thô, các ảnh đồ họa, tiếng nói số, hình ảnh số hoặc bất cứ loại thông tin một ứng dụng có thể đòi hỏi.

Công việc của các cơ chế hệ thống nhập/xuất là di chuyển các byte từ các thiết bị tới bộ nhớ và ngược lại theo lối chắc và đáng tin cậy. Như thế các di chuyển thường bao gồm sự di chuyển cơ học như sự quay của một đĩa hoặc một băng từ, hoặc nhấn phím tại một bàn phím. Thời gian các di chuyển này thông thường không lồ so với thời gian bộ xử lý thao tác dữ liệu nội tại. Vì thế, các thao tác nhập/xuất đòi hỏi có kế hoạch cẩn thận và điều chỉnh để bảo đảm sự thi hành tối đa.

C++ cung cấp cả hai khả năng nhập/xuất "mức thấp" (low-level) và "mức cao" (high-level). Các khả năng nhập/xuất mức thấp (nghĩa là nhập/xuất không định dạng) chỉ định cụ thể số byte nào đó phải được di chuyển hoàn toàn từ thiết bị tới bộ nhớ hoặc từ bộ nhớ tới thiết bị. Trong các di chuyển như thế, byte riêng rẽ là mục cần quan tâm. Vì thế các khả năng mức thấp cung cấp tốc độ cao, các di chuyển dung lượng cao, nhưng các khả năng này không phải là tiện lợi lắm cho lập trình viên.

Các lập trình viên ưu thích quan điểm nhập/xuất mức cao, nghĩa là nhập/xuất có định dạng, trong đó các byte được nhóm thành các đơn vị có ý nghĩa như các số nguyên, các số chấm động, các ký tự, các chuỗi và các kiểu do người dùng định nghĩa.

8.2.1 Các file header của thư viện iostream:

■ Thư viện **iostream** của C++ cung cấp hàng trăm khả năng của nhập/xuất. Một vài tập tin header chứa các phần của giao diện thư viện.

■ Phần lớn chương trình C++ thường include tập tin header **<iostream.h>** mà chứa các thông tin cơ bản đòi hỏi tất cả các thao tác dòng nhập/xuất. Tập tin header **<iostream.h>** chứa các đối tượng **cin**, **cout**, **cerr** và **clog** mà tương ứng với dòng nhập chuẩn, dòng xuất chuẩn, dòng lỗi chuẩn không vùng đệm và dòng lỗi chuẩn vùng đệm. Cả hai khả năng nhập/xuất định dạng và không định dạng được cung cấp.

■ Header **<iomanip.h>** chứa thông tin hữu ích cho việc thực hiện nhập/xuất định dạng với tên gọi là các bộ xử lý dòng biểu hiện bằng tham số (parameterized stream manipulators).

■ Header **<fstream.h>** chứa các thông tin quan trọng cho các thao tác xử lý file do người dùng kiểm soát.

■ Header **<strstream.h>** chứa các thông tin quan trọng cho việc thực hiện các định dạng trong bộ nhớ. Điều này tương tự xử lý file, nhưng các thao tác nhập/xuất tới và từ mảng các ký tự hơn là file.

■ Header **<stdiostream.h>** chứa các thông tin quan trọng cho các chương trình trộn các kiểu nhập/xuất của C và C++. Các chương trình mới phải tránh kiểu nhập/xuất C, nhưng cần thì hiệu chỉnh các chương trình C, hoặc tiến triển chương trình C thành C++.

8.2.2 Các lớp và các đối tượng của dòng nhập/xuất:

Thư viện **iostream** chứa nhiều lớp để xử lý một sự đa dạng rộng của các thao tác nhập/xuất. Lớp **istream** hỗ trợ các thao tác dòng nhập. Lớp **ostream** hỗ trợ các thao tác dòng xuất. Lớp **iostream** hỗ trợ cả hai thao tác dòng nhập và dòng xuất. Lớp **istream** và lớp **ostream** đều kế thừa đơn từ lớp cơ sở **ios**. Lớp **iostream** được kế thừa thông qua đa kế thừa từ hai lớp **istream** và **ostream**.



Hình 8.1: Một phần của phân cấp lớp dòng nhập/xuất

Đa năng hóa toán tử cung cấp một ký hiệu thích hợp cho việc thực hiện nhập/xuất. Toán tử dịch chuyển trái (<<) được đa năng hóa để định rõ dòng xuất và được tham chiếu như là toán tử chèn dòng. Toán tử dịch chuyển phải (>>) được đa năng hóa để định rõ dòng nhập và được tham chiếu như là toán tử trích dòng. Các toán tử này được sử dụng với các đối tượng dòng chuẩn **cin**, **cout**, **cerr** và **clog**, và bình thường với các đối tượng dòng do người dùng định nghĩa.

■ **cin** là một đối tượng của lớp **istream** và được nói là "bị ràng buộc tới" (hoặc kết nối tới) thiết bị nhập chuẩn, thông thường là bàn phím. Toán tử trích dòng được sử dụng ở lệnh sau tạo ra một giá trị cho biến nguyên *X* được nhập từ **cin** tới bộ nhớ:

```
int X;  
  
cin >> X;
```

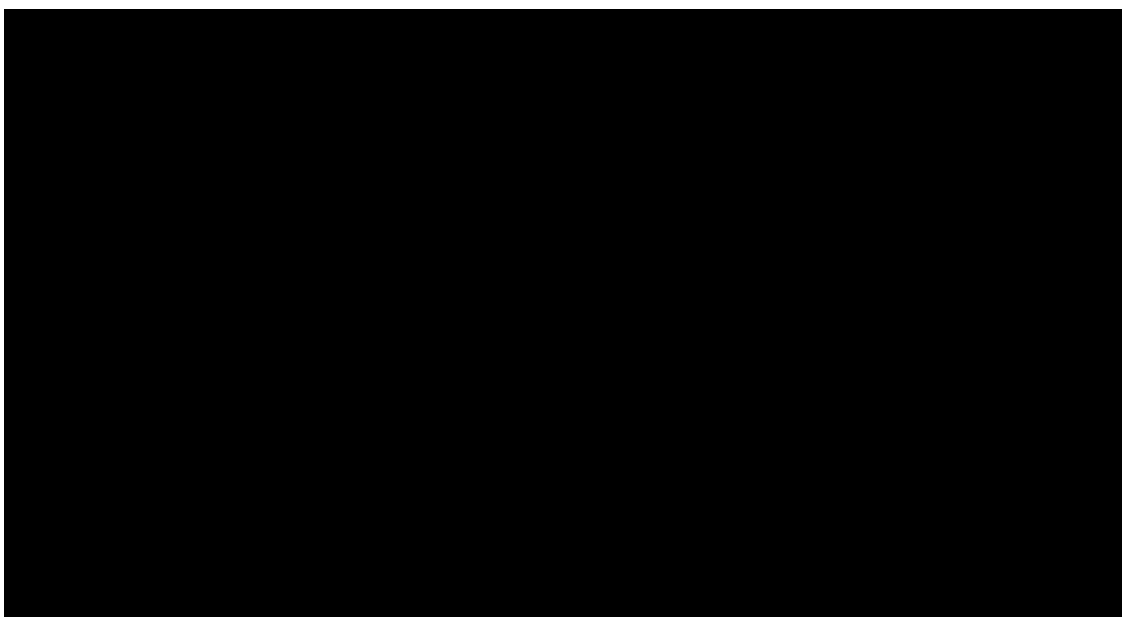
■ **cout** là một đối tượng của lớp **ostream** và được nói là "bị ràng buộc tới" thiết bị xuất chuẩn, thông thường là màn hình. Toán tử chèn dòng được sử dụng ở lệnh sau tạo ra một giá trị cho biến nguyên *X* được xuất từ bộ nhớ tới thiết bị chuẩn:

```
cout << X;
```

■ **cerr** là một đối tượng của lớp **ostream** và được nói là "bị ràng buộc tới" thiết bị lỗi chuẩn. Việc xuất đối tượng **cerr** là không vùng đệm. Điều này có nghĩa là mỗi lần chèn tới **cerr** tạo ra kết xuất của nó xuất hiện ngay tức thì; Điều này thích hợp cho việc thông báo nhanh chóng người dùng khi có sự cố.

■ **clog** là một đối tượng của lớp **ostream** và được nói là "bị ràng buộc tới" thiết bị lỗi chuẩn. Việc xuất đối tượng **clog** là có vùng đệm. Điều này có nghĩa là mỗi lần chèn tới **clog** tạo ra kết xuất của nó được giữ trong vùng đệm cho đến khi vùng đệm đầy hoặc vùng đệm được flush.

Việc xử lý file của C++ sử dụng các lớp **ifstream** để thực hiện các thao tác nhập file, **ofstream** cho các thao tác xuất file, và **fstream** cho các thao tác nhập/xuất file. Lớp **ifstream** kế thừa từ **istream**, **ofstream** lớp kế thừa từ **ostream**, và lớp **fstream** kế thừa từ **iostream**.



Hình 8.2: Một phần của phân cấp lớp dòng nhập/xuất với việc xử lý file.

8.3 DÒNG XUẤT

ostream của C++ cung cấp khả năng để thực hiện xuất định dạng và không định dạng. Các khả năng xuất bao gồm: xuất các kiểu dữ liệu chuẩn với toán tử chèn dòng; xuất các ký tự với hàm thành viên **put()**; xuất không định dạng với hàm thành viên **write**; xuất các số nguyên dạng thập phân, bát phân và thập lục phân; xuất các giá trị chấm động với độ chính xác khác nhau, với dấu chấm thập phân, theo ký hiệu khoa học và theo ký hiệu cố định; xuất dữ liệu theo các trường độn thêm các ký tự chỉ định; và xuất các mẫu tự chữ hoa theo ký hiệu khoa học và ký hiệu thập lục phân.

8.3.1 Toán tử chèn dòng:

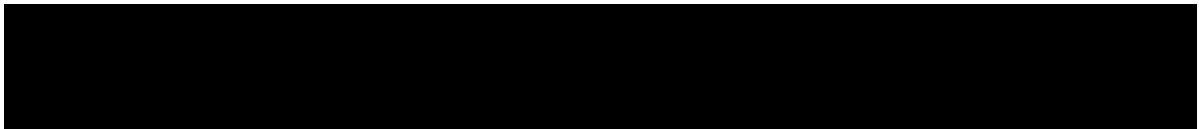
Dòng xuất có thể được thực hiện với toán tử chèn dòng, nghĩa là toán tử << đã đa năng hóa. Toán tử << đã được đa năng hóa để xuất các mục dữ liệu của các kiểu có sẵn, xuất chuỗi, và xuất các giá trị con trỏ.

■ Ví dụ 8.1: Minh họa xuất chuỗi sử dụng một lệnh chèn dòng.

CT8_1.CPP

```
1: //Chương trình 8.1:Xuất một chuỗi sử dụng chèn dòng
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout<<"Welcome to C++!\n";
7:     return 0;
8: }
```

Chúng ta [chạy ví dụ 8.1](#), kết quả ở hình 8.3



Hình 8.3: Kết quả của ví dụ 8.1

Ví dụ 8.2: Minh họa xuất chuỗi sử dụng nhiều lệnh chèn dòng.

CT8_2.CPP

```
1: //Chương trình 8.2:Xuất một chuỗi sử dụng hai chèn dòng
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout<<"Welcome to";
7:     cout<<"C++!\n";
8:     return 0;
9: }
```

Chúng ta [chạy ví dụ 8.2](#), kết quả ở hình 8.4



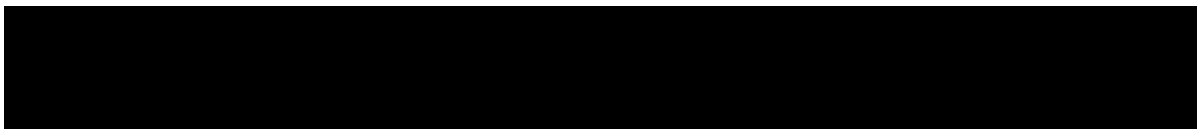
Hình 8.4: Kết quả của ví dụ 8.2

Hiệu quả của chuỗi thoát `\n` (newline) cũng đạt được bởi bộ xử lý dòng (stream manipulator) **`endl`** (end line).

■ Ví dụ 8.3:

```
CT8_3.CPP
1: //Chương trình 8.3:Sử dụng bộ xử lý dòng endl
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout<<"Welcome to";
7:     cout<<"C++!";
8:     cout<<endl;
9:     return 0;
10: }
```

Chúng ta [chạy ví dụ 8.3](#), kết quả ở hình 8.5



Hình 8.5: Kết quả của ví dụ 8.3

Bộ xử lý dòng **`endl`** đưa ra một ký tự newline, và hơn nữa, flush vùng đệm xuất (nghĩa là tạo ra vùng đệm xuất được xuất ngay lập tức kể cả nó chưa đầy). Vùng đệm xuất cũng có thể được flush bằng:

```
cout<<flush;
```

■ Ví dụ 8.4: Các biểu thức có thể xuất

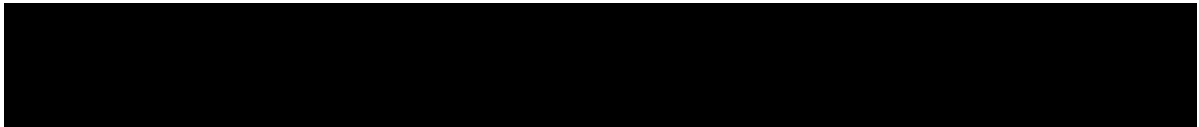


```

1: //Chương trình 8.4: Xuất giá trị biểu thức.
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout<<"47 plus 53 is ";
7:     cout<< (47+53);
8:     cout<<endl;
9:     return 0;
10: }

```

Chúng ta [chạy ví dụ 8.4](#), kết quả ở hình 8.6



Hình 8.6: Kết quả của ví dụ 8.4

8.3.2 Nối các toán tử chèn dòng và trích dòng:

Các toán tử đã đa năng hóa << và >> có thể được theo dạng nối vào nhau.

■ [Ví dụ 8.5:](#) Nối các toán tử đã đa năng hóa

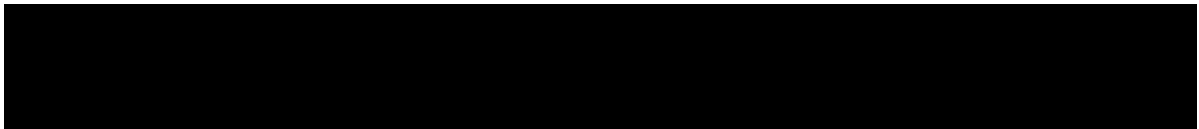


```

1: //Chương trình 8.5: Nối toán tử << đã đa năng hóa.
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout<<"47 plus 53 is "<< (47+53)<<endl;
7:     return 0;
8: }

```

Chúng ta [chạy ví dụ 8.5](#), kết quả ở hình 8.7



Hình 8.7: Kết quả của ví dụ 8.5

Nhiều chèn dòng ở dòng 6 trong ví dụ 8.5 được thực thi nếu có thể viết:

```
((cout<<"47 plus 53 is "<< (47+53))<<endl);
```

nghĩa là << liên kết từ trái qua phải. Loại liên kết của các toán tử chèn dòng được phép bởi vì toán tử đa năng hóa << trả về một tham chiếu tới đối tượng toán hạng bên trái của nó, nghĩa là **cout**. Vì thế biểu thức đặt trong ngoặc bên cực trái:

```
(cout<<"47 plus 53 is ")
```

xuất ra một chuỗi đã chỉ định và trả về một tham chiếu tới **cout**. Điều này cho phép biểu thức đặt trong ngoặc ở giữa được ước lượng:

```
(cout<< (47+53))
```

xuất giá trị nguyên 100 và trả về một tham chiếu tới **cout**. Sau đó biểu thức đặt trong ngoặc bên cực phải được ước lượng:

```
cout<<endl;
```

xuất một newline, flush cout và trả về một tham chiếu tới **cout**. Trả về cuối cùng này không được sử dụng.

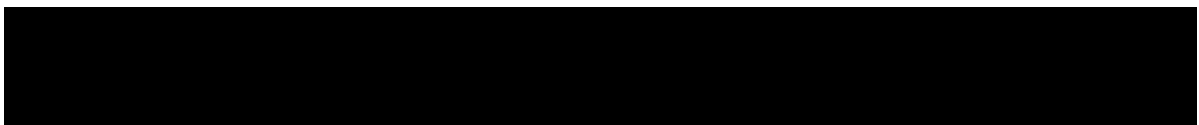
8.3.3 Xuất các biến kiểu char *:

Trong nhập/xuất kiểu C, thật cần thiết cho lập trình viên để cung cấp thông tin kiểu. C++ xác định các kiểu dữ liệu một cách tự động – một cải tiến hay hơn C. Đôi khi điều này là một trở ngại. Chẳng hạn, chúng ta biết rằng một chuỗi ký tự là kiểu **char ***. Mục đích của chúng ta in giá trị của con trỏ đó, nghĩa là địa chỉ bộ nhớ của ký tự đầu tiên của chuỗi đó. Nhưng toán tử << đã được đa năng hóa để in dữ liệu của kiểu **char *** như là chuỗi kết thúc ký tự null. Giải pháp là ép con trỏ thành kiểu **void ***.

Ví dụ 8.6: In địa chỉ lưu trong một biến kiểu **char ***

```
CT8_6.CPP
1: //Chương trình 8.6
2: #include <iostream.h>
3:
4: int main()
5: {
6:     char *Str = "test";
7:     cout << "Value of Str is: " << Str
8:         << "\nValue of (void *)Str is: "
9:         << (void *)Str << endl;
10:    return 0;
11: }
```

Chúng ta [chạy ví dụ 8.6](#), kết quả ở hình 8.8



Hình 8.8: Kết quả của ví dụ 8.6

8.3.4 Xuất ký tự với hàm thành viên put(); Nói với nhau hàm put():

Hàm thành viên **put()** của lớp **ostream** xuất một ký tự có dạng :

ostream& put(char ch);

Chẳng hạn:

cout.put('A');

Gọi **put()** có thể được nối vào nhau như:

```
cout.put('A').put('\n');
```

Hàm **put()** cũng có thể gọi với một biểu thức có giá trị là mã ASCII như:

```
cout.put(65);
```

8.4 DÒNG NHẬP

Dòng nhập có thể được thực hiện với toán tử trích, nghĩa là toán tử đã đa năng hóa **>>**. Bình thường toán tử này bỏ qua các ký tử khoảng trắng (như các blank, tab và newline). trong dòng nhập. Toán tử trích dòng trả về zero (false) khi kết thúc file (end-of-file) được bắt gặp trên một dòng; Ngược lại, toán tử trích dòng trả về một tham chiếu tới đối tượng xuyên qua đó nó được kéo theo. Mỗi dòng chứa một tập các bit trạng thái (state bit) sử dụng để điều khiển trạng thái của dòng (nghĩa là định dạng, ấn định các trạng thái lỗi,...). Trích dòng sinh ra **failbit** của dòng được thiết lập nếu dữ liệu của kiểu sai được nhập, và sinh ra **badbit** của dòng được thiết lập nếu thao tác sai.

8.4.1 Toán tử trích dòng:

Để đọc hai số nguyên sử dụng đối tượng **cin** và toán tử trích dòng đã đa năng hóa **>>**.

Ví dụ 8.7:

```
CT8_7.CPP
1: //Chương trình 8.7
2: #include <iostream.h>
3:
4: int main()
5: {
6:     int X, Y;
7:     cout << "Enter two integers: ";
8:     cin >> X >> Y;
9:     cout << "Sum of " << X << " and " << Y << " is: "
10:         << (X + Y) << endl;
11:     return 0;
12: }
```

Chúng ta [chạy ví dụ 8.7](#), kết quả ở hình 8.9



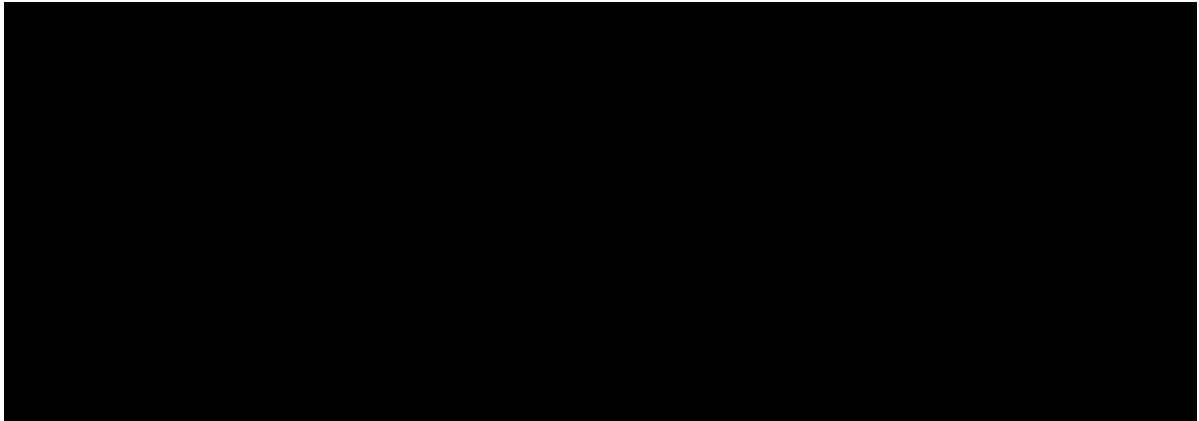
Hình 8.9: Kết quả của ví dụ 8.7

Một cách phổ biến để nhập một dãy các giá trị là sử dụng toán tử trích dòng trong vòng lặp **while**. Toán tử trích dòng trả về false (0) khi end-of-file được bắt gặp:

■ [Ví dụ 8.8](#):

```
CT8_8.CPP
1: //Chương trình 8.8
2: #include <iostream.h>
3:
4: int main()
5: {
6:     int Grade, HighestGrade = -1;
7:     cout << "Enter grade (enter end-of-file to end): ";
8:     while (cin >> Grade)
9:     {
10:         if (Grade > HighestGrade )
11:             HighestGrade = Grade;
12:         cout << "Enter grade (enter end-of-file to end): ";
13:     }
14:     cout << endl << "Highest grade is: " << HighestGrade<< endl;
15:     return 0;
16: }
```

Chúng ta [chạy ví dụ 8.8](#), kết quả ở hình 8.10



Hình 8.10: Kết quả của ví dụ 8.8

8.4.2 Các hàm thành viên `get()` và `getline()`:

Hàm `istream::get()` có các dạng sau:

- (1) `int get();`
- (2) `istream& get(unsigned char &ch);`
- (3) `istream& get(signed char &ch);`
- (4) `istream& get(unsigned char * puch, int len, char delim='\n');`
- (5) `istream& get(signed char * psch, int len, char delim='\n');`

Dạng (1) trích ký tự đơn từ dòng và trả về nó hoặc **EOF** khi end-of-file trên dòng được bắt gặp.

Dạng (2) và (3) trích một ký tự đơn từ dòng và lưu trữ nó vào *ch*.

Dạng (4) và (5) trích các ký tự từ dòng cho đến khi hoặc *delim* được tìm thấy, giới hạn *len* đạt đến, hoặc end-of-file được bắt gặp. Các ký tự được lưu trong con trỏ chỉ đến mảng ký tự *puch* hoặc *psch*.

■ Ví dụ 8.9: Sử dụng hàm `get()` dạng (1)

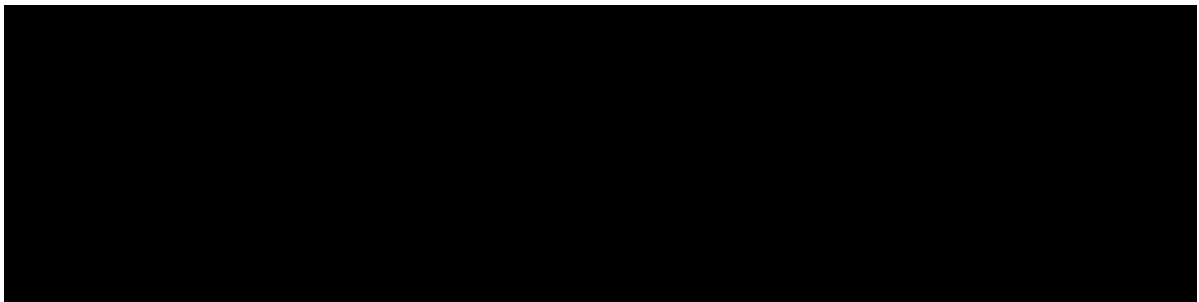



```

1: //Chương trình 8.9
2: #include <iostream.h>
3: int main()
4: {
5:     int Ch;
6:     cout << "Before input, cin.eof() is " << cin.eof() << endl
7:         << "Enter a sentence followed by end-of-file:" <<
endl;
8:     while ( ( Ch = cin.get() ) != EOF)
9:         cout.put(Ch);
10:    cout << endl << "EOF in this system is: " << Ch << endl;
11:    cout << "After input, cin.eof() is " << cin.eof() << endl;
12:    return 0;
13: }

```

Chúng ta [chạy ví dụ 8.9](#), kết quả ở hình 8.11



Hình 8.11: Kết quả của ví dụ 8.9

Trong ví dụ 8.9 trên, chúng ta có sử dụng hàm **ios::eof()** có dạng sau:

```
int eof();
```

Hàm trả về giá trị khác zero nếu end-of-file bắt gặp.

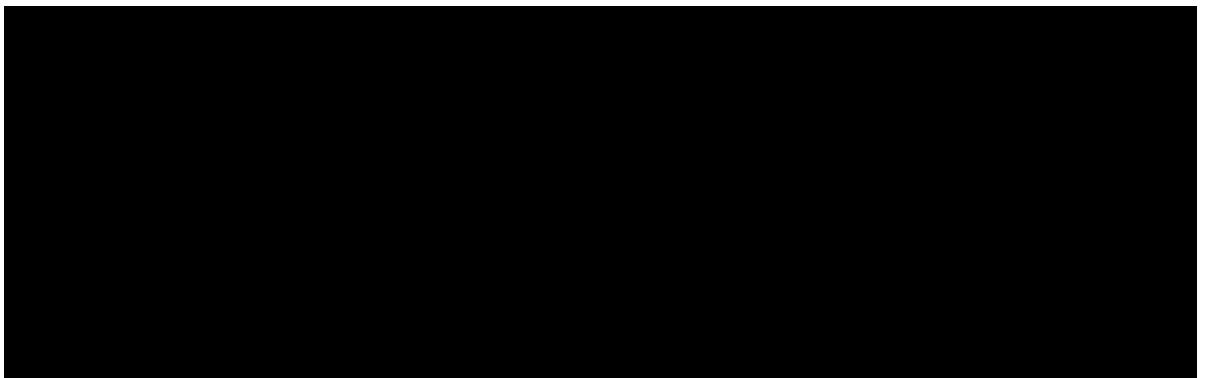
Ví dụ 8.10: Sử dụng hàm **get()** dạng (5)

```

1: //Chương trình 8.10
2: #include <iostream.h>
3:
4: const int SIZE = 80;
5:
6: int main()
7: {
8:     char Buffer1[SIZE], Buffer2[SIZE];
9:     cout << "Enter a sentence:" << endl;
10:    cin >> Buffer1;
11:    cout << endl << "The string readwith cin was:" << endl
12:         << Buffer1 << endl << endl;
13:    cin.get(Buffer2, SIZE);
14:    cout << "The string read with cin.get was:" << endl
15:         << Buffer2 << endl;
16:    return 0;
17: }

```

Chúng ta [chạy ví dụ 8.10](#), kết quả ở hình 8.12



Hình 8.12: Kết quả của ví dụ 8.10

Hàm `istream::getline()` có các dạng sau:

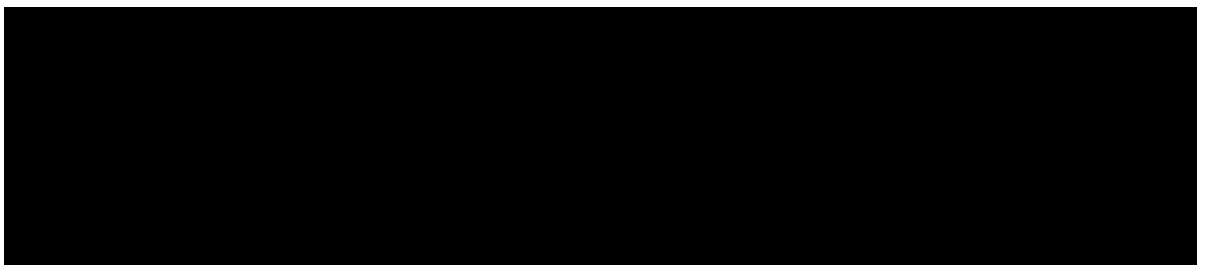
(1) `istream& getline(unsigned char * puch, int len, char delim='\n');`

(2) `istream& getline(signed char * psch, int len, char delim='\n');`

Ví dụ 8.11: Sử dụng hàm `getline()`

```
CT8_11.CPP
1: //Chương trình 8.11
2: #include <iostream.h>
3:
4: const SIZE = 80;
5:
6: int main()
7: {
8:     char Buffer[SIZE];
9:     cout << "Enter a sentence:" << endl;
10:    cin.getline(Buffer, SIZE);
11:    cout << endl << "The sentence entered is:" << endl
12:         << Buffer << endl;
13:    return 0;
14: }
```

Chúng ta [chạy ví dụ 8.11](#), kết quả ở hình 8.13



Hình 8.13: Kết quả của ví dụ 8.11

8.4.3 Các hàm thành viên khác của `istream`:

Hàm `ignore()`:

istream& ignore(int nCount = 1, int delim = EOF);

Trích và loại bỏ lên đến *nCount* ký tự. Việc trích dừng nếu *delim* được bắt gặp hoặc nếu end-of-file bắt gặp.

Hàm **putback()**:

istream& putback(char ch);

Đặt một ký tự ngược lại dòng nhập.

Hàm **peek()**:

int peek();

Hàm trả về ký tự kế tiếp mà không trích nó từ dòng.

8.4.4 Nhập/xuất kiểu an toàn:

C++ cung cấp nhập/xuất kiểu an toàn (type-safe). Các toán tử << và >> được đa năng hóa để nhận các mục dữ liệu của kiểu cụ thể. Nếu dữ liệu bất ngờ được xử lý, các cờ hiệu lỗi khác nhau được thiết lập mà người dùng có thể kiểm tra để xác định nếu một thao tác nhập/xuất thành công hoặc thất bại. Phần sau chúng ta sẽ khảo sát kỹ hơn.

8.5 NHẬP/ XUẤT KHÔNG ĐỊNH DẠNG VỚI READ(),GCOUNT() VÀ WRITE()

Nhập/xuất không định dạng được thực hiện với các hàm thành viên **istream::read()** và **ostream::write()**.

Hàm **istream::read()**:

istream& read(unsigned char* puch, int nCount);

istream& read(signed char* psch, int nCount);

Trích các byte từ dòng cho đến khi giới hạn *nCount* đạt đến hoặc cho đến khi end-of-file đạt đến. Hàm này có ích cho dòng nhập nhị phân.

Hàm **ostream::write()**:

ostream& write(const unsigned char* puch, int nCount);

ostream& write(const signed char* psch, int nCount);

Chèn *nCount* byte vào từ vùng đệm (được trỏ bởi *puch* và *psch*) vào dòng. Nếu file được mở ở chế độ text, các ký tự CR có thể được chèn vào. Hàm này có ích cho dòng xuất nhị phân. Chẳng hạn:

```
char Buff[]="HAPPY BIRTHDAY";
```

```
cout.write(Buff,10);
```

Hàm `istream::gcount()`:

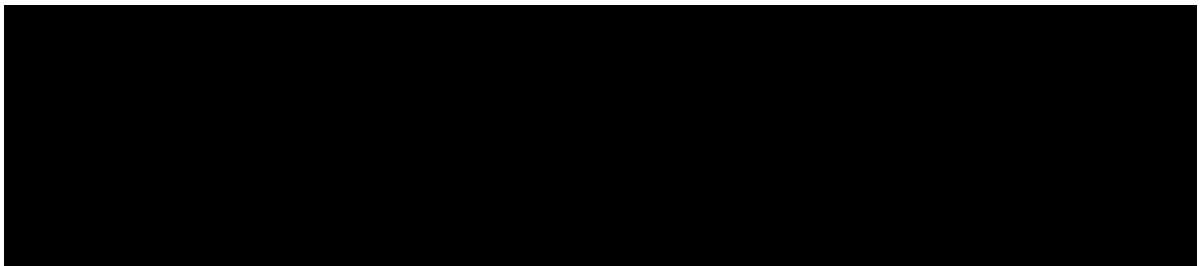
`int gcount();`

Hàm trả về số ký tự đã trích bởi hàm nhập không định dạng cuối cùng.

■ [Ví dụ 8.12](#): Sử dụng hàm `getline()`

```
CT8_12.CPP
1: //Chương trình 8.12
2: #include <iostream.h>
3:
4: const int SIZE = 80;
5:
6: int main()
7: {
8:     char Buffer[SIZE];
9:     cout << "Enter a sentence:" << endl;
10:    cin.read(Buffer, 20);
11:    cout << endl << "The sentence entered was:" << endl;
12:    cout.write(Buffer, cin.gcount());
13:    cout << endl;
14:    return 0;
15: }
```

Chúng ta [chạy ví dụ 8.12](#), kết quả ở hình 8.14



Hình 8.14: Kết quả của ví dụ 8.12

8.6 CÁC BỘ XỬ LÝ DÒNG (STREAM MANIPULATOR)

C++ cung cấp các bộ xử lý dòng khác nhau mà thực hiện các công việc định dạng. Các bộ xử lý dòng cung cấp các khả năng như ấn định các chiều rộng trường, ấn định độ chính xác, ấn định và không ấn định các cờ định dạng, ấn định lấp đầy (fill) ký tự trong các trường, flush dòng, chèn một ký tự newline vào dòng xuất và flush dòng, chèn một ký tự null vào dòng xuất và nhảy khoảng trắng trong dòng nhập.

8.6.1 Các bộ xử lý `dec`, `oct`, `hex` và `setbase()`:

Các số nguyên bình thường được thể hiện các giá trị thập phân (cơ số 10). Để thay đổi mà các số nguyên được thể hiện trên dòng, chèn bộ xử lý `ios::hex` để ấn định cơ số 16 hoặc chèn bộ xử lý `ios::oct` để ấn định cơ số 8. Chèn bộ xử lý dòng `ios::dec` để xác lập lại cơ số dòng thập phân.

Cơ số của dòng cũng có thể thay đổi bởi bộ xử lý dòng `setbase()` có một tham số là số nguyên với các giá trị:

0: Dòng cơ sở mặc định (cơ số 10). Trong trường hợp này muốn nhập số ở dạng bát phân, chúng ta phải ghi chữ số 0 phía trước số đó.

8: Dòng cơ số 8.

10: Dòng cơ số 10.

16: Dòng cơ số 16.

Vì hàm `setbase()` có một tham số nên nó được gọi là một bộ xử lý dòng có tham số. Sử dụng hoặc bất kỳ bộ xử lý dòng có tham số nào phải include tập tin `<iomanip.h>`. Trong file này có ba lớp làm mẫu **OMANIP**, **IMANIP**, và **SMANIP** lần lượt cần cho việc định nghĩa bộ xử lý có tham số trên dòng xuất, dòng nhập và trên dòng nhập/xuất.

Lớp mẫu (class template) còn gọi là lớp chung chung (generic class). Gọi như vậy là vì lớp được định nghĩa với nhiều yếu tố được bỏ ngoặc. Chẳng hạn, lớp **OMANIP(class Type)** có dùng một kiểu dữ liệu chung chung gọi là *Type*. Khi *Type* thay bằng kiểu cụ thể, chẳng hạn **OMANIP(int)**, chúng ta có một lớp cụ thể tương ứng.

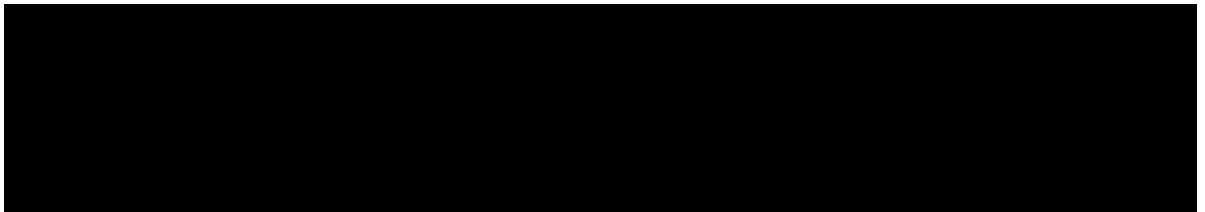
■ Ví dụ 8.13:

```

1: //Chương trình 8.13
2: #include <iostream.h>
3: #include <iomanip.h>
4:
5: int main()
6: {
7:     int n;
8:     cout << "Enter a decimal number: ";
9:     cin >> n;
10:    cout << n << " in hexadecimal is: "
11:         << hex << n << endl
12:         << dec << n << " in octal is: "
13:         << oct << n << endl
14:         << setbase(10) << n << " in decimal is: "
15:         << n << endl;
16:    return 0;
17: }

```

Chúng ta [chạy ví dụ 8.13](#), kết quả ở hình 8.15



Hình 8.15: Kết quả của ví dụ 8.13

8.6.2 Độ chính xác dấu chấm động (`ios::precision()`, `setprecision()`):

Chúng ta có thể điều khiển độ chính xác của các số chấm động, nghĩa là số các chữ số bên phải dấu chấm thập phân, bằng cách sử dụng các hàm `ios::precision()` và `setprecision()`.

Hàm **ios::precision()**:

(1) **int precision(int *np*);**

(2) **int precision();**

Dạng (2) trả về giá trị chính xác hiện giờ. Dạng (1) ấn định độ chính xác và trả về độ chính xác giá trị chính xác trước đó. Nếu *np* bằng 0 thì khôi phục độ chính xác mặc định (là 6).

Hàm **setprecision()**:

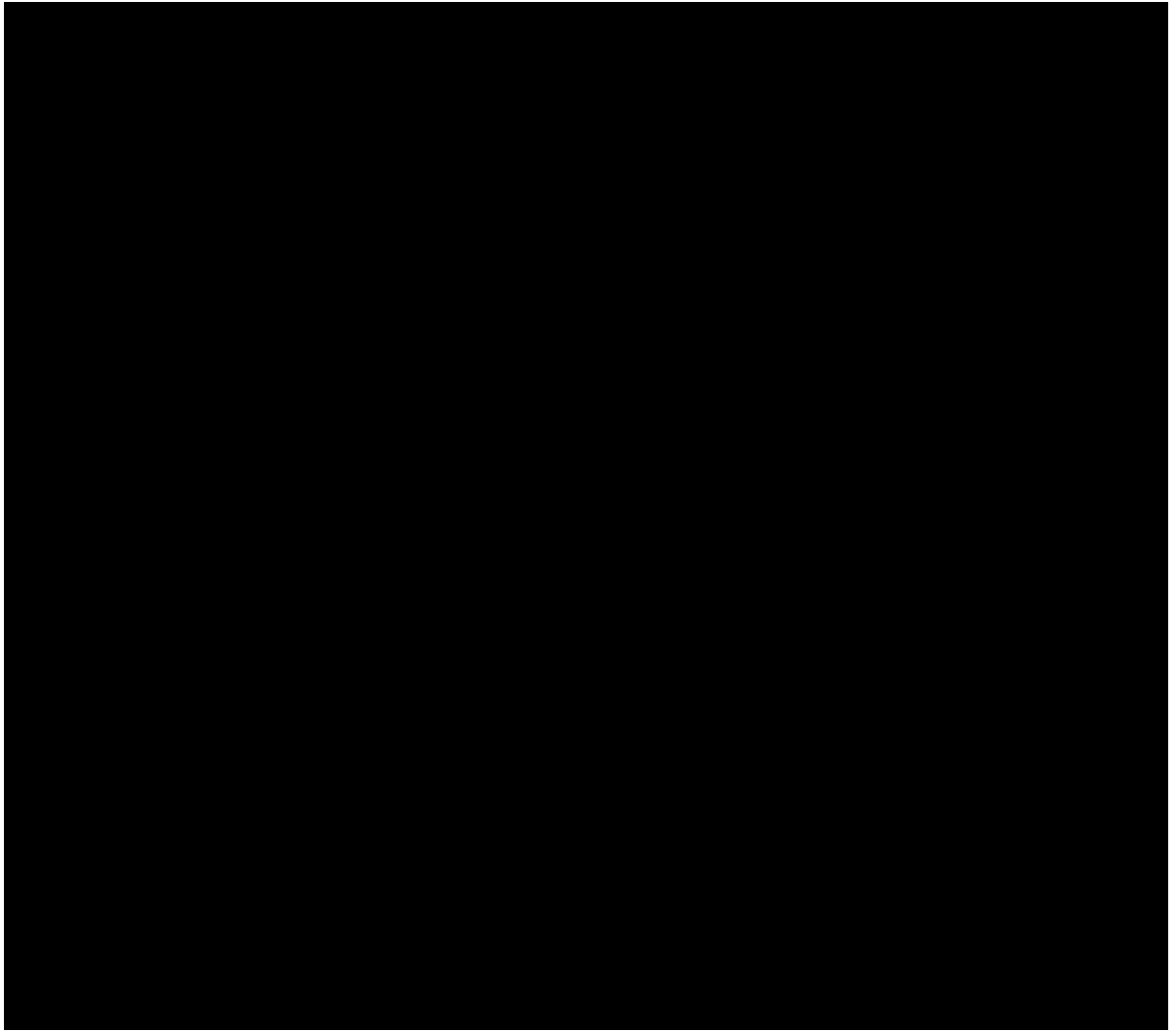
SMANIP(int) setprecision(int *np*);

np cho biết độ chính xác của số chấm động.

■ [Ví dụ 8.14:](#)


```
1: //Chương trình 8.14
2: #include <iostream.h>
3: #include <iomanip.h>
4: #include <math.h>
5:
6: int main()
7: {
8:     double Root2 = sqrt(2.0);
9:     cout << "Square root of 2 with precisions 0-9." << endl
10:          << "Precision set by the "
11:          << "precision member function:" << endl;
12:     for (int Places = 0; Places <= 9; Places++)
13:     {
14:         cout.precision(Places);
15:         cout << Root2 << endl;
16:     }
17:     cout << endl << "Precision set by the "
18:          << "setprecision manipulator:" << endl;
19:     for (Places = 0; Places <= 9; Places++)
20:         cout << setprecision(Places) << Root2 << endl;
21:     return 0;
22: }
```

Chúng ta [chạy ví dụ 8.14](#), kết quả ở hình 8.16



Hình 8.16: Kết quả của ví dụ 8.14

8.6.3 Độ rộng trường (setw(), ios::width()):

Hàm **ios::width()**:

(1) **int width(int *nw*);**

(2) **int width();**

Dạng (1) sẽ ấn định độ rộng trường nội tại của dòng với tham số *nw*. Khi độ rộng là 0 (mặc định). Việc chèn chỉ chèn số các ký tự cần thiết để biểu diễn giá trị đã chèn. Khi độ rộng khác 0, việc chèn độn vào trường với ký tự lấp đầy của dòng, lên tới *nw*. Giá trị độ rộng nội tại xác lập lại 0 sau mỗi lần trích hoặc chèn. Dạng (2) trả về giá trị hiện tại của biến độ rộng của dòng.

Hàm **setw()**:

SMANIP(int) setw(int *nw*);

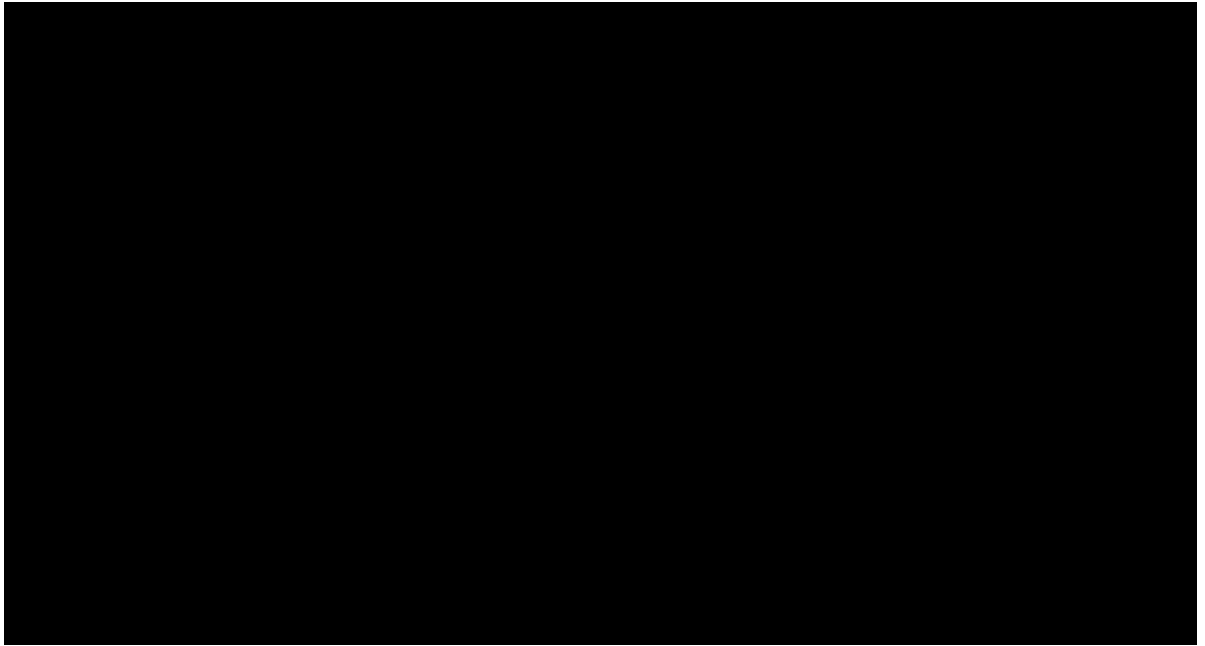
nw là số nguyên cho biết độ rộng của trường.

Ví dụ 8.15:

CT8_15.CPP

```
1: //Chương trình 8.15
2: #include <iostream.h>
3:
4: int main()
5: {
6:     int W = 4;
7:     char St[10];
8:     cout << "Enter a sentence:" << endl;
9:     cin.width(5);
10:    while (cin >> St)
11:    {
12:        cout.width(W++);
13:        cout << St << endl;
14:        cin.width(5);
15:    }
16:    return 0;
17: }
```

Chúng ta [chạy ví dụ 8.15](#), kết quả ở hình 8.17



Hình 8.17: Kết quả của ví dụ 8.15

8.6.4 Các bộ xử lý người dùng định nghĩa:

Người dùng có thể tạo các bộ xử lý dòng cho riêng mình. Định nghĩa bộ xử lý không tham số:

```
istream & function_name (istream & in)
```

```
{
```

```
    .....
```

```
    return in;
```

```
}
```

```
ostream & function_name (ostream & out)
```

```
{
```

```
    .....
```

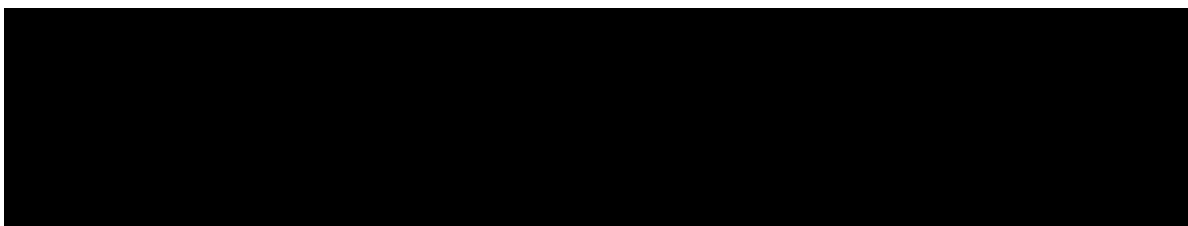
```
    return out;
```

```
}
```

■ [Ví dụ 8.16:](#) Bộ xử lý không tham số.

```
1: //Chương trình 8.16
2: #include <iostream.h>
3:
4: //Bộ xử lý bell (sử dụng chuỗi thoát \a)
5: ostream& bell(ostream& output)
6: {
7: return output << '\a';
8: }
9:
10: //Bộ xử lý ret (sử dụng chuỗi thoát \r)
11: ostream& ret(ostream& output)
12: {
13: return output << '\r';
14: }
15:
16: //Bộ xử lý tab (sử dụng chuỗi thoát \t)
17: ostream& tab(ostream& output)
18: {
19: return output << '\t';
20: }
21:
22: //Bộ xử lý endl (sử dụng chuỗi thoát \n và hàm flush())
23: ostream& endl(ostream& output)
24: {
25: return output << '\n' << flush;
26: }
27:
28: int main()
29: {
```

Chúng ta [chạy ví dụ 8.16](#), kết quả ở hình 8.18



Hình 8.18: Kết quả của ví dụ 8.16

Muốn định nghĩa bộ xử lý có tham số, chúng ta phải include tập tin **<iomanip.h>**. Bộ xử lý có tham số trên dòng xuất có dạng:

```
ostream &function_name (ostream & out, parameter_type parameter)  
  
{  
  
    .....  
  
    return out;  
  
}  
  
OMANIP(type)function_name (parameter_type parameter)  
  
{  
  
    return OMANIP(type) (function_name, parameter);  
  
}
```

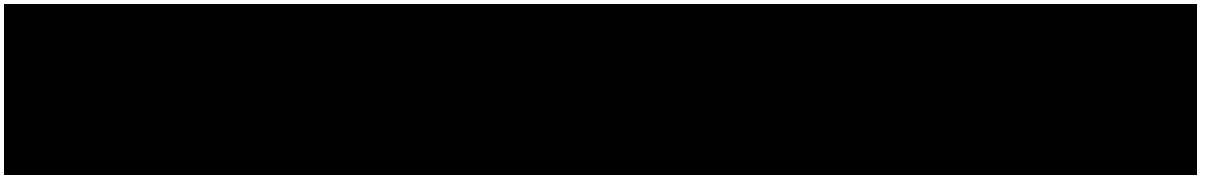
Tương tự dạng định nghĩa bộ xử lý trên dòng nhập và trên dòng nhập/xuất.

■ [Ví dụ 8.17](#): Bộ xử lý có tham số.



```
1: //Chương trình 8.17
2: #include <iostream.h>
3: #include <iomanip.h>
4:
5: ostream & indent(ostream & Stream,int Len)
6: {
7:     for(int I=0;I<Len;++I)
8:         cout<<" ";
9:     return Stream;
10: }
11:
12: OMANIP (int) indent(int Len)
13: {
14:     return OMANIP(int) (indent,Len);
15: }
16:
17: int main()
18: {
19:     cout<<indent(10)<<"Hello"<<endl;
20:     cout<<indent(20)<<"Peter"<<endl;
21:     cout<<indent(5)<<"John"<<endl;
22:     return 0;
23: }
```

Chúng ta [chạy ví dụ 8.17](#), kết quả ở hình 8.19



Hình 8.19: Kết quả của ví dụ 8.17

8.7 CÁC TRẠNG THÁI ĐỊNH DẠNG DÒNG

Các cờ định dạng khác nhau mô tả các loại định dạng được thực thi trong suốt quá trình thao tác nhập/xuất dòng. Các hàm `ios::setf()`, `ios::unsetf()`, và `ios::flags()` điều khiển các thiết lập cờ.

Hàm `ios::setf()`:

(1) `long setf(long lFlags);`

(2) `long setf(long lFlags, long lMask);`

Trong đó:

lFlags: Định dạng các giá trị bit cờ. Có thể kết hợp các bit cờ bằng cách sử dụng toán tử `|`. Các bit cờ:

Bit cờ	Ý nghĩa
<code>ios::skipws</code>	Bỏ qua khoảng trắng trên nhập.
<code>ios::left</code>	Giá trị canh lề trái; Độn thêm bên phải với ký tự lấp đầy.
<code>ios::right</code>	Giá trị canh lề phải; Độn thêm bên trái với ký tự lấp đầy (mặc định).
<code>ios::internal</code>	Thêm các ký tự lấp đầy sau bất kỳ dấu chỉ dẫn hoặc dấu hiệu cơ số nào, nhưng trước giá trị.
<code>ios::dec</code>	Giá trị số định dạng theo cơ số 10 (mặc định).
<code>ios::oct</code>	Giá trị số định dạng theo cơ số 8.
<code>ios::hex</code>	Giá trị số định dạng theo cơ số 16.
<code>ios::showbase</code>	Hiển thị các hằng số theo dạng mà có thể đọc bởi trình biên dịch C++.
<code>ios::showpoint</code>	Xuất hiện dấu thập phân và dạng thức zero theo đuôi cho các giá trị chấm động.
<code>ios::uppercase</code>	Hiển thị chữ hoa từ A đến F đối với giá trị hệ 16 và E đối với các giá trị dạng khoa học.

ios::showpos	Hiển thị dấu + đối với các số dương.
ios::scientific	Hiển thị số chấm động theo dạng khoa học.
ios::fixed	Hiển thị số chấm động theo dạng cố định

lMask: Định dạng mặt nạ bit cờ.

Hàm trả về giá trị các bit cờ trước đó.

Hàm **ios::flags()**:

(1) **long flags(long lFlags);**

(2) **long flags();**

Dạng (2) trả về các bit cờ hiện tại của dòng. Dạng (1) giống như dạng (1) của hàm **ios::setf()**.

Hàm **ios::unsetf()**:

long unsetf(long lFlags);

Hàm xóa các bit cờ chỉ định bởi *lFlags*.

Ngoài ra chúng ta còn có hàm **setiosflags()** giống như dạng (1) của **ios::setf()** và hàm **resetiosflags()** giống như hàm **unsetf()**.

Các cờ bit **ios::left**, **ios::right**, và **ios::internal** được chứa trong biến thành viên **ios::adjustfield**. Chẳng hạn:

```
ostream os;

if( ( os.flags() & ios::adjustfield ) == ios::left )

    .....
```

Tham số **ios::adjustfield** phải được cung cấp như tham số thứ 2 ở dạng (2) của hàm **ios::setf()** khi ấn định các cờ bit căn lề (**ios::left**, **ios::right**, và **ios::internal**). Điều này cho phép **ios::setf()** bảo đảm chỉ có một trong ba cờ bit căn lề được ấn định. Chẳng hạn:

```
cout.setf(ios::left,ios::adjustfield);
```

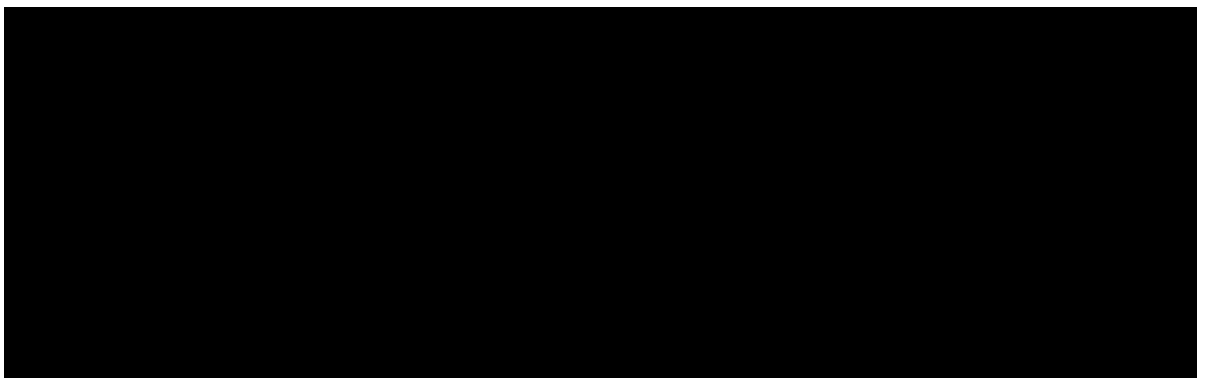
Ví dụ 8.18: Cờ bit **ios::showpoint**.

```

1: //Chương trình 8.18
2: #include <iostream.h>
3: #include <iomanip.h>
4: #include <math.h>
5:
6: int main()
7: {
8:     cout << "cout prints 9.9900 as: " << 9.9900 << endl
9:         << "cout prints 9.9000 as: " << 9.9000 << endl
10:        << "cout prints 9.0000 as: " << 9.0000 << endl <<
endl
11:        << "After setting the ios::showpoint flag" << endl;
12:     cout.setf(ios::showpoint);
13:     cout << "cout prints 9.9900 as: " << 9.9900 << endl
14:         << "cout prints 9.9000 as: " << 9.9000 << endl
15:         << "cout prints 9.0000 as: " << 9.0000 << endl;
16:     return 0;
17: }

```

Chúng ta [chạy ví dụ 8.18](#), kết quả ở hình 8.20



Hình 8.20: Kết quả của ví dụ 8.18

[Ví dụ 8.19](#): Cờ bit `ios::left`, `ios::right`, `ios::internal`.

```

1: //Chương trình 8.19

2: #include <iostream.h>

3: #include <iomanip.h>

4:

5: int main()

6: {

7: int X = 12345;

8: cout << "Default is right justified:" << endl

9:         << setw(10) << X << endl << endl

10:        << "USING MEMBER FUNCTIONS" << endl

11:        << "Use setf to set ios::left:" << endl << setw(10);

12: cout.setf(ios::left, ios::adjustfield);

13: cout << X << endl << "Use unsetf to restore default:" <<
endl;

14: cout.unsetf(ios::left);

15: cout << setw(10) << X << endl << endl

16:        << "USING PARAMETERIZED STREAM MANIPULATORS" << endl

17:        << "Use setiosflags to set ios::left:" << endl

18:        << setw(10) << setiosflags(ios::left) << X << endl

19:        << "Use resetiosflags to restore default:" << endl

20:        << setw(10) << resetiosflags(ios::left)

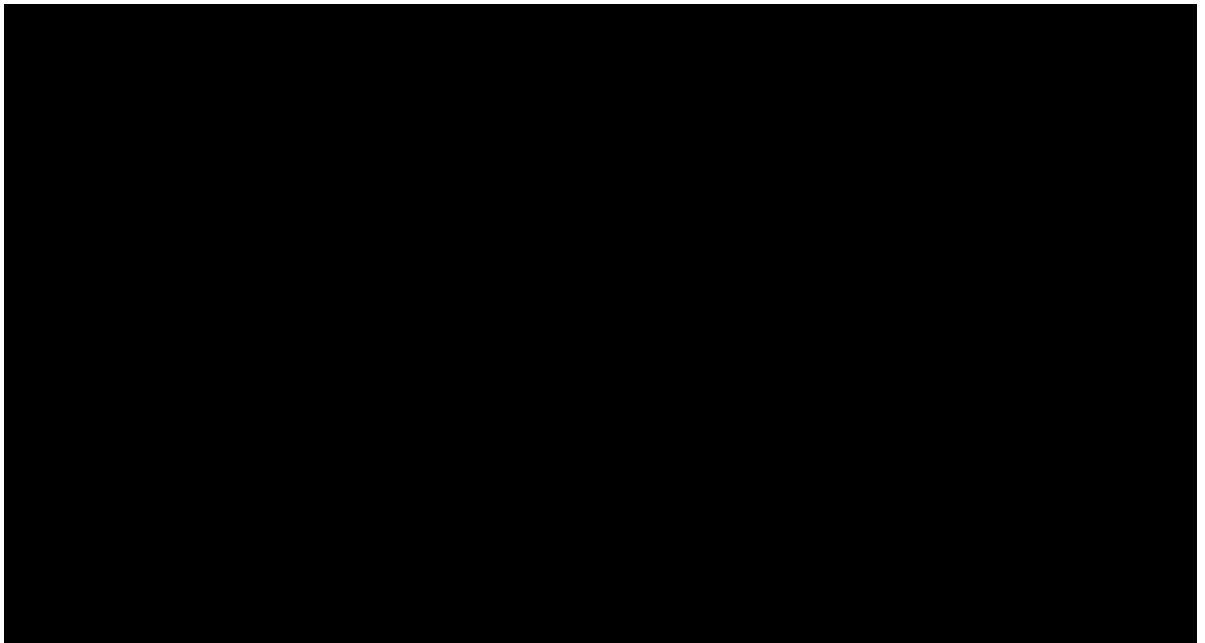
21:        << X << endl;

22: return 0;

23: }

```

Chúng ta [chạy ví dụ 8.19](#), kết quả ở hình 8.21



Hình 8.21: Kết quả của ví dụ 8.19

Ví dụ 8.20: Cờ bit `ios::showpos` và `ios::internal`.

```
CT8_20.CPP
1: //Chương trình 8.20
2: #include <iostream.h>
3: #include <iomanip.h>
4:
5: int main()
6: {
7:     cout << setiosflags(ios::internal | ios::showpos)
8:         << setw(10) << 123 << endl;
9:     return 0;
10: }
```

Chúng ta chạy ví dụ 8.20, kết quả ở hình 8.22



Hình 8.22: Kết quả của ví dụ 8.20

Để ký tự lấp đầy chúng ta dùng hàm **ios::fill()** và **setfill()**:

Hàm **ios::fill()**:

(1) **char fill(char cFill);**

(2) **char fill();**

Dạng (1) ấn định ký tự lấp đầy nội tại và trả về ký tự lấp đầy trước đó. Dạng (2) trả về ký tự lấp đầy của dòng.

Hàm **setfill()**:

SMANIP(int) setfill(int nFill);

Ví dụ 8.21: Hàm **ios::fill()** và **setfill()**.



```
1: //Chương trình 8.21
2: #include <iostream.h>
3: #include <iomanip.h>
4:
5: int main()
6: {
7:     int X = 10000;
8:     cout << X
9:         << " printed as int right and left justified" << endl
10:        << "and as hex with internal justification." << endl
11:        << "Using the default pad character (space):" <<
endl;
12:     cout.setf(ios::showbase);
13:     cout << setw(10) << X << endl;
14:     cout.setf(ios::left, ios::adjustfield);
15:     cout << setw(10) << X << endl;
16:     cout.setf(ios::internal, ios::adjustfield);
17:     cout << setw(10) << hex << X << endl << endl;
18:     cout << "Using various padding characters:" << endl;
19:     cout.setf(ios::right, ios::adjustfield);
20:     cout.fill('*');
21:     cout << setw(10) << dec << X << endl;
22:     cout.setf(ios::left, ios::adjustfield);
23:     cout << setw(10) << setfill('%') << X << endl;
24:     cout.setf(ios::internal, ios::adjustfield);
25:     cout << setw(10) << setfill('^') << hex
26:         << X << endl;
27:     return 0;
28: }
```

Chúng ta [chạy ví dụ 8.21](#), kết quả ở hình 8.23



Hình 8.23: Kết quả của ví dụ 8.21

[Ví dụ 8.22](#): Cờ bit `ios::showbase`.

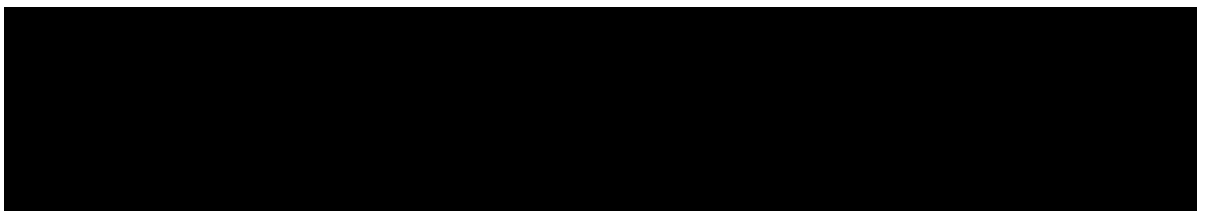


```

1: //Chương trình 8.22
2: #include <iostream.h>
3: #include <iomanip.h>
4:
5: int main()
6: {
7:     int X = 100;
8:     cout << setiosflags(ios::showbase)
9:           << "Printing integers preceded by their base:" <<
endl
10:          << X << endl
11:          << oct << X << endl
12:          << hex << X << endl;
13:     return 0;
14: }

```

Chúng ta [chạy ví dụ 8.22](#), kết quả ở hình 8.24



Hình 8.24: Kết quả của ví dụ 8.22

[Ví dụ 8.23](#): Hiển thị các giá trị chấm động theo dạng mặc định hệ thống, dạng khoa học và dạng cố định.


```
1: //Chương trình 8.23
2: #include <iostream.h>
3:
4: int main()
5: {
6:     double X = .001234567, Y = 1.946e
7:     cout << "Displayed in default format:" << endl
8:         << X << '\t' << Y << endl;
9:     cout.setf(ios::scientific, ios::floatfield);
10:    cout << "Displayed in scientific format:" << endl
11:        << X << '\t' << Y << endl;
12:    cout.unsetf(ios::scientific);
13:    cout << "Displayed in default format after unsetf:" << endl
14:        << X << '\t' << Y << endl;
15:    cout.setf(ios::fixed, ios::floatfield);
16:    cout << "Displayed in fixed format:" << endl
17:        << X << '\t' << Y << endl;
18:    return 0;
19: }
```

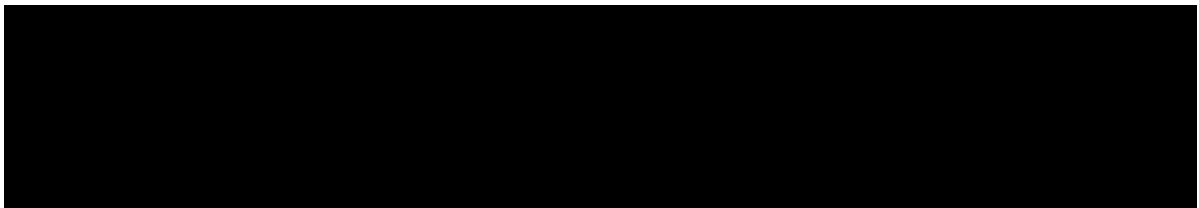
Chúng ta [chạy ví dụ 8.23](#), kết quả ở hình 8.25

Hình 8.25: Kết quả của ví dụ 8.23

Ví dụ 8.24: Cờ bit `ios::uppercase`.

```
CT8_24.CPP
1: //Chương trình 8.24
2: #include <iostream.h>
3: #include <iomanip.h>
4:
5: int main()
6: {
7:     cout << setiosflags(ios::uppercase)
8:         << "Printing uppercase letters in scientific" << endl
9:         << "notation exponents and hexadecimal values:" << endl
10:        << 4.345e10 << endl
11:        << hex << 123456789 << endl;
12:     return 0;
13: }
```

Chúng ta chạy ví dụ 8.24, kết quả ở hình 8.26



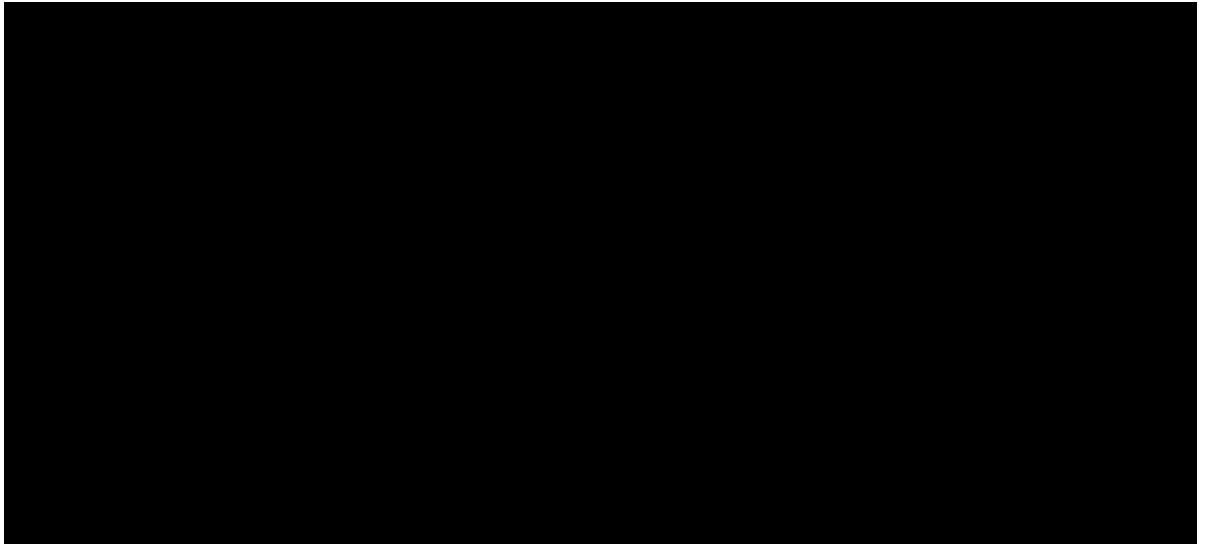
Hình 8.26: Kết quả của ví dụ 8.24

Ví dụ 8.25: Ấn định và xác lập lại các cờ định dạng (`ios::flags`, `setiosflags()`, `resetiosflags()`).



```
1: //Chương trình 8.25
2: #include <iostream.h>
3: int main()
4: {
5:     int I = 1000;
6:     double D = 0.0947628;
7:     cout << "The value of the flags variable is: "
8:         << cout.flags() << endl
9:         << "Print int and double in original format:" << endl
10:        << I << '\t' << D << endl << endl;
11:     long originalFormat = cout.flags(ios::oct | ios::scientific);
12:     cout << "The value of the flags variable is: "
13:         << cout.flags() << endl
14:         << "Print int and double in a new format" << endl
15:         << "specified using the flags member function:" <<
endl
16:        << I << '\t' << D << endl << endl;
17:     cout.flags(originalFormat);
18:     cout << "The value of the flags variable is: "
19:         << cout.flags() << endl
20:         << "Print values in original format again:" << endl
21:        << I << '\t' << D << endl;
22:     return 0;
23: }
```

Chúng ta [chạy ví dụ 8.25](#), kết quả ở hình 8.27



Hình 8.27: Kết quả của ví dụ 8.25

8.8 CÁC TRẠNG THÁI LỖI DÒNG

Trạng thái của dòng có thể được kiểm tra thông qua các bit trong lớp **ios**-lớp cơ sở cho các lớp **istream**, **ostream**, và **iostream** chúng ta sử dụng cho nhập/xuất. Các bit trạng thái được định nghĩa kiểu liệt kê thuộc lớp **ios** như sau:

Bit trạng thái	Ý nghĩa
ios::eofbit	Bật lên nếu end-of-bit đạt đến.
ios::failbit	Bật lên nếu một lỗi định dạng xảy ra trên dòng nhưng các ký tự không bị mất.
ios::badbit	Bật lên nếu có lỗi nghiêm trọng xảy ra .
ios::goodbit	Bật lên nếu các bit eofbit , failbit , và badbit không bật lên.

Để biết được trạng thái của các bit, chúng ta có hai cách đọc:

Cách 1: Tương ứng với các bit, chúng ta có các hàm:

Bit trạng thái	Hàm tương ứng
ios::eofbit	ios::eof() int eof();
ios::failbit	ios::fail() int fail();

ios::badbit	ios::bad() int bad();
ios::goodbit	ios::good() int good();

Cách 2: Dùng hàm **ios::rdstate()**:

```
int rdstate();
```

Hàm trả về trạng thái lỗi của stream bởi các mặt nạ: **ios::eofbit**, **ios::failbit**, **ios::badbit**, và **ios::goodbit**.

Để xóa bit trạng thái nào chúng ta dùng hàm **ios::clear()**:

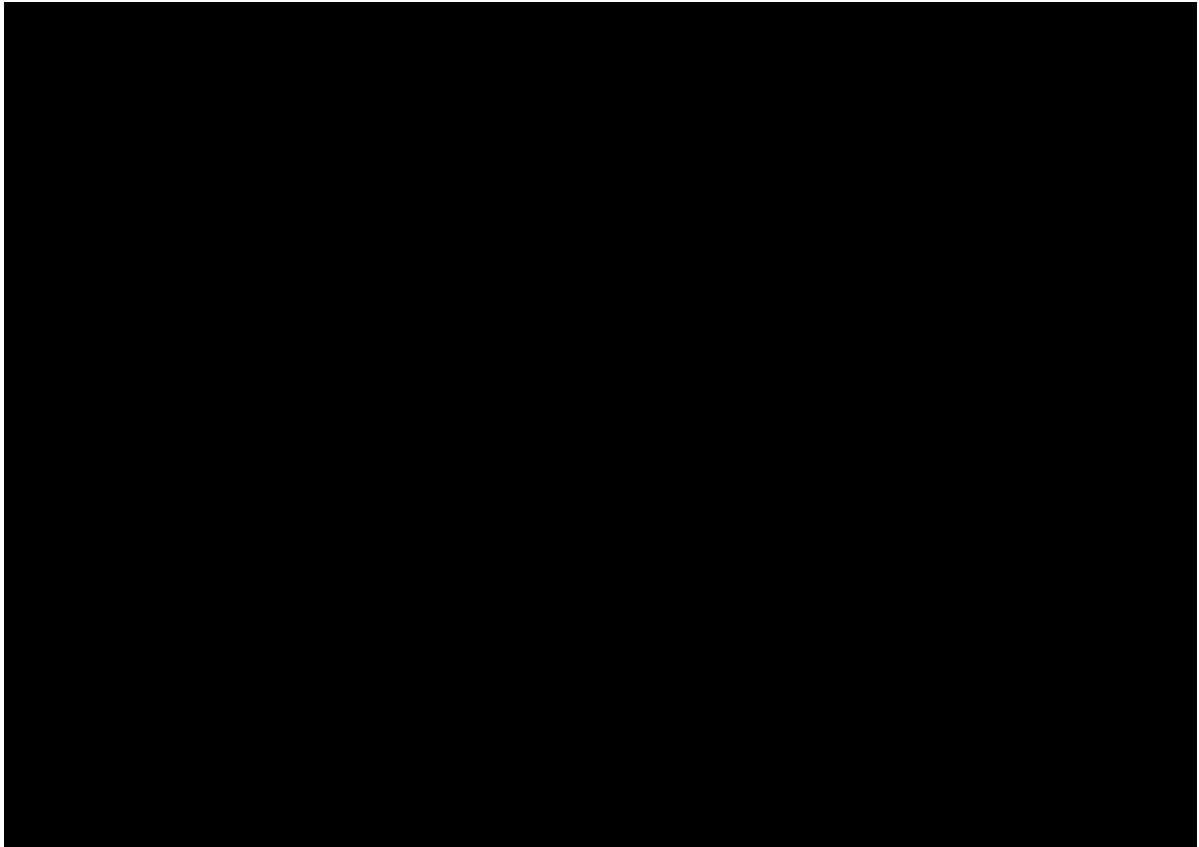
```
void clear(int nState = 0);
```

Trong đó *nState*=0 thì tất cả các bit lỗi đều xóa. Chúng ta có thể dùng toán tử | để kết hợp các bit lại.

Ví dụ 8.26: Kiểm tra các trạng thái lỗi.

```
1: //Chương trình 8.26
2: #include <iostream.h>
3:
4: int main()
5: {
6: int X;
7: cout << "Before a bad input operation:" << endl
8:     << "cin.rdstate(): " << cin.rdstate() << endl
9:     << " cin.eof(): " << cin.eof() << endl
10:    << " cin.fail(): " << cin.fail() << endl
11:    << " cin.bad(): " << cin.bad() << endl
12:    << " cin.good(): " << cin.good() << endl << endl
13:    << "Expects an integer, but enter a character: ";
14: cin >> X;
15: cout << endl << "After a bad input operation:" << endl
16:     << "cin.rdstate(): " << cin.rdstate() << endl
17:     << " cin.eof(): " << cin.eof() << endl
18:     << " cin.fail(): " << cin.fail() << endl
19:     << " cin.bad(): " << cin.bad() << endl
20:     << " cin.good(): " << cin.good() << endl << endl;
21: cin.clear();
22: cout << "After cin.clear()" << endl
23:     << "cin.fail(): " << cin.fail() << endl
24:     << "cin.good(): " << cin.good() << endl;
25: return 0;
26: }
```

Chúng ta [chạy ví dụ 8.26](#), kết quả ở hình 8.28



Hình 8.28: Kết quả của ví dụ 8.26

Chúng ta có thể dùng toán tử ! trên dòng nhập/xuất, chẳng hạn:

```
if (!cin)
{
    cout<<"Error input"<<endl;
}
```

Tuy nhiên không thể viết:

```
if (cin)
{
    cout<<"OK!"<<endl;
}
```

8.9 NHẬP XUẤT CÁC KIỂU NGƯỜI DÙNG ĐỊNH NGHĨA

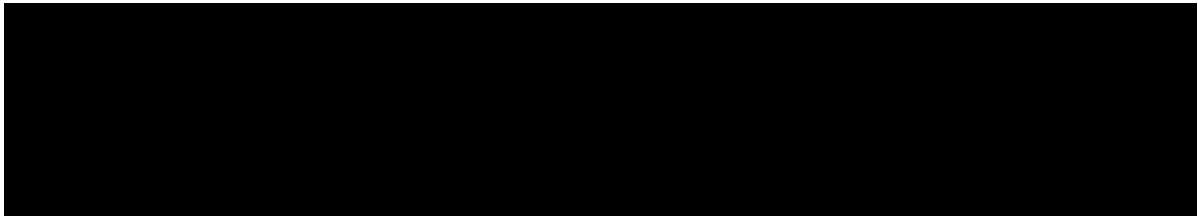
C++ có thể nhập và xuất các kiểu dữ liệu chuẩn sử dụng toán tử trích dòng >> và toán tử chèn dòng <<. Các toán tử này được đa năng hóa để xử lý mỗi kiểu dữ liệu chuẩn bao gồm các chuỗi và các địa chỉ bộ nhớ. Lập trình viên có thể đa năng hóa toán tử trích dòng và toán tử chèn dòng để thực hiện nhập/xuất cho các kiểu người dùng định nghĩa (Chúng ta đã tìm hiểu kỹ ở chương 4).

Ví dụ 8.27:




```
1: //Chương trình 8.27
2: #include <iostream.h>
3:
4: class PhoneNumber
5: {
6: private:
7: char AreaCode[4];
8: char Exchange[4];
9: char Line[5];
10: friend ostream& operator <<(ostream& Output, PhoneNumber&
Num);
11: friend istream& operator >>(istream& Input, PhoneNumber&
Num);
12: };
13:
14: ostream& operator <<(ostream& Output, PhoneNumber& Num)
15: {
16: Output << "(" << Num.AreaCode << ")" "
17: << Num.Exchange << "-" << Num.Line;
18: return Output;
19: }
20:
21: istream& operator>>(istream& Input, PhoneNumber& Num)
22: {
23: Input.ignore(); //Bỏ qua (
24: Input.getline(Num.AreaCode, 4);
25: Input.ignore(2); //Bỏ qua ) và khoảng trắng
26: Input.getline(Num.Exchange, 4);
27: Input.ignore(); //Bỏ qua -
28: Input.getline(Num.Line, 5);
```

Chúng ta [chạy ví dụ 8.27](#), kết quả ở hình 8.29



Hình 8.29: Kết quả của ví dụ 8.27

8.10 GẮN MỘT DÒNG XUẤT TỚI MỘT DÒNG NHẬP

C++ cung cấp hàm **ios::tie()** để gắn liền dòng xuất với dòng nhập. Với hàm dòng xuất sẽ được làm sạch trước khi có bất kỳ phép nhập nào được thực hiện trên dòng nhập. Hàm này có dạng:

```
ostream* tie(ostream* pos);
```

```
ostream* tie();
```

Chẳng hạn:

```
cin.tie(cout);
```

8.11 DÒNG NHẬP/ XUẤT FILE

Để thực thi xử lý file trong C++, các chương trình phải include tập tin **<iostream.h>** và **<fstream.h>**. Header **<fstream.h>** gồm định nghĩa cho các lớp dòng **ifstream** cho nhập (đọc) từ một file, **ofstream** cho xuất (ghi) tới một file) và **fstream** cho nhập/xuất (đọc/ghi) tới một file. Các file được mở bằng cách tạo các đối tượng của các lớp dòng này. Cây phả hệ của các lớp này ở hình 8.2.

Constructor của lớp **ofstream**:

(1) **ofstream();**

(2) **ofstream(const char* szName, int nMode=ios::out, int nProt=filebuf::openprot);**

(3) **ofstream(int fd);**

(4) **ofstream(filedesc fd, char* pch, int nLength);**

Trong đó:

szName: Tên file được mở.

nMode: Một số nguyên chứa các bit mode định nghĩa là kiểu liệt kê của **ios**. Có thể kết hợp bằng toán tử **|**. Tham số này có thể một trong các giá trị sau:

Mode	Ý nghĩa
ios::app	Hàm di chuyển con trỏ file tới end-of-file. Khi các byte mới được ghi lên file, chúng luôn luôn nối thêm vào cuối, ngay cả vị trí được di chuyển với hàm ostream::seekp() .
ios::ate	Hàm di chuyển con trỏ file tới end-of-file. Khi byte mới đầu tiên được ghi lên file, chúng luôn luôn nối thêm vào cuối, nhưng khi các byte kế tiếp được ghi, chúng ghi vào vị trí hiện hành.
ios::in	Mở file để đọc. Với dòng ifstream , việc mở file đương nhiên được thực hiện ở chế độ này.
ios::out	Mở file để đọc. Với dòng ofstream , việc mở file đương nhiên được thực hiện ở chế độ này.
ios::trunc	Xóa file hiện có trên đĩa và tạo file mới cùng tên. Cũng có hiểu đây là chặt cụt file cũ, làm cho kích thước của nó bằng 0, chuẩn bị ghi nội dung mới. Mode này được áp dụng nếu ios::out được chỉ định và ios::app , ios::ate , và ios::in không được chỉ định.
ios::nocreate	Nếu file không tồn tại thì thao tác mở thất bại.
ios::noreplace	Nếu file tồn tại thì thao tác mở thất bại.
ios::binary	Mở file ở chế độ nhị phân (mặc định là ở chế độ văn bản).

nProt: Đặc tả chế độ bảo vệ file.

fd: Mã nhận diện file.

pch: Con trỏ tới vùng dành riêng chiều dài *nLength*. Giá trị **NULL** (hoặc *nLength*=0) dẫn đến dòng không vùng đệm.

nLength: Chiều dài tính theo byte của vùng dành riêng (0=không vùng đệm).

Dạng (1) xây dựng một đối tượng **ofstream** mà không mở file.

Dạng (2) xây dựng một đối tượng **ofstream** và mở file đã chỉ định.

Dạng (3) xây dựng một đối tượng **ofstream** và gắn (attach) với một file mở.

Dạng (4) xây dựng một đối tượng **ofstream** mà liên kết với đối tượng **filebuf**. Đối tượng **filebuf** được gắn tới file mở và vùng dành riêng.

Constructor của lớp **ifstream**:

(1) **ifstream();**

(2) **ifstream(const char* szName,int nMode=ios::in,int nProt=filebuf::openprot);**

(3) **ifstream(int fd);**

(4) **ifstream(filedesc fd, char* pch, int nLength);**

Dạng (1) xây dựng một đối tượng **ifstream** mà không mở file.

Dạng (2) xây dựng một đối tượng **ifstream** và mở file đã chỉ định.

Dạng (3) xây dựng một đối tượng **ifstream** và gắn (attach) với một file mở.

Dạng (4) xây dựng một đối tượng **ifstream** mà liên kết với đối tượng **filebuf**. Đối tượng **filebuf** được gắn tới file mở và vùng dành riêng.

Constructor của lớp **fstream**:

(1) **fstream();**

(2) **fstream(const char* szName, int nMode, int nProt=filebuf::openprot);**

(3) **fstream(int fd);**

(4) **fstream(filedesc fd, char* pch, int nLength);**

Dạng (1) xây dựng một đối tượng **fstream** mà không mở file.

Dạng (2) xây dựng một đối tượng **fstream** và mở file đã chỉ định.

Dạng (3) xây dựng một đối tượng **fstream** và gắn (attach) với một file mở.

Dạng (4) xây dựng một đối tượng **fstream** mà liên kết với đối tượng **filebuf**. Đối tượng **filebuf** được gắn tới file mở và vùng dành riêng.

Nếu chúng ta sử dụng constructor ở dạng (1) thì chúng ta dùng hàm **open()** để mở file:

Hàm **ofstream::open()**:

```
void open(const char* szName, int nMode=ios::out, int  
nProt=filebuf::openprot);
```

Hàm **ifstream::open()**:

```
void open(const char* szName, int nMode=ios::in, int  
nProt=filebuf::openprot);
```

Hàm **fstream::open()**:

```
void open(const char* szName, int nMode, int nProt=filebuf::openprot);
```

Để đóng file chúng ta dùng hàm **close()**, hàm này ở các lớp **ifstream**, **ofstream**, và **fstream** đều có dạng:

void close();

Các hàm liên quan đến con trỏ file:

Lớp **istream**:

Hàm **seekg()**: (*seek get*)

(1) **istream& seekg(streampos pos);**

(2) **istream& seekg(streamoff off,ios::seek_dir dir);**

Trong đó:

+ *pos*: Vị trí mới. **streampos** là tương đương **typedef** với **long**.

+ *off*: Giá trị offset mới. là tương đương **typedef** với **long**.

+ *dir*: hướng seek. Có một trong các trị sau:

ios::begin Seek từ bắt đầu của dòng.

ios::cur Seek từ vị trí hiện hành của dòng

ios::end Seek từ cuối của dòng

Hàm **tellg()**: (*tell get*)

streampos tellg();

Hàm trả về vị trí hiện hành của con trỏ file.

Lớp **ostream**:

Hàm **seekp()**: (*seek put*)

(1) **ostream& seekp(streampos pos);**

(2) **ostream& seekp(streamoff off,ios::seek_dir dir);**

Hàm **tellp()**: (*tell put*)

streampos tellp();

Hàm trả về vị trí hiện hành của con trỏ file.

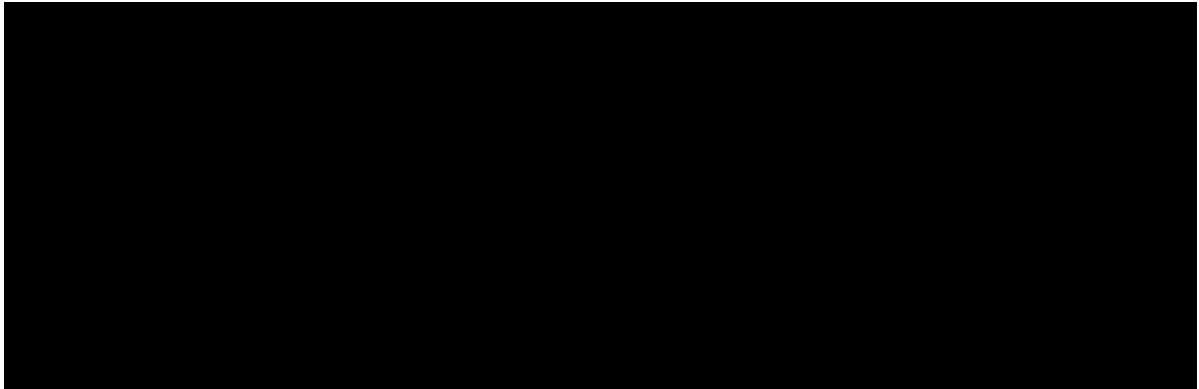
8.11.1 Nhập/xuất file văn bản:

Nếu dòng được gắn với file văn bản, việc nhập/xuất file được thực hiện một cách đơn giản bởi các toán tử >> và <<, giống như khi chúng ta làm việc với **cin** và **cout**. File văn bản chứa dữ liệu ở dạng mã ASCII, kết thúc bởi ký tự EOF.

Ví dụ 8.28: Tạo file văn bản có thể được sử dụng trong hệ thống có thể nhận được các tài khoản để giúp đỡ quản lý tiền nợ bởi các khách hàng tín dụng của công ty. Mỗi khách hàng, chương trình chứa một số tài khoản, tên và số dư (balance).

```
1: //Chương trình 8.28
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <stdlib.h>
5:
6: int main()
7: {
8:     ofstream OutClientFile("clients.dat", ios::out);
9:     if (!OutClientFile)
10:     {
11:         cerr << "File could not be opened" << endl;
12:         exit(1);
13:     }
14:     cout << "Enter the Account, Name, and Balance." << endl
15:          << "Enter EOF to end input." << endl << "? ";
16:     int Account;
17:     char Name[10];
18:     float Balance;
19:     while (cin >> Account >> Name >> Balance)
20:     {
21:         OutClientFile << Account << " " << Name
22:                      << " " << Balance << endl;
23:         cout << "? ";
24:     }
25:     OutClientFile.close();
26:     return 0;
27: }
```

Chúng ta [chạy ví dụ 8.28](#), kết quả ở hình 8.30



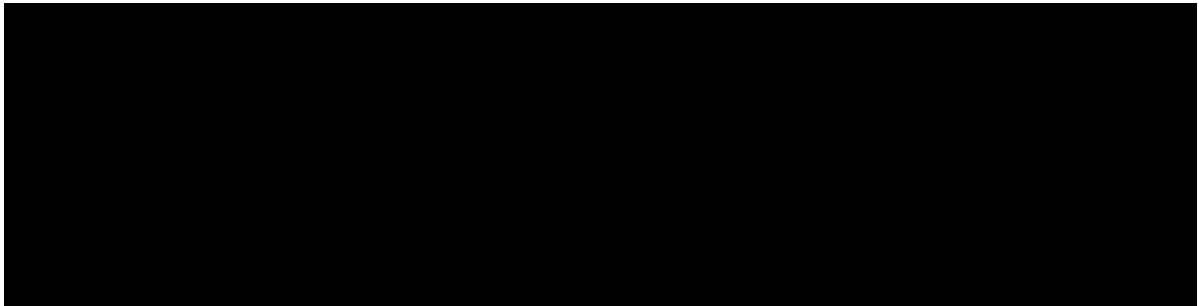
Hình 8.30: Kết quả của ví dụ 8.28

[Ví dụ 8.29](#): Đọc file văn bản tạo ở ví dụ 8.28 và xuất ra màn hình.




```
1: //Chương trình 8.29
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <iomanip.h>
5: #include <stdlib.h>
6:
7: void OutputLine(int, char*, float);
8:
9: int main()
10: {
11:     ifstream InClientFile("clients.dat", ios::in);
12:     if (!InClientFile)
13:     {
14:         cerr << "File could not be opened" << endl;
15:         exit(1);
16:     }
17:     int Account;
18:     char Name[10];
19:     float Balance;
20:     cout << setiosflags(ios::left) << setw(10) << "Account"
21:          << setw(13) << "Name" << "Balance" << endl;
22:     while (InClientFile >> Account >> Name >> Balance)
23:         OutputLine(Account, Name, Balance);
24:     InClientFile.close();
25:     return 0;
26: }
27:
28: void OutputLine(int Acct, char *Name, float Bal)
29: {
```

Chúng ta [chạy ví dụ 8.29](#), kết quả ở hình 8.31



Hình 8.31: Kết quả của ví dụ 8.29

Do lớp **ifstream** dẫn xuất từ lớp **istream** nên chúng ta có thể dùng các hàm **istream::get()**, **istream::getline()**.

[Ví dụ 8.30](#): Đọc file văn bản tạo ở ví dụ 8.28 bằng hàm **istream::getline()** và xuất ra màn hình.



```
1: //Chương trình 8.30
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <iomanip.h>
5: #include <stdlib.h>
6:
7: #define MAXLINE 256
8:
9: int main()
10: {
11:     ifstream InClientFile("clients.dat", ios::in);
12:     if (!InClientFile)
13:     {
14:         cerr << "File could not be opened" << endl;
15:         exit(1);
16:     }
17:     char Line[MAXLINE];
18:     while (!InClientFile.eof())
19:     {
20:         InClientFile.getline(Line, MAXLINE-1);
21:         cout << Line << endl;
22:     }
23:     InClientFile.close();
24:     return 0;
25: }
```

Chúng ta [chạy ví dụ 8.30](#), kết quả ở hình 8.32.(nội dung tập tin [clients.dat](#))



Hình 8.32: Kết quả của ví dụ 8.30

8.11.2 Nhập/xuất file nhị phân:

Đối với file nhị phân (còn gọi là file truy cập ngẫu nhiên), chúng ta mở ở chế độ **ios::binary**. Đối với file nhị phân, chúng ta có thể dùng hàm **istream::get()** và **ostream::put()** để đọc và ghi từng byte một. Để đọc và ghi nhiều byte cùng lúc chúng ta có thể dùng **istream::read()** và **ostream::write()**.

Ví dụ 8.31: Lấy lại ví dụ 8.28 nhưng lưu dữ liệu dưới dạng nhị phân.



```
1: //Chương trình 8.31
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <stdlib.h>
5:
6: typedef struct
7: {
8:     int Account;
9:     char Name[10];
10:    float Balance;
11: }Client;
12:
13: int main()
14: {
15:     ofstream OutClientFile("credit.dat", ios::out|ios::binary);
16:     if (!OutClientFile)
17:     {
18:         cerr << "File could not be opened" << endl;
19:         exit(1);
20:     }
21:     cout << "Enter the Account, Name, and Balance." << endl
22:           << "Enter EOF to end input." << endl << "? ";
23:     Client C;
24:     while (cin >> C.Account >> C.Name >> C.Balance)
25:     {
26:         OutClientFile.write((char *)&C, sizeof(Client));
27:         cout << "? ";
28:     }
29:     OutClientFile.close();
```

Chúng ta [chạy ví dụ 8.31](#), kết quả ở hình 8.33 (nội dung tập tin [credit.dat](#))



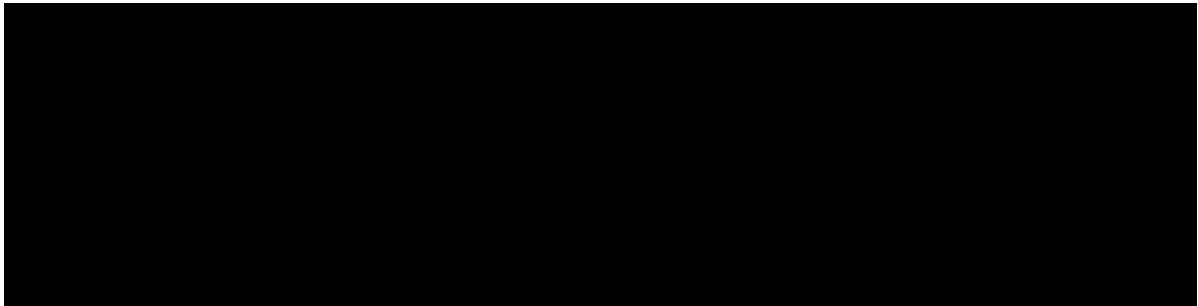
Hình 8.33: Kết quả của ví dụ 8.31

[Ví dụ 8.32](#): Đọc file tạo ở [ví dụ 8.31](#) và xuất ra màn hình.



```
1: //Chương trình 8.32
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <iomanip.h>
5: #include <stdlib.h>
6:
7: typedef struct
8: {
9:     int Account;
10:    char Name[10];
11:    float Balance;
12: }Client;
13:
14: void OutputLine(Client);
15:
16: int main()
17: {
18:     ifstream InClientFile("credit.dat", ios::in|ios::binary);
19:     if (!InClientFile)
20:     {
21:         cerr << "File could not be opened" << endl;
22:         exit(1);
23:     }
24:     cout << setiosflags(ios::left) << setw(10) << "Account"
25:          << setw(13) << "Name" << "Balance" << endl;
26:     Client C;
27:     while (InClientFile.read((char *)&C,sizeof(Client)))
28:         OutputLine(C);
29:     InClientFile.close();
```

Chúng ta [chạy ví dụ 8.32](#), kết quả ở hình 8.34



Hình 8.34: Kết quả của ví dụ 8.32

9.1 TEMPLATE

Trong phần này, chúng ta tìm hiểu về một trong các đặc tính còn lại của C++, đó là template (khuôn mẫu). Các template cho phép chúng ta để định rõ, với một đoạn mã đơn giản, một toàn bộ phạm vi của các hàm có liên quan (đa năng hóa)—gọi là các hàm template—hoặc một toàn bộ phạm vi của các lớp có liên quan—gọi là lớp template.

Chúng ta có thể viết một hàm template đơn giản cho việc sắp xếp một mảng và C++ tự động phát sinh các hàm template riêng biệt mà sẽ sắp xếp một mảng **int**, sắp xếp một mảng **float**, ...

Chúng ta có thể viết một lớp template cho lớp stack và sau đó C++ tự động phát sinh các lớp template riêng biệt như lớp stack của **int**, lớp stack của **float**,...

9.1.1 Các hàm template:

Các hàm đa năng hóa bình thường được sử dụng để thực hiện các thao tác tương tự trên các kiểu khác nhau của dữ liệu. Nếu các thao tác đồng nhất cho mỗi kiểu, điều này có thể thực hiện mạch lạc và thuận tiện hơn sử dụng các hàm template. Lập trình viên viết định nghĩa hàm template đơn giản. Dựa vào các kiểu tham số cung cấp trong lời gọi hàm này, trình biên dịch tự động phát sinh các hàm mã đối tượng riêng biệt để xử lý mỗi kiểu của lời gọi thích hợp. Trong C, điều này có thể được thực hiện bằng cách sử dụng các macro tạo với tiền xử lý **#define**. Tuy nhiên, các macro biểu thị khả năng đối với các hiệu ứng lề nghiêm trọng và không cho phép trình biên dịch thực hiện việc kiểm tra kiểu. Các hàm template cung cấp một giải pháp mạch lạc giống như các macro, nhưng cho phép kiểm tra kiểu đầy đủ. Chẳng hạn, chúng ta muốn viết hàm lấy trị tuyệt đối của một số, chúng ta có thể viết nhiều dạng khác nhau như sau:

```
int MyAbs(int X)

{

    return X>=0?X:-X;

}
```



```

long MyAbs(long X)

{

    return X>=0?X:-X;

}

double MyAbs(double X)

{

    return X>=0?X:-X;

}

```

Tuy nhiên với các hàm này chúng ta vẫn chưa có giải pháp tốt, mang tính tổng quát nhất như hàm có tham số kiểu int nhưng giá trị trả về là double và ngược lại.

Tất cả các hàm template định nghĩa bắt đầu với từ khóa **template** theo sau một danh sách các tham số hình thức với hàm template vây quanh trong các ngoặc nhọn (< và >); Mỗi tham số hình thức phải được đặt trước bởi từ khóa **class** như:

```
template <class T>
```

hoặc

```
template <class T1, class T2,...>
```

Các tham số hình thức của một định nghĩa template được sử dụng để mô tả các kiểu của các tham số cho hàm, để mô tả kiểu trả về của hàm, và để khai báo các biến bên trong hàm. Phần định nghĩa hàm theo sau và được định nghĩa giống như bất kỳ hàm nào. Chú ý từ khóa **class** sử dụng để mô tả các kiểu tham số của hàm template thực sự nghĩa là "kiểu có sẵn và kiểu người dùng định nghĩa bất kỳ".

Khi đó, hàm trị tuyệt đối ở trên viết theo hàm template:

```

template <class T>

T MyAbs(T x)

{

    return (x>=0)?x:-x;

}

```

Hàm template *MyAbs()* khai báo một tham số hình thức *T* cho kiểu của một số. *T* được tham khảo như một tham số kiểu. Khi trình biên dịch phát hiện ra một lời gọi hàm *MyAbs()* trong chương trình, kiểu của tham số thứ nhất của hàm *MyAbs()* được thay thế cho *T* thông qua định nghĩa template, và

C++ tạo một hàm template đầy đủ để trả về trị tuyệt đối của một số của kiểu dữ liệu đã cho. Sau đó, hàm mới tạo được biên dịch. Chẳng hạn:

```
cout<<MyAbs(-2)<<endl;
```

```
cout<<MyAbs(3.5)<<endl;
```

Trong lần gọi thứ nhất, hàm *MyAbs()* có tham số thực là **int** nên trình biên dịch tạo ra hàm **int** *MyAbs(int)* theo dạng của hàm template, lần thứ hai sẽ tạo ra hàm **float** *MyAbs(float)*.

Mỗi tham số hình thức trong một định nghĩa hàm template phải xuất hiện trong danh sách tham số của hàm tối thiểu một lần. Tên của tham số hình thức chỉ có thể sử dụng một lần trong danh sách tham số của phần đầu template.

■ Ví dụ 9.1: Sử dụng hàm template để in các giá trị của một mảng có kiểu bất kỳ.

```
1: //Chương trình 9.1
2: #include <iostream.h>
3:
4: template<class T>
5: void PrintArray(T *Array, const int Count)
6: {
7:     for (int I = 0; I < Count; I++)
8:         cout << Array[I] << " ";
9:
10:    cout << endl;
11: }
12:
13: int main()
14: {
15:     const int Count1 = 5, Count2 = 7, Count3 = 6;
16:     int A1[Count1] = {1, 2, 3, 4, 5};
17:     float A2[Count2] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
18:     char A3[Count3] = "HELLO";
19:     cout << "Array A1 contains:" << endl;
20:     PrintArray(A1, Count1); //Hàm template kiểu int
21:     cout << "Array A2 contains:" << endl;
22:     PrintArray(A2, Count2); //Hàm template kiểu float
23:     cout << "Array A3 contains:" << endl;
24:     PrintArray(A3, Count3); //Hàm template kiểu char
25:     return 0;
26: }
```

Chúng ta [chạy ví dụ 9.1](#), kết quả ở hình 9.1



Hình 9.1: Kết quả của ví dụ 9.1

■ [Ví dụ 9.2](#): Hàm template có thể có nhiều tham số.

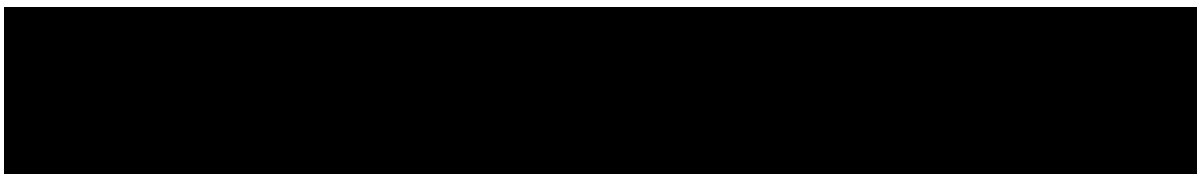


```

1: //Chương trình 9.2
2: #include <iostream.h>
3:
4: template<class T>
5: T Max(T a, T b)
6: {
7:     return (a>b)?a:b;
8: }
9:
10: int main()
11: {
12:     float A,B;
13:     cout<<"Enter first number:";
14:     cin>>A;
15:     cout<<"Enter second number:";
16:     cin>>B;
17:     cout<<"Maximum:"<<Max(A,B) ;
18:     return 0;
19: }

```

Chúng ta [chạy ví dụ 9.2](#), kết quả ở hình 9.2



Hình 9.2: Kết quả của ví dụ 9.2

Một hàm template có thể được đa năng hóa theo vài cách. Chúng ta có thể cung cấp các hàm template khác mà mô tả cùng tên hàm nhưng các tham số hàm khác nhau. Một hàm template cũng có thể được đa năng hóa bởi cung cấp hàm non-template với cùng tên hàm nhưng các tham số hàm khác

nhau. Trình biên dịch thực hiện một xử lý so sánh để xác định hàm gọi khi một hàm được gọi. Đầu tiên trình biên dịch cố gắng tìm và sử dụng một đối sánh chính xác mà các tên hàm và các kiểu tham số đối sánh chính xác. Nếu điều này thất bại, trình biên dịch kiểm tra nếu một hàm template đã có mà có thể phát sinh một hàm template với một đối sánh chính xác của tên hàm và các kiểu tham số. Nếu một hàm template như thế được tìm thấy, trình biên dịch phát sinh và sử dụng hàm template thích hợp. Chú ý xử lý đối sánh này với các template đòi yêu các đối sánh chính xác trên tất cả kiểu tham số-không có các chuyển đổi tự động được áp dụng.

9.1.2 Các lớp template:

Bên cạnh hàm template, ngôn ngữ C++ còn trang bị thêm lớp template, lớp này cũng mang đầy đủ ý tưởng của hàm template. Các lớp template được gọi là các kiểu có tham số (parameterized types) bởi vì chúng đòi hỏi một hoặc nhiều tham số để mô tả làm thế nào tùy chỉnh một lớp template chung để tạo thành một lớp template cụ thể.

Chúng ta cài đặt một lớp *Stack*, thông thường chúng ta phải định nghĩa trước một kiểu dữ liệu cho từng phần tử của stack. Nhưng điều này chỉ mang lại sự trong sáng cho một chương trình và không giải quyết được vấn đề tổng quát. Do đó chúng ta định nghĩa lớp template *Stack*.

Ví dụ 9.3:

File TSTACK.H:

```
1: //TSTACK.H
2: //Lớp template Stack
3: #ifndef TSTACK_H
4: #define TSTACK_H
5:
6: #include <iostream.h>
7:
8: template<class T>
9: class Stack
10: {
11: private:
12: int Size; //Kích thước stack
13: int Top;
14: T *StackPtr;
15: public:
16: Stack(int = 10);
17: ~Stack()
18: {
19: delete [] StackPtr;
20: }
21: int Push(const T&);
22: int Pop(T&);
23: int IsEmpty() const
24: {
25: return Top == -1;
26: }
27: int IsFull() const
28: {
29: return Top == Size - 1;
```

[REDACTED]

[REDACTED]

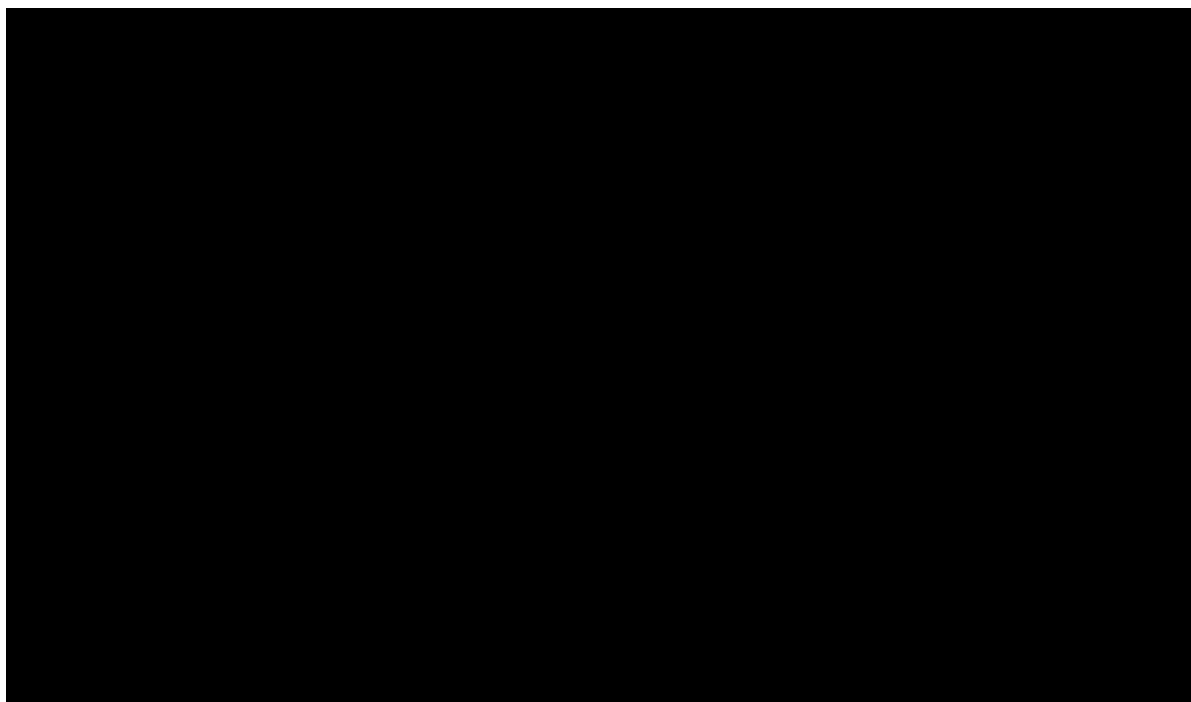
[REDACTED]

■ File CT9_3.CPP:

[REDACTED]


```
1: //CT9_3.CPP
2: //Chương trình 9.3
3: #include "tstack.h"
4:
5: int main()
6: {
7:     Stack<float> FloatStack(5);
8:     float F = 1.1;
9:     cout << "Pushing elements onto FloatStack" << endl;
10:    while (FloatStack.Push(F))
11:    {
12:        cout << F << ' ';
13:        F += 1.1;
14:    }
15:    cout << endl << "Stack is full. Cannot push " << F << endl;
16:        << endl << "Popping elements from FloatStack" <<
endl;
17:    while (FloatStack.Pop(F))
18:    {
19:        cout << F << ' ';
20:        cout << endl << "Stack is empty. Cannot pop" << endl;
21:    }
22:    Stack<int> IntStack;
23:    int I = 1;
24:    cout << endl << "Pushing elements onto IntStack" << endl;
25:    while (IntStack.Push(I))
26:    {
27:        cout << I << ' ';
28:        ++I ;
29:    }
30:    cout << endl << "Stack is full. Cannot push " << I << endl;
```

Chúng ta [chạy ví dụ 9.3](#), kết quả ở hình 9.3



Hình 9.3: Kết quả của ví dụ 9.3

Hàm thành viên định nghĩa bên ngoài lớp template bắt đầu với phần đầu là

```
template <class T>
```

Sau đó mỗi định nghĩa tương tự một định nghĩa hàm thường ngoại trừ kiểu phần tử luôn luôn được liệt kê tổng quát như tham số kiểu *T*. Chẳng hạn:

```
template<class T>  
  
int Stack<T>::Push(const T &Item)  
  
{  
  
    .....  
  
}
```

Ngôn ngữ C++ còn cho phép chúng ta tạo ra các lớp template linh động hơn bằng cách cho phép thay đổi giá trị của các thành viên dữ liệu bên trong lớp. Khi đó lớp có dạng của một hàm với tham số hình thức.

[Ví dụ 9.4:](#)

[File TSTACK.H:](#)



```
1: //TSTACK.H
2: //Lóp template Stack
3: #ifndef TSTACK_H
4: #define TSTACK_H
5:
6: #include <iostream.h>
7:
8: template<class T,int Elements>
9: class Stack
10: {
11: private:
12: int Top;
13: T StackArray[Elements];
14: public:
15: Stack();
16: int Push(const T&);
17: int Pop(T&);
18: int IsEmpty() const
19: {
20: return Top == -1;
21: }
22: int IsFull() const
23: {
24: return Top == Elements - 1;
25: }
26: };
27:
28: template<class T,int Elements>
29: Stack<T,Elements>::Stack()
```

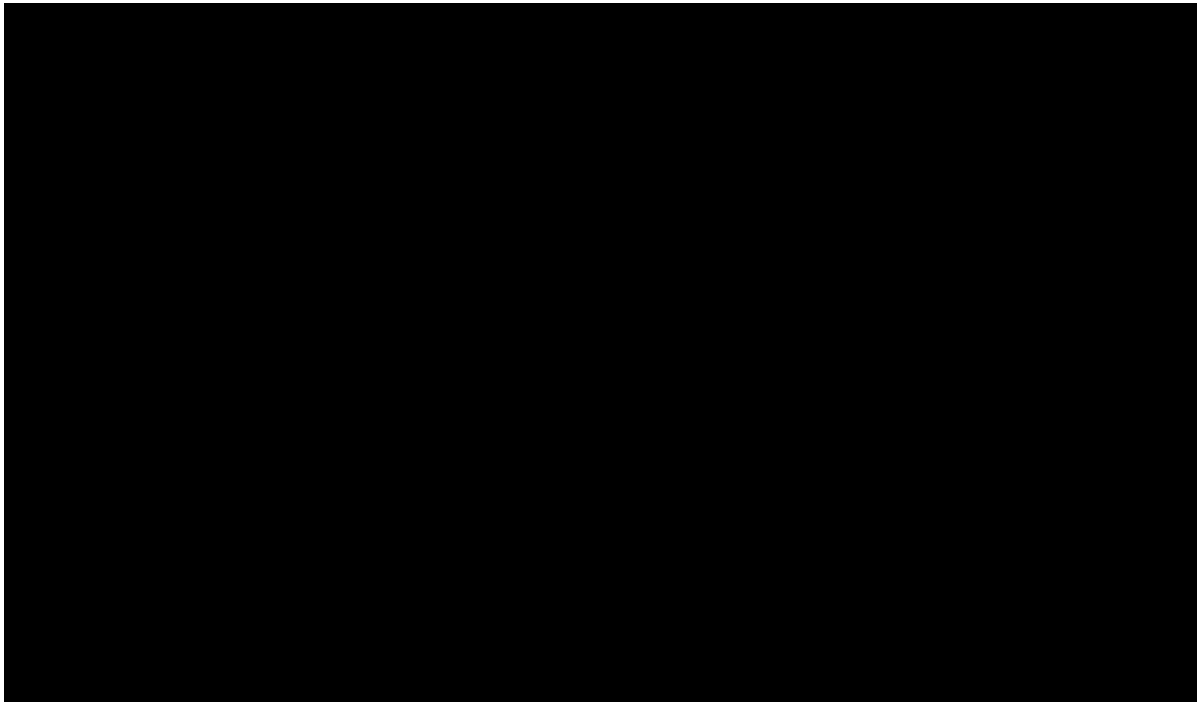
[REDACTED]

File CT9_4.CPP:

[REDACTED]

```
1: //CT9_3.CPP
2: //Chương trình 9.4
3: #include "tstack.h"
4:
5: int main()
6: {
7:     Stack<float,5> FloatStack;
8:     float F = 1.1;
9:     cout << "Pushing elements onto FloatStack" << endl;
10:    while (FloatStack.Push(F))
11:    {
12:        cout << F << ' ';
13:        F += 1.1;
14:    }
15:    cout << endl << "Stack is full. Cannot push " << F << endl;
16:        << endl << "Popping elements from FloatStack" <<
endl;
17:    while (FloatStack.Pop(F))
18:    {
19:        cout << F << ' ';
20:        cout << endl << "Stack is empty. Cannot pop" << endl;
21:        Stack<int,10> IntStack;
22:        int I = 1;
23:        cout << endl << "Pushing elements onto IntStack" << endl;
24:        while (IntStack.Push(I))
25:        {
26:            cout << I << ' ';
27:            ++I;
28:        }
29:        cout << endl << "Stack is full. Cannot push " << I << endl
```

Chúng ta [chạy ví dụ 9.4](#), kết quả ở hình 9.4



Hình 9.4: Kết quả của ví dụ 9.4

9.2 XỬ LÝ EXCEPTION (EXCEPTION HANDLING)

Xử lý exception là phương thức thống nhất của C++ giúp lập trình viên quản lý theo cùng một phong cách đối với các lỗi phát sinh bởi chương trình vào lúc chạy. Nhờ phương thức này, chúng ta có thể quy định các thủ tục cần được thực hiện một cách tự động trong từng trường hợp lỗi. Điểm quan trọng nhất mà phương thức xử lý exception là sự bảo đảm cho chương trình không chấm dứt tức thời, đột ngột hoặc nhảy tới đoạn mã thực hiện không đúng.

9.2.1 Các từ khóa **try**, **catch** và **throw**:

Xử lý exception của C++ dùng ba câu lệnh được thêm vào ngôn ngữ C++, đó là **try**, **catch** và **throw**. Ba từ khóa này tạo nên một cơ chế cho phép các chương trình thông báo với nhau về các vấn đề bất thường và nghiêm trọng xảy ra. Trong đoạn mã của chúng ta có ba nói tham gia vào việc phát hiện exception:

Khối **try** đánh dấu đoạn mã mà chúng ta nghi ngờ có thể xảy ra sai lầm khi chạy.

Khối **catch** nằm ngay liền sau khối **try**, chứa đoạn mã lo giải quyết vấn đề bất thường xảy ra gọi là exception handler. Có thể có nhiều lệnh **catch**, mỗi lệnh chuyên một loại exception. Trong mỗi khối **catch** có thể khai báo cho biết loại exception nào chúng ta muốn giải quyết.

Lệnh **throw** là làm thế nào đoạn mã có vấn đề thông báo exception handler biết vấn đề phải giải quyết.

Nếu ở bất cứ chỗ nào trong chương trình, chúng ta phát hiện một điều kiện gây ra chương trình phải thoát khỏi **try** nhảy về một exception handler, chúng ta có thể ném ra exception riêng của chúng ta bằng cách dùng lệnh **throw**. Lệnh throw sử dụng cú pháp tương tự như lệnh **return** cho phép chúng ta ném ra bất cứ biến nào đặc biệt (hoặc biểu thức gán) mà chúng ta muốn. Chú ý rằng kiểu biến chúng ta ném ra phải khớp với kiểu được khai báo trong khối **catch** đầu đó.

```
try
{
    .....
}

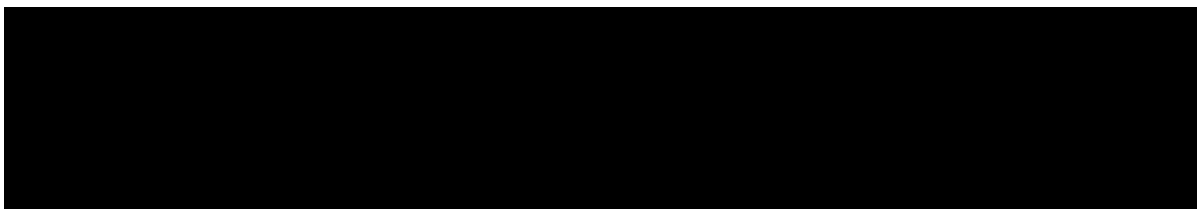
catch(<exception declaration>)
{
    .....
}

catch(<exception declaration>)
{
    .....
}
```

Ví dụ 9.5: Minh họa các từ khóa **try**, **catch** và **throw** (Các chương trình về exception phải chạy trên Borland C++ 4.0 trở lên)


```
1: //Chương trình 9.5
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout<<"Start"<<endl;
7:     try
8:     {
9:         cout<<"Inside block try"<<endl;
10:        throw 100;
11:        cout<<"This will not execute"<<endl;
12:    }
13:    catch(int I)
14:    {
15:        cout<<"Caught One! Number is "<<I<<endl;
16:    }
17:    cout<<"End"<<endl;
18:    return 0;
19: }
```

Chúng ta [chạy ví dụ 9.5](#), kết quả ở hình 9.5



Hình 9.5: Kết quả của ví dụ 9.5

Trong ví dụ 9.5, khối **try** chỉ có vài câu lệnh (dòng 7 đến 12), trong đó có lệnh:

`throw 100; //Ném ra trị 100`

trị này sẽ được bắt bởi khối **catch** ngay sau khối **try**. Chính nhờ khối này chúng ta ứng phó với lỗi xảy ra. Đây chính là exception handler. Nó chỉ in trị bắt lỗi được ra màn hình. Sau hoạt động của exception handler, dòng điều khiển của chương trình không trở về câu lệnh ở dòng 11 mà tiếp tục với câu lệnh sau khối **catch** (dòng 17).

Như vậy một sai lầm xảy ra, và một exception được ném ra trong khối **try** thì quyền điều khiển sẽ nhảy về khối **try** và bắt đầu dò tìm trên các khối **catch**, khối **catch** nào sẽ giải quyết "bệnh tình" của mình (phần còn lại của khối **try** sẽ không được thi hành).

Nếu khối **catch** nào khớp với loại exception được ném ra thì đoạn mã trong khối **catch** này sẽ được thi hành, và việc thi hành chấm dứt ở cuối khối **catch**.

Nếu không có khối **catch** nào khớp với loại exception được ném ra thì chương trình sẽ dò tìm trên call stack (stack chứa các hàm được gọi) đi tìm lần lên cấp cao các exception handler thích hợp được tìm thấy.

Nếu không có exception nào được ném ra thì các **catch** sẽ không được thi hành (Hàm **terminate()** được gọi, hàm này mặc định gọi hàm **abort()**).

Biến được khai báo trong câu lệnh **catch** có thể thuộc bất cứ kiểu dữ liệu nào chúng ta muốn ném ra. Đây có thể là kiểu dữ liệu đơn giản (**int**, **char** *, ...) hoặc bất cứ lớp nào chúng ta đã định nghĩa. Ngoài ra chúng ta có thể sử dụng ba dấu chấm (...) như là kiểu exception trong khối **catch** để xử lý bất kỳ kiểu của exception nào và chỉ dùng ở cuối đối với một khối **try**. Exception handler sử dụng ba dấu chấm:

catch(...)

{

.....

}

và được gọi là ellipsis handler.

Chú ý:

Bên trong exception handler chương trình có thể tiếp tục hoặc kết thúc bằng cách gọi hàm **exit()** hoặc **abort()**.

Nếu bên trong khối **try** có đối tượng được tạo ra thì khi chương trình bị đổi hướng do lệnh **throw** thì đối tượng sẽ bị hủy trước khi chương trình bị đổi hướng.

9.2.2 Các tình huống xử lý exception:

- Khi một exception không bị bắt

Nếu một exception được ném ra trong trường hợp sau khối **try** không có exception handler không khớp thì hàm **terminate()** được gọi nhằm kết thúc chương trình.

■ [Ví dụ 9.6:](#)

```
CT9_6.CPP
1: //Chương trình 9.6
2: #include <iostream.h>
3:
4: int main()
5: {
6:     cout<<"Start"<<endl;
7:     try
8:     {
9:         cout<<"Inside try block"<<endl;
10:        throw 100;
11:        cout<<"This will not execute"<<endl;
12:    }
13:    catch(long I)
14:    {
15:        cout<<"Caught One! Number is "<<I<<endl;
16:    }
17:    cout<<"End"<<endl;
18:    return 0;
19: }
```

Chúng ta [chạy ví dụ 9.6](#), kết quả ở hình 9.6



Hình 9.6: Kết quả của ví dụ 9.6

□ Ném một exception trong hàm bất kỳ

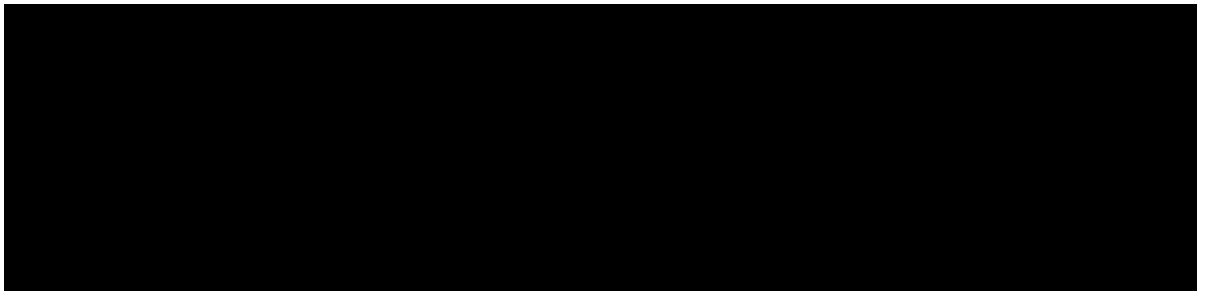
Chúng ta có thể ném một exception từ trong bất kỳ hàm nào miễn là hàm đó sẽ được gọi trong khối **try**.

■ Ví dụ 9.7:



```
1: //Chương trình 9.7
2: #include <iostream.h>
3:
4: void MyTest(int Test)
5: {
6:     cout<<"Inside MyTest(), Test is "<<Test<<endl;
7:     if (Test)
8:         throw Test;
9: }
10:
11: int main()
12: {
13:     cout<<"Start"<<endl;
14:     try
15:     {
16:         cout<<"Inside try block"<<endl;
17:         MyTest(0);
18:         MyTest(1);
19:         MyTest(2);
20:     }
21:     catch(int I)
22:     {
23:         cout<<"Caught One! Number is "<<I<<endl;
24:     }
25:     cout<<"End"<<endl;
26:     return 0;
27: }
```

Chúng ta [chạy ví dụ 9.7](#), kết quả ở hình 9.7



Hình 9.7: Kết quả của ví dụ 9.7

Trị được ném ra không phải là trị trả về của hàm và có thể thuộc kiểu bất kỳ tùy chúng ta chọn (không liên quan gì đến kiểu trả về của hàm). Chú ý rằng lệnh **return** phục hồi stack về trạng thái trước khi gọi hàm, trong khi lệnh **throw** tái lập stack như trước khi vào khối **try**.

□ [Khối **try** bên trong hàm](#)

Chúng ta có thể đặt khối **try** bên trong hàm bất kỳ để xử lý exception lúc chạy sinh bởi đoạn mã nào đó thuộc hàm.

■ [Ví dụ 9.8:](#)



```
1: //Chương trình 9.8
2: #include <iostream.h>
3:
4: void MyHandler(int Test)
5: {
6: try
7: {
8: if (Test)
9: throw Test;
10: }
11: catch(int I)
12: {
13: cout<<"Caught One! Ex. #"<<I<<endl;
14: }
15: }
16:
17: int main()
18: {
19: cout<<"Start"<<endl;
20: MyHandler(1);
21: MyHandler(2);
22: MyHandler(0);
23: MyHandler(3);
24: cout<<"End"<<endl;
25: return 0;
26: }
```

Chúng ta [chạy ví dụ 9.8](#), kết quả ở hình 9.8



Hình 9.8: Kết quả của ví dụ 9.8

□ [Nhiều khối **catch**](#)

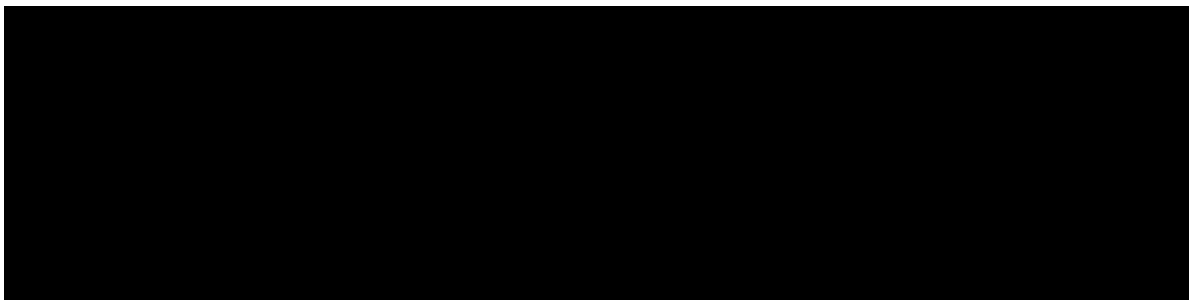
Chúng ta có thể tạo nhiều exception handler. Mỗi exception handler có nhiệm vụ bắt một exception thuộc kiểu nào đó.

■ [Ví dụ 9.9:](#)




```
1: //Chương trình 9.9
2: #include <iostream.h>
3:
4: void MyHandler(int Test)
5: {
6: try
7: {
8: if (Test)
9: throw Test;
10: else
11: throw "Value is zero";
12: }
13: catch(int I)
14: {
15: cout<<"Caught One! Ex. #"<<I<<endl;
16: }
17: catch(char *Str)
18: {
19: cout<<"Caught a string:"<<Str<<endl;
20: }
21: }
22:
23: int main()
24: {
25: cout<<"Start"<<endl;
26: MyHandler(1);
27: MyHandler(2);
28: MyHandler(0);
29: MyHandler(3);
```

Chúng ta [chạy ví dụ 9.9](#), kết quả ở hình 9.9



Hình 9.9: Kết quả của ví dụ 9.9

□ [Bắt exception có kiểu bất kỳ](#)

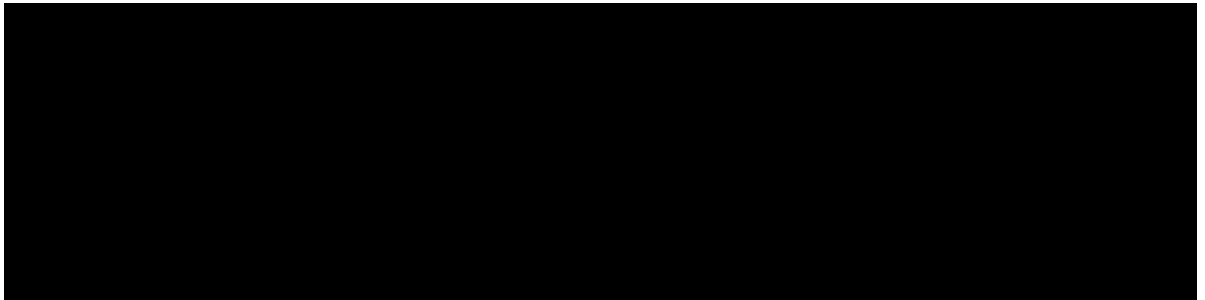
Chúng ta có thể tạo ra một exception handler có khả năng bắt exception có kiểu bất kỳ, đây chính là ellipsis handler.

■ [Ví dụ 9.10:](#)



```
1: //Chương trình 9.10
2: #include <iostream.h>
3:
4: void MyHandler(int Test)
5: {
6: try
7: {
8: if (Test==0)
9: throw Test;
10: if (Test==1)
11: throw 'a';
12: if (Test==2)
13: throw 123.34;
14: if (Test==3)
15: throw "Test";
16: }
17: catch(int)
18: {
19: cout<<"Caught int"<<endl;
20: }
21: catch(double)
22: {
23: cout<<"Caught double"<<endl;
24: }
25: catch(...)
26: {
27: cout<<"Caught One"<<endl;
28: }
29: }
```

Chúng ta [chạy ví dụ 9.10](#), kết quả ở hình 9.10



Hình 9.10: Kết quả của ví dụ 9.10

□ [Nén exception từ một exception handler](#)

C++ cho phép một exception handler ném một exception mà nó bắt được. Muốn vậy, trong exception handler chúng ta chỉ ghi câu lệnh:

throw;

Bằng cách như vậy, một exception handler chuyển exception mà nó bắt được cho khối **try** khác chứa nó.

■ [Ví dụ 9.11:](#)



```
1: //Chương trình 9.11
2: #include <iostream.h>
3:
4: void MyHandler()
5: {
6: try
7: {
8: throw "Hello!";
9: }
10: catch(char *)
11: {
12: cout<<"Caught inside MyHandler()"<<endl;
13: throw; //Ném tiếp ra ngoài
14: }
15: }
16:
17: int main()
18: {
19: cout<<"Start"<<endl;
20: try
21: {
22: MyHandler();
23: }
24: catch(char *)
25: {
26: cout<<"Caught inside main()"<<endl;
27: }
28: cout<<"End"<<endl;
29: return 0;
```

Chúng ta [chạy ví dụ 9.11](#), kết quả ở hình 9.11



Hình 9.11: Kết quả của ví dụ 9.11

Trong chương trình ở ví dụ 9.11, khối **catch(char *)** trong hàm *MyHandler()* bắt được một chuỗi và ném tiếp ra ngoài. Do vậy, khi hàm *MyHandler()* được gọi bên trong khối **try** ở hàm **main()**, khối **catch(char *)** trong hàm **main()** lại bắt được chuỗi đó.

Với cơ chế này, chúng ta có thể tạo ra khả năng xử lý exception theo nhiều cấp bằng cấu trúc **try/catch** lồng nhau. Những gì cấp dưới (**catch** bên trong) không giải quyết thì chuyển lên cấp trên (**catch** bên ngoài) để dứt điểm.

□ [Hạn chế về kiểu của một exception](#)

Như đã nói, exception được ném ra từ trong một hàm không có liên quan gì đến trị trả về của hàm đó. Exception có thể thuộc kiểu tùy ý. Tuy nhiên, nếu cần chúng ta cũng có thể quy định kiểu của exception. Điều này gọi là đặc tả exception (exception specification). Để thực hiện hạn chế, chúng ta phải thêm **throw** vào định nghĩa hàm. Dạng tổng quát:

<return type> <function name> (<parameter list>) throw (<type list>)

{

.....

}

Trong đó chỉ có các exception có kiểu trong danh sách *<type list>* (phân cách bằng dấu phẩy) là được ném ra từ hàm. Ngược lại, việc ném ra các exception thuộc kiểu khác là chương trình kết thúc bất thường (gọi hàm **unexpected()**, mặc định hàm này gọi hàm **abort()**).

Nếu *<type list>* rỗng thì cấm ném bất kỳ cái gì ra bên ngoài. Nếu hàm đang xét cố tình vi phạm thì chương trình sẽ kết thúc ngay bởi hàm **unexpected()**.

■ [Ví dụ 9.12:](#)



```
1: //Chương trình 9.12
2: #include <iostream.h>
3:
4: void MyHandler(int Test) throw (int,char,double)
5: {
6: if (Test==0)
7: throw Test;
8: if (Test==1)
9: throw 'a';
10: if (Test==2)
11: throw 123.34;
12: }
13:
14: int main()
15: {
16: cout<<"Start"<<endl;
17: try
18: {
19: MyHandler(0);
20: }
21: catch(int)
22: {
23: cout<<"Caught int"<<endl;
24: }
25: catch(double)
26: {
27: cout<<"Caught double"<<endl;
28: }
29: catch(char)
```

Chúng ta [chạy ví dụ 9.12](#), kết quả ở hình 9.12



Hình 9.12: Kết quả của ví dụ 9.12

Chú ý rằng quy định vừa nêu có hiệu lực đối với exception ném ra khỏi hàm tức là trong trường hợp hàm được gọi bên trong một khối **try**. Đối với trường hợp khối **try** đặt bên trong, exception được ném và bắt bên trong hàm vẫn không bị hạn chế.

9.2.3 Ví dụ:

■ [Ví dụ 9.13](#): Chương trình sau là một ví dụ về xử lý exception để kiểm tra lỗi "chia cho zero".




```
1: //Chương trình 9.13

2: #include <iostream.h>

3: #include <string.h>

4:

5: //Định nghĩa một lớp DivideByZeroError để sử dụng trong xử lý
exception

6: //cho việc ném một exception với một phép chia cho zero.

7: class DivideByZeroError

8: {

9: private:

10: char Message[15];

11: public:

12: DivideByZeroError()

13: {

14: strcpy(Message, "Divide by zero");

15: }

16: void PrintMessage() const

17: {

18: cout << Message;

19: }

20: };

21:

22: //Định nghĩa hàm thương số. Được sử dụng để minh họa ném

23: //một exception khi lỗi chia cho zero bắt gặp

24: float Quotient(int Num1, int Num2)

25: {

26: if (Num2 == 0)

27: throw DivideByZeroError();

28: return (float) Num1 / Num2;
```

Chúng ta [chạy ví dụ 9.13](#), kết quả ở hình 9.13



Hình 9.13: Kết quả của ví dụ 9.13

Toàn bộ các ví dụ về exception (từ ví dụ 9.5 đến 9.13) được đưa vào một [project của Borland C++ 4.0](#) với tên file nén là [Project_BC4.zip](#) hoặc [workspace trong Visual C++ 6.0](#) với tên file nén là [Project_VC6.zip](#).

**ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG CAO ĐẲNG CÔNG NGHỆ THÔNG TIN**

BÀI GIẢNG

**LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG
VỚI C++**

ĐINH THỊ MỸ HẠNH

ĐÀ NẴNG - 2009

MỤC LỤC

MỤC LỤC.....	3
TỔNG QUAN VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG	6

I.	PHƯƠNG PHÁP LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG	6
I.1.	Vì sao lập trình hướng đối tượng?.....	6
I.2.	Bài toán quan hệ gia đình	6
II.	MỘT SỐ KHÁI NIỆM TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG.....	9
II.1.	Lập trình hướng đối tượng	9
II.2.	Trừu tượng hoá	10
II.3.	Đối tượng và Lớp	10
II.4.	Thuộc tính và Phương thức	10
II.5.	Nguyên tắc đóng gói dữ liệu	11
II.6.	Tính kế thừa (inheritance).....	11
II.7.	Tính đa hình (polymorphime)	12
III.	NHỮNG ĐẶC ĐIỂM CỦA LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG	13
III.1.	Những đặc tính chủ yếu của LTHĐT	13
III.2.	Ưu điểm của LTHĐT.....	14
IV.	NGÔN NGỮ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG	14
	NHỮNG MỞ RỘNG CỦA C++ SO VỚI C.....	16
I.	LỊCH SỬ NGÔN NGỮ LẬP TRÌNH C++	16
II.	NHỮNG ĐIỂM KHÔNG TƯƠNG THÍCH VỚI C.....	16
II.1.	Định nghĩa hàm	16
II.2.	Sự tương thích giữa con trỏ void và các con trỏ khác.....	17
III.	CÁC MỞ RỘNG CỦA C++.....	18
III.1.	Các khả năng vào/ra mới của C++	18
III.2.	Chỉ thích cuối dòng.....	19
III.3.	Khai báo mọi nơi	19
III.4.	Các biến kiểu const.....	20
III.5.	Về struct, union và enum.....	20
III.6.	Toán tử phạm vi “::”	21
III.7.	Hàm nội tuyến.....	21
III.8.	Giá trị tham số ngầm định.....	22
III.9.	Toán tử new và delete	23
IV.	PHÉP THAM CHIẾU.....	25
IV.1.	Biến tham chiếu	25
IV.2.	Truyền tham số cho hàm bằng tham chiếu.....	26
IV.3.	Giá trị trả về của hàm là tham chiếu	27
V.	PHÉP ĐA NĂNG HOÁ.....	28
V.1.	Đa năng hóa các hàm (Functions overloading)	28
V.2.	Đa năng hóa các toán tử (Operators overloading)	30
	ĐỐI TƯỢNG VÀ LỚP	33
I.	KHÁI NIỆM ĐỐI TƯỢNG VÀ LỚP	33
II.	CÀI ĐẶT MỘT LỚP	34
II.1.	Khai báo lớp	34
II.2.	Tạo đối tượng	36
III.	PHẠM VI LỚP VÀ TRUY CẬP CÁC THÀNH VIÊN LỚP	37
III.1.	Phạm vi lớp.....	37
III.2.	Điều khiển truy cập đến các thành viên.....	38
III.3.	Con trỏ this.....	40
IV.	CÁC HÀM TRUY CẬP VÀ CÁC HÀM TIỆN ÍCH.....	40
V.	THIẾT LẬP VÀ HỦY BỎ ĐỐI TƯỢNG	42
V.1.	Hàm thiết lập - CONSTRUCTOR	42
V.2.	Hàm hủy bỏ - DESTRUCTOR	44
V.3.	Hàm thiết lập sao chép (copy constructor)	45
VI.	ĐỐI TƯỢNG HẲNG VÀ CÁC HÀM THÀNH VIÊN CONST.....	47
VI.1.	Đối tượng hằng.....	47
VI.2.	Các hàm thành viên const	48
VII.	LỚP NHƯ LÀ CÁC THÀNH VIÊN CỦA CÁC LỚP KHÁC	48
VIII.	CÁC THÀNH PHẦN TĨNH CỦA LỚP	50
VIII.1.	Thành phần dữ liệu tĩnh.....	50
VIII.2.	Các hàm thành phần tĩnh.....	51

Lập trình hướng đối tượng C++	5
IX. CÁC HÀM VÀ CÁC LỚP BẠN	53
IX.1. Hàm tự do bạn của một lớp	53
IX.2. Hàm thành phần của lớp là bạn của lớp khác	53
IX.3. Tất cả các hàm của lớp là bạn của lớp khác	54
ĐA NĂNG HOÁ TOÁN TỬ	ERROR! BOOKMARK NOT DEFINED.
I. BÀI TOÁN SẮP XẾP	ERROR! BOOKMARK NOT DEFINED.
KỸ THUẬT THỪA KẾ	74
I. SỰ DẪN XUẤT VÀ TÍNH KẾ THỪA	74
I.1. Lớp cơ sở và lớp dẫn xuất	74
I.2. Đơn thừa kế	75
I.3. Tính thừa kế trong lớp dẫn xuất	79
I.4. Hàm thiết lập và hàm hủy trong lớp dẫn xuất	82
I.5. Các từ khóa quy định phạm vi truy nhập của lớp cơ sở	89
I.6. Các kiểu dẫn xuất khác nhau	89
Bảng tổng kết các kiểu dẫn xuất	90
II. ĐA THỪA KẾ	91
II.1. Đặt vấn đề	91
II.2. Danh sách móc nối các đối tượng	98
III. HÀM ẢO VÀ TÍNH ĐA HÌNH	106
III.1. Đặt vấn đề	106
III.2. Tổng quát về hàm ảo	112
KHUÔN HÌNH	125
I. MỘT SỐ KHÁI NIỆM	125
TÀI LIỆU THAM KHẢO	126

CHƯƠNG 1

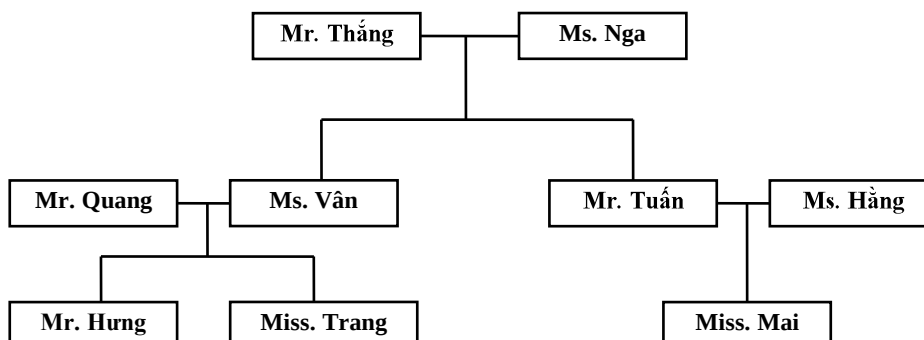
TỔNG QUAN VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

I. PHƯƠNG PHÁP LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

I.1. Vì sao lập trình hướng đối tượng?

Như ta đã biết phương pháp lập trình **cấu trúc** (TLCT), tiếp cận theo quan điểm *Chương trình = Cấu trúc dữ liệu + Giải thuật*. 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 nhận được đủ đơn giản. 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. Khi sử dụng phương pháp lập trình này còn gặp một khó khăn lớn là tổ chức dữ liệu của hệ thống như thế nào trong máy tính, 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. Đặc biệt là với những dự án lớn, chương trình được giao cho nhiều nhóm cùng phát triển, những vấn đề không đồng bộ dữ liệu, viết lại mã chương trình khi có sự điều chỉnh hoặc thay đổi, điều đó thật sự phức tạp.

Phương pháp **lập trình hướng đối tượng** (LTHĐT) ra đời, tiếp cận theo quan điểm 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 LTHĐT, ta khảo sát một bài toán cụ thể về quan hệ gia đình như biểu diễn ở hình 1.1.



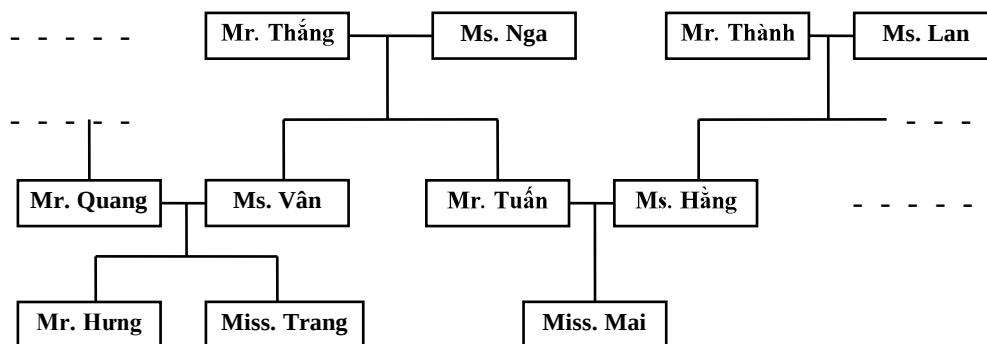
Hình 1.1 Cây quan hệ trong một gia đình

I.2. Bài toán quan hệ gia đình

Yêu cầu của bài toán là 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. Thông thường, để thể hiện các mối quan hệ này người ta biểu diễn bằng một sơ đồ cây quan hệ như ở hình 1.1.

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. Tiếp theo đó là 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. Vấn đề quan hệ sẽ càng phức tạp khi 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

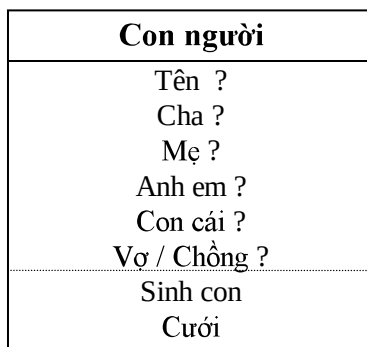
Theo cách tiếp cận LTHĐT, bài toán quan hệ gia đình được xem xét dưới góc độ quản lý tập các đối tượng Con người. Để biết mối quan hệ gia đình của mỗi cá thể, cần thể hiện một số quan hệ cơ bản như cha, mẹ, anh em, con cái, vợ chồng của cá thể đó. Như vậy, mỗi đối tượng con người của bài toán có các 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

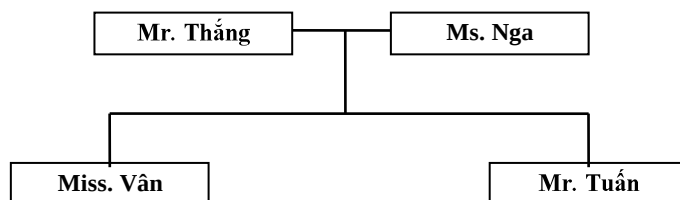
Nếu chỉ có như vậy thì chẳng khác gì một cấu trúc hay bản ghi trong cấu trúc dữ liệu được sử dụng ở phương pháp LTCT. Vấn đề ở đây là phương pháp LTHĐT xem các mối quan hệ trong gia đình được hình thành một cách tự nhiên do các sự kiện cụ thể trong cuộc sống tạo nên. Ví dụ, khi người phụ nữ sinh con, 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. Dễ dàng thấy rằng có hai sự kiện chính tác động đến mỗi 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 đồng 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.



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

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 Nga.



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

Thắng.Cưới (Nga)

Nga.Sinh con (gái, Vân)

Nga.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.

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.. Nếu đứng trên đối tượng X thì câu hỏi sẽ là: “Đối tượng có phải là anh của Y không?”, “Đối tượng 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.

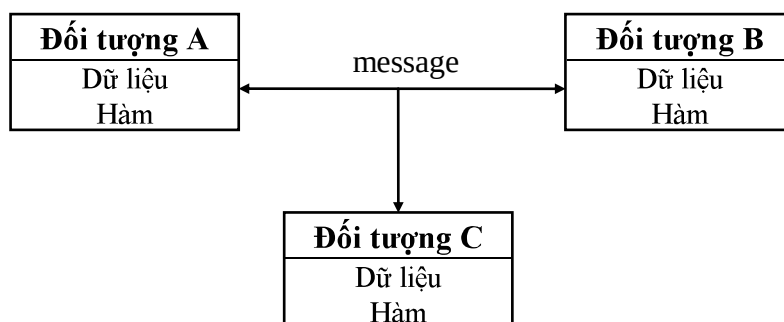


Ta xem xét các đối tượng trả lời các câu hỏi như thế nào? Chẳng hạn X trả lời câu hỏi “Đối tượng có phải là anh của Y không ?” hoàn toàn đơn giản. Nó chỉ cần kiểm tra xem Y có phải là anh em mà trong thuộc tính của nó lưu giữ không. Hoàn toàn tương tự đối với các câu hỏi quan hệ gần như là em, là chị,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

II. MỘT SỐ KHÁI NIỆM TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

II.1. 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 tùy 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

- ❖ **Lập trình hướng đối tượng (OOP)** là một phương pháp thiết kế và phát triển phần mềm dựa trên kiến trúc lớp và đối tượng.

II.2. Trừu tượng hoá

Trừu tượng hóa là một kỹ thuật chỉ trình bày những các đặc điểm cần thiết của vấn đề mà không trình bày những chi tiết cụ thể hay những lời giải thích phức tạp của vấn đề đó. Hay nói khác hơn nó là một kỹ thuật tập trung vào thứ cần thiết và phớt lờ đi những thứ không cần thiết.

Ví dụ những thông tin sau đây là các đặc tính gắn kết với con người:

- Tên
- Tuổi
- Địa chỉ
- Chiều cao
- Màu tóc

Giả sử ta cần phát triển ứng dụng khách hàng mua sắm hàng hóa thì những chi tiết thiết yếu là tên, địa chỉ còn những chi tiết khác (tuổi, chiều cao, màu tóc, ..) là không quan trọng đối với ứng dụng. Tuy nhiên, nếu chúng ta phát triển một ứng dụng hỗ trợ cho việc điều tra tội phạm thì những thông tin như chiều cao và màu tóc là thiết yếu.

Trong OOP, sự trừu tượng hóa được phân thành sự trừu tượng hóa dữ liệu và trừu tượng hóa chương trình.

- ❖ **Trừu tượng hóa dữ liệu (data abstraction)** là tiến trình xác định và nhóm các thuộc tính, các hành động liên quan đến một thực thể đặc thù trong ứng dụng đang phát triển.
- ❖ **Trừu tượng hóa chương trình (program abstraction)** là một sự trừu tượng hóa dữ liệu mà làm cho các dịch vụ thay đổi theo dữ liệu.

II.3. Đối tượng và Lớp

Đối tượng (Object) 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:

Đối tượng = Dữ liệu + Phương thức

Lớp (Class) 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.

II.4. Thuộc tính và Phương thức

Các thuộc tính trình bày trạng thái của đối tượng. Các thuộc tính nắm giữ các giá trị dữ liệu trong một đối tượng, chúng định nghĩa một đối tượng đặc thù.

Thuộc tính (attribute) là dữ liệu trình bày các đặc điểm về một đối tượng.

Các phương thức thực thi các hoạt động của đối tượng. Các phương thức là nhân tố làm thay đổi các thuộc tính của đối tượng. Các phương thức xác định cách thức hoạt động của một đối tượng và được thực thi khi đối tượng cụ thể được tạo ra.

***Phương thức (method)** có liên quan tới những thứ mà đối tượng có thể làm. Một phương thức đáp ứng một chức năng tác động lên dữ liệu của đối tượng (thuộc tính).*

Các phương thức định nghĩa trong một lớp có thể được gọi bởi các đối tượng của lớp đó. Điều này được gọi là gửi một **thông điệp** (Message) cho đối tượng. Các thông điệp này phụ thuộc vào đối tượng, chỉ đối tượng nào nhận thông điệp mới phải làm việc theo thông điệp đó. Các đối tượng đều độc lập với nhau vì vậy các thay đổi trên các biến thể hiện của đối tượng này không ảnh hưởng gì trên các biến thể hiện của các đối tượng khác và việc gửi thông điệp cho một đối tượng này không ảnh hưởng gì đến các đối tượng khác.

II.5. 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++, vẫn có những ngoại lệ nhất định vi phạm nguyên tắc đóng gói dữ liệu.

II.6. 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. Qua cơ cấu kế thừa này, dạng hình cây của các lớp được hình thành. Dạng cây của các lớp trông giống như các cây gia phả vì thế các lớp cơ sở còn được gọi là lớp **cha** (**parent class / superclass**) và các lớp dẫn xuất được gọi là lớp **con** (**child class / subclass**).

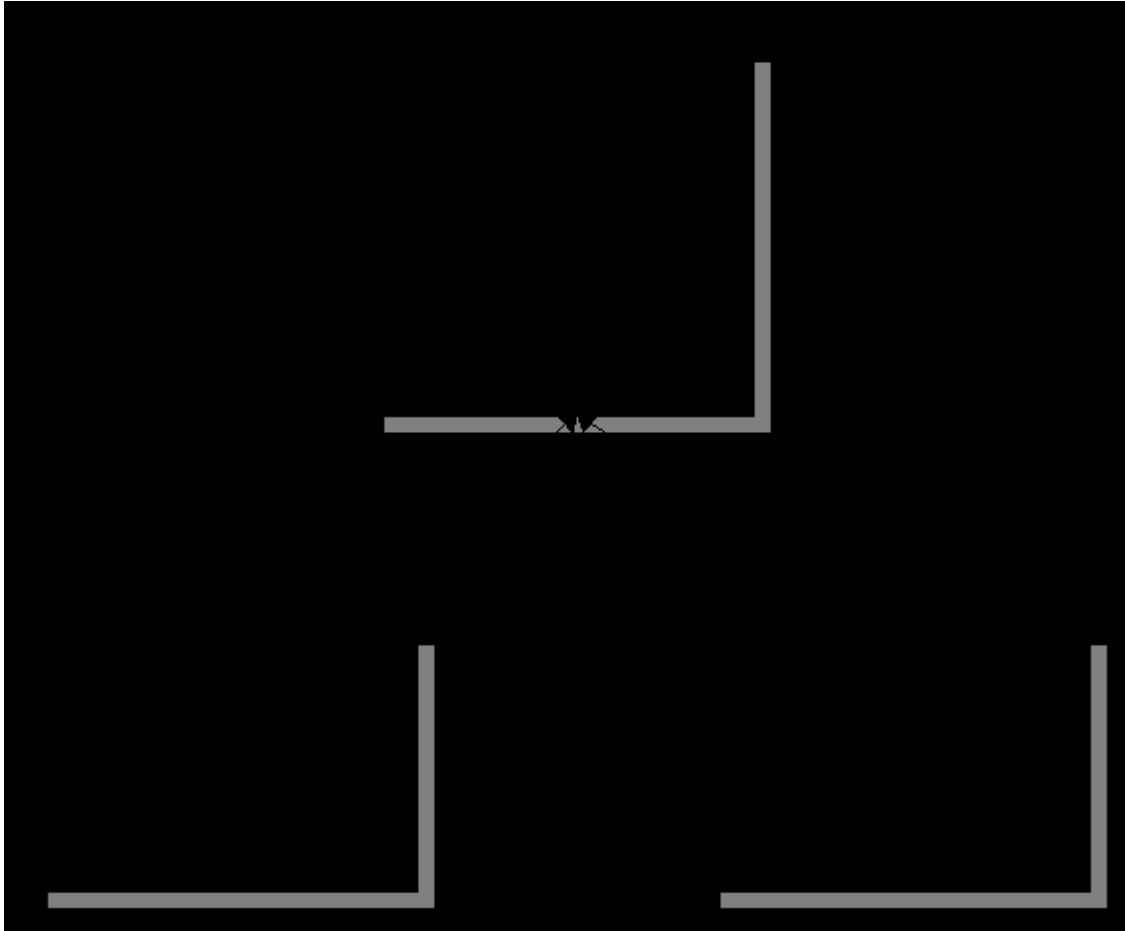
***Thừa kế (inheritance)** nghĩa là các hành động (phương thức) và các thuộc tính được định nghĩa trong một lớp có thể được thừa kế hoặc được sử dụng lại bởi lớp khác.*

***Lớp cha (superclass)** là lớp có các thuộc tính hay hành động được thừa hưởng bởi một hay nhiều lớp khác.*

***Lớp con (subclass)** là lớp thừa hưởng một vài đặc tính chung của lớp cha và thêm vào những đặc tính riêng khác.*

Ví dụ 1.1: Chúng ta sẽ xây dựng một tập các lớp mô tả cho thư viện các ấn phẩm (xem hình 1.7). Có hai kiểu ấn phẩm: tạp chí và sách. Chúng ta có thể tạo một ấn phẩm tổng quát bằng cách định nghĩa các thành phần dữ liệu tương ứng với số trang, mã số tra cứu, ngày tháng xuất

bản, bản quyền và nhà xuất bản. Các ấn phẩm có thể được lấy ra, cất đi và đọc. Đó là các phương thức thực hiện trên một ấn phẩm. Tiếp đó chúng ta định nghĩa hai lớp dẫn xuất tên là tạp chí và sách. Tạp chí có tên, số ký phát hành và chứa nhiều bài của các tác giả khác nhau. Các thành phần dữ liệu tương ứng với các yếu tố này được đặt vào định nghĩa của lớp tạp chí. Tạp chí cũng cần có một phương thức nữa đó là đặt mua. Các thành phần dữ liệu xác định cho sách sẽ bao gồm tên của (các) tác giả, loại bìa (cứng hay mềm) và số hiệu ISBN của nó. Như vậy chúng ta có thể thấy, sách và tạp chí có chung các đặc trưng ấn phẩm, trong khi vẫn có các thuộc tính riêng.



Hình 1.7: Lớp ấn phẩm và các lớp dẫn xuất của nó.

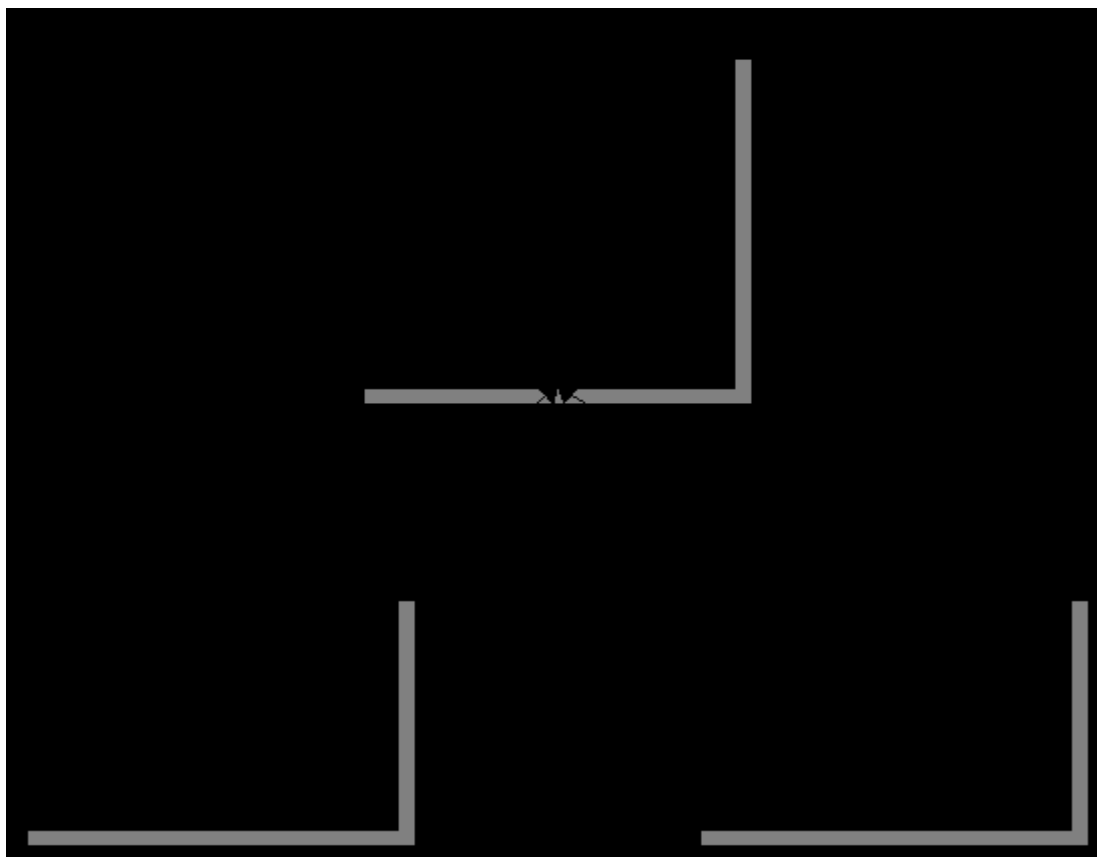
II.7. Tính đa hình (polymorphime)

Tính đa hình xuất hiện khi có khái niệm kế thừa. Khi một lớp dẫn xuất được tạo ra, nó có thể thay đổi cách thực hiện các phương thức nào đó mà nó thừa hưởng từ lớp cơ sở của nó. Một thông điệp khi được gửi đến một đối tượng của lớp cơ sở, sẽ dùng phương thức đã định nghĩa cho nó trong lớp cơ sở. Nếu một lớp dẫn xuất định nghĩa lại một phương thức thừa hưởng từ lớp cơ sở của nó thì một thông điệp có cùng tên với phương thức này, khi được gửi tới một đối tượng của lớp dẫn xuất sẽ gọi phương thức đã định nghĩa cho lớp dẫn xuất.

Đa hình (polymorphism) nghĩa là “nhiều hình thức”, hành động cùng tên có thể được thực hiện khác nhau đối với các đối tượng/các lớp khác nhau.

Ví dụ 1.2: Xét lại ví dụ 1.1, chúng ta thấy rằng cả tạp chí và sách đều phải có khả năng lấy ra. Tuy nhiên phương pháp lấy ra cho tạp chí có khác so với phương pháp lấy ra cho sách,

mặc dù kết quả cuối cùng giống nhau. Khi phải lấy ra tạp chí, thì phải sử dụng phương pháp lấy ra riêng cho tạp chí (dựa trên một bản tra cứu) nhưng khi lấy ra sách thì lại phải sử dụng phương pháp lấy ra riêng cho sách (dựa trên hệ thống phiếu lưu trữ). Tính đa hình cho phép chúng ta xác định một phương thức để lấy ra một tạp chí hay một cuốn sách. Khi lấy ra một tạp chí nó sẽ dùng phương thức lấy ra dành riêng cho tạp chí, còn khi lấy ra một cuốn sách thì nó sử dụng phương thức lấy ra tương ứng với sách. Kết quả là chỉ cần một tên phương thức duy nhất được dùng cho cả hai công việc tiến hành trên hai lớp dẫn xuất có liên quan, mặc dù việc thực hiện của phương thức đó thay đổi tùy theo từng lớp. (xem mô tả lớp ở hình 1.8)



Hình 1.8: Minh họa tính đa hình đối với lớp ẩn phẩm và các lớp dẫn xuất của nó.

Tính đa hình dựa trên sự nối kết (Binding), đó là quá trình gắn một phương thức với một hàm thực sự. Khi các phương thức kiểu đa hình được sử dụng thì trình biên dịch chưa thể xác định hàm nào tương ứng với phương thức nào sẽ được gọi. Hàm cụ thể được gọi sẽ tùy thuộc vào việc phần tử nhận thông điệp lúc đó là thuộc lớp nào, do đó hàm được gọi chỉ xác định được vào lúc chương trình chạy. Điều này gọi là sự kết nối muộn (Late binding) hay kết nối lúc chạy (Runtime binding) vì nó xảy ra khi chương trình đang thực hiện.

III. NHỮNG ĐẶC ĐIỂM CỦA LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

III.1. Những đặc tính chủ yếu của LTHĐT

- Tập trung vào dữ liệu thay cho các hàm
- Chương trình được chia thành các đối tượng.
- Các cấu trúc dữ liệu được thiết kế sao cho đặc tả được đối tượng.
- Các hàm thao tác trên các vùng dữ liệu của đối tượng được gắn với cấu trúc dữ liệu đó.

- Dữ liệu được đóng gói lại, được che giấu và không cho phép các hàm ngoại lai truy nhập tự do.
- Các đối tượng tác động và trao đổi thông tin với nhau qua các hàm
- Có thể dễ dàng bổ sung dữ liệu và các hàm mới vào đối tượng nào đó khi cần thiết
- Chương trình được thiết kế theo cách tiếp cận từ dưới lên (bottom-up).

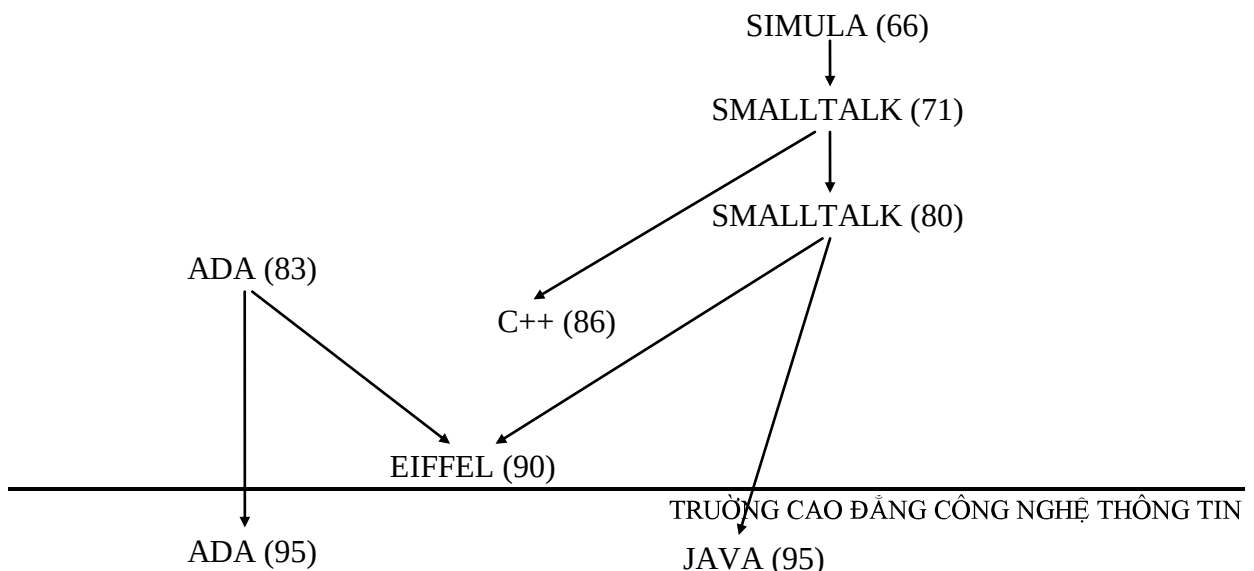
III.2. Ưu điểm của LTHĐT

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. Những ưu điểm chính của LTHĐT là:

- 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.
- Chương trình được xây dựng từ những đơn thể (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.
- 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.
- Có thể xây dựng được ánh xạ các đối tượng của bài toán vào đối tượng chương trình.
- 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.
- Các hệ thống hướng đối tượng dễ mở rộng, nâng cấp thành những hệ lớn hơn.
- 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.
- Có thể quản lý được độ phức tạp của những sản phẩm phần mềm.

IV. NGÔN NGỮ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

LTHĐT không phải là đặc quyền của một ngôn ngữ đặc biệt nào. 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 1.9 cho chúng ta một cái nhìn tổng quan về sự phát triển các ngôn ngữ LTHĐT.



Hình 1.9 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, sử dụng cú pháp của ngôn ngữ C. Trên C++ vẫn có thể sử dụng tính cấu trúc và đối tượng của chương trình.

CHƯƠNG 2

NHỮNG MỞ RỘNG CỦA C++ SO VỚI C

I. LỊCH SỬ NGÔN NGỮ LẬP TRÌNH C++

Vào những năm đầu thập niên 1980, người dùng biết C++ với tên gọi "C with Classes" được mô tả trong hai bài báo của Bjarne Stroustrup (thuộc AT&T Bell Laboratories) với nhan đề "Classes: An Abstract Data Type Facility for the C Language" và "Adding Classes to C : An Exercise in Language Evolution". Trong công trình này, tác giả đã đề xuất khái niệm lớp, bổ sung việc kiểm tra kiểu tham số của hàm, các chuyển đổi kiểu và một số mở rộng khác vào ngôn ngữ C. Bjarne Stroustrup nghiên cứu mở rộng ngôn ngữ C nhằm đạt đến một ngôn ngữ mô phỏng (simulation language) với những tính năng hướng đối tượng.

Trong năm 1983, 1984, ngôn ngữ "C with Classes" được thiết kế lại, mở rộng hơn rồi một trình biên dịch ra đời. Và chính từ đó, xuất hiện tên gọi "C++". Bjarne Stroustrup mô tả ngôn ngữ C++ lần đầu tiên trong bài báo có nhan đề "Data Abstraction in C". Sau một vài hiệu chỉnh C++ được công bố rộng rãi trong quyển "The C++ Programming Language" của Bjarne Stroustrup xuất hiện đánh dấu sự hiện diện thực sự của C++, người lập trình chuyên nghiệp từ đây đã có một ngôn ngữ đủ mạnh cho các dự án thực tiễn của mình.

Về thực chất C++ giống như C nhưng bổ sung thêm một số mở rộng quan trọng, đặc biệt là ý tưởng về đối tượng, lập trình định hướng đối tượng. Thật ra các ý tưởng về cấu trúc trong C++ đã xuất phát vào các năm 1970 từ Simula 70 và Algol 68. Các ngôn ngữ này đã đưa ra các khái niệm về lớp và đơn thể. Ada là một ngôn ngữ phát triển từ đó, nhưng C++ đã khẳng định vai trò thực sự của mình.

II. NHỮNG ĐIỂM KHÔNG TƯƠNG THÍCH VỚI C

II.1. Định nghĩa hàm

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

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.

Ví dụ:

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

```
/*cả C và C++ cho phép*/
double vidu(int u,double v)
{}
```


Đối với C++, 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 vidu (int, double) /*khai báo hàm vidu*/
....
main() {
int n;
char c;
double z, res1, res2, res3;
....
res1 = vidu(n, z); /* không có chuyển đổi kiểu*/
res2 = vidu(c, z); /* có chuyển đổi kiểu, từ char (c) thành int*/
res3 = vidu(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 phần 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**.

II.2. 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;
```

Hai câu lệnh trên đã ngầm định thực hiện phép chuyển đổi kiểu:

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

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

Trong C++, chỉ chấp nhận ngầm định chuyển đổi kiểu từ một kiểu trỏ tùy ý thành **void***. Nếu muốn chuyển đổi ngược lại, ta phải thực hiện chuyển kiểu tường minh như sau:

```
gen = adj;
```

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

III. CÁC MỞ RỘNG CỦA C++

III.1. Các khả năng vào/ra mới của C++

Ngoài các tiện ích vào/ra (hàm hoặc macro) của thư viện C chuẩn đều có thể sử dụng được, C++ còn cài đặt thêm các khả năng vào/ra mới dựa trên hai toán tử "<<" (xuất) và ">>" (nhập).

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.

Ví dụ 2.2 : Sử dụng cout và "<<" để xuất ra màn hình

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

```
Value : 25
```

Chúng ta có thể sử dụng toán tử "<<" cùng với cout để đưa ra màn hình giá trị của một biểu thức có kiểu: cơ sở (char, int, float, double), chuỗi ký tự: (char *), con trỏ (trừ con trỏ char *). Trong trường hợp muốn đưa ra địa chỉ biến chuỗi ký tự phải thực hiện phép chuyển kiểu tương minh, chẳng hạn (**char***)--> (**void***).

Ví dụ 2.3 : Xuất ra màn hình dữ liệu các kiểu khác nhau

```
#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 << "Adresse 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++
Adresse of n : 0xffff2
Adresse de ch : 0x00b2
```

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

Ví dụ 2.4 : Dùng cin và “>>” để nhập dữ liệu vào từ bàn phím

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

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

- 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.
- 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”. Mã 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
```

III.2.Chú thích cuối dòng

Ngoài đoạn chú thích giới hạn bởi “/*” và “*/” giống như C, trong C++, mọi ký hiệu đi sau “/” cho đến hết dòng cũng được coi là chú thích, được chương trình dịch bỏ qua khi biên dịch chương trình. Ví dụ có thể viết chú thích như sau:

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

III.3.Khai báo mọi nơi

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 có thể khai báo như trong đoạn chương trình sau:

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

III.4. Các biến kiểu const

Trong ANSI C, muốn định nghĩa một hằng có kiểu nhất định thì chúng ta dùng biến **const** (vì nếu dùng **#define** thì tạo ra các hằng không có chứa thông tin về kiểu). Trong C++, các biến **const** linh hoạt hơn một cách đáng kể.

C++ xem **const** cũng như **#define** nếu như chúng ta muốn dùng hằng có tên trong chương trình. Chính vì vậy chúng ta có thể dùng **const** để quy định kích thước của một mảng như đoạn mã sau:

```
const int ArraySize = 100;
int X[ArraySize];
```

Khi khai báo một biến **const** trong C++ thì chúng ta phải khởi tạo một giá trị ban đầu nhưng đối với ANSI C thì không nhất thiết phải làm như vậy (vì trình biên dịch ANSI C tự động gán trị zero cho biến **const** nếu chúng ta không khởi tạo giá trị ban đầu cho nó).

Phạm vi của các biến **const** giữa ANSI C và C++ khác nhau. Trong ANSI C, các biến **const** được khai báo ở bên ngoài mọi hàm thì chúng có phạm vi toàn cục, điều này nghĩa là chúng có thể nhìn thấy cả ở bên ngoài file mà chúng được định nghĩa, trừ khi chúng được khai báo là **static**. Nhưng trong C++, các biến **const** được hiểu mặc định là **static**.

III.5. Về struct, union và enum

Trong C++, các struct và union thực sự các các kiểu class. Tuy nhiên có sự thay đổi đối với C++. Đó là tên của struct và union được xem luôn là tên kiểu giống như khai báo bằng lệnh typedef vậy. Ta có thể khai báo đơn giản như sau:

```
struct Complex
{
    float Real;
    float Imaginary;
};
.....
Complex C; // với C thì phải khai báo là struct Complex C;
```

Quy định này cũng áp dụng cho cả **union** và **enum**. Tuy nhiên để tương thích với C, C++ vẫn chấp nhận cú pháp cũ.

Một kiểu **union** đặc biệt được thêm vào C++ gọi là **union** nặc danh (anonymous union). Nó chỉ khai báo một loạt các trường(field) dùng chung một vùng địa chỉ bộ nhớ. Một **union** nặc danh không có tên **tag**, các trường có thể được truy xuất trực tiếp bằng tên của chúng. Chẳng hạn như đoạn mã sau:

```

union
{
    int Num;
    float Value;
};

.....

/*Cả hai Num và Value đều dùng chung một vị trí và không gian bộ nhớ. Tuy nhiên
không giống như kiểu union có tên, các trường của union nặc danh thì được truy
xuất trực tiếp như sau*/

Num = 12;
Value = 30.56;

.....

```

III.6. Toán tử phạm vi “::”

Bình thường, biến cục bộ che lấp biến toàn cục cùng tên. Trong những trường hợp cần thiết, khi muốn truy xuất tới biến toàn cục phải sử dụng toán tử “::” trước tên biến.

Ví dụ 2.5 : Dùng toán tử “::” để truy xuất biến toàn cục

```

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

```

III.7. Hàm nội tuyến

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. Nếu một hàm được định nghĩa nội tuyến ở trong một tập tin thì nó không sẵn dùng cho các tập tin khác. Do đó, các hàm nội tuyến thường được đặt vào trong các tập tin **header** để mà chúng có thể được chia sẻ.

Ví dụ 2.6: Giả sử một chương trình thường xuyên yêu cầu tìm giá trị tuyệt đối của một số các số nguyên. Ta sẽ xây dựng một hàm nội tuyến xác định giá trị tuyệt đối của một số nguyên.

```

#include <iostream.h>
#include <conio.h>
inline int Abs(int n) // định nghĩa hàm inline

```

```

{
    return(n>0? n:-n);
}

void main()
{
    clrscr();
    int a=-5, b=5;
    cout <<"Abs(a) = "<<Abs(a) <<" and Abs(b) = "<<Abs(b) <<"\n";
    getch();
}

```

```
Abs(a) = 5 and Abs(b) = 5
```

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**.

Hàm đệ qui không thể khai báo **inline**.

III.8. Giá trị tham số ngầm định

Một trong các đặc tính nổi bật nhất của C++ là khả năng định nghĩa các giá trị tham số mặc định cho các hàm. Nếu thường xuyên gọi hàm với một giá trị tham số cố định nào đó, ta có thể định nghĩa hàm với tham số mặc định.

Ví dụ 2.7: tính thể tích hình hộp

```

#include <iostream.h>
int BoxVolume(int Length = 1, int Width = 1, int Height = 1);
int main()
{
    cout << "The tích hình hộp mac dinh: "
    << BoxVolume() << endl << endl
    << "The tích hình hộp voi chieu dai=10,do rong=1,chieu cao=1:"
    << BoxVolume(10) << endl << endl
    << "The tích hình hộp voi chieu dai=10,do rong=5,chieu cao=1:"
    << BoxVolume(10, 5) << endl << endl
    << "The tích hình hộp voi chieu dai=10,do rong=5,chieu cao=2:"
    << BoxVolume(10, 5, 2)<< endl;
    return 0;
}
//Tính thể tích của hình hộp
int BoxVolume(int Length, int Width, int Height)
{
    return Length * Width * Height;
}

```

```

The tích hình hộp mac dinh: 1
The tích hình hộp voi chieu dai=10,do rong=1,chieu cao=1: 10
The tích hình hộp voi chieu dai=10,do rong=5,chieu cao=1: 50

```

```
The tích hình hộp voi chiều dài=10,do rộng=5,chiều cao=2: 100
```

■ Các tham số có giá trị mặc định chỉ được cho trong prototype của hàm và không được lặp lại trong định nghĩa hàm (Vì trình biên dịch sẽ dùng các thông tin trong prototype chứ không phải trong định nghĩa hàm để tạo một lệnh gọi).

■ Một hàm có thể có nhiều tham số có giá trị mặc định. Các tham số có giá trị mặc định cần phải được nhóm lại vào các tham số cuối cùng (hoặc duy nhất) của một hàm. Khi gọi hàm có nhiều tham số có giá trị mặc định, chúng ta chỉ có thể bỏ bớt các tham số theo thứ tự từ phải sang trái và phải bỏ liên tiếp nhau.

III.9.Toán tử new và delete

Trong các chương trình C, tất cả các cấp phát động bộ nhớ đều được xử lý thông qua các hàm thư viện như **malloc()**, **calloc()** và **free()**. C++ định nghĩa một phương thức mới để thực hiện việc cấp phát động bộ nhớ bằng cách dùng hai toán tử **new** và **delete**. Sử dụng hai toán tử này sẽ linh hoạt hơn rất nhiều so với các hàm thư viện của C.

Đoạn chương trình sau dùng để cấp phát vùng nhớ động trong C.

```
int *P;
P = malloc(sizeof(int));
if (P==NULL)
printf("Khong con du bo nho de cap phat\n");
else {
    *P = 290;
    printf("%d\n", *P);
    free(P);
}
```

Trong C++, chúng ta có thể viết lại đoạn chương trình trên như sau:

```
int *P;
P = new int;
if (P==NULL)
cout<<"Khong con du bo nho de cap phat\n";
else {
    *P = 290;
    cout<<*P<<"\n";
    delete P;
}
```

➤ Cú pháp dùng toán tử new

```
new type_name
new ( type_name )
new type_name initializer
new ( type_name ) initializer
```

Trong đó :

- **type_name**: Mô tả kiểu dữ liệu được cấp phát. Nếu kiểu dữ liệu mô tả phức tạp, nó có thể được đặt bên trong các dấu ngoặc.
- **initializer**: Giá trị khởi động của vùng nhớ được cấp phát.
- Nếu toán tử new cấp phát không thành công thì nó sẽ trả về giá trị NULL.

➤ Cú pháp dùng toán tử delete

```
delete pointer
delete [] pointer
```

Ví dụ 2.8: 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 ma trận 10*10\n";
    Nhap(mat);
    /*In ma trận*/
    cout<<"Ma trận vừa nhập \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<<"Thành phần thu ["<i><<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";
}
}
```

IV. PHÉP 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.

IV.1. Biến tham chiếu

Biến tham chiếu có thể sử dụng như một bí danh của biến khác (bí danh đơn giản như một tên khác của biến gốc). Ví dụ có khai báo như sau:

```
int Count = 1;
int & Ref = Count; //Tạo biến Ref như là một bí danh của biến Count
++Ref; //Tăng biến Count lên 1 (sử dụng bí danh của biến Count)
```

Khi một tham chiếu được khai báo như một bí danh của biến khác, mọi thao tác thực hiện trên bí danh chính là thực hiện trên biến gốc của nó. Chúng ta có thể lấy địa chỉ của biến tham chiếu và có thể so sánh các biến tham chiếu với nhau (phải tương thích về kiểu tham chiếu).

Ví dụ 2.9: Sử dụng biến tham chiếu như là 1 bí danh của biến

```
#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
```

■ Các biến tham chiếu phải được khởi động trong phần khai báo của chúng và chúng ta không thể gán lại một bí danh của biến khác cho chúng.

■ Chúng ta có thể tạo ra biến tham chiếu với việc khởi động là một hằng. Trong trường hợp này, trình biên dịch tạo ra một biến tạm thời chứa trị hằng và biến tham chiếu chính

là bí danh của biến tạm thời này. Điều này gọi là tham chiếu độc lập (independent reference). Chẳng hạn như đoạn mã sau :

```
int & Ref = 45;
```

- Mặc dù biến tham chiếu trông giống như là biến con trỏ nhưng chúng không thể là biến con trỏ do đó chúng không thể được dùng cấp phát động.
- Chúng ta không thể khai báo một biến tham chiếu chỉ đến biến tham chiếu hoặc biến con trỏ chỉ đến biến tham chiếu. Tuy nhiên chúng ta có thể khai báo một biến tham chiếu về biến con trỏ như đoạn mã sau:

```
int X;
int *P = &X;
int * &Ref = P;
```

IV.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. Tuy nhiên, 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 (xem hình 2.1).



Hình 2.1: Một tham số kiểu tham chiếu nhận một tham chiếu tới một biến được chuyển cho tham số của hàm.

Ví dụ 2.10: Chương trình sau đưa ra hai cách viết khác nhau của hàm thực hiện việc hoán đổi nội dung của hai biến.

```
/*swap.cpp*/
#include <conio.h>
#include <iostream.h>
/*Hàm swap1 thực hiện việc truyền tham số bằng tham trỏ*/
void swap1(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp; }
/*Hàm swap2 thực hiện việc truyền tham số bằng tham chiếu*/
void swap2(int &x, int &y) {
    int temp = x;
```

```

    x = y;
    y = temp; }
void main() {
    int a=3, b=4;
    clrscr();
    cout <<"Truoc khi goi swap1:\n";
    cout<<"a = "<<a<<" b = "<<b<<"\n";
    swap1 (&a,&b);
    cout <<"Sau khi goi swap1:\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";
    getch();
}

```

```

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

```

IV.3.Giá trị trả về của hàm là tham chiếu

Các hàm có thể trả về một tham chiếu, nhưng điều này rất nguy hiểm. Khi hàm trả về một tham chiếu tới một biến cục bộ của hàm thì biến này phải được khai báo là **static**, nếu không thì khi hàm kết thúc biến cục bộ này sẽ bị huỷ mất.

Khi giá trị trả về của hàm là tham chiếu, ta có thể gặp các câu lệnh gán “kỳ dị” trong đó về 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, vế trái của lệnh gán (biểu thức gán) có thể là lời gọi đến một hàm có giá trị trả về là một tham chiếu.

Ví dụ 2.11: Giá trị trả về của hàm là một tham chiếu

```

#include <iostream.h>
#include <conio.h>
int X;
//prototype
int & MyFunc();
int main()

```

```

{ clrscr();
  X=4;
  cout<<"X="<<X<<endl;
  cout<<"X="<<MyFunc()<<endl;
  MyFunc() = 20; //Nghĩa là X = 20
  cout<<"X="<<X<<endl;
  getch();
  return 0;
}
int & MyFunc()
{
    return X;
}

```

```

X=6
X=4
X=20

```

V. PHÉP ĐA NĂNG HOÁ

Với ngôn ngữ C++, chúng ta có thể đa năng hóa các hàm và các toán tử (operator). Đa năng hóa là phương pháp cung cấp nhiều hơn một định nghĩa cho tên hàm đã cho trong cùng một phạm vi. Trình biên dịch sẽ lựa chọn phiên bản thích hợp của hàm hay toán tử dựa trên các tham số mà nó được gọi.

V.1. Đa năng hóa các hàm (Functions overloading)

Trong ngôn ngữ C cũng như mọi ngôn ngữ máy tính khác, mỗi hàm đều phải có một tên phân biệt. Đôi khi đây là một điều phiền toái. Chẳng hạn như trong ngôn ngữ C, có rất nhiều hàm trả về trị tuyệt đối của một tham số là số, vì cần thiết phải có tên phân biệt nên C phải có hàm riêng cho mỗi kiểu dữ liệu số, do vậy chúng ta có tới ba hàm khác nhau để trả về trị tuyệt đối của một tham số :

```

int abs(int i);
long labs(long l);
double fabs(double d);

```

Tất cả các hàm này đều cùng thực hiện một chức năng nên chúng ta thấy điều này nghịch lý khi phải có ba tên khác nhau. C++ giải quyết điều này bằng cách cho phép chúng ta tạo ra các hàm khác nhau có cùng một tên. Đây chính là đa năng hóa hàm. Do đó trong C++ chúng ta có thể định nghĩa lại các hàm trả về trị tuyệt đối để thay thế các hàm trên như sau :

```

int abs(int i);
long abs(long l);
double abs(double d);

```

Ví dụ 2.12: Định nghĩa chồng hàm trị tuyệt đối

```
#include <iostream.h>
#include <math.h>
int MyAbs(int X);
long MyAbs(long X);
double MyAbs(double X);
int main()
{   int X = -7;
    long Y = 200000;
    double Z = -35.678;
    cout<<"Tri tuyet doi cua so nguyen (int) "<<X<<" la "<<MyAbs(X)<<endl;
    cout<<"Tri tuyet doi cua so nguyen (long int) "<<Y<<" la "<<MyAbs(Y)<<endl;
    cout<<"Tri tuyet doi cua so thuc "<<Z<<" la "<<MyAbs(Z)<<endl;
    return 0; }
int MyAbs(int X)
{   return abs(X); }
long MyAbs(long X)
{   return labs(X); }
double MyAbs(double X)
{   return fabs(X); }
```

```
Tri tuyet doi cua so nguyen (int) -7 la 7
Tri tuyet doi cua so nguyen (long int) 200000 la 200000
Tri tuyet doi cua so thuc -35.678 la 35.678
```

Trình biên dịch dựa vào sự khác nhau về số các tham số, kiểu của các tham số để có thể xác định chính xác phiên bản cài đặt nào của hàm là thích hợp với một lệnh gọi hàm được cho, chẳng hạn như những lời gọi hàm `MyAbs()` như sau:

```
MyAbs(-7); //Gọi hàm int MyAbs(int)
MyAbs(-7l); //Gọi hàm long MyAbs(long)
MyAbs(-7.5); //Gọi hàm double MyAbs(double)
```

Quá trình tìm được hàm được đa năng hóa cũng là quá trình được dùng để giải quyết các trường hợp nhập nhằng của C++. Chẳng hạn như nếu tìm thấy một phiên bản định nghĩa nào đó của một hàm được đa năng hóa mà có kiểu dữ liệu các tham số của nó trùng với kiểu các tham số đã gọi tới trong lệnh gọi hàm thì phiên bản hàm đó sẽ được gọi. Nếu không trình biên dịch C++ sẽ gọi đến phiên bản nào cho phép chuyển kiểu dễ dàng nhất. Các phép chuyển kiểu có sẵn sẽ được ưu tiên hơn các phép chuyển kiểu mà chúng ta tạo ra. Ví dụ các lời gọi hàm `MyAbs()`:

```
MyAbs('c'); //Gọi int MyAbs(int)
MyAbs(2.34f); //Gọi double MyAbs(double)
```

Lưu ý:

- Bất kỳ hai hàm nào trong tập các hàm đã đa năng phải có các tham số khác nhau.
- Các khai báo bằng lệnh **typedef** không định nghĩa kiểu mới. Chúng chỉ thay đổi tên gọi của kiểu đã có. Chúng không có giá trị phân biệt trong cơ chế đa năng hóa hàm.

Kiểu mảng và con trỏ được xem như đồng nhất trong việc phân biệt khác sự nhau giữa các phiên bản hàm đa năng hóa.

V.2. Đa năng hóa các toán tử (Operators overloading)

Trong ngôn ngữ C, khi chúng ta tự tạo ra một kiểu dữ liệu mới, chúng ta thực hiện các thao tác liên quan đến kiểu dữ liệu đó thường thông qua các hàm, điều này trở nên không thoải mái. Ví dụ như khi chúng ta xây dựng cấu trúc dữ liệu và lập trình với bài toán số phức, chúng ta phải xây dựng các hàm thực hiện các phép toán, trong khi bản chất của chúng là các toán tử. Để khắc phục yếu điểm này, trong C++ cho phép chúng ta có thể định nghĩa lại chức năng của các toán tử đã có sẵn một cách tiện lợi. Điều này gọi là đa năng hóa toán tử.

Cú pháp khai báo hàm định nghĩa một toán tử như sau:

```
data_type operator operator_symbol ( parameters )
{
.....
}
```

Trong đó: *data_type*: Kiểu trả về.

operator_symbol: Ký hiệu của toán tử.

parameters: Các tham số (nếu có).

Ví dụ 2.13: Chương trình cài đặt các phép toán cộng và trừ số phức

```
#include <iostream.h>
// Định nghĩa số phức
typedef struct
{
    double Real;
    double Imaginary;
}Complex;
Complex SetComplex(double R,double I);
void DisplayComplex(Complex C);
Complex operator + (Complex C1,Complex C2);
Complex operator - (Complex C1,Complex C2);
int main(void)
{
    Complex C1,C2,C3,C4;
    C1 = SetComplex(1.0,2.0);
    C2 = SetComplex(-3.0,4.0);
    cout<<"\nSo phuc thu nhat:";
    DisplayComplex(C1);
    cout<<"\nSo phuc thu hai:";
    DisplayComplex(C2);
    C3 = C1 + C2;
    C4 = C1 - C2;
```

```

    cout<<"\nTong hai so phuc nay:";
    DisplayComplex(C3);
    cout<<"\nHieu hai so phuc nay:";
    DisplayComplex(C4);
    return 0;
}
//Đặt giá trị cho một số phức
Complex SetComplex(double R,double I)
{
    Complex Tmp;
    Tmp.Real = R;
    Tmp.Imaginary = I;
    return Tmp;
}
//Cộng hai số phức
Complex operator + (Complex C1,Complex C2)
{
    Complex Tmp;
    Tmp.Real = C1.Real+C2.Real;
    Tmp.Imaginary = C1.Imaginary+C2.Imaginary;
    return Tmp;
}
//Trừ hai số phức
Complex operator - (Complex C1,Complex C2)
{
    Complex Tmp;
    Tmp.Real = C1.Real-C2.Real;
    Tmp.Imaginary = C1.Imaginary-C2.Imaginary;
    return Tmp;
}
//Hiển thị số phức
void DisplayComplex(Complex C)
{
    cout<<"("<<C.Real<<","<<C.Imaginary<<")";
}

```

```

So phuc thu nhat:(1,2)
So phuc thu hai:(-3,4)
Tong hai so phuc nay:(-2,6)
Hieu hai so phuc nay:(4,-2)

```

Trong chương trình ví dụ 2.13, toán tử + là toán tử gồm hai toán hạng (gọi là toán tử hai ngôi; toán tử một ngôi là toán tử chỉ có một toán hạng) và trình biên dịch biết tham số đầu tiên là ở bên trái toán tử, còn tham số thứ hai thì ở bên phải của toán tử. Trong trường hợp lập trình viên quen thuộc với cách gọi hàm, C++ vẫn cho phép bằng cách viết như sau:

```

C3 = operator + (C1,C2);
C4 = operator - (C1,C2);

```

Các toán tử được đa năng hóa sẽ được lựa chọn bởi trình biên dịch cũng theo cách thức tương tự như việc chọn lựa giữa các hàm được đa năng hóa là khi gặp một toán tử làm việc trên các kiểu không phải là kiểu có sẵn, trình biên dịch sẽ tìm một hàm định nghĩa của toán tử nào đó có các tham số đối sánh với các toán hạng để dùng.

Lưu ý:

- Chúng ta không thể định nghĩa các toán tử mới.
- Các toán tử có thể đa năng hoá

+	-	*	/	%	^
!	=	<	>	+=	-=
^=	&=	=	<<	>>	<<=
<=	>=	&&		++	--
()	[]	new	delete	&	
~	*=	/=	%=	>>=	==
!=	,	->	->*		

- Các toán tử không được đa năng hóa

Toán tử	Ý nghĩa
::	Toán tử định phạm vi.
.*	Truy cập đến con trỏ là trường của struct hay thành viên của class.
.	Truy cập đến trường của struct hay thành viên của class.
?:	Toán tử điều kiện
sizeof	Trả về kiểu của tham số

- Chúng ta không thể thay đổi thứ tự ưu tiên của một toán tử hay không thể thay đổi số các toán hạng của nó.
- Chúng ta không thể thay đổi ý nghĩa của các toán tử khi áp dụng cho các kiểu có sẵn.
- Đa năng hóa các toán tử không thể có các tham số có giá trị mặc định.

CHƯƠNG 3

ĐỐI TƯỢNG VÀ LỚP

I. KHÁI NIỆM ĐỐI TƯỢNG VÀ LỚP

Đối tượng là một khái niệm trong lập trình hướng đối tượng (OOP) biểu thị sự liên kết giữa dữ liệu và các thủ tục (gọi là các phương thức) thao tác trên dữ liệu đó. OOP đóng gói dữ liệu (các thuộc tính) và các hàm (hành vi) thành gói gọi là các đối tượng.

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. Việc đóng gói làm cho một đối tượng, khi nhìn từ bên ngoài 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.

Trong C và các ngôn ngữ lập trình thủ tục, lập trình có định hướng hành động, trong khi trong lập trình C++ là định hướng đối tượng. Trong C, đơn vị của lập trình là **hàm**; trong C++, đơn vị của lập trình là **lớp (class)**.

Các lập trình viên C tập trung vào viết các **hàm**. Các nhóm của các hành động mà thực hiện vài công việc được tạo thành các **hàm**, và các **hàm** được nhóm thành các chương trình. Dữ liệu trong C đóng vai trò hỗ trợ các hành động mà hàm thực hiện. Các **động từ** trong một hệ thống giúp cho lập trình viên C xác định tập các hàm mà sẽ hoạt động cùng với việc thực thi hệ thống.

Các lập trình viên C++ tập trung vào việc tạo ra "**các kiểu do người dùng định nghĩa**" (user-defined types) gọi là các lớp. Các lớp cũng được tham chiếu như "**các kiểu do lập trình viên định nghĩa**" (programmer-defined types). Mỗi lớp chứa dữ liệu cũng như tập các hàm xử lý dữ liệu. Các thành phần dữ liệu của một lớp được gọi là "**các thành viên dữ liệu**" (data members). Các thành phần hàm của một lớp được gọi là "**các hàm thành viên**" (member functions), còn gọi là các giao diện của lớp. Giống như thực thể của kiểu có sẵn như **int** được gọi là một biến, một thực thể của kiểu do người dùng định nghĩa (nghĩa là một lớp) được gọi là một **đối tượng**. Các **danh từ** trong một hệ thống giúp cho lập trình viên C++ xác định tập các **lớp**. Các lớp này được sử dụng để tạo các đối tượng cho việc thực thi hệ thống.

Các thành viên lớp được liệt kê vào một trong ba loại quyền truy xuất khác nhau:

- Thành viên chung (public) có thể được truy xuất bởi tất cả các thành phần sử dụng lớp.
- Thành viên riêng (private) chỉ có thể được truy xuất bởi các thành viên thuộc lớp.
- Thành viên được bảo vệ (protected) chỉ có thể được truy xuất bởi các thành viên thuộc lớp và các thành viên của một lớp dẫn xuất.

II. CÀI ĐẶT MỘT LỚP

II.1. Khai báo lớp

Các lớp trong C++ được tiến hóa tự nhiên của khái niệm **struct** trong C. Trước khi tiến hành việc trình bày các lớp trong C++, chúng ta tìm hiểu về cấu trúc, và chúng ta xây dựng một kiểu do người dùng định nghĩa dựa trên một cấu trúc.

Ví dụ 3.1: Chúng ta xây dựng kiểu cấu trúc **Point** với hai thành viên số nguyên là hai tọa độ **x và y**.

```
struct point
{
    int x, y;
};

void init(point p, int ox, int oy); //hàm khởi tạo điểm p
void move(point p, int dx, int dy); //hàm di chuyển điểm p
```

Với cách khai báo cấu trúc như trên, theo quan điểm lập trình cấu trúc, chúng ta phải lập trình xây dựng các hàm tách rời thao tác với cấu trúc dữ liệu trên.

Đối với OOP, các lớp cho phép lập trình viên mô hình các đối tượng mà có các thuộc tính (biểu diễn như các thành viên dữ liệu – Data members) và các hành vi hoặc các thao tác (biểu diễn như các hàm thành viên – Member functions). Các kiểu chứa các thành viên dữ liệu và các hàm thành viên được định nghĩa thông thường trong C++ sử dụng từ khóa **class**, có cú pháp khai báo lớp và đối tượng như sau:

```
class <class-name>
{
    <member-list> //Thân của lớp
};

<class-name> <object-name>; // khai báo đối tượng của lớp
```

Trong đó: *class-name*: tên lớp.

member-list: đặc tả các thành viên dữ liệu và các hàm thành viên.

Các hàm thành viên đôi khi được gọi là các **phương thức** (methods) trong các ngôn ngữ lập trình hướng đối tượng khác, và được đưa ra trong việc đáp ứng các **message** gửi tới một đối tượng. Một **message** tương ứng với việc gọi hàm thành viên.

Các kiểu khai báo lớp khác nhau đều có thể chuẩn hoá để đưa về dạng sau:

```
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>
...
```

Ví dụ 3.2: Chúng ta xây dựng lớp **Point** thay cho cấu trúc ở 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
Ham thanh phan display
Toa do: 3 6

```

Chúng ta nhận thấy rằng, tất cả các thành viên dữ liệu của một lớp không thể khởi tạo tại nơi mà chúng được khai báo trong thân lớp. Các thành viên dữ liệu này phải được khởi tạo bởi constructor của lớp hay chúng có thể gán giá trị bởi các hàm thiết lập.

Khi một lớp được định nghĩa và các hàm thành viên của nó được khai báo, các hàm thành viên này phải được định nghĩa. Mỗi hàm thành viên của lớp có thể được định nghĩa trực tiếp trong thân lớp (hiển nhiên bao gồm prototype hàm của lớp), hoặc hàm thành viên có thể được định nghĩa sau thân lớp. Khi một hàm thành viên được định nghĩa sau định nghĩa lớp tương ứng, tên hàm được đặt trước bởi tên lớp và toán tử định **phạm vi (::)**. Khi đó ta sử dụng cú pháp:

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

Mặc dù một hàm thành viên khai báo trong định nghĩa một lớp có thể định nghĩa bên ngoài định nghĩa lớp này, hàm thành viên đó vẫn còn bên trong phạm vi của lớp, nghĩa là tên của nó chỉ được biết tới các thành viên khác của lớp ngoại trừ tham chiếu thông qua một đối tượng của lớp, một tham chiếu tới một đối tượng của lớp, hoặc một con trỏ trỏ tới một đối tượng của lớp.

Nếu một hàm thành viên được định nghĩa trong định nghĩa một lớp, hàm thành viên này chính là hàm **inline**. Các hàm thành viên định nghĩa bên ngoài định nghĩa một lớp có thể là hàm **inline** bằng cách sử dụng từ khóa **inline**.

Hàm thành viên cùng tên với tên lớp nhưng đặt trước là một ký tự ngã (~) được gọi là **destructor** của lớp này. Hàm **destructor** làm "công việc nội trợ kết thúc" trên mỗi đối tượng của lớp trước khi vùng nhớ cho đối tượng được phục hồi bởi hệ thống.

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

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

II.2. Tạo đối tượng

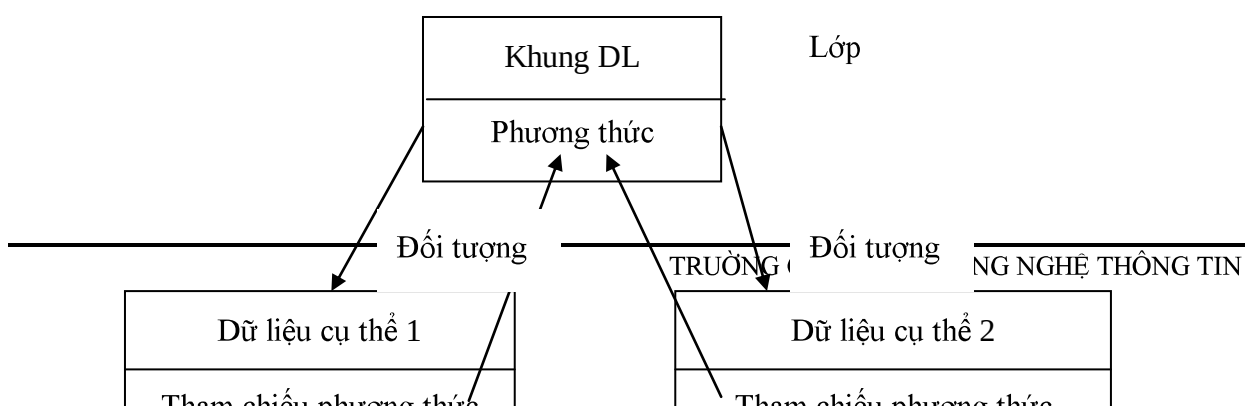
Trong C++, một đối tượng có thể được xác lập thông qua một biến/hằng có kiểu lớp. Cú pháp tạo đối tượng như sau:

```
<tên lớp> <tên 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.



III. PHẠM VI LỚP VÀ TRUY CẬP CÁC THÀNH VIÊN LỚP

III.1. Phạm vi lớp

Trong lập trình cấu trúc thường đề cập đến phạm vi toàn cục (tệp, chương trình) và phạm vi cục bộ (khối lệnh, hàm). Mục đích của phạm vi là để kiểm soát việc truy xuất đến các biến/hằng/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 viên dữ liệu của một lớp (các biến khai báo trong định nghĩa lớp) và các hàm thành viên (các hàm khai báo trong định nghĩa lớp) thuộc vào phạm vi của 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.

Các toán tử được sử dụng để truy cập các thành viên của lớp được đồng nhất với các toán tử sử dụng để truy cập các thành viên của cấu trúc. Toán tử lựa chọn thành viên **dấu chấm (.)** được kết hợp với một tên của đối tượng hay với một tham chiếu tới một đối tượng để truy cập các thành viên của đối tượng. Toán tử lựa chọn thành viên **mũi tên (->)** được kết hợp với một con trỏ tới một truy cập để truy cập các thành viên của đối tượng.

Ví dụ 3.3: Chương trình minh họa việc truy cập các thành viên của một lớp với các toán tử lựa chọn thành viên.

```
#include <iostream.h>

class Count
{
    public:
        int X;
        void Print()
        {   cout << X << endl;   }
};

int main()
{
    Count Counter, //Tạo đối tượng Counter
    *CounterPtr = &Counter, //Con trỏ trỏ tới Counter
    &CounterRef = Counter; //Tham chiếu tới Counter
    cout << "Assign 7 to X and Print using the object's name: ";
    Counter.X = 7; //Gán 7 cho thành viên dữ liệu X
}
```

```

Counter.Print(); //Gọi hàm thành viên Print
cout << "Assign 8 to X and Print using a reference: ";
CounterRef.X = 8; //Gán 8 cho thành viên dữ liệu X
CounterRef.Print(); //Gọi hàm thành viên Print
cout << "Assign 10 to X and Print using a pointer: ";
CounterPtr->X = 10; // Gán 10 cho thành viên dữ liệu X
CounterPtr->Print(); //Gọi hàm thành viên Print
return 0;
}

```

```

Assign 7 to X and Print using the object's name: 7
Assign 8 to X and Print using a reference: 8
Assign 10 to X and Print using a pointer: 10

```

III.2. Điều khiển truy cập đến các thành viên

Các thuộc tính truy cập **public** và **private** (và **protected** chúng ta sẽ xem xét sau) được sử dụng để điều khiển truy cập tới các thành viên dữ liệu và các hàm thành viên của lớp. Chế độ truy cập mặc định đối với lớp là **private** vì thế tất cả các thành viên sau phần **header** của lớp và trước nhãn đầu tiên là **private**. Sau mỗi nhãn, chế độ mà được kéo theo bởi nhãn đó áp dụng cho đến khi gặp nhãn kế tiếp hoặc cho đến khi gặp dấu **móc phải** (**}**) của phần định nghĩa lớp. Các nhãn **public**, **private** và **protected** có thể được lặp lại nhưng cách dùng như vậy thì hiếm có và có thể gây khó hiểu.

Các thành viên **private** chỉ có thể được truy cập bởi các hàm thành viên (và các hàm **friend**) của lớp đó. Các thành viên **public** của lớp có thể được truy cập bởi bất kỳ hàm nào trong chương trình.

Mục đích chính của các thành viên **public** là để biểu thị cho **client** của lớp một cái nhìn của các dịch vụ (services) mà lớp cung cấp. Tập hợp này của các dịch vụ hình thành giao diện **public** của lớp. Các **client** của lớp không cần quan tâm làm thế nào lớp hoàn thành các thao tác của nó. Các thành viên **private** của lớp cũng như các định nghĩa của các hàm thành viên **public** của nó thì không phải có thể truy cập tới **client** của một lớp. Các thành phần này hình thành sự thi hành của lớp.

Thuộc tính truy cập mặc định đối với các thành viên của lớp là **private**. Thuộc tính truy cập các thành viên của một lớp có thể được thiết lập rõ ràng là **public**, **protected** hoặc **private**.

Truy cập đến một dữ liệu **private** cần phải được điều khiển cẩn thận bởi việc sử dụng của các hàm thành viên, gọi là các hàm truy cập (access functions).

Ví dụ 3.4: Chương trình thể hiện các thành viên **private** chỉ có thể truy cập thông qua giao diện **public** sử dụng các hàm thành viên **public**.

```

#include <iostream.h>
class MyClass
{ private:
    int X,Y;
public:

```

```

    void Print();
};
void MyClass::Print()
{   cout <<X<<Y<<endl; }
int main()
{   MyClass M;
    M.X = 3; //Error: 'MyClass::X' is not accessible
    M.Y = 4; //Error: 'MyClass::Y' is not accessible
    M.Print();
    return 0;
}

```

➤ **Gọi một hàm thành phần trong một hàm thành phần khác:**

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 “::”.

➤ **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::init(int abs,int ord)
{   x=abs;
    y=ord; }

```

➤ **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 trỏ 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); }
}

```

```
}

```

➤ 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;    }
};

```

III.3.Con trỏ this

Mỗi đối tượng duy trì một con trỏ trỏ tới chính nó – gọi là con trỏ **this** – Đó là một tham số ẩn trong tất cả các tham chiếu tới các thành viên bên trong đối tượng đó. 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**.

Ví dụ: dùng con trỏ **this** trong hàm **point::coincide(point pt)** như sau:

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

```

IV. CÁC HÀM TRUY CẬP VÀ CÁC HÀM TIỆN ÍCH

Không phải tất cả các hàm thành viên đều là **public** để phục vụ như bộ phận giao diện của một lớp. Một vài hàm còn lại là **private** và phục vụ như các **hàm tiện ích** (utility functions) cho các hàm khác của lớp.

Các hàm truy cập có thể đọc hay hiển thị dữ liệu. Sử dụng các hàm truy cập để kiểm tra tính đúng hoặc sai của các điều kiện – các hàm như thế thường được gọi là các hàm khẳng định (predicate functions).

Một hàm tiện ích không là một phần của một giao diện của lớp. Hơn nữa nó là một hàm thành viên **private** mà hỗ trợ các thao tác của các hàm thành viên **public**. Các hàm tiện ích không dự định được sử dụng bởi các **client** của lớp.

Ví dụ 3.5: Minh họa hàm tiện ích

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
class SalesPerson
{

```



```
public:
    SalesPerson();           //constructor
    void SetSales(int, double); //Người dùng cung cấp con số doanh thu của một
    tháng
    void PrintAnnualSales();
private:
    double Sales[12];        //12 con số doanh thu của các tháng
    double TotalAnnualSales(); //Hàm tiện ích
};

SalesPerson::SalesPerson()    //Hàm constructor khởi tạo mảng
{ for (int I = 0; I < 12; I++)
    Sales[I] = 0.0;
}

//Hàm thiết lập một trong 12 con số doanh thu của các tháng
void SalesPerson::SetSales(int Month, double Amount)
{ if (Month >= 1 && Month <= 12 && Amount > 0)
    Sales[Month - 1] = Amount;
    else
        cout << "Invalid month or sales figure" << endl;
}

//Hàm tiện ích để tính tổng doanh thu hàng năm
double SalesPerson::TotalAnnualSales()
{ double Total = 0.0;
    for (int I = 0; I < 12; I++)
        Total += Sales[I];
    return Total;
}

void SalesPerson::PrintAnnualSales() //In tổng doanh thu hàng năm
{ cout << setprecision(2)
    << setiosflags(ios::fixed | ios::showpoint)
    << endl << "The total annual sales are: $"
    << TotalAnnualSales() << endl;
}

int main()
{ SalesPerson S;
    double salesFigure;
    clrscr();
    for (int I = 1; I <= 12; I++)
    { cout << "Enter sales amount for month "<< I << ": ";
        cin >> salesFigure;
        S.SetSales(I, salesFigure);
    }
```

```

    }
    S.PrintAnnualSales();
    getch();
    return 0;
}

```

```

Enter sales amount for month 1: 120.5
Enter sales amount for month 2: 100.75
Enter sales amount for month 3: 180
Enter sales amount for month 4: 200
Enter sales amount for month 5: 350.06
Enter sales amount for month 6: 300
Enter sales amount for month 7: 120
Enter sales amount for month 8: 321
Enter sales amount for month 9: 123.45
Enter sales amount for month 10: 129
Enter sales amount for month 11: 124.5
Enter sales amount for month 12: 125.55

```

```

The total annual sales are: $2194.81

```

V. THIẾT LẬP VÀ HUỖ BỎ ĐỐI TƯỢNG

V.1. Hàm thiết lập - CONSTRUCTOR

Hàm thiết lập (constructor) là một hàm thành phần đặc biệt không thể thiếu được trong một lớp, là một hàm thành viên với tên giống như tên của 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.

Các constructor không thể mô tả các kiểu trả về hoặc các giá trị trả về. Các constructor có thể được đa năng hóa để cung cấp sự đa dạng để khởi tạo các đối tượng của lớp.

Ví dụ 3.6: Dùng hàm thiết lập thay cho hàm **init()** trong lớp **point** ở ví dụ 3.2.

```

#include <iostream.h>
#include <conio.h>
class point
{ /*khai báo các thành phần dữ liệu*/
    int x;
    int y;
public:
    /*khai báo các thành phần hàm*/
    point() {x=0; y=0;} // hàm thiết lập không tham số - constructor mặc định
    point(int ox,int oy) {x=ox;y=oy;} //hàm thiết lập có 2 tham số
    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 có hai tham số
    cout<<"Diem A: \n";
    a.display();
    a.move(-2,4); a.display();
    point b;      //sử dụng hàm thiết lập không có tham số
    cout<<"Diem B: \n";
    b.move(-2,4); b.display();
    point m[10]; //tạo mảng m có 10 đối tượng point - sử dụng constructor mặc định
    getch();
}

```

```

Diem A:
Toa do : 5 2
Toa do : 3 6
Diem B:
Toa do : -2 4

```

Nếu không có constructor nào được định nghĩa trong một lớp thì trình biên dịch tạo một constructor mặc định (default constructor). Constructor này không thực hiện bất kỳ sự khởi tạo nào, vì vậy khi đối tượng được tạo, nó không đảm bảo được ở một trạng thái phù hợp.

Constructor có thể có các tham số mặc định. Một constructor mà hoặc tất cả các tham số của nó có giá trị mặc định hoặc không có tham số nào thì cũng được gọi là constructor mặc định. Như vậy bằng cách này, ta có thể tạo ra constructor mặc định để đảm bảo đối tượng tạo ra có một trạng thái phù hợp.

➤ Một số lưu ý:

- Hàm thiết lập có cùng tên với tên của lớp.
- Hàm thiết lập phải có thuộc tính **public**.
- Hàm thiết lập không có giá trị trả về, và không cần khai báo **void**.
- Có thể có nhiều hàm thiết lập trong cùng lớp (chồng các hàm thiết lập).
- 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.
- Chỉ có thể có một **constructor mặc định** cho mỗi lớp.
- Đối với mảng các đối tượng, thì nhất thiết phải có constructor mặc định.

➤ Con trỏ và cấp phát động đối tượng:

- Ta có thể tạo con trỏ đối tượng và truy xuất các thành phần của đối tượng như sau:

```
point a;          //tạo đối tượng a
```

```

point *ptr;          //khai báo con trỏ đối tượng
.....
ptr = &a;            //cho con trỏ trỏ vào đối tượng a
.....
ptr = new point(5,2) //dùng toán tử new cấp phát động cho đối tượng
.....
ptr->display();       //gọi hàm thành phần của đối tượng được trỏ
ptr->move(-2,3);      //gọi hàm thành phần của đối tượng được trỏ

```

➤ 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);        //tạo đối tượng a
point &ra=a;          //ra là bí danh của đối tượng a
a.display();
ra.display();
ra.move(2,3);
a.display();

```

```

Toa do : 2 5
Toa do : 2 5
Toa do : 4 8

```

V.2. Hàm hủy bỏ - DESTRUCTOR

Một destructor là một hàm thành viên đặc biệt của một lớp. Tên của destructor của một lớp chính là tên lớp đó có thêm một ký tự ngã (~) ở phía trước.

Destructor của một lớp được gọi khi đối tượng được hủy bỏ nghĩa là khi sự thực hiện chương trình rời khỏi phạm vi mà trong đó đối tượng của lớp đó được khởi tạo. Destructor không thực sự hủy bỏ đối tượng – nó thực hiện "công việc dọn dẹp sạch sẽ" trước khi hệ thống phục hồi không gian bộ nhớ dành cho đối tượng để có thể dùng lưu giữ các đối tượng mới.

Ví dụ 3.7 : Lớp có hàm hủy bỏ

```

#include <iostream.h>
class Simple
{private:
    int *X;
public:
    Simple(); //Constructor
    ~Simple(); //Destructor
    void SetValue(int V);
    int GetValue();
};

```

```

Simple::Simple() {
    X = new int; //Cấp phát vùng nhớ cho X
}
Simple::~~Simple() {
    delete X; //Giải phóng vùng nhớ khi đối tượng bị hủy bỏ.
}
void Simple::SetValue(int V) {
    *X = V;
}
int Simple::GetValue() {
    return *X;
}
int main()
{ Simple S;
  int X;
  cout<<"Enter a number:";
  cin>>X;
  S.SetValue(X);
  cout<<"The value of this number:"<<S.GetValue();
  return 0;
}

```

```

Enter a number: 12
The value of this number: 12

```

➤ Một số lưu ý:

- Một destructor không nhận các tham số và không trả về giá trị.
- Hàm hủy bỏ phải có thuộc tính **public**.
- Một lớp chỉ có duy nhất một destructor – đa năng hóa destructor là không cho phép.
- Nếu trong một lớp không có định nghĩa một destructor thì trình biên dịch sẽ tạo một destructor mặc định không làm gì cả. Đối với các lớp không có khai báo các thành phần bộ nhớ động, có thể dùng hàm hủy bỏ mặc định. Trái lại, phải khai báo hàm hủy bỏ tường minh để đảm bảo quản lý tốt việc giải phóng bộ nhớ động do các đối tượng chiếm giữ khi chúng hết thời gian làm việc.

V.3. Hàm thiết lập sao chép (copy constructor)

Ta có thể khai báo một đối tượng và gán cho nó nội dung của một đối tượng cùng lớp đã tồn tại trước đó. Khi đó ta sẽ có 2 đối tượng cùng nội dung.

```

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

```

Như ta biết, 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. Trong khi hàm thiết lập thực hiện đồng thời hai nhiệm vụ: tạo đối tượng và sao chép nội dung từ đối tượng đã có sang đối tượng mới tạo ra đó.

Việc truyền đối tượng cho hàm bằng tham trị hoặc hàm trả về một đối tượng cũng dùng đến **hàm thiết lập sao chép** để tạo một đối tượng giống hệt một đối tượng cùng lớp đã có trước đó.

Giống như hàm thiết lập ngầm định, 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**.

➤ Khai báo và định nghĩa hàm thiết lập sao chép

Trong một số trường hợp cần thực hiện những thao tác theo yêu cầu khi thiết lập đối tượng, như trong trường hợp cấp phát động các thành phần dữ liệu chẳng hạn, ta cần phải định nghĩa hàm thiết lập sao chép tường minh. Dạng khai báo của hàm thiết lập sao chép cho lớp **Point** như sau:

```
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.

Ví dụ 3.8: Bổ sung thêm hàm thiết lập sao chép vào lớp **point**.

```
#include <conio.h>
#include <iostream.h>
class point {
    int x;
    int y;
public:
    //Hàm thiết lập
    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"; }
//hàm tịnh tiến điểm a thành điểm khác
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
Toa do : 5 2
Tao doi tuong : 0xffea
Dung ham thiet lap sao chep
Tao doi tuong : 0xffde
Dung ham thiet lap sao chep
Tao doi tuong : 0xffee
Dung ham thiet lap sao chep
Toa do : 7 5

```

VI. ĐỐI TƯỢNG HẲNG VÀ CÁC HÀM THÀNH VIÊN CONST

VI.1. Đối tượng hằng

Cũng như các phần tử dữ liệu khác, một vài đối tượng cần được thay đổi và một vài đối tượng thì không. Nếu muốn đối tượng không thể thay đổi được, có thể được khai báo là **hằng**. Trừ khi có các chỉ định cụ thể, phương thức duy nhất sử dụng các đối tượng hằng là các **hàm thiết lập** và **hàm huỷ bỏ**. Bởi lẽ các đối tượng hằng không thể thay đổi, mà chỉ được tạo ra hoặc huỷ bỏ đi. Ngoài hàm thiết lập và huỷ bỏ, các đối tượng hằng chỉ có thể gọi được các hàm thành phần được khai báo với từ khoá **const**. Dĩ nhiên các **hàm thành viên const** không thể thay đổi đối tượng - trình biên dịch cấm điều này.

```

class point {
    int x,y;
public:
    point(...);
    void display() const;
    void move(...);
};
.....
int main()
.....
const point p(2,4);

```

VI.2. Các hàm thành viên 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 bằng một hàm khác không phải **const**.

Lưu ý: Hàm thành viên **const** có thể được đa năng hóa với một phiên bản **non-const**. Việc lựa chọn hàm thành viên đa năng hóa nào để sử dụng được tạo một cách tự động bởi trình biên dịch dựa vào nơi mà đối tượng được khai báo **const** hay không.

Một đối tượng **const** không thể được thay đổi bởi phép gán vì thế nó phải được khởi động. Khi một thành viên dữ liệu của một lớp được khai báo **const**, một bộ khởi tạo thành viên (member initializer) phải được sử dụng để cung cấp cho **constructor** với giá trị ban đầu của thành viên dữ liệu đối với một đối tượng của lớp.

VII.LỚP NHƯ LÀ CÁC THÀNH VIÊN CỦA CÁC LỚP KHÁC

Một lớp có thể có các đối tượng của các lớp khác như các thành viên. Khi một đối tượng đi vào phạm vi, constructor của nó được gọi một cách tự động, vì thế chúng ta cần mô tả các tham số được truyền như thế nào tới các constructor của đối tượng thành viên. Các đối tượng thành viên được xây dựng theo thứ tự mà trong đó chúng được khai báo (không theo thứ tự mà chúng được liệt kê trong danh sách bộ khởi tạo thành viên của constructor) và trước các đối tượng của lớp chứa đựng chúng được xây dựng.

Ví dụ 3.9: Lớp **Date** là thành viên của lớp **Employee**.

```
#include <iostream.h>
#include <string.h>
class Date
{public:
    Date(int = 1, int = 1, int = 1900); //Constructor mặc định
    void Print() const; //In ngày theo dạng Month/Day/Year
private:
    int Month; //1-12
    int Day; //1-31
    int Year; //Năm bất kỳ
    //Hàm tiện ích để kiểm tra Day tương thích đối với Month và Year
    int CheckDay(int);
};

class Employee
{public:
```



```

Employee(char *, char *, int, int, int, int, int, int);
void Print() const;
private:
char LastName[25];
char FirstName[25];
Date BirthDate;
Date HireDate;
};
//Constructor: xác nhận giá trị tương thích của Month
//Gọi hàm CheckDay() để xác nhận giá trị tương thích của Day
Date::Date(int Mn, int Dy, int Yr)
{if (Mn > 0 && Mn <= 12)
    Month = Mn;
else
    {   Month = 1;
        cout << "Month " << Mn << " invalid. Set to Month 1."<< endl;    }
    Year = Yr;
    Day = CheckDay(Dy);
    cout << "Date object constructor for date ";
    Print();
    cout << endl;
}
//Hàm tiện ích để xác nhận giá trị Day tương thích dựa vào Month và Year
int Date::CheckDay(int TestDay)
{
    static int DaysPerMonth[13]={0, 31, 28, 31, 30, 31, 30, 31, 31, 30,31, 30, 31};
    if (TestDay > 0 && TestDay <= DaysPerMonth[Month])
        return TestDay;
    if (Month == 2 && TestDay == 29 && (Year % 400 == 0 || (Year % 4 == 0 && Year % 100 != 0)))
        return TestDay;
    cout << "Day " << TestDay << "invalid. Set to Day 1." << endl;
    return 1;
}
//In đối tượng Date dạng Month/Day/Year
void Date::Print() const
{ cout << Month << '/' << Day << '/' << Year; }
Employee::Employee(char *FName, char *LName,int BMonth, int BDay, int BYear,int HMonth, int HDay, int HYear)
    :BirthDate(BMonth, BDay, BYear), HireDate(HMonth, HDay, HYear)

```

```

{ //Sao chép FName vào FirstName và phải chắc chắn rằng nó phù hợp
    int Length = strlen(FName);
    Length = Length < 25 ? Length : 24;
    strncpy(FirstName, FName, Length);
    FirstName[Length] = '\0';
//Sao chép LName vào LastName và phải chắc chắn rằng nó phù hợp
    Length = strlen(LName);
    Length = Length < 25 ? Length : 24;
    strncpy(LastName, LName, 24);
    LastName[Length] = '\0';
    cout << "Employee object constructor: " << FirstName << " " << LastName << endl;
}

void Employee::Print() const
{cout << LastName << ", " << FirstName << endl << "Hired: ";
    HireDate.Print();
    cout << " Birthday: ";
    BirthDate.Print();
    cout << endl; }

int main()
{Employee E("Bob", "Jones", 7, 24, 49, 3, 12, 88);
    cout << endl;
    E.Print();
    cout << endl << "Test Date constructor with invalid values:" << endl;
    Date D(14, 35, 94); //Các giá trị Date không hợp lệ
    return 0;
}

```

Lưu ý: Một đối tượng thành viên không cần được khởi tạo thông qua một bộ khởi tạo thành viên. Nếu một bộ khởi tạo thành viên không được cung cấp, **constructor** mặc định của đối tượng thành viên sẽ được gọi một cách tự động.

VIII.CÁC THÀNH PHẦN TĨNH CỦA LỚP

VIII.1. Thành phần dữ liệu tĩnh

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. Tuy nhiên, có thể cho phép nhiều đối tượng cùng chia sẻ dữ liệu bằng cách đặt từ khóa **static** trước khai báo thành phần dữ liệu tương ứng.

Ví dụ: khai báo lớp **exple** với thành phần dữ liệu tĩnh và tạo 2 đối tượng **a** và **b**.

```

class exple {
    static int n; // n là thành phần dữ liệu tĩnh
    float x;

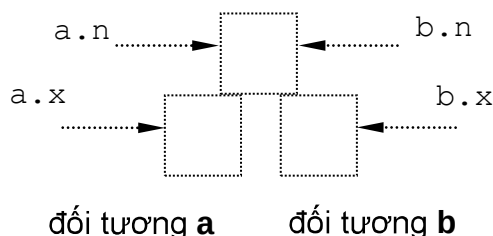
```

```

.... };
int exple::n=5; //khởi tạo thành viên dữ liệu tĩnh
.....
exple a,b; //tạo 2 đối tượng a và b
.....

```

Khi đó 2 đối tượng **a** và **b** tạo ra sẽ có chung thành phần dữ liệu **n**



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 thành viên dữ liệu tĩnh phải được khởi tạo một lần (và chỉ một lần) tại phạm vi file. Tức là chúng được một cách tường minh bên ngoài khai báo lớp. Cú pháp khởi tạo như sau:

```
<kiểu của thành phần dữ liệu> <tên lớp>::<tên thành phần dữ liệu>=<giá trị>;
```

Các thành viên tĩnh có thể là **public**, **private** hoặc **protected**. Các thành viên lớp tĩnh **public** có thể được truy cập thông qua bất kỳ đối tượng nào của lớp đó, hoặc chúng có thể được truy cập thông qua tên lớp sử dụng toán tử định phạm vi. Các thành viên lớp tĩnh **private** và **protected** phải được truy cập thông qua các hàm thành viên **public** của lớp hoặc thông qua các **friend** của lớp. Các thành viên lớp tĩnh tồn tại ngay cả khi đối tượng của lớp đó không tồn tại. Để truy cập một thành viên lớp tĩnh **public** khi các đối tượng của lớp không tồn tại, đơn giản thêm vào đầu tên lớp và toán tử định phạm vi cho thành viên dữ liệu. Để truy cập một thành viên lớp tĩnh **private** hoặc **protected** khi các đối tượng của lớp không tồn tại, phải dùng đến một hàm thành viên **public** và hàm phải được gọi với cú pháp;

```
<tên lớp>::<tên hàm thành viên public>(<tham số nếu có>;
```

VIII.2. Các hàm thành phần tĩnh

Một hàm thành phần được khai báo bắt đầu với từ khoá **static** được gọi là hàm thành phần **tĩnh**, hàm thành phần **static** cũng độc lập với bất kỳ đối tượng nào của lớp. Hàm thành phần **static** không có tham số ngầm định. Có thể gọi hàm thành phần **static** cho dù đối tượng của lớp đó không tồn tại. 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**.

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

Ví dụ 3.10: dùng thành phần dữ liệu **static** để đếm số đối tượng đang sử dụng

```
#include <iostream.h>
#include <conio.h>
class counter {
    static int count; //đếm số đối tượng được tạo ra
public :
    counter (); //constructor
    ~ counter (); //destructor
    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 ()
{ cout<<"--Xoa : bay gio con "<<--count<<" doi tuong \n"; }
void fct();
void main() {
    clrscr();
    counter a;
    fct();
    counter::counter_display();
    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
```

IX. CÁC HÀM VÀ CÁC LỚP BẠN

Khi định nghĩa một lớp, có thể khai báo một hay nhiều hàm “bạn” (bên ngoài lớp); các hàm này truy xuất được tới các thành phần **private** của lớp giống như các hàm thành phần của lớp đó. Có nhiều kiểu bạn bè mà chúng ta sẽ lần lượt nghiên cứu sau đây.

IX.1. Hàm tự do bạn của một lớp

Trong khai định nghĩa lớp phải có khai báo bạn bè

```
friend <kiểu> <tên hàm tự do>(tham số nếu có);
```

Ví dụ 3.11: hàm tự do là bạn của lớp

```
#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); // khai báo hàm coincide là bạn bè của lớp
};
int coincide (point p, point q) { //định nghĩa hàm coincide
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
```

IX.2. Hàm thành phần của lớp là bạn của lớp khác

Tương tự như trường hợp trên, tuy nhiên tên hàm phải ghi đầy đủ, bao gồm tên lớp, toán tử phạm vi và tên hàm thành phần bạn bè.

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

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

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

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

Để biên dịch được các khai báo của lớp A có chứa khai báo bạn bè như trên, 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 hàm f bên trong lớp B, chương trình dịch không nhất thiết phải biết chi tiết nội dung của A, nó chỉ cần biết đó là một lớp. Như vậy ta cần 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 ...) {
...
}
```

Lưu ý: Tất cả các hàm tự do, hay hàm thành phần của một lớp đều có thể là bạn của nhiều lớp khác nhau.

IX.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 bạn bè 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; //chỉ thị B là lớp bạn
```

Lưu ý: 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.

CHƯƠNG 4

ĐỊNH NGHĨA TOÁN TỬ TRÊN LỚP**SƠ LƯỢC VỀ HÀM TOÁN TỬ**

Trong chương 2 chúng ta đã tìm hiểu về khái niệm Đa năng hoá toán tử trong lập trình C++, định nghĩa toán tử trên lớp ở đây chính là Đa năng hoá toán tử áp dụng trên lớp. Như ta đã biết, việc thực hiện Đa năng hoá toán tử được thực tiến hành bằng cách định nghĩa các hàm toán tử. Tương tự đối với lớp, chúng ta cũng có thể Đa năng hoá toán tử bằng cách định nghĩa các hàm thành phần hoặc các hàm tự do. Trong chương này, chúng ta sẽ tìm hiểu cách thức định nghĩa Hàm toán tử trên lớp để cho phép các toán tử của C++ làm việc với các đối tượng của lớp.

Thông thường các thao tác trên đối tượng của lớp được thực hiện bởi gọi các thông điệp (dưới dạng các lời gọi hàm thành viên) tới các đối tượng. Phương pháp gọi hàm này thì không gọn gàng cho một số lớp nhất định, đặc biệt là các lớp toán học. Đối với các loại lớp này sẽ gọn gàng hơn nếu sử dụng tập các toán tử có sẵn phong phú của C++ để chỉ rõ các thao tác của đối tượng. Ví dụ khi định nghĩa một lớp **sophuc** (số phức) để biểu diễn các số phức, có thể viết: $a+b$, $a-b$, $a*b$, a/b với a, b là các đối tượng **sophuc**. 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).

Các nguyên tắc cơ bản

C++ không cho phép tạo ra các toán tử mới, mà cho phép đa năng hoá các toán tử đã tồn tại sao cho khi các toán tử này được sử dụng với các đối tượng của lớp, các toán tử có ý nghĩa thích hợp các kiểu mới.

Các toán tử được đa năng hóa bằng cách viết một định nghĩa hàm (bao gồm phần đầu và thân) như khi chúng ta viết một hàm bình thường, ngoại trừ tên hàm bây giờ trở thành từ khóa **operator** theo sau bởi **ký hiệu của toán tử** được đa năng hóa. Cú pháp của nó có dạng như sau:

<kiểu> operator <ký hiệu toán tử> (danh sách tham số);

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 :

1. 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.
2. 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. Và khi đó hàm toán tử phải được khai báo là bạn (**friend**) của các lớp có các đối tượng mà hàm thao tác.

Để sử dụng một toán tử đối với các đối tượng của một lớp, cần phải đa năng hoá toán tử đó cho lớp tương ứng ngoại trừ hai trường hợp sau:

1. Toán tử gán có thể sử dụng với mọi lớp mà không cần đa năng hóa. Cách cư xử mặc định của toán tử gán là phép gán các thành viên dữ liệu của lớp. Đối với các lớp có thành phần cấp phát động thì việc sao chép thành viên mặc định sẽ rất nguy hiểm. Khi đó, chúng ta sẽ đa năng hóa một cách tường minh toán tử gán đối với các lớp như thế.
2. Toán tử địa chỉ (&) cũng có thể được sử dụng với các đối tượng của bất kỳ lớp nào mà không cần đa năng hóa; Nó trả về địa chỉ của đối tượng trong bộ nhớ. Toán tử địa chỉ cũng có thể được đa năng hóa như các toán tử khác.

Không nên định nghĩa những hàm toán tử khác nhau cùng làm những công việc giống nhau vì dễ xảy ra nhập nhằng.

Những giới hạn

Phần lớn các toán tử đều có thể Đa năng hoá và chỉ có một số toán tử không thể đa năng hoá, chẳng hạn như: toán tử truy nhập thành phần cấu trúc “.”, toán tử phạm vi “::”, toán tử điều kiện “?:” (đã liệt kê chi tiết ở chương 2).

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. Có một số toán tử cần phải tuân theo các ràng buộc như sau:

1. phép =, [] nhất định phải được định nghĩa như hàm thành phần của lớp.
2. phép << và >> dùng với **cout** và **cin** phải được định nghĩa như hàm bạn.
3. 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--. Như vậy phải có hai hàm toán tử khác nhau.

Hàm cho dạng tiền tố	Hàm cho dạng hậu tố
operator++()	operator++(int)
operator--()	operator--(int)

Lưu ý rằng tham số **int** trong dạng hậu tố chỉ mang ý nghĩa tượng trưng (dump type)

Các toán tử được đa năng hoá 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 (phép đảo dấu) và hai ngôi (phép trừ số học) trên lớp tương ứng, 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.

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 tùy tiện cho các toán tử được đa năng hoá. Tính kết hợp của một toán tử không thể được thay đổi bởi đa năng hóa.

Thứ tự ưu tiên của một toán tử không thể được thay đổi bởi đa năng hóa. Trong trường hợp độ ưu tiên của toán tử không còn phù hợp, chúng ta có thể sử dụng các dấu ngoặc đơn để đặt thứ tự thực hiện của các toán tử đã đa năng hóa trong một biểu thức.

Không thể đa năng hoá để thay đổi ý nghĩa của toán tử trên các đối tượng của các kiểu có sẵn. Việc đa năng hóa toán tử chỉ làm việc với các đối tượng của các kiểu do người dùng định

nghĩa hoặc với một sự pha trộn của một đối tượng của kiểu do người dùng định nghĩa và một đối tượng của một kiểu có sẵn.

Các tham số mặc định không thể sử dụng với một hàm toán tử.

CHIẾN LƯỢC SỬ DỤNG HÀM TOÁN TỬ

Việc đa năng hoá một phép toán là khá đơn giản, nhưng việc sử dụng phép toán đa năng hoá lại không đơn giản chút nào, mà cần phải được 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 thủy 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: *, &, ~, !, ++, --, 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: *, /, %, +, -, <<, >>, <, >, <=, >=, ==, !=, &, |, ^, &&, ||

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: =, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=

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 con trỏ cấu trúc: ->

Phép toán này được dùng để truy xuất các thành phần của đối tượng thuộc một cấu trúc hay một lớp khi mà một con trỏ trỏ vào đối tượng đó. Có thể định nghĩa phép toán -> giống như đối 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 tọa độ nào đó **vector**. Khi đa năng hoá phải được định nghĩa như hàm thành viên có một đối số tường minh.

➤ Toán tử gọi hàm: ()

Toán tử **()** được dùng để gọi hàm, gồm có hai toán hạng, toán hạng thứ nhất là tên hàm, toán hạng thứ hai là danh sách tham số. Toán tử này có dạng giống như toán tử **[]** và khi đa năng toán tử này thì hàm toán tử tương ứng phải là thành viên của một lớp.

ĐỊNH NGHĨA TOÁN TỬ HAI NGÔI TRÊN LỚP

Các toán tử hai ngôi được đa năng hoá cho ở bảng kèm theo sau đây:

Toán tử	Ví dụ	Toán tử	Ví dụ	Toán tử	Ví dụ
+	a+b	+=	a+=b	<<=	a<<=b
-	a-b	-=	a-=b	==	a==b
*	a*b	*=	a*=b	!=	a!=b
/	a/b	/=	a/=b	<=	a<=b
%	a%b	%=	a%=b	>=	a>=b
^	a^b	^=	a^=b	&&	a&& b
&	a&b	&=	a&=b		a b
	a b	=	a =b	,	a, b
=	a=b	<<	a<<b	[]	a[b]
<	a>	a>>b	->*	a->*b
>	a>b	>>=	a>>=b		

Hình 4.1: Các toán tử hai ngôi được đa năng hóa

Trường hợp định nghĩa hàm toán tử là hàm thành viên, ít nhất toán hạng đầu tiên của toán hạng là đối tượng thuộc lớp đang định nghĩa và hàm chỉ có 1 tham số. Cú pháp hàm như sau:

<kiểu> operator <ký hiệu toán tử> (<toán hạng thứ hai>);

Trường hợp định nghĩa hàm toán tử là hàm tự do, hàm phải có đầy đủ hai tham số. Cú pháp hàm như sau:

<kiểu> operator <ký hiệu toán tử> (<toán hạng thứ nhất, toán hạng thứ hai>);

Ví dụ 4.1: Xây dựng lớp số phức với tên lớp là Complex và đa năng hóa các toán tử tính toán + - += -= và các toán tử so sánh == != > >= < <= bằng cách dùng các hàm thành viên.

```
#include <iostream.h>
#include <math.h>
#include <conio.h>
class Complex
{private:
    double Real, Imaginary;
public:
    Complex(); // Constructor mặc định
    Complex(double R,double I);
    Complex (const Complex & Z); // Constructor sao chép
    Complex (double R); // Constructor chuyển đổi
```

```
void Print(); // Hiển thị số phức
// Các toán tử tính toán
Complex operator + (Complex Z);
Complex operator - (Complex Z);
Complex operator += (Complex Z);
Complex operator -= (Complex Z);
// Các toán tử so sánh
int operator == (Complex Z);
int operator != (Complex Z);
int operator > (Complex Z);
int operator >= (Complex Z);
int operator < (Complex Z);
int operator <= (Complex Z);
private:
double Abs(); // Giá trị tuyệt đối của số phức
};

Complex::Complex()
{ Real = 0.0;
  Imaginary = 0.0; }
Complex::Complex(double R, double I)
{ Real = R;
  Imaginary = I; }
Complex::Complex(const Complex & Z)
{ Real = Z.Real;
  Imaginary = Z.Imaginary; }
Complex::Complex(double R)
{ Real = R;
  Imaginary = 0.0; }
void Complex::Print()
{ cout<<'('<<Real<<','<<Imaginary<<')'; }
Complex Complex::operator + (Complex Z)
{ Complex Tmp;
  Tmp.Real = Real + Z.Real;
  Tmp.Imaginary = Imaginary + Z.Imaginary;
  return Tmp; }
Complex Complex::operator - (Complex Z)
{ Complex Tmp;
  Tmp.Real = Real - Z.Real;
  Tmp.Imaginary = Imaginary - Z.Imaginary;
  return Tmp; }
```

```

Complex Complex::operator += (Complex Z)
{ Real += Z.Real;
  Imaginary += Z.Imaginary;
  return *this; }
Complex Complex::operator -= (Complex Z)
{ Real -= Z.Real;
  Imaginary -= Z.Imaginary;
  return *this; }
int Complex::operator == (Complex Z)
{ return (Real == Z.Real) && (Imaginary == Z.Imaginary); }
int Complex::operator != (Complex Z)
{ return (Real != Z.Real) || (Imaginary != Z.Imaginary); }
int Complex::operator > (Complex Z)
{ return Abs() > Z.Abs(); }
int Complex::operator >= (Complex Z)
{ return Abs() >= Z.Abs(); }
int Complex::operator < (Complex Z)
{ return Abs() < Z.Abs(); }
int Complex::operator <= (Complex Z)
{ return Abs() <= Z.Abs(); }
double Complex::Abs()
{ return sqrt(Real*Real+Imaginary*Imaginary); }
int main()
{
  clrscr();
  Complex X, Y(4.3,8.2), Z(3.3,1.1), T;
  cout<<"X: ";
  X.Print();
  cout<<endl<<"Y: ";
  Y.Print();
  cout<<endl<<"Z: ";
  Z.Print();
  cout<<endl<<"T: ";
  T.Print();
  T=5.3;// Gọi constructor chuyển đổi
  cout<<endl<<endl<<"T = 5.3"<<endl;
  cout<<"T: ";
  T.Print();
  X = Y + Z;
  cout<<endl<<endl<<"X = Y + Z: ";

```

```

X.Print();
cout<<" = ";
Y.Print();
cout<<" + ";
Z.Print();
X = Y - Z;
cout<<endl<<"X = Y - Z: ";
X.Print();
cout<<" = ";
Y.Print();
cout<<" - ";
Z.Print();
cout<<endl<<endl<<"Y += T i.e ";
Y.Print();
cout<<" += ";
T.Print();
Y += T;
cout<<endl<<"Y: ";
Y.Print();
cout<<endl<<"Z -= T i.e ";
Z.Print();
cout<<" -= ";
T.Print();
Z -= T;
cout<<endl<<"Z: ";
Z.Print();
Complex U(X); // Gọi constructor sao chép
cout<<endl<<endl<<"U: ";
U.Print();
cout<<endl<<endl<<"Evaluating: X==U"<<endl;
if (X==U) cout<<"They are equal"<<endl;
cout<<"Evaluating: Y!=Z"<<endl;
if (Y!=Z) cout<<"They are not equal => ";
if (Y>Z) cout<<"Y>Z";
else      cout<<"Y<Z";
getch();
return 0;
}

```

```

X: (0,0)
Y: (4.3,8.2)
Z: (3.3,1.1)

```

```

T: (0,0)

T = 5.3
T: (5.3,0)

X = Y + Z: (7.6,9.3) = (4.3,8.2) + (3.3,1.1)
X = Y - Z: (1,7.1) = (4.3,8.2) - (3.3,1.1)

Y += T i.e (4.3,8.2) += (5.3,0)
Y: (9.6,8.2)
Z -= T i.e (3.3,1.1) -= (5.3,0)
Z: (-2,1.1)

U: (1,7.1)

Evaluating: X==U
They are equal
Evaluating: Y!=Z
They are not equal => Y>Z

```

Ví dụ 4.2: Xây dựng lớp số phức với tên lớp là Complex và đa năng hóa các toán tử tính toán + và toán tử so sánh == bằng cách dùng các hàm tự do và được định nghĩa là hàm bạn trong khai báo lớp.

```

#include <iostream.h>
#include <math.h>
#include <conio.h>
class Complex
{private:
    double Real,Imaginary;
public:
    Complex ();
    Complex (double R,double I);
    void Print(); // Hiện thị số phức
    friend Complex operator + (Complex Z1,Complex Z2);
    friend int operator == (Complex Z1,Complex Z2);
};

Complex::Complex()
{
    Real = 0.0;
    Imaginary = 0.0; }
Complex::Complex(double R,double I)
{ Real = R;
    Imaginary = I; }
void Complex::Print()
{ cout<<'('<<Real<<','<<Imaginary<<')'; }
Complex operator + (Complex Z1,Complex Z2)
{ Complex Tmp;

```

```

Tmp.Real = Z1.Real + Z2.Real;
Tmp.Imaginary = Z1.Imaginary + Z2.Imaginary;
return Tmp; }

int operator == (Complex Z1, Complex Z2)
{ return (Z1.Real == Z2.Real) && (Z1.Imaginary == Z2.Imaginary); }

int main()
{
clrscr();
Complex X,Y(4.3,8.2),Z(3.3,1.1);
cout<<"X: ";
X.Print();
cout<<endl<<"Y: ";
Y.Print();
cout<<endl<<"Z: ";
Z.Print();
X = Y + Z;
cout<<endl<<endl<<"X = Y + Z: ";
X.Print();
cout<<" = ";
Y.Print();
cout<<" + ";
Z.Print();
getch();
return 0;
}

```

```

X: (0,0)
Y: (4.3,8.2)
Z: (3.3,1.1)

```

```

X = Y + Z: (7.6,9.3) = (4.3,8.2) + (3.3,1.1)

```

ĐỊNH NGHĨA TOÁN TỬ MỘT NGÔI TRÊN LỚP

Các toán tử một ngôi được đa năng hoá cho ở bảng kèm thao sau đây:

Toán tử	Ví dụ	Toán tử	Ví dụ
+	+c	~	~c
-	-c	!	!a
*	*c	++	++c, c++

&	&c	--	--c, c--
->	c->		

Hình 4.2: Các toán tử một ngôi được đa năng hóa

Một toán tử một ngôi của lớp được đa năng hóa bằng một hàm thành viên không tĩnh không có tham số, cú pháp như sau:

<kiểu> operator <ký hiệu toán tử> ();

hoặc bằng một hàm tự do có một tham số, cú pháp như sau:

<kiểu> operator <ký hiệu toán tử> (<toán hạng>);

Lưu ý: Tham số phải hoặc là một đối tượng lớp hoặc là một tham chiếu đến đối tượng lớp.

Ví dụ 4.2: Xây dựng lớp số phức với tên lớp là Complex và đa năng hóa các toán tử - (đổi dấu) bằng cách dùng hàm thành phần trong khai báo lớp.

```
#include <iostream.h>
#include <math.h>
#include <conio.h>
class Complex
{
private:
double Real, Imaginary;
public:
Complex();
Complex(double R, double I);
void Print(); // Hiện thị số phức
Complex operator -();
};
Complex::Complex()
{ Real = 0.0;
Imaginary = 0.0; }
Complex::Complex(double R, double I)
{ Real = R;
Imaginary = I; }
void Complex::Print()
{ cout<<'('<<Real<<','<<Imaginary<<')'; }
Complex Complex::operator -()
{ Complex Tmp;
Tmp.Real = -Real;
Tmp.Imaginary = -Imaginary;
return Tmp; }
```



```
int main()
{
    clrscr();
    Complex X(4.3,8.2);
    cout<<"X: ";
    X.Print();
    cout<<endl;
    cout<<"-X: ";
    (-X).Print();
    getch();
    return 0;
}
```

```
X: (4.3,8.2)
-X: (-4.3,-8.2)
```

ĐỊNH NGHĨA MỘT SỐ TOÁN TỬ KHÁC TRÊN LỚP

Toán tử []

Toán tử [] để truy cập đến một phần tử của mảng hay chuỗi. Đây là toán tử hai ngôi, có dạng $a[b]$ và khi đa năng toán tử này thì hàm toán tử tương ứng phải là thành viên của một lớp.

Ví dụ 4.3: Đa năng hóa toán tử [] để truy cập đến một phần tử của vector.

```
#include <iostream.h>
#include <conio.h>
class Vector
{private:
    int Size;
    int *Data;
public:
    Vector(int S=2,int V=0);
    ~Vector();
    void Print() const;
    int & operator [] (int I); //hàm trả về 1 tham chiếu để có thể sử dụng như 1 biến
};
Vector::Vector(int S,int V)
{ Size = S;
  Data=new int[Size]; // cấp phát động cho vector
  for(int I=0;I<Size;++I)
    Data[I]=V;}
Vector::~~Vector()
```

```

{ delete []Data;} // thu hồi bộ nhớ động
void Vector::Print() const
{ cout<<"Vector: (";
  for(int I=0;I<Size-1;++I)
    cout<<Data[I]<<" ";
  cout<<Data[Size-1]<<" "<<endl;}
int & Vector::operator [] (int I)
{ return Data[I]; }
int main()
{ clrscr();
  Vector V(5,1);
  V.Print();
  for(int I=0;I<5;++I)
    V[I]*=(I+1); //sử dụng hàm như một biến
  V.Print();
  V[0]=10;
  V.Print();
  getch();
  return 0;}

```

```

Vector: (1,1,1,1,1)
Vector: (1,2,3,4,5)
Vector: (10,2,3,4,5)

```

Toán tử ()

Toán tử **()** được dùng để gọi hàm, toán tử này gồm hai toán hạng: toán hạng đầu tiên là tên hàm, toán hạng thứ hai là danh sách các tham số của hàm. Gần tương tự như toán tử **[]**, đa năng toán tử này bằng hàm thành viên của một lớp.

Ví dụ 4.4: Điều chỉnh Ví dụ 4.3 bằng cách Đa năng hóa toán tử **()** để truy cập đến một phần tử của vector thay cho toán tử **[]**.

```

#include <iostream.h>
#include <conio.h>
class Vector
{private:
  int Size;
  int *Data;
public:
  Vector(int S=2,int V=0);
  ~Vector();
  void Print() const;
  int & operator () (int I); //hàm trả về 1 tham chiếu để có thể sử dụng như 1 biến

```

```
};
Vector::Vector(int S,int V)
{ Size = S;
  Data=new int[Size]; // cấp phát động cho vector
  for(int I=0;I<Size;++I)
    Data[I]=V;}
Vector::~~Vector()
{ delete []Data;} //thu hồi bộ nhớ động
void Vector::Print() const
{ cout<<"Vector: (";
  for(int I=0;I<Size-1;++I)
    cout<<Data[I]<<" ";
  cout<<Data[Size-1]<<" "<<endl;}
int & Vector::operator ()(int I)
{ return Data[I]; }
int main() // kết quả tương tự như ví dụ 4.3
{ clrscr();
  Vector V(5,1);
  V.Print();
  for(int I=0;I<5;++I)
    V(I)*=(I+1); //sử dụng hàm như một biến
  V.Print();
  V(0)=10;
  V.Print();
  getch();
  return 0;}
```

Toán tử gán =

Chúng ta có thể đa năng hóa toán tử gán “=” bằng hàm toán tử là thành viên của lớp. Toán tử này có 2 toán hạng dạng a=b, khi đó hàm toán tử có dạng a.operator(b).

Ví dụ 4.5: Đa năng hóa toán tử gán.

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class String
{private:
  char *St;
  int Len;
public:
  String(char *S);
```

```

~String();
char *GetStr();
String & operator=(String &Obj);
};
String::String(char *S)
{Len=strlen(S);
 St=new char[Len+1];
 strcpy(St,S);}
String::~~String()
{ delete []St;}
char * String::GetStr()
{ return St;}
String & String::operator=(String &Obj)
{ if (Len< Obj.Len) { delete [] St;
                      St=new char[Obj.Len+1]; }

 Len=Obj.Len;
 strcpy(St,Obj.St);
 return *this; }
int main()
{clrscr();
 String S1("Hello"), S2("Chao");
 cout<<"S1="<<S1.GetStr()<<endl;
 cout<<"S2="<<S2.GetStr()<<endl;
 S2=S1;
 cout<<"S1="<<S1.GetStr()<<endl;
 cout<<"S2="<<S2.GetStr()<<endl;
 getch();
 return 0; }

```

```

S1=Hello
S2=Chao
S1=Hello
S2=Hello

```

Toán tử ++ và --

Các toán tử ++ và – được thực hiện ở hai dạng tiền tố và hậu tố. Để có thể phân biệt được tiền tố và hậu tố khi đa năng hóa, toán tử ++a (hoặc --a) là toán tử một ngôi, còn toán tử a++ (hoặc a--) được xem là toán tử hai ngôi trong đó toán hạng thứ hai kiểu **int**.

Ví dụ 4.6: Đa năng hóa toán tử ++ và – có phân biệt tiền tố và hậu tố.

```

#include <iostream.h>
#include <conio.h>
class Point

```

```
{private:
    int X,Y;
public:
    Point(int A=0,int B=0);
    Point operator ++();
    Point operator ++(int); //Hậu tố
    Point operator --(); //Tiền tố
    Point operator --(int); //Hậu tố
    void Print() const;
};

Point::Point(int A,int B)
{X = A;
 Y = B;}

Point Point::operator++()
{ ++X;
 ++Y;
 return *this; }

Point Point::operator++(int)
{ Point Tmp=*this; //Sao chép đối tượng đang xét vào đối tượng tạm
 ++X;
 ++Y;
 return Tmp; }

Point Point::operator--()
{ --X;
 --Y;
 return *this; }

Point Point::operator--(int)
{ Point Tmp=*this; //Sao chép đối tượng đang xét vào đối tượng tạm
 --X;
 --Y;
 return Tmp; }

void Point::Print() const
{ cout<<"X="<<X<<" ,Y="<<Y<<endl; }

int main()
{clrscr();
 Point P1(2,6),P2(5,8),P3,P4;
 cout<<"Point 1:";
 P1.Print();
 cout<<"Point 2:";
 P2.Print();
```

```

cout<<"Point 3:";
P3.Print();
cout<<"Point 4:";
P4.Print();
P3=++P1;
P4=P2++;
cout<<"Point 1:";
P1.Print();
cout<<"Point 2:";
P2.Print();
cout<<"Point 3:";
P3.Print();
cout<<"Point 4:";
P4.Print();
getch();
return 0; }

```

```

Point 1:X=2,Y=6
Point 2:X=5,Y=8
Point 3:X=0,Y=0
Point 4:X=0,Y=0
Point 1:X=3,Y=7
Point 2:X=6,Y=9
Point 3:X=3,Y=7
Point 4:X=5,Y=8

```

Toán tử ->

Toán tử dấu mũi tên -> (hoặc toán tử con trỏ cấu trúc) cho phép chúng ta truy cập thành viên của đối tượng thuộc class, struct và union khi được áp dụng vào con trỏ chỉ đến đối tượng đó. Đa năng hóa toán tử này bằng cách dùng hàm toán tử -> là hàm thành viên của lớp. Nếu toán tử -> có dạng a-> thì hàm toán tử là a.operator->()

Ví dụ 4.7: Đa năng hóa toán tử ->.

```

#include <iostream.h>
#include <conio.h>
class MyClass
{public:
    int Data;
    MyClass * operator ->()
    { return this; }
};
int main()
{clrscr();
    MyClass M;

```

```
M->Data = 10;
cout<<M.Data<<" "<<M->Data;
getch();
return 0;}
```

```
10 10
```

Toán tử new và delete

Nếu muốn toán tử **new** và **delete** có tính chất đặc biệt chỉ khi áp dụng cho đối tượng của lớp nào đó, chúng ta đã nâng hóa toán tử **new** và **delete** với tư cách là hàm thành viên của lớp đó.

Ví dụ 4.8: Đa năng hóa toán tử new và delete cho một lớp.

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
class Number
{private:
    int Data;
public:
    Number(int X=0)
    { Data=X; }
    void * operator new(size_t Size)
    { cout<<"Toan tu new cua lop!"<<endl;
      return ::new unsigned char[Size]; }
    void operator delete(void *Ptr)
    { cout<<"Toan tu delete cua lop!"<<endl;
      ::delete Ptr; }
    void Print() const
    { cout<<"Data:"<<Data<<endl; }
};
int main()
{clrscr();
  Number *N;
  N= new Number(10);
  if (N==NULL) { cout<<"Out of memory!"<<endl; return 1; }
  int *X=new int;
  if (X==NULL) { cout<<"Out of memory!"<<endl; return 1; }
  *X=10;
  cout<<"X="<<*X<<endl;
  N->Print();
  delete N;
  delete X;
```

```
getch();
return 0; }
```

```
Toan tu new cua lop!
X=10
Data:10
Toan tu delete cua lop!
```

Toán tử chèn dòng << và trích dòng >>

Hàm toán tử << trả về tham chiếu chỉ đến dòng xuất ostream. Tham số thứ nhất của hàm toán tử << là một tham chiếu chỉ đến dòng xuất ostream, tham số thứ hai là đối tượng được chèn vào dòng. Khi sử dụng, dòng trao cho toán tử << (tham số thứ nhất) là toán hạng bên trái và đối tượng được đưa vào dòng (tham số thứ hai) là toán hạng bên phải. Để bảo đảm cách dùng toán tử << luôn nhất quán, chúng ta không thể định nghĩa hàm toán tử << như là hàm thành viên của lớp đang xét, thông thường nó chính là hàm friend và có cú pháp như sau:

ostream & operator << (ostream & stream, ClassName Object);

Hàm toán tử >> trả về tham chiếu chỉ đến dòng nhập istream. Tham số thứ nhất của hàm toán tử này là một tham chiếu chỉ đến dòng nhập istream, tham số thứ hai là đối tượng của lớp đang xét mà chúng ta muốn tạo dựng nhờ vào dữ liệu lấy từ dòng nhập. Khi sử dụng, dòng nhập đóng vai toán hạng bên trái, đối tượng nhận dữ liệu đóng vai toán hạng bên phải. Cũng như trường hợp toán tử <<, hàm toán tử >> không là hàm thành viên của lớp, thông thường nó chính là hàm friend và có cú pháp như sau:

istream & operator >> (istream & stream, ClassName Object);

Ví dụ 4.9: Đa năng hoá toán tử << và >>

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

class Point
{private:
    int X,Y;
public:
    Point();
    friend ostream & operator << (ostream & Out,Point & P);
    friend istream & operator >> (istream & In,Point & P);
};

Point::Point()
{ X=Y=0; }

ostream & operator << (ostream & Out,Point & P)
{ Out<<"X="<<P.X<<" ,Y="<<P.Y<<endl;
  return Out; //Cho phép cout<<a<<b<<c;
}

istream & operator >> (istream &In,Point & P)
{ cout<<"X:";
```



```
In>>P.X;
cout<<"Y:";
In>>P.Y;
return In; //Cho phép cin>>a>>b>>c;
}
int main()
{clrscr();
  Point P;
  cin>>P;
  cout<<"Point:"<<P;
  getch();
  return 0; }
```

```
X:12
Y:5
Point:X=12,Y=5
```

KỸ THUẬT THỪA KẾ

I. SỰ DẪN XUẤT VÀ TÍNH KẾ THỪA

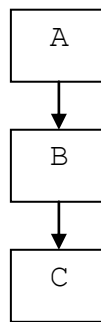
I.1. Lớp cơ sở và lớp dẫn xuất

Một lớp được xây dựng thừa kế một lớp khác gọi là lớp dẫn xuất. Lớp dùng để xây dựng lớp dẫn xuất gọi là lớp cơ sở.

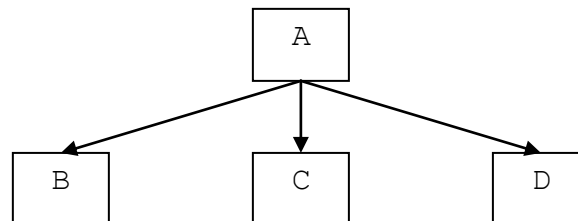
Lớp nào cũng có thể là lớp cơ sở và 1 lớp có thể là lớp cơ sở của nhiều lớp dẫn xuất khác nhau. Một lớp dẫn xuất cũng có thể là một lớp cơ sở để xây dựng các lớp dẫn xuất khác.

Dưới đây là một số sơ đồ về quan hệ dẫn xuất của các lớp:

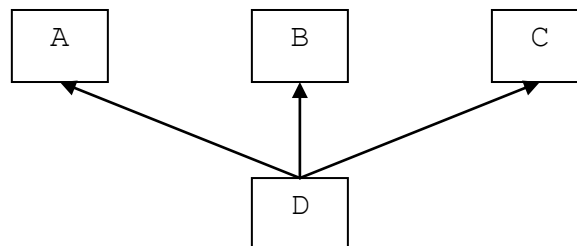
Sơ đồ 1: Lớp B dẫn xuất từ lớp A, lớp C dẫn xuất từ lớp B



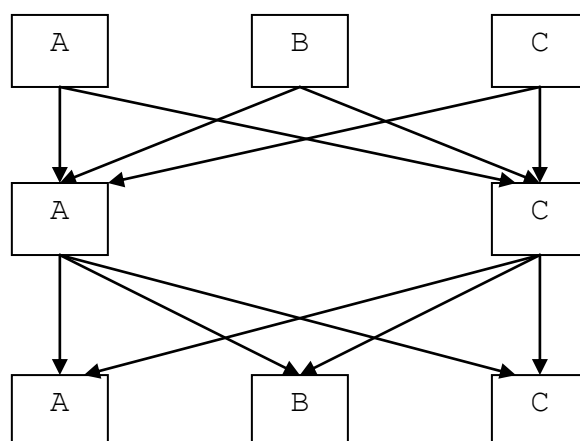
Sơ đồ 2: Lớp A là lớp cơ sở của lớp B, C, D



Sơ đồ 3: Lớp D dẫn xuất từ 3 lớp A, B, C



Sơ đồ 4: Sơ đồ dẫn xuất tổng quát



Tính kế thừa: Một lớp dẫn xuất, ngoài các thành phần của riêng nó, nó còn được kế thừa tất cả các thành phần của các lớp cơ sở có liên quan. Nếu lớp B được kế thừa từ lớp A thì toàn bộ các thành phần của lớp A cũng có mặt trong lớp B mà không cần phải khai báo trong lớp B. Lớp A chính là lớp cơ sở hay lớp cha, lớp B là lớp dẫn xuất hay lớp con.

Khai báo: Cách khai báo 1 lớp dẫn xuất trong C++

```

class A
{
    //Khai báo các thành phần của lớp A
};

class B: public A
{
    //Khai báo các thành phần của lớp B
};
  
```

I.2. Đơn thừa kế

I.2.1. Ví dụ về đơn thừa kế

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 nói 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 (đề) 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; //tọa độ điểm
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 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();
    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();
}
```

```
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()
```

1.2.2. Thừa kế **private** và **public**

- Nếu thừa kế **public** thì tất cả các thành phần **public** của lớp cơ sở cũng là các thành phần của lớp dẫn xuất
- Nếu thừa kế **private** thì tất cả các thành phần **public** của lớp cơ sở sẽ trở thành các thành phần **private** của lớp dẫn xuất

Chú ý: C++ mặc định thừa kế **private**

```
class C: public A, B
{
    //khai báo thành phần lớp C
};
```

Trong ví dụ trên, lớp A là lớp cơ sở **public** của C, còn lớp B là lớp cơ sở **private** của C

1.2.3. 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 cơ sở, 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.

1.2.4. 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 chồng 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 chồng 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.

1.3. Tính thừa kế trong lớp dẫn xuất

1.3.1. 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 `inher11.cpp` ta thấy rằng: đối tượng `m` của lớp `coloredpoint` có đồng thời hai thành phần tọa 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();
coloredpointcol pc(2,3,5);
```

câu lệnh

```
p=pc;
p.display();
pc.display();
```

cho kết quả

```
Diem p khong mau
Goi ham point::display()
Toa do :0 0
Diem pc co mau
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
p =pc
Goi ham point::display()
Toa do :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 chồng 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).

I.3.2. Thừa kế các thành phần dữ liệu (thuộc tính)

Các thuộc tính của lớp cơ sở đều được thừa kế trong lớp dẫn xuất. Trong lớp dẫn xuất gồm có thuộc tính vừa mới định nghĩa trong lớp dẫn xuất và thuộc tính của lớp cơ sở. Tuy nhiên, trong lớp dẫn xuất không có quyền truy nhập đến các thuộc tính `private` của lớp cơ sở. Cho phép đặt trùng tên các thuộc tính trong lớp cơ sở và lớp dẫn xuất.

Ví dụ:

```
class A
{
    private:
        int a, b;
    ...
};
class B
{
    private:
        float a, b;
    ...
};
class C: public A, B
{
    private:
        int b;
        char *a;
    ...
};
```

Lúc này lớp C sẽ có các thuộc tính:

`A::a; A::b` (kiểu `int`) - thừa kế từ lớp A

`B::a; B::b` (kiểu `float`) - thừa kế từ lớp B

`A` (kiểu `char*`) và `b` (kiểu `int`) - được định nghĩa trong C

Các phương thức trong C chỉ có quyền truy nhập trực tiếp đến các thuộc tính khai báo trong lớp C

I.3.3. Thừa kế phương thức

Các phương thức trong lớp cơ sở đều được thừa kế trong lớp dẫn xuất, ngoại trừ:

- Hàm tạo
- Hàm hủy
- Toán tử gán

Ví dụ 5.4

```
#include <conio.h>
#include <iostream.h>
class Point
{
    private:
        int x,y; //toa do cua diem
    public:
        Point() //ham thiet lap khong co tham so
        {
            x = y =0;
        }
        Point(int x1, int y1) //Ham thiet lap co tham so
        {
            x = x1;
            y = y1;
        }
        void display()
        {
            cout<<"Toa do diem x= <<x<<" y= "<<y;
        }
};

class HINH_TRON: public Point
{
    private:
        float R;
    public:
        HINH_TRON():public Point() //Ham thiet lap khong co
tham so
        {
```

```

        R=0.0;
    }
    HINH_TRON(int x1,int y1,float R1):public Point(x1,y1)
    //Ham thiet lap co tham so
    {
        R=R1;
    }
    float getR()
    {
        return R;
    }
};

void main()
{
    HINH_TRON c(3, 4, 6);
    clrscr();
    cout<<"Tam hình tron: ";
    c.display();
    cout<<"\n Ban kinh duong tron: "<<c.getR();
    getch();
}

```

I.4. Hàm thiết lập và hàm hủy trong lớp dẫn xuất

Lớp dẫn xuất không thừa kế các hàm thiết lập, hàm hủy, toán tử gán của các lớp cơ sở

I.4.1. Cách xây dựng hàm thiết lập trong lớp dẫn xuất

- Hàm tạo cần có các đối để khởi gán cho các thuộc tính (các thành phần dữ liệu) của lớp
- Có thể phân thuộc tính làm 3 loại, ứng với 3 cách khởi gán khác nhau:
 - Các thuộc tính mới khai báo trong lớp dẫn xuất. Trong thân của lớp dẫn xuất có thể truy nhập đến các thuộc tính này, vì vậy chúng thường được khởi gán bằng các câu lệnh gán viết trong thân hàm thiết lập
 - Các thành phần kiểu đối tượng. Trong lớp dẫn xuất không cho phép truy nhập đến các thuộc tính của các đối tượng này. Vì vậy để khởi gán cho các đối tượng thành phần cần dùng hàm tạo của lớp tương ứng.
 - Các thuộc tính kế thừa từ lớp cơ sở. Trong lớp dẫn xuất không được phép truy nhập đến các thuộc tính này. Vì vậy để khởi gán cho các thuộc tính trên, cần sử dụng hàm thiết lập của lớp cơ sở. Cách thức cũng giống như khởi gán cho các đối tượng thành phần, chỉ khác: để khởi gán cho các đối tượng thành phần ta dùng

tên đối tượng thành phần, để khởi gán cho các thuộc tính thừa kế từ các lớp cơ sở ta dùng tên lớp:

Tên_đối_tượng_thành_phần (danh sách giá trị);

Tên_lớp_cơ_sở (danh sách giá trị);

Danh sách giá trị lấy từ các đối của hàm thiết lập của lớp cơ sở đang xây dựng

I.4.2. Hàm hủy trong lớp dẫn xuất

Khi một đối tượng của lớp dẫn xuất được giải phóng (bị hủy), thì các đối tượng thành phần và các đối tượng thừa kế từ lớp cơ sở cũng bị giải phóng theo. Do đó các hàm hủy tương ứng sẽ được gọi đến.

Khi xây dựng hàm hủy của lớp dẫn xuất, ta chỉ cần quan tâm đến các thuộc tính (không phải là đối tượng) khai báo thêm trong lớp dẫn xuất mà không cần để ý đến các đối tượng thành phần và các thuộc tính thừa kế từ lớp cơ sở.

I.4.3. Ví dụ

Ví dụ 5.5

```
#include <conio.h>
#include <iostream.h>
#include <string.h>
Class MON_HOC
{
    private:
        char *monhoc;
        int st; //số tiết
    public:
        MON_HOC()
        {
            monhoc=NULL;
            st=0;
        }
        MON_HOC(char *monhoc1, int st1)
        {
            int n=strlen(monhoc1);
            monhoc=new char[n+1];
            strcpy(monhoc, monhoc1);
            st = st1;
        }
        ~ MON_HOC()
        {
            if(monhoc!=NULL)
```

```
        {
            delete monhoc;
            st = 0;
        }
    }
    void in()
    {
        cout << "\nTen mon: <<monhoc<<so tiet: " << st;
    }
};
class NGUOI
{
    private:
        char *ht;
        int ns;
    public:
        NGUOI()
        {
            ht=NULL;
            ns = 0;
        }
        NGUOI(char *ht1, int ns1)
        {
            int n=strlen(ht1);
            ht = new char [n+1];
            strcpy(ht, ht1);
            ns = ns1;
        }
        ~NGUOI()
        {
            if (ht=NULL)
            {
                delete ht;
                ns = 0;
            }
        }
        void in()
        {
```

```
        Cout<<"\n Ho ten: "<<ht<< "nam sinh: "<<ns;
    }
};
class GIAO_VIEN: public NGUOI
{
    private:
        char *bomon;
        MON_HOC mh;
    public:
        GIAO_VIEN():mh, NGUOI//sử dụng hàm tạo không đối
        {
            Bomon = NULL;
        }
        GIAO_VIEN(char *ht1, int ns1, char *monhoc1, int st1,
char bomon1): NGUOI(ht1, ns1), mh(monhoc1, st1)
        {
            int n=strlen(bomon1);
            bomon = newchar[n+1];
            strcpy(bomon,bomon1);
        }
        ~GIAO_VIEN()
        {
            if(bomon!=NULL)
                delete bomon;
        }
        void in()
        {
            //su dung phuong thuc in
            NGUOI::in();
            cout<<"\n Cong tacs tai bo mon"<<bomon;
            mh.in();
        }
};
void main()
{
    clrscr();
    GIAO_VIEN g1;//Goi cac ham tao khong doi
    GIAO_VIEN *g2;
    //Goi cac ham tao co doi
```

```

g2= new GIAO_VIEN("NGUYEN VAN AN",1981,"CNTT",45,"TIN");
g2->in(); //hoac viet g2->GIAO_VIEN::in();
g2->NGUOI::in();
getch();
delete g2; //goi toi ham huy
getch();
}

```

I.4.4.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 5.6

```

#include <iostream.h>
#include <conio.h>
/*khai báo lớp point mà không có hàm thiết lập tường minh*/
class point {
    float x,y;
public:
    void display() {
        cout<<"Goi ham point::display() \n";
        cout<<"Toa do : "<<x<<" "<<y<<endl;
    }
    void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
};
void main() {
    clrscr();
    cout<<"Diem p \n";
    point p;
    p.display();
    getch();
}

```

```

Diem p
Goi ham point::display()
Toa do :8.187236e-34 2.637535e-11

```

I.4.5. Phân 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.

Giải pháp của C++: 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.

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 phan 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;
}

```

I.4.6. 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 5.6

```

/*inheri4.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;
    }
};

/*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
```

1.5. Các từ khóa quy định phạm vi truy nhập của lớp cơ sở

- Lớp dẫn xuất được thừa kế tất cả các thành phần của lớp cơ sở, nhưng trong lớp dẫn xuất không thể truy nhập tới tất cả các thành phần này. Do đó ta thường sử dụng các phương thức của lớp cơ sở để truy nhập đến các thuộc tính của chính lớp cơ sở đó. Ngoài ra có thể sử dụng các giải pháp khác dưới đây:
- Các thành phần private của lớp cơ sở không cho phép truy nhập trong lớp dẫn xuất
- Các thành phần public của lớp cơ sở có thể truy nhập từ bất kỳ chỗ nào trong chương trình. Như vậy trong lớp dẫn xuất có thể truy nhập được các thành phần này.
- Các thành phần khai báo protected chỉ được mở rộng phạm vi truy nhập cho các lớp dẫn xuất trực tiếp từ lớp này

1.6. 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

Public	dẫn xuất <code>public</code>
Private	dẫn xuất <code>private</code>
protected	dẫn xuất <code>protected</code>

I.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.

I.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.

I.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 <code>public</code>		DẪN XUẤT <code>protected</code>		DẪN XUẤT <code>private</code>	
TTĐ	TN FMA	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	DIỄN 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

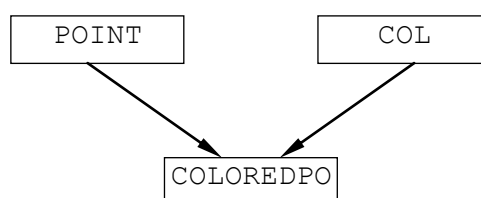
II. Đa thừa kế

II.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 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 cùng 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:

```

class point {
    float x,y;
public:
    point(...) {...}
    ~point(...) {...}
    void display() {...}
};
  
```

```

class col {
    unsigned color;
public:
    col (...) {...}
    ~color() {...}
    void display() {...}
};
  
```

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:

```
coloredpoint(.....):point(.....),col(.....)
```

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, một 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.

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

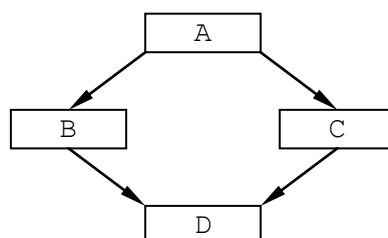
II.1.1.Sự trùng tên

Các yêu cầu về cách đặt tên:

- Tên các lớp không được trùng nhau
- Tên các thành phần của lớp cũng không được trùng lặp
- Tên các thành phần trong các lớp khác nhau có quyền trùng nhau

II.1.2.Lớp cơ sở ảo

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 NHỜ:

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 khoá **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 khoá **virtual** có thể được đặt trước hoặc sau từ khoá **public** (hoặc **private**, **protected**). Ta xét chương trình ví dụ sau đây:

Ví dụ 5.7

```
/*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();
}

```

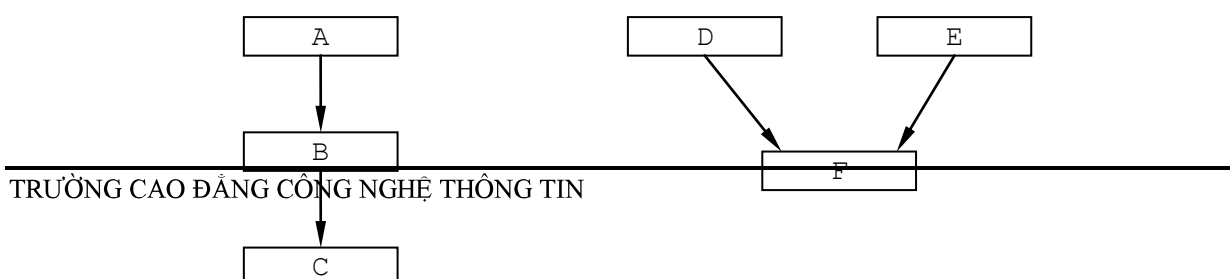
```

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

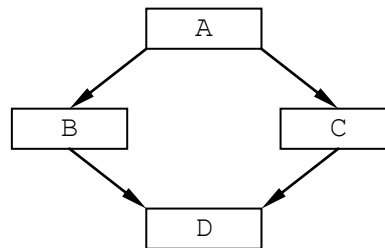
```

II.1.3.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:

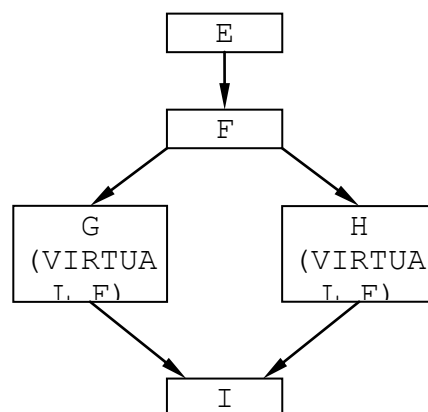


Trong trường hợp này, ta chỉ xây dựng một đối tượng duy nhất có kiểu A. Các tham số được truyền cho hàm thiết lập A được mô tả ngay khi định nghĩa hàm thiết lập D. Và như vậy, không cần mô tả các thông tin truyền cho A ở mức B và C. Cần phải tuân theo quy tắc, quy định sau đây:

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.

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 5.8

```

/*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

```

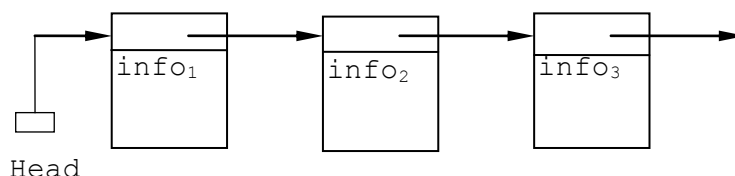
II.2. Danh sách móc nối các đối tượng

Trong phần này, chúng ta đưa ra cách xây dựng một lớp, cho phép quản lý một danh sách móc nối các đối tượng kiểu `point`. Các công việc cần phải làm là thiết kế một lớp đặc biệt để quản lý danh sách móc nối độc lập với kiểu đối tượng liên quan. Một lớp như vậy hoàn toàn

mang ý nghĩa trừu tượng, cùng với `point` sẽ là lớp thừa kế cho một lớp dẫn xuất chứa danh sách móc nối.

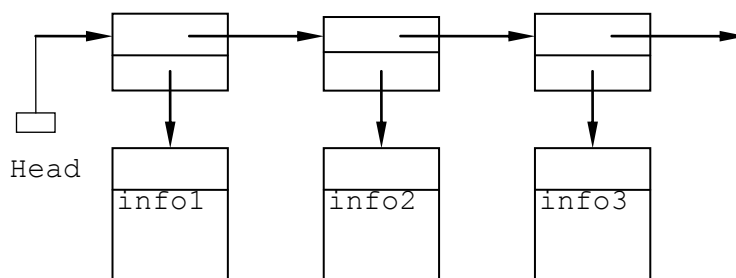
II.2.1. Xây dựng lớp trừu tượng

Ta giới hạn ở xây dựng một danh sách móc nối đơn giản, với các phần tử có hai thành phần: thành phần dữ liệu và thành phần con trỏ, chỉ tới phần tử đi tiếp sau. Sơ đồ của danh sách như vậy có dạng như sau:



Ở đây `Head` là con trỏ đối tượng, chỉ đến phần tử đầu tiên của danh sách móc 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:

- (iii) một thành phần dữ liệu: con trỏ đến phần tử đầu tiên (`Head`),
- (iv) 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 hủy 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.

Sau đây là chương trình hoàn chỉnh:

Ví dụ 5.10

```

/*list_hom.cpp
homogenous list*/
#include <iostream.h>
#include <stddef.h> //de định nghĩa 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 *chose) {
    element *adel= new element;
```

```

    adel->next =head;
    adel->content=chose;
    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
-----

```

II.2.2. Tạo danh sách móc nối không đồng nhất

Phần trước đã xét ví dụ về danh sách móc 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 trỏ (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 móc nối. Trong khai báo của lớp cơ sở này, có một hàm thành phần ảo 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 trỏ 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 trỏ để gọi tới các thành phần hàm hiển thị, nên tốt nhất ta là khai báo hàm đó như là hàm ảo thuần túy và lớp tương ứng sẽ là lớp trừu tượng.

Ví dụ 5.11

```

/*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 thuần túy
};
//lớp quản lý danh sách móc nối
struct element //cấu trúc một thành phần của danh sách
{
    element *next; //chỉ đến thành phần đi 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;
    }
    /*cho địa chỉ của phần tử hiện tại , xác định thành phần tiếp theo*/
    void *nextelement() {
        void *adnext=NULL;
        if (current !=NULL)
        {
            adnext=current->content;
            current=current->next;
        }
        return adnext;
    }
    /*liệt kê tất cả các phần tử trong danh sách*/
    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* chu không phải 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);

```

```

1.add(&x);
1.display_list();
cout<<"-----\n";
1.add(&y);
1.add(&b);
1.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
```

III.Hàm ảo và tính đa hình

III.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 trỏ tới 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 trỏ đến. Ta đã có dịp minh chứng điều này trong ví dụ ở phần **Error! Reference source not found.** 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 nổi điểm chỉ được xác định một cách tường 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 trỏ đến, 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 5.16

```
/*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";
    ptr->display();
    getch();
}

```

```

coloredpoint pc(2,3,5);
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)

```

```

pc.display();
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
point *ptr=&pc;
ptr->display();
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
point p(10,20);
point::point(float, float)
ptr = &p
p.display()
Goi ham point::display()
Toa do :10 20
ptr->display();
Goi ham point::display()
Toa do :10 20

```

Nhận xét

1. 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 ảo 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 5.17

```

/*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.

III.2. Tổng quát về hàm ảo

III.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 5.18

```
#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 coloredtpoint::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
threedimpoint p3d(2,3,4);
point::point(float, float)
p3d.display();
Toa do : 2 3
Toa do z : 4
coloredthreedimpoint p3dc(2,3,4,10);
point::point(float, float)
p3dc.display();
Toa do : 2 3
Diem mau : 10

```

III.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 5.19

```

#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;
    }
};

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()
Goi ham point::display()
Toa do :10 20
ptr->display();
Goi ham point::display()
Toa do :10 20

```

III.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.

III.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 5.20

```

/*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;
    }
    threedimpoint(float ox, float oy, float oz):point (ox, oy) {
        z = oz;
    }
    threedimpoint(threedimpoint &p) :point(p) {
        z = p.z;
    }
    virtual 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;
    }
};

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

```

III.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 dọn 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 5.21

```

#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()
```

III.2.6. Lớp trừu tượng và hàm ảo thuần túy

Hàm ảo thuần túy là hàm không có phần định nghĩa. Định nghĩa một hàm ảo như thế được viết như sau:

```
class mere {
public:
    virtual display() = 0; //hàm ảo thuần túy
};
```

Lớp có ít nhất một hàm thành phần ảo thuần túy 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 trừu tượng.
- (iii) Một hàm ảo thuần túy khai báo trong một lớp trừu 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 trừu tượng. Trong phần II.2.2 sẽ có một ví dụ về việc sử dụng lớp trừu tượng để xây dựng một danh sách các đối tượng không đồng nhất.

CHƯƠNG 6

KHUÔN HÌNH

I. MỘT SỐ KHÁI NIỆM

TAÌI LIỆU THAM KHẢO

- [1] Phạm Văn Át, *C++ và Lập Trình Hướng Đối Tượng*, NXB KHKT, 2000.
- [2] Nguyễn Thanh Thủy, *Lập trình hướng đối tượng với C++ và Bài tập Lập trình hướng đối tượng với C++*, NXB Khoa học và kỹ thuật, 2001.
- [3] Bruce Eckel, *Thinking in C++*, Prentice Hall Inc., 2000.
- [4] Peter **Müller**, *Introduction to Object-Oriented Programming Using C++*, www.gnacademy.org
- [5] Cox, *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, Massachusetts, (2nd), 1993.
- [6] Budd, *An Introduction to Object Oriented Programming (2nd)*, Addison-Wesley, Reading, Massachusetts, 1997.

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG



NGÔN NGỮ LẬP TRÌNH C++

(Dùng cho sinh viên hệ đào tạo đại học từ xa)

Lưu hành nội bộ

HÀ NỘI - 2006

NGÔN NGỮ LẬP TRÌNH C++

PGS.TS. Trần Đình Quế
KS. Nguyễn Mạnh Hùng



Lập trình nâng cao với C++
Lập trình hướng đối tượng với C++



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04).5541221; Fax: (04).5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhtx@o-pit.edu.vn

GIỚI THIỆU

C++ là ngôn ngữ lập trình hướng đối tượng được mở rộng từ ngôn ngữ C. Do vậy, C++ có ưu điểm là kế thừa được các điểm mạnh truyền thống của ngôn ngữ C như uyển chuyển, tương thích với các thiết bị phần cứng. Hiện nay, C++ là một ngôn ngữ lập trình phổ biến, được giảng dạy tại các trường đại học trong nước và trên thế giới và đặc biệt được sử dụng rộng rãi cho nhu cầu phát triển của công nghiệp phần mềm hiện nay. Tài liệu này không những nhằm giới thiệu cho sinh viên ngôn ngữ lập trình C++, mà còn mong muốn qua đó sinh viên có thể hiểu được tư tưởng của phương pháp lập trình hướng đối tượng nói chung. Nội dung của tài liệu bao gồm hai phần chính:

- Phần thứ nhất là lập trình nâng cao với C++, bao gồm lập trình C++ với con trỏ và mảng, các kiểu dữ liệu có cấu trúc cùng các thao tác vào ra trên tệp.
- Phần thứ hai là lập trình hướng đối tượng với C++, bao gồm các định nghĩa và các thao tác trên lớp đối tượng, tính kế thừa và tương ứng bội trong C++, cách sử dụng một số lớp cơ bản trong thư viện C++.

Nội dung tài liệu được tổ chức thành 7 chương:

Chương 1: Giới thiệu tổng quan về các phương pháp lập trình

Trình bày các phương pháp lập trình tuyến tính, lập trình cấu trúc và đặc biệt, làm quen với các khái niệm trong lập trình hướng đối tượng.

Chương 2: Con trỏ và mảng

Trình bày cách khai báo và sử dụng các kiểu con trỏ và mảng trong ngôn ngữ C++.

Chương 3: Kiểu dữ liệu có cấu trúc

Trình bày cách biểu diễn và cài đặt một số kiểu cấu trúc dữ liệu trừu tượng trong C++. Sau đó, trình bày cách áp dụng các kiểu dữ liệu này trong các ứng dụng cụ thể.

Chương 4: Vào ra trên tệp

Trình bày các thao tác đọc, ghi dữ liệu trên các tệp tin khác nhau: tệp tin văn bản và tệp tin nhị phân. Trình bày các cách truy nhập tệp tin trực tiếp.

Chương 5: Lớp đối tượng

Trình bày các khái niệm mở đầu cho lập trình hướng đối tượng trong C++, bao gồm cách khai báo và sử dụng lớp, các thuộc tính của lớp; cách khởi tạo và hủy bỏ đối tượng, các quy tắc truy nhập đến các thành phần của lớp.

Chương 6: Tính kế thừa và tương ứng bội

Trình bày cách thức kế thừa giữa các lớp trong C++, các nguyên tắc truy nhập trong kế thừa, định nghĩa nạp chồng các phương thức và tính đa hình trong lập trình hướng đối tượng với C++.

Chương 7: Một số lớp quan trọng

Trình bày cách sử dụng một số lớp có sẵn trong thư viện chuẩn của C++, bao gồm các lớp: lớp tập hợp, lớp chuỗi, lớp ngăn xếp, lớp hàng đợi và lớp danh sách liên kết.

Để đọc được cuốn sách này, sinh viên phải quen biết các khái niệm cơ bản về lập trình, có một số kỹ năng lập trình với ngôn ngữ C hoặc C++. Cuốn sách này cũng có thể dùng tài liệu tham khảo cho những sinh viên muốn tìm hiểu các kỹ thuật lập trình nâng cao và lập trình hướng đối tượng

với C++. Cuốn sách này có kèm theo một đĩa chương trình chứa toàn bộ các chương trình được lấy làm minh họa và các bài tập trong cuốn sách.

Mặc dù các tác giả đã có nhiều cố gắng trong việc biên soạn tài liệu này, song không thể tránh khỏi những thiếu sót. Rất mong nhận được những ý kiến đóng góp quý báu từ các sinh viên và các bạn đồng nghiệp.



CHƯƠNG 1

GIỚI THIỆU VỀ CÁC PHƯƠNG PHÁP LẬP TRÌNH

Nội dung của chương này tập trung trình bày các phương pháp lập trình:

- Phương pháp lập trình tuyến tính
- Phương pháp lập trình hướng cấu trúc
- Phương pháp lập trình hướng đối tượng.

1.1 LẬP TRÌNH TUYẾN TÍNH

Đặc trưng cơ bản của lập trình tuyến tính là tư duy theo lối tuần tự. Chương trình sẽ được thực hiện theo thứ tự từ đầu đến cuối, lệnh này kế tiếp lệnh kia cho đến khi kết thúc chương trình.

Đặc trưng

Lập trình tuyến tính có hai đặc trưng:

- **Đơn giản:** chương trình được tiến hành đơn giản theo lối tuần tự, không phức tạp.
- **Đơn luồng:** chỉ có một luồng công việc duy nhất, và các công việc được thực hiện tuần tự trong luồng đó.

Tính chất

- **Ưu điểm:** Do tính đơn giản, lập trình tuyến tính được ứng dụng cho các chương trình đơn giản và có ưu điểm dễ hiểu.
- **Nhược điểm:** Với các ứng dụng phức tạp, người ta không thể dùng lập trình tuyến tính để giải quyết.

Ngày nay, lập trình tuyến tính chỉ tồn tại trong phạm vi các modul nhỏ nhất của các phương pháp lập trình khác. Ví dụ trong một chương trình con của lập trình cấu trúc, các lệnh cũng được thực hiện theo tuần tự từ đầu đến cuối chương trình con.

1.2 LẬP TRÌNH HƯỚNG CẤU TRÚC

1.2.1 Đặc trưng của lập trình hướng cấu trúc

Trong lập trình hướng cấu trúc, chương trình chính được chia nhỏ thành các chương trình con và mỗi chương trình con thực hiện một công việc xác định. Chương trình chính sẽ gọi đến chương trình con theo một giải thuật, hoặc một cấu trúc được xác định trong chương trình chính.

Các ngôn ngữ lập trình cấu trúc phổ biến là Pascal, C và C++. Riêng C++ ngoài việc có đặc trưng của lập trình cấu trúc do kế thừa từ C, còn có đặc trưng của lập trình hướng đối tượng. Cho nên C++ còn được gọi là ngôn ngữ lập trình nửa cấu trúc, nửa hướng đối tượng.

Đặc trưng

Đặc trưng cơ bản nhất của lập trình cấu trúc thể hiện ở mối quan hệ:

Chương trình = Cấu trúc dữ liệu + Giải thuật

Trong đó:

- **Cấu trúc dữ liệu** là cách tổ chức dữ liệu cho việc xử lý bởi một hay nhiều chương trình nào đó.
- **Giải thuật** là một quy trình để thực hiện một công việc xác định

Trong chương trình, giải thuật có quan hệ phụ thuộc vào cấu trúc dữ liệu:

- Một cấu trúc dữ liệu chỉ phù hợp với một số hạn chế các giải thuật.
- Nếu thay đổi cấu trúc dữ liệu thì phải thay đổi giải thuật cho phù hợp.
- Một giải thuật thường phải đi kèm với một cấu trúc dữ liệu nhất định.

Tính chất

- Mỗi chương trình con có thể được gọi thực hiện nhiều lần trong một chương trình chính.
- Các chương trình con có thể được gọi đến để thực hiện theo một thứ tự bất kì, tùy thuộc vào giải thuật trong chương trình chính mà không phụ thuộc vào thứ tự khai báo của các chương trình con.
- Các ngôn ngữ lập trình cấu trúc cung cấp một số cấu trúc lệnh điều khiển chương trình.

Ưu điểm

- Chương trình sáng sủa, dễ hiểu, dễ theo dõi.
- Tư duy giải thuật rõ ràng.

Nhược điểm

- Lập trình cấu trúc không hỗ trợ mạnh việc sử dụng lại mã nguồn: Giải thuật luôn phụ thuộc chặt chẽ vào cấu trúc dữ liệu, do đó, khi thay đổi cấu trúc dữ liệu, phải thay đổi giải thuật, nghĩa là phải viết lại chương trình.
- Không phù hợp với các phần mềm lớn: tư duy cấu trúc với các giải thuật chỉ phù hợp với các bài toán nhỏ, nằm trong phạm vi một modul của chương trình. Với dự án phần mềm lớn, lập trình cấu trúc tỏ ra không hiệu quả trong việc giải quyết mối quan hệ vĩ mô giữa các modul của phần mềm.

Vấn đề

Vấn đề cơ bản của lập trình cấu trúc là bằng cách nào để phân chia chương trình chính thành các chương trình con cho phù hợp với yêu cầu, chức năng và mục đích của mỗi bài toán. Thông thường, để phân rã bài toán trong lập trình cấu trúc, người ta sử dụng phương pháp thiết kế trên xuống (top-down).

1.2.2 Phương pháp thiết kế trên xuống (top-down)

Phương pháp thiết kế top-down tiếp cận bài toán theo hướng từ trên xuống dưới, từ tổng quát đến chi tiết. Theo đó, một bài toán được chia thành các bài toán con nhỏ hơn. Mỗi bài toán con lại được chia nhỏ tiếp, nếu có thể, thành các bài toán con nhỏ hơn nữa. Quá trình này còn được gọi là quá trình làm mịn dần. Quá trình này sẽ dừng lại khi các bài toán con không cần chia nhỏ thêm

nữa. Nghĩa là khi mỗi bài toán con đều có thể giải quyết bằng một chương trình con với một giải thuật đơn giản.

Ví dụ, sử dụng phương pháp top-down để giải quyết bài toán xây một căn nhà mới. Chúng ta có thể phân rã bài toán theo các bước như sau:

- Ở mức thứ nhất, chia bài toán xây nhà thành các bài toán nhỏ hơn như *làm móng*, đổ cột, đổ trần, xây tường, lợp mái.
- Ở mức thứ hai, phân rã các công việc ở mức thứ nhất như việc *làm móng* nhà có thể phân rã tiếp thành các công việc *đào móng*, gia cố nền, làm khung sắt, đổ bê tông; công việc đổ cột được phân rã thành ...
- Ở mức thứ ba, phân rã các công việc của mức thứ hai như việc *đào móng* có thể phân chia tiếp thành các công việc như đo đạc, cắm mốc, chằng dây, đào và kiểm tra móng. Việc gia cố nền được phân rã thành ...

Quá trình phân rã có thể dừng ở mức này, bởi vì các công việc con thu được như đo đạc, cắm mốc, chằng dây, đào... có thể thực hiện được ngay, không cần chia nhỏ thêm nữa.

Lưu ý:

- Cùng sử dụng phương pháp top-down với cùng một bài toán, nhưng có thể cho ra nhiều kết quả khác nhau. Nguyên nhân là do sự khác nhau trong tiêu chí để phân rã một bài toán thành các bài toán con.

Ví dụ, vẫn áp dụng phương pháp top-down để giải quyết bài toán xây nhà, nhưng nếu sử dụng một cách khác để phân chia bài toán, ta có thể thu được kết quả khác biệt so với phương pháp ban đầu:

- Ở mức thứ nhất, chia bài toán xây nhà thành các bài toán nhỏ hơn như *làm phần gỗ*, làm phần sắt, làm phần bê tông và làm phần gạch.
- Ở mức thứ hai, phân rã các công việc ở mức thứ nhất là *làm phần gỗ* có thể chia thành các công việc như xẻ gỗ, gia công gỗ, tạo khung, lắp vào nhà. Việc làm sắt có thể chia nhỏ thành...

Rõ ràng, với cách làm mịn thế này, ta sẽ thu được một kết quả khác hẳn với cách thức đã thực hiện ở phần trên.

1.3 LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

1.3.1 Lập trình hướng đối tượng

Trong lập trình hướng đối tượng:

- Người ta coi các thực thể trong chương trình là các đối tượng và sau đó trừu tượng hoá đối tượng thành lớp đối tượng.
- Dữ liệu được tổ chức thành các thuộc tính của lớp. Người ta ngăn chặn việc thay đổi tùy tiện dữ liệu trong chương trình bằng các cách giới hạn truy nhập như chỉ cho phép truy nhập dữ liệu thông qua đối tượng, thông qua các phương thức mà đối tượng được cung cấp...
- Quan hệ giữa các đối tượng là quan hệ ngang hàng hoặc quan hệ kế thừa: Nếu lớp B kế thừa từ lớp A thì A được gọi là *lớp cơ sở* và B được gọi là *lớp dẫn xuất*.

Ngôn ngữ lập trình hướng đối tượng phổ biến hiện nay là Java, C++, C#... Mặc dù C++ cũng có những đặc trưng cơ bản của lập trình hướng đối tượng nhưng vẫn không phải là ngôn ngữ lập trình thuần hướng đối tượng.

Đặc trưng

Lập trình hướng đối tượng có hai đặc trưng cơ bản:

- **Đóng gói dữ liệu:** dữ liệu luôn được tổ chức thành các thuộc tính của lớp đối tượng. Việc truy nhập đến dữ liệu phải thông qua các phương thức của đối tượng lớp.
- **Sử dụng lại mã nguồn:** việc sử dụng lại mã nguồn được thể hiện thông qua cơ chế kế thừa. Cơ chế này cho phép các lớp đối tượng có thể kế thừa từ các lớp đối tượng khác. Khi đó, trong các lớp dẫn xuất, có thể sử dụng các phương thức (mã nguồn) của các lớp cơ sở mà không cần phải định nghĩa lại.

Ưu điểm

Lập trình hướng đối tượng có một số ưu điểm nổi bật:

- Không còn nguy cơ dữ liệu bị thay đổi tự do trong chương trình. Vì dữ liệu đã được đóng gói vào các đối tượng. Nếu muốn truy nhập vào dữ liệu phải thông qua các phương thức được cho phép của đối tượng.
- Khi thay đổi cấu trúc dữ liệu của một đối tượng, không cần thay đổi mã nguồn của các đối tượng khác, mà chỉ cần thay đổi một số thành phần của đối tượng dẫn xuất. Điều này hạn chế sự ảnh hưởng xấu của việc thay đổi dữ liệu đến các đối tượng khác trong chương trình.
- Có thể sử dụng lại mã nguồn, tiết kiệm tài nguyên, chi phí thời gian. Vì nguyên tắc kế thừa cho phép các lớp dẫn xuất sử dụng các phương thức từ lớp cơ sở như những phương thức của chính nó, mà không cần thiết phải định nghĩa lại.
- Phù hợp với các dự án phần mềm lớn, phức tạp.

1.3.2 Một số khái niệm cơ bản

Trong mục này, chúng ta sẽ làm quen với một số khái niệm cơ bản trong lập trình hướng đối tượng. Bao gồm:

- Khái niệm đối tượng (object)
- Khái niệm đóng gói dữ liệu (encapsulation)
- Khái niệm kế thừa (inheritance)
- Khái niệm đa hình (polymorphism)

Đối tượng (Object)

Trong lập trình hướng đối tượng, đối tượng được coi là đơn vị cơ bản nhỏ nhất. Các dữ liệu và cách xử lý chỉ là thành phần của đối tượng mà không được coi là thực thể. Một đối tượng chứa các dữ liệu của riêng nó, đồng thời có các phương thức (hành động) thao tác trên các dữ liệu đó:

Đối tượng = dữ liệu + phương thức

Lớp (Class)

Khi có nhiều đối tượng giống nhau về mặt dữ liệu và phương thức, chúng được nhóm lại với nhau và gọi chung là lớp:

- Lớp là sự trừu tượng hoá của đối tượng
- Đối tượng là một thể hiện của lớp.

Đóng gói dữ liệu (Encapsulation)

- Các dữ liệu được đóng gói vào trong đối tượng. Mỗi dữ liệu có một phạm vi truy nhập riêng.
- Không thể truy nhập đến dữ liệu một cách tự do như lập trình cấu trúc
- Muốn truy nhập đến các dữ liệu đã được bảo vệ, phải thông qua các đối tượng, nghĩa là phải sử dụng các phương thức mà đối tượng cung cấp mới có thể truy nhập đến dữ liệu của đối tượng đó.

Tuy nhiên, vì C++ chỉ là ngôn ngữ lập trình nửa đối tượng, cho nên C++ vẫn cho phép định nghĩa các biến dữ liệu và các hàm tự do, đây là kết quả kế thừa từ ngôn ngữ C, một ngôn ngữ lập trình thuần cấu trúc.

Kế thừa (Inheritance)

Tính kế thừa của lập trình hướng đối tượng cho phép một lớp có thể kế thừa từ một số lớp đã tồn tại. Khi đó, lớp mới có thể sử dụng dữ liệu và phương thức của các lớp cơ sở như là của mình. Ngoài ra, lớp dẫn xuất còn có thể bổ sung thêm một số dữ liệu và phương thức. Ưu điểm của kế thừa là khi thay đổi dữ liệu của một lớp, chỉ cần thay đổi các phương thức trong phạm vi lớp cơ sở mà không cần thay đổi trong các lớp dẫn xuất.

Đa hình (Polymorphsim)

Đa hình là khái niệm luôn đi kèm với kế thừa. Do tính kế thừa, một lớp có thể sử dụng lại các phương thức của lớp khác. Tuy nhiên, nếu cần thiết, lớp dẫn xuất cũng có thể định nghĩa lại một số phương thức của lớp cơ sở. Đó là sự nạp chồng phương thức trong kế thừa. Nhờ sự nạp chồng phương thức này, ta chỉ cần gọi tên phương thức bị nạp chồng từ đối tượng mà không cần quan tâm đó là đối tượng của lớp nào. Chương trình sẽ tự động kiểm tra xem đối tượng là thuộc kiểu lớp cơ sở hay thuộc lớp dẫn xuất, sau đó sẽ gọi phương thức tương ứng với lớp đó. Đó là tính đa hình.

1.3.3 Lập trình hướng đối tượng trong C++

Vì C++ là một ngôn ngữ lập trình được mở rộng từ một ngôn ngữ lập trình cấu trúc C nên C++ được xem là ngôn ngữ lập trình nửa hướng đối tượng, nửa hướng cấu trúc.

Những đặc trưng hướng đối tượng của C++

- Cho phép định nghĩa lớp đối tượng.
- Cho phép đóng gói dữ liệu vào các lớp đối tượng. Cho phép định nghĩa phạm vi truy nhập dữ liệu của lớp bằng các từ khoá phạm vi: **public**, **protected**, **private**.

- Cho phép kế thừa lớp với các kiểu kế thừa khác nhau tùy vào từ khoá dẫn xuất.
- Cho phép lớp dẫn xuất sử dụng các phương thức của lớp cơ sở (trong phạm vi quy định).
- Cho phép định nghĩa chồng phương thức trong lớp dẫn xuất.

Những hạn chế hướng đối tượng của C++

Những hạn chế này là do C++ được phát triển từ một ngôn ngữ lập trình thuần cấu trúc C.

- Cho phép định nghĩa và sử dụng các biến dữ liệu tự do.
- Cho phép định nghĩa và sử dụng các hàm tự do.
- Ngay cả khi dữ liệu được đóng gói vào lớp, dữ liệu vẫn có thể truy nhập trực tiếp như dữ liệu tự do bởi các hàm bạn, lớp bạn (friend) trong C++.

TỔNG KẾT CHƯƠNG 1

Chương 1 đã trình bày tổng quan về các phương pháp lập trình hiện nay. Nội dung tập trung vào ba phương pháp lập trình có liên quan trực tiếp đến ngôn ngữ lập trình C++:

- Lập trình tuyến tính
- Lập trình hướng cấu trúc
- Lập trình hướng đối tượng.

C++ là ngôn ngữ lập trình được mở rộng từ ngôn ngữ lập trình cấu trúc C. Do đó, C++ vừa có những đặc trưng của lập trình cấu trúc, vừa có những đặc trưng của lập trình hướng đối tượng.

CHƯƠNG 2

CON TRỎ VÀ MẢNG

Nội dung của chương này tập trung trình bày các vấn đề cơ bản liên quan đến các thao tác trên kiểu dữ liệu con trỏ và mảng trong C++:

- Khái niệm con trỏ, cách khai báo và sử dụng con trỏ.
- Mối quan hệ giữa con trỏ và mảng
- Con trỏ hàm
- Cấp phát bộ nhớ cho con trỏ

2.1 KHÁI NIỆM CON TRỎ

2.1.1 Khai báo con trỏ

Con trỏ là một biến đặc biệt chứa địa chỉ của một biến khác. Con trỏ có cùng kiểu dữ liệu với kiểu dữ liệu của biến mà nó trỏ tới. Cú pháp khai báo một con trỏ như sau:

```
<Kiểu dữ liệu> *<Tên con trỏ>;
```

Trong đó:

- **Kiểu dữ liệu:** Có thể là các kiểu dữ liệu cơ bản của C++, hoặc là kiểu dữ liệu có cấu trúc, hoặc là kiểu đối tượng do người dùng tự định nghĩa.
- **Tên con trỏ:** Tuân theo qui tắc đặt tên biến của C++:
 - Chỉ được bắt đầu bằng một kí tự (chữ), hoặc dấu gạch dưới “_”.
 - Bắt đầu từ kí tự thứ hai, có thể có kiểu kí tự số.
 - Không có dấu trống (space bar) trong tên biến.
 - Có phân biệt chữ hoa và chữ thường.
 - Không giới hạn độ dài tên biến.

Ví dụ, để khai báo một biến con trỏ có kiểu là int và tên là pointerInt, ta viết như sau:

```
int *pointerInt;
```

Lưu ý

- Có thể viết dấu con trỏ “*” ngay sau kiểu dữ liệu, nghĩa là hai cách khai báo sau là tương đương:

```
int *pointerInt;  
int* pointerInt;
```

- Các cách khai báo con trỏ như sau là sai cú pháp:

```
*int pointerInt; // Khai báo sai con trỏ  
int pointerInt*; // Khai báo sai con trỏ
```

2.1.2 Sử dụng con trỏ

Con trỏ được sử dụng theo hai cách:

- Dùng con trỏ để lưu địa chỉ của biến để thao tác
- Lấy giá trị của biến do con trỏ trỏ đến để thao tác

Dùng con trỏ để lưu địa chỉ của biến

Bản thân con trỏ sẽ được trỏ vào địa chỉ của một biến có cùng kiểu dữ liệu với nó. Cú pháp của phép gán như sau:

```
<Tên con trỏ> = &<tên biến>;
```

Lưu ý

- Trong phép toán này, tên con trỏ không có dấu “*”.

Ví dụ:

```
int x, *px;  
px = &x;
```

sẽ cho con trỏ px có kiểu int trỏ vào địa chỉ của biến x có kiểu nguyên. Phép toán &<Tên biến> sẽ cho địa chỉ của biến tương ứng.

Lấy giá trị của biến do con trỏ trỏ đến

Phép lấy giá trị của biến do con trỏ trỏ đến được thực hiện bằng cách gọi tên:

```
*<Tên con trỏ>;
```

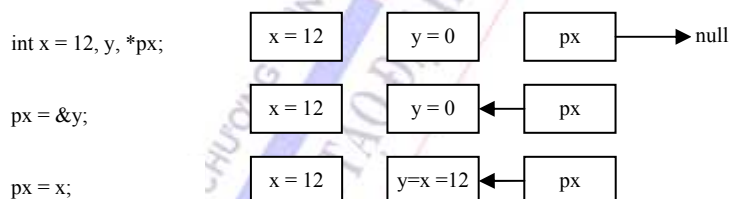
Lưu ý

- Trong phép toán này, phải có dấu con trỏ “*”. Nếu không có dấu con trỏ, sẽ trở thành phép lấy địa chỉ của biến do con trỏ trỏ tới.

Ví dụ:

```
int x = 12, y, *px;  
px = &y;  
*px = x;
```

Quá trình diễn ra như sau:



con trỏ px vẫn trỏ tới địa chỉ biến y và giá trị của biến y sẽ là 12.

Phép gán giữa các con trỏ

Các con trỏ cùng kiểu có thể gán cho nhau thông qua phép gán và lấy địa chỉ con trỏ:

```
<Tên con trỏ 1> = <Tên con trỏ 2>;
```

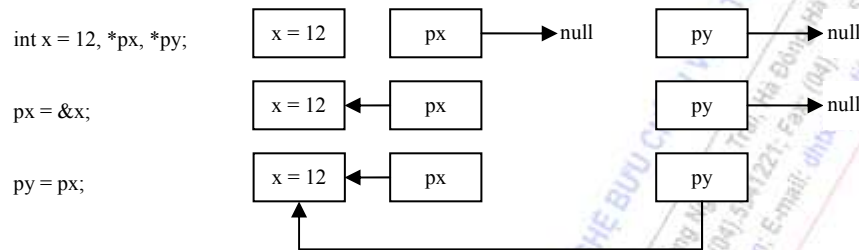
Lưu ý

- Trong phép gán giữa các con trỏ, bắt buộc phải dùng phép lấy **địa chỉ của biến** do con trỏ trỏ tới (không có dấu “*” trong tên con trỏ) mà không được dùng phép lấy **giá trị của biến** do con trỏ trỏ tới.

- Hai con trỏ phải cùng kiểu. Trong trường hợp hai con trỏ khác kiểu, phải sử dụng các phương thức ép kiểu tương tự như trong phép gán các biến thông thường có kiểu khác nhau.

Ví dụ:

```
int x = 12, *px, *py;
px = &x;
py = px;
```



con trỏ py cũng trỏ vào địa chỉ của biến x như con trỏ px. Khi đó *py cũng có giá trị 12 giống như *px và là giá trị của biến x.

Chương trình 2.1 minh họa việc dùng con trỏ giữa các biến của một chương trình C++.

Chương trình 2.1

```
#include <stdio.h>
#include <conio.h>
void main(void) {
    int x = 12, *px, *py;
    cout << "x = " << x << endl;

    px = &x;          // Con trỏ px trỏ tới địa chỉ của x
    cout << "px = &x, *px = " << *px << endl;

    *px = *px + 20;    // Nội dung của px là 32
    cout << "*px = *px+20, x = " << x << endl;

    py = px;          // Cho py trỏ tới chỗ mà px trỏ: địa chỉ của x
    *py += 15;         // Nội dung của py là 47
    cout << "py = px, *py +=15, x = " << x << endl;
}
```

Trong chương trình 2.1, ban đầu biến x có giá trị 12. Sau đó, con trỏ px trỏ vào địa chỉ của biến x nên con trỏ px cũng có giá trị 12. Tiếp theo, ta tăng giá trị của con trỏ px thêm 20, giá trị của con trỏ px là 32. Vì px đang trỏ đến địa chỉ của x nên x cũng có giá trị là 32. Sau đó, ta cho con trỏ py trỏ đến vị trí mà px đang trỏ tới (địa chỉ của biến x) nên py cũng có giá trị 32. Cuối cùng, ta tăng giá trị của con trỏ py thêm 15, py sẽ có giá trị 37. Vì py cũng đang trỏ đến địa chỉ của x nên x cũng có giá trị 37. Do đó, ví dụ 2.1 sẽ in ra kết quả như sau:

```
x = 12
px = &x, *px = 12
*px = *px + 20, x = 32
py = px, *py += 15, x = 37
```

2.2 CON TRỎ VÀ MẢNG

2.2.1 Con trỏ và mảng một chiều

Mảng một chiều

Trong C++, tên một mảng được coi là một kiểu con trỏ hằng, được định vị tại một vùng nhớ xác định và địa chỉ của tên mảng trùng với địa chỉ của phần tử đầu tiên của mảng.

Ví dụ khai báo:

```
int A[5];
```

thì địa chỉ của mảng A (cũng viết là A) sẽ trùng với địa chỉ phần tử đầu tiên của mảng A (là &A[0]) nghĩa là:

```
A = &A[0];
```

Quan hệ giữa con trỏ và mảng

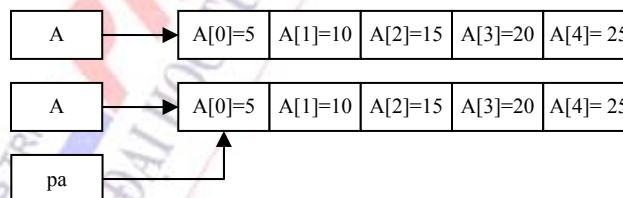
Vì tên của mảng được coi như một con trỏ hằng, nên nó có thể được gán cho một con trỏ có cùng kiểu.

Ví dụ khai báo:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = A;
```

```
int A[5] = {5, 10, 15, 20, 25};
```

```
int *pa = A;
```



thì con trỏ pa sẽ trỏ đến mảng A, tức là trỏ đến địa chỉ của phần tử A[0], cho nên hai khai báo sau là tương đương:

```
pa = A;
pa = &A[0];
```

Với khai báo này, thì địa chỉ trỏ tới của con trỏ pa là địa chỉ của phần tử A[0] và giá trị của con trỏ pa là giá trị của phần tử A[0], tức là *pa = 5;

Phép toán trên con trỏ và mảng

Khi một con trỏ trỏ đến mảng, thì các phép toán tăng hay giảm trên con trỏ sẽ tương ứng với phép dịch chuyển trên mảng.

Ví dụ khai báo:

```
int A[5] = {5, 10, 15, 20, 25};
```

```
int *pa = &A[2];
```

thì con trỏ pa sẽ trỏ đến địa chỉ của phần tử A[2] và giá trị của pa là: $*pa = A[2] = 15$.

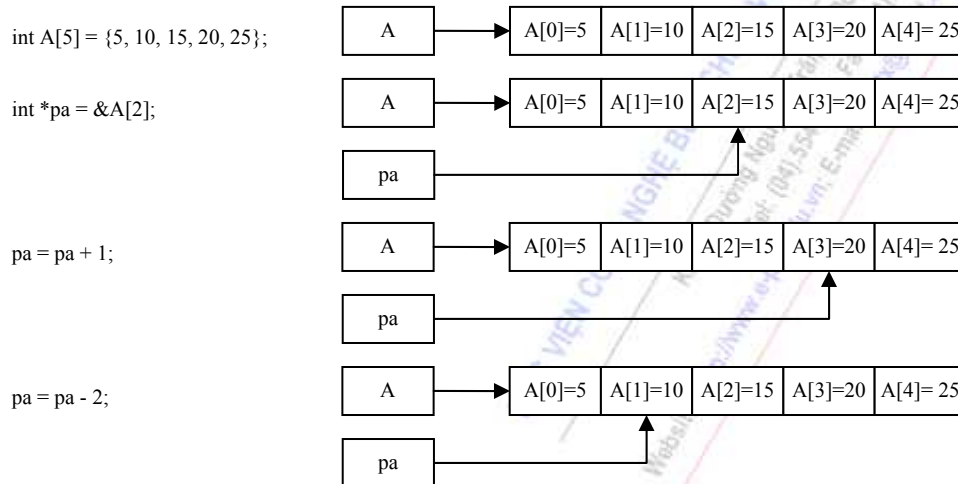
Khi đó, phép toán:

```
pa = pa + 1;
```

sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử tiếp theo của mảng A, đó là địa chỉ của A[3]. Sau đó, phép toán:

```
pa = pa - 2;
```

sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử A[1].



Lưu ý:

- Hai phép toán `pa++` và `*pa++` có tác dụng hoàn toàn khác nhau trên mảng, `pa++` là thao tác trên con trỏ, tức là trên bộ nhớ, nó sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử tiếp theo của mảng. `*pa++` là phép toán trên giá trị, nó tăng giá trị hiện tại của phần tử mảng lên một đơn vị.

Ví dụ:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = &A[2];
```

thì `pa++` là tương đương với `pa = &A[3]` và `*pa = 20`.

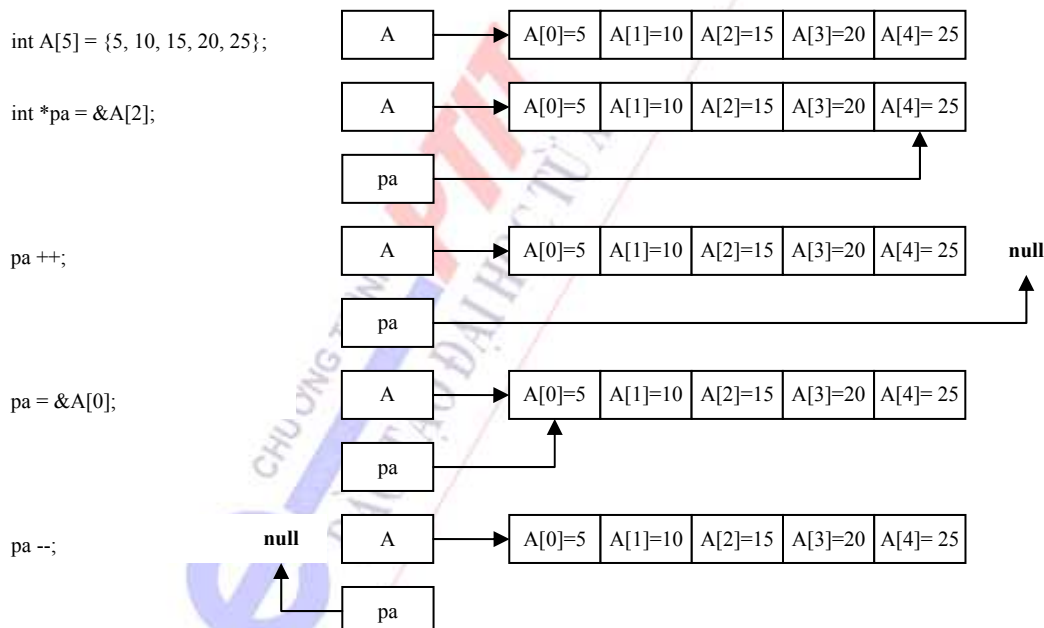
nhưng `*pa++` lại tương đương với `pa = &A[2]` và `*pa = 15+1 = 16`, `A[2] = 16`.



- Trong trường hợp:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = &A[4];
```

thì phép toán pa++ sẽ đưa con trỏ pa trở đến một địa chỉ không xác định. Lí do là A[4] là phần tử cuối của mảng A, nên pa++ sẽ trở đến địa chỉ ngay sau địa chỉ của A[4], địa chỉ này nằm ngoài vùng chỉ số của mảng A nên không xác định. Tương tự với trường hợp pa=&A[0], phép toán pa-- cũng đưa pa trở đến một địa chỉ không xác định.



- Vì mảng A là con trỏ hằng, cho nên không thể thực hiện các phép toán trên A mà chỉ có thể thực hiện trên các con trỏ trỏ đến A: các phép toán pa++ hoặc pa-- là hợp lệ, nhưng các phép toán A++ hoặc A-- là không hợp lệ.

Chương trình 2.2a minh họa việc cài đặt một thủ tục sắp xếp các phần tử của một mảng theo cách thông thường.

Chương trình 2.2a

```
void SortArray(int A[], int n){
    int temp;
    for(int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(A[i] > A[j]){
                temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
}
```

Chương trình 2.2b cài đặt một thủ tục tương tự bằng con trỏ. Hai thủ tục này có chức năng hoàn toàn giống nhau.

Chương trình 2.2b

```
void SortArray(int *A, int n){
    int temp;
    for(int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(*(A+i) > *(A+j)){
                temp = *(A+i);
                *(A+i) = *(A+j);
                *(A+j) = temp;
            }
}
```

Trong chương trình 2.2b, thay vì dùng một mảng, ta dùng một con trỏ để trỏ đến mảng cần sắp xếp. Khi đó, ta có thể dùng các thao tác trên con trỏ thay vì các thao tác trên các phần tử mảng.

2.2.2 Con trỏ và mảng nhiều chiều

Con trỏ và mảng nhiều chiều

Một câu hỏi đặt ra là nếu một ma trận một chiều thì tương đương với một con trỏ, vậy một mảng nhiều chiều thì tương đương với con trỏ như thế nào?

Xét ví dụ:

```
int A[3][3] = {
    {5, 10, 15},
    {20, 25, 30},
    {35, 40, 45}
};
```


Khi đó, địa chỉ của ma trận A chính là địa chỉ của hàng đầu tiên của ma trận A, và cũng là địa chỉ của phần tử đầu tiên của hàng đầu tiên của ma trận A:

- Địa chỉ của ma trận A: $A = A[0] = *(A+0) = \&A[0][0];$
- Địa chỉ của hàng thứ nhất: $A[1] = *(A+1) = \&A[1][0];$
- Địa chỉ của hàng thứ i: $A[i] = *(A+i) = \&A[i][0];$
- Địa chỉ phần tử $\&A[i][j] = (*(A+i)) + j;$
- Giá trị phần tử $A[i][j] = *((*(A+i)) + j);$

Như vậy, một mảng hai chiều có thể thay thế bằng một mảng một chiều các con trỏ cùng kiểu:

```
int A[3][3];
```

có thể thay thế bằng:

```
int (*A)[3];
```

Con trỏ trỏ tới con trỏ

Vì một mảng hai chiều `int A[3][3]` có thể thay thế bằng một mảng các con trỏ `int (*A)[3]`. Hơn nữa, một mảng `int A[3]` lại có thể thay thế bằng một con trỏ `int *A`. Do vậy, một mảng hai chiều có thể thay thế bằng một mảng các con trỏ, hoặc một con trỏ trỏ đến con trỏ. Nghĩa là các cách viết sau là tương đương:

```
int A[3][3];
int (*A)[3];
int **A;
```

2.3 CON TRỎ HÀM

Mặc dù hàm không phải là một biến cụ thể nên không có một địa chỉ xác định. Nhưng trong khi chạy, mỗi một hàm trong C++ cũng có một vùng nhớ xác định, do vậy, C++ cho phép dùng con trỏ để trỏ đến hàm. Con trỏ hàm được dùng để truyền tham số có dạng hàm.

Khai báo con trỏ hàm

Con trỏ hàm được khai báo tương tự như khai báo nguyên mẫu hàm thông thường trong C++, ngoại trừ việc có thêm kí hiệu con trỏ “*” trước tên hàm. Cú pháp khai báo con trỏ hàm như sau:

```
<Kiểu dữ liệu trả về> (*<Tên hàm>) (<Các tham số>);
```

Trong đó:

- **Kiểu dữ liệu trả về:** là các kiểu dữ liệu thông thường của C++ hoặc kiểu do người dùng tự định nghĩa.
- **Tên hàm:** tên do người dùng tự định nghĩa, tuân thủ theo quy tắc đặt tên biến trong C++.
- **Các tham số:** có thể có hoặc không (phần trong dấu “[]” là tùy chọn). Nếu có nhiều tham số, mỗi tham số được phân cách nhau bởi dấu phẩy.

Ví dụ khai báo:

```
int (*Calcul)(int a, int b);
```

là khai báo một con trỏ hàm, tên là `Calcul`, có kiểu `int` và có hai tham số cũng là kiểu `int`.

Lưu ý:

- Dấu “()” bao bọc tên hàm là cần thiết để chỉ ra rằng ta đang khai báo một con trỏ hàm. Nếu không có dấu ngoặc đơn này, trình biên dịch sẽ hiểu rằng ta đang khai báo một hàm thông thường và có giá trị trả về là một con trỏ.

Ví dụ, hai khai báo sau là khác nhau hoàn toàn:

```
// Khai báo một con trỏ hàm
int (*Calcul)(int a, int b);

// Khai báo một hàm trả về kiểu con trỏ
int *Calcul(int a, int b);
```

Sử dụng con trỏ hàm

Con trỏ hàm được dùng khi cần gọi một hàm như là tham số của một hàm khác. Khi đó, một hàm được gọi phải có khuôn mẫu giống với con trỏ hàm đã được khai báo.

Ví dụ, với khai báo:

```
int (*Calcul)(int a, int b);
```

thì có thể gọi các hàm có hai tham số kiểu int và trả về cũng kiểu int như sau:

```
int add(int a, int b);
int sub(int a, int b);
```

nhưng không được gọi các hàm khác kiểu tham số hoặc kiểu trả về như sau:

```
int add(float a, int b);
int add(int a);
char* sub(char* a, char* b);
```

Chương trình 2.3 minh họa việc khai báo và sử dụng con trỏ hàm.

Chương trình 2.3

```
#include <ctype.h>
#include <string>

// Hàm có sử dụng con trỏ hàm như tham số
void Display(char[] str, int (*Xtype)(int c)){
    int index = 0;
    while(str[index] != '\0'){
        cout << (*Xtype)(str[index]); // Sử dụng con trỏ hàm
        index ++;
    }
    return;
}

// Hàm main, dùng lời gọi hàm đến con trỏ hàm
void main(){
    char input[500];
    cout << "Enter the string: ";
    cin >> input;
```

```
char reply;
cout << "Display the string in uppercase or lowercase (u,l): ";
cin >> reply;
if(reply == 'l') // Hiển thị theo dạng lowercase
    Display(str, tolower);
else // Hiển thị theo dạng uppercase
    Display(str, toupper);
return;
}
```

Chương trình 2.3 khai báo hàm Display() có sử dụng con trỏ hàm có khuôn mẫu

```
int (*Xtype)(int c);
```

Trong hàm main, con trỏ hàm này được gọi bởi hai thẻ hiện là các hàm tolower() và hàm toupper(). Hai hàm này được khai báo trong thư viện ctype.h với mẫu như sau:

```
int tolower(int c);
int toupper(int c);
```

Hai khuôn mẫu này phù hợp với con trỏ hàm Xtype trong hàm Display() nên lời gọi hàm Display() trong hàm main là hợp lệ.

2.4 CẤP PHÁT BỘ NHỚ ĐỘNG

Xét hai trường hợp sau đây:

- Trường hợp 1, khai báo một con trỏ và gán giá trị cho nó:
- Trường hợp 2, khai báo con trỏ đến phần tử cuối cùng của mảng rồi tăng thêm một đơn vị cho nó:

```
int *pa = 12;

int A[5] = {5, 10, 15, 20, 25};
int *pa = &A[4];
pa++;
```

Trong cả hai trường hợp, ta đều không biết thực sự con trỏ pa đang trỏ đến địa chỉ nào trong bộ nhớ: trường hợp 1 chỉ ra rằng con trỏ pa đang trỏ tới một địa chỉ không xác định, nhưng lại chứa giá trị là 12 do được gán vào. Trường hợp 2, con trỏ pa đã trỏ đến địa chỉ ngay sau địa chỉ phần tử cuối cùng của mảng A, đó cũng là một địa chỉ không xác định. Các địa chỉ không xác định này là các địa chỉ nằm ở vùng nhớ tự do còn thừa của bộ nhớ. Vùng nhớ này có thể bị chiếm dụng bởi bất kỳ một chương trình nào đang chạy.

Do đó, rất có thể các chương trình khác sẽ chiếm mất các địa chỉ mà con trỏ pa đang trỏ tới. Khi đó, nếu các chương trình thay đổi giá trị của địa chỉ đó, giá trị pa cũng bị thay đổi theo mà ta không thể kiểm soát được. Để tránh các rủi ro có thể gặp phải, C++ yêu cầu phải cấp phát bộ nhớ một cách tường minh cho con trỏ trước khi sử dụng chúng.

2.4.1 Cấp phát bộ nhớ động cho biến

Cấp phát bộ nhớ động

Thao tác cấp phát bộ nhớ cho con trỏ thực chất là gán cho con trỏ một địa chỉ xác định và đưa địa chỉ đó vào vùng đã bị chiếm dụng, các chương trình khác không thể sử dụng địa chỉ đó. Cú pháp cấp phát bộ nhớ cho con trỏ như sau:

```
<tên con trỏ> = new <kiểu con trỏ>;
```

Ví dụ, khai báo:

```
int *pa;
pa = new int;
```

sẽ cấp phát bộ nhớ hợp lệ cho con trỏ pa.

Lưu ý:

- Ta có thể vừa cấp phát bộ nhớ, vừa khởi tạo giá trị cho con trỏ theo cú pháp sau:

```
int *pa;
pa = new int(12);
```

sẽ cấp phát cho con trỏ pa một địa chỉ xác định, đồng thời gán giá trị của con trỏ *pa = 12.

Giải phóng bộ nhớ động

Địa chỉ của con trỏ sau khi được cấp phát bởi thao tác **new** sẽ trở thành vùng nhớ đã bị chiếm dụng, các chương trình khác không thể sử dụng vùng nhớ đó ngay cả khi ta không dùng con trỏ nữa. Để tiết kiệm bộ nhớ, ta phải huỷ bỏ vùng nhớ của con trỏ ngay sau khi không dùng đến con trỏ nữa. Cú pháp huỷ bỏ vùng nhớ của con trỏ như sau:

```
delete <tên con trỏ>;
```

Ví dụ:

```
int *pa = new int(12); // Khai báo con trỏ pa, cấp phát bộ nhớ
                        // và gán giá trị ban đầu cho pa là 12.
delete pa;             // Giải phóng vùng nhớ vừa cấp cho pa.
```

Lưu ý:

- Một con trỏ, sau khi bị giải phóng địa chỉ, vẫn có thể được cấp phát một vùng nhớ mới hoặc trỏ đến một địa chỉ mới:

```
int *pa = new int(12); // Khai báo con trỏ pa, cấp phát bộ nhớ
                        // và gán giá trị ban đầu cho pa là 12.
delete pa;             // Giải phóng vùng nhớ vừa cấp cho pa.
int A[5] = {5, 10, 15, 20, 25};
pa = A;                // Cho pa trỏ đến địa chỉ của mảng A
```

- Nếu có nhiều con trỏ cùng trỏ vào một địa chỉ, thì chỉ cần giải phóng bộ nhớ của một con trỏ, tất cả các con trỏ còn lại cũng bị giải phóng bộ nhớ:

```
int *pa = new int(12); // *pa = 12
int *pb = pa;          // pb trỏ đến cùng địa chỉ pa.
*pb += 5;              // *pa = *pb = 17
delete pa;             // Giải phóng cả pa lẫn pb
```

- Một con trỏ sau khi cấp phát bộ nhớ động bằng thao tác **new**, cần phải phóng bộ nhớ trước khi trỏ đến một địa chỉ mới hoặc cấp phát bộ nhớ mới:

```
int *pa = new int(12); // pa được cấp bộ nhớ và *pa = 12
*pa = new int(15);    // pa trỏ đến địa chỉ khác và *pa = 15.
                      // địa chỉ cũ của pa vẫn bị coi là bận
```

2.4.2 Cấp phát bộ nhớ cho mảng động một chiều

Cấp phát bộ nhớ cho mảng động một chiều

Mảng một chiều được coi là tương ứng với một con trỏ cùng kiểu. Tuy nhiên, cú pháp cấp phát bộ nhớ cho mảng động một chiều là khác với cú pháp cấp phát bộ nhớ cho con trỏ thông thường:

```
<Tên con trỏ> = new <Kiểu con trỏ>[<Độ dài mảng>];
```

Trong đó:

- **Tên con trỏ:** tên do người dùng đặt, tuân thủ theo quy tắc đặt tên biến của C++.
- **Kiểu con trỏ:** Kiểu dữ liệu cơ bản của C++ hoặc là kiểu do người dùng tự định nghĩa.
- **Độ dài mảng:** số lượng các phần tử cần cấp phát bộ nhớ của mảng.

Ví dụ:

```
int *A = new int[5];
```

sẽ khai báo một mảng A có 5 phần tử kiểu int được cấp phát bộ nhớ động.

Lưu ý:

- Khi cấp phát bộ nhớ cho con trỏ có khởi tạo thông thường, ta dùng dấu “()”, khi cấp phát bộ nhớ cho mảng, ta dùng dấu “[]”. Hai lệnh cấp phát sau là hoàn toàn khác nhau:

```
// Cấp phát bộ nhớ và khởi tạo cho một con trỏ int
int *A = new int(5);
// Cấp phát bộ nhớ cho một mảng 5 phần tử kiểu int
int *A = new int[5];
```

Giải phóng bộ nhớ của mảng động một chiều

Để giải phóng vùng nhớ đã được cấp phát cho một mảng động, ta dùng cú pháp sau:

```
delete [] <tên con trỏ>;
```

Ví dụ:

```
// Cấp phát bộ nhớ cho một mảng có 5 phần tử kiểu int
int *A = new int[5];
// Giải phóng vùng nhớ do mảng A đang chiếm giữ.
delete [] A;
```

Chương trình 2.4 minh họa hai thủ tục khởi tạo và giải phóng một mảng động một chiều.

Chương trình 2.4

```
void InitArray(int *A, int length){
    A = new int[length];
    for(int i=0; i<length; i++)
```

```

        A[i] = 0;
    return;
}

void DeleteArray(int *A){
    delete [] A;
    return;
}

```

2.4.3 Cấp phát bộ nhớ cho mảng động nhiều chiều

Cấp phát bộ nhớ cho mảng động nhiều chiều

Một mảng hai chiều là một con trỏ đến một con trỏ. Do vậy, ta phải cấp phát bộ nhớ theo từng chiều theo cú pháp cấp phát bộ nhớ cho mảng động một chiều.

Ví dụ:

```

int **A;
const int length = 10;
A = new int*[length];    // Cấp phát bộ nhớ cho số dòng của ma trận A
for(int i=0; i<length; i++)
    // Cấp phát bộ nhớ cho các phần tử của mỗi dòng
    A[i] = new int[length];

```

sẽ cấp phát bộ nhớ cho một mảng động hai chiều, tương đương với một ma trận có kích thước 10*10.

Lưu ý:

- Trong lệnh cấp phát `A = new int*[length]`, cần phải có dấu “*” để chỉ ra rằng cần cấp phát bộ nhớ cho một mảng các phần tử có kiểu là con trỏ `int` (`int*`), khác với kiểu `int` bình thường.

Giải phóng bộ nhớ của mảng động nhiều chiều

Ngược lại với khi cấp phát, ta phải giải phóng lần lượt bộ nhớ cho con trỏ tương ứng với cột và hàng của mảng động.

Ví dụ:

```

int **A;
...;                                // cấp phát bộ nhớ
...
for(int i=0; i<length; i++)
    delete [] A[i];    // Giải phóng bộ nhớ cho mỗi dòng
delete [] A;           // Giải phóng bộ nhớ cho mảng các dòng

```

sẽ giải phóng bộ nhớ cho một mảng động hai chiều.

Chương trình 2.5 minh họa việc dùng mảng động hai chiều để tính tổng của hai ma trận.

Chương trình 2.5

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

/* Khai báo nguyên mẫu hàm */
void InitArray(int **A, int row, int colum);
void AddArray(int **A, int **B, int row, int colum);
void DisplayArray(int **A, int row, int colum);
void DeleteArray(int **A, int row);

void InitArray(int **A, int row, int colum){
    A = new int*[row];
    for(int i=0; i<row; i++){
        A[i] = new int[colum];
        for(int j=0; j<colum; j++){
            cout << "Phan tu [" << i << ", " << j << "] = ";
            cin >> A[i][j];
        }
    }
    return;
}

void AddArray(int **A, int **B, int row, int colum){
    for(int i=0; i<row; i++)
        for(int j=0; j<colum; j++)
            A[i][j] += B[i][j];
    return;
}

void DisplayArray(int **A, int row, int colum){
    for(int i=0; i<row; i++){
        for(int j=0; j<colum; j++)
            cout << A[i][j] << " ";
        cout << endl; // Xuống dòng
    }
    return;
}

void DeleteArray(int **A, int row){
    for(int i=0; i<row; i++)
        delete [] A[i];
    delete [] A;
    return;
}
```

```
void main() {
    clrscr();
    int **A, **B, row, column;
    cout << "So dong: ";
    cin >> row;
    cout << "So cot: ";
    cin >> column;

    /* Khởi tạo các ma trận */
    cout << "Khoi tao mang A:" << endl;
    InitArray(A, row, column);

    cout << "Khoi tao mang B:" << endl;
    InitArray(B, row, column);

    // Cộng hai ma trận
    AddArray(A, B, row, column);

    // Hiển thị ma trận kết quả
    cout << "Tong hai mang A va mang B:" << endl;
    DisplayArray(A, row, column);

    // Giải phóng bộ nhớ
    DeleteArray(A, row);
    DeleteArray(B, row);
    return;
}
```

TỔNG KẾT CHƯƠNG 2

Nội dung chương 2 đã trình bày các vấn đề liên quan đến việc khai báo và sử dụng con trỏ và mảng trong ngôn ngữ C++:

- Con trỏ là một kiểu biến đặc biệt, nó trỏ đến địa chỉ của một biến khác. Có hai cách truy nhập đến con trỏ là truy nhập đến địa chỉ hoặc truy nhập đến giá trị của địa chỉ mà con trỏ trỏ đến.
- Con trỏ có thể tham gia vào các phép toán như các biến thông thường bằng phép lấy giá trị.
- Một con trỏ có sự tương ứng với một mảng một chiều có cùng kiểu.
- Một ma trận hai chiều có thể thay thế bằng một mảng các con trỏ hoặc một con trỏ trỏ đến con trỏ.
- Một con trỏ có thể trỏ đến một hàm, khi đó, nó được dùng để gọi một hàm như là một tham số cho hàm khác.

- Một con trỏ cần phải trỏ vào một địa chỉ xác định hoặc phải được cấp phát bộ nhớ qua phép toán **new** và giải phóng bộ nhớ sau khi dùng bằng thao tác **delete**.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 2

1. Trong các khai báo con trỏ sau, những khai báo nào là đúng:

- a. `int A*;`
- b. `*int A;`
- c. `int* A, B;`
- d. `int* A, *B;`
- e. `int *A, *B;`

2. Với khai báo:

```
int a = 12;  
int *pa;
```

Các phép gán nào sau đây là hợp lệ:

- a. `pa = &a;`
- b. `pa = a;`
- c. `*pa = &a;`
- d. `*pa = a;`

3. Với khai báo:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A+2;
```

Khi đó, `*pa = ?`

- a. 10
- b. 20
- c. 30
- d. 40
- e. 50

4. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
*pa += 2;
```

Khi đó, `*pa = ?`

- a. 10
- b. 12
- c. 30
- d. 32

5. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;
```

```
pa += 2;
```

Khi đó, *pa = ?

- a. 10
- b. 12
- c. 30
- d. 32

6. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
pa += 2;
```

Khi đó, pa = ?

- a. &A[0]
- b. A[2]
- c. &A[2]
- d. Không xác định

7. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
pa -= 2;
```

Khi đó, pa = ?

- a. &A[0]
- b. &A[2]
- c. &A[4]
- d. Không xác định

8. Với đoạn chương trình:

```
int A[3][3] = {  
    {10, 20, 30},  
    {40, 50, 60},  
    {70, 80, 90}  
};  
  
int *pa;
```

Khi đó, để có được kết quả *pa = 50, các lệnh nào sau đây là đúng?

- a. pa = A + 4;
- b. pa = (*(A+1)) + 1;
- c. pa = &A[1][1];
- d. pa = *((*(A+1)) + 1);

9. Giả sử ta khai báo một hàm có sử dụng con trỏ hàm với khuôn mẫu như sau:

```
int Calcul(int a, int b, int (*Xcalcul)(int x, int y)){}  
Và ta có cài đặt một số hàm như sau:
```

```
int add(int a, int b);  
void cal(int a, int b);  
int square(int a);
```

Khi đó, lời gọi hàm nào sau đây là đúng:

- a. `Calcul(5, 10, add);`
- b. `Calcul(5, 10, add(2, 3));`
- c. `Calcul(5, 10, cal);`
- d. `Calcul(5, 10, square);`

10. Ta muốn cấp phát bộ nhớ cho một con trỏ kiểu `int` và khởi đầu giá trị cho nó là 20. Lệnh nào sau đây là đúng:

- a. `int *pa = 20;`
- b. `int *pa = new int{20};`
- c. `int *pa = new int(20);`
- d. `int *pa = new int[20];`

11. Ta muốn cấp phát bộ nhớ cho một mảng động kiểu `int` có chiều dài là 20. Lệnh nào sau đây là đúng:

- a. `int *pa = 20;`
- b. `int *pa = new int{20};`
- c. `int *pa = new int(20);`
- d. `int *pa = new int[20];`

12. Xét đoạn chương trình sau:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
pa = new int(2);
```

Khi đó, `*pa = ?`

- a. 10
- b. 30
- c. 2
- d. Không xác định

13. Xét đoạn chương trình sau:

```
1> int A[5] = {10, 20, 30, 40, 50};  
2> int *pa = A;  
3> pa += 15;  
4> delete pa;
```

Đoạn chương trình trên có lỗi ở dòng nào?

- a. 1
- b. 2
- c. 3
- d. 4

14. Viết chương trình thực hiện các phép toán cộng, trừ, nhân, chia trên đa thức. Các đa thức được biểu diễn bằng mảng động một chiều. Bậc của đa thức và các hệ số tương ứng được nhập từ bàn phím.
15. Viết chương trình thực hiện các phép toán cộng, trừ, nhân hai ma trận kích thước $m \times n$. Các ma trận được biểu diễn bằng mảng động hai chiều. Giá trị kích cỡ ma trận (m, n) và giá trị các phần tử của ma trận được nhập từ bàn phím.



CHƯƠNG 3

Kiểu dữ liệu cấu trúc

Nội dung chương này tập trung trình bày các vấn đề liên quan đến kiểu dữ liệu có cấu trúc trong C++:

- Định nghĩa một cấu trúc
- Sử dụng một cấu trúc bằng các phép toán cơ bản trên cấu trúc
- Con trỏ cấu trúc, khai báo và sử dụng con trỏ cấu trúc
- Mảng các cấu trúc, khai báo và sử dụng mảng các cấu trúc
- Một số kiểu dữ liệu trừu tượng khác như ngăn xếp, hàng đợi, danh sách liên kết.

3.1 ĐỊNH NGHĨA CẤU TRÚC

Kiểu dữ liệu có cấu trúc được dùng khi ta cần nhóm một số biến dữ liệu luôn đi kèm với nhau. Khi đó, việc xử lý trên một nhóm các biến được thực hiện như trên các biến cơ bản thông thường.

3.1.1 Khai báo cấu trúc

Trong C++, một cấu trúc do người dùng tự định nghĩa được khai báo thông qua từ khoá **struct**:

```
struct <Tên cấu trúc>{  
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;  
    <Kiểu dữ liệu 2> <Tên thuộc tính 2>;  
    ...  
    <Kiểu dữ liệu n> <Tên thuộc tính n>;  
};
```

Trong đó:

- **struct**: là tên từ khoá để khai báo một cấu trúc, bắt buộc phải có khi định nghĩa cấu trúc.
- **Tên cấu trúc**: là tên do người dùng tự định nghĩa, tuân thủ theo quy tắc đặt tên biến trong C++. Tên này sẽ trở thành tên của kiểu dữ liệu có cấu trúc tương ứng.
- **Thuộc tính**: mỗi thuộc tính của cấu trúc được khai báo như khai báo một biến thuộc kiểu dữ liệu thông thường, gồm có kiểu dữ liệu và tên biến tương ứng. Mỗi khai báo thuộc tính phải kết thúc bằng dấu chấm phẩy “;” như một câu lệnh C++ thông thường.

Ví dụ, để quản lý nhân viên của một công ty, khi xử lý thông tin về mỗi nhân viên, ta luôn phải xử lý các thông tin liên quan như:

- Tên
- Tuổi
- Chức vụ
- Lương

Do đó, ta sẽ dùng cấu trúc để lưu giữ thông tin về mỗi nhân viên bằng cách định nghĩa một cấu trúc có tên là Employee với các thuộc tính như sau:

```
struct Employeee{
    char name[20];    // Tên nhân viên
    int age;          // Tuổi nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
};
```

Lưu ý:

- Cấu trúc chỉ cần định nghĩa một lần trong chương trình và có thể được khai báo biến cấu trúc nhiều lần. Khi cấu trúc đã được định nghĩa, việc khai báo biến ở lần khác trong chương trình được thực hiện như khai báo biến thông thường:

<Tên cấu trúc> <tên biến 1>, <tên biến 2>;

Ví dụ, sau khi đã định nghĩa cấu trúc Employeee, muốn có biến myEmployeee, ta khai báo như sau:

```
Employee myEmployeee;
```

3.1.2 Cấu trúc lồng nhau

Các cấu trúc có thể được định nghĩa lồng nhau khi một thuộc tính của một cấu trúc cũng cần có kiểu là một cấu trúc khác. Khi đó, việc định nghĩa cấu trúc cha được thực hiện như một cấu trúc bình thường, với khai báo về thuộc tính đó là một cấu trúc con:

```
struct <Tên cấu trúc cha>{
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;
    // Có kiểu cấu trúc
    <Kiểu cấu trúc con> <Tên thuộc tính 2>;
    ...
    <Kiểu dữ liệu n> <Tên thuộc tính n>;
};
```

Ví dụ, với kiểu cấu trúc Employee, ta không quan tâm đến tuổi nhân viên nữa, mà quan tâm đến ngày sinh của nhân viên. Vì ngày sinh cần có các thông tin luôn đi với nhau là ngày sinh, tháng sinh, năm sinh. Do đó, ta định nghĩa một kiểu cấu trúc con cho kiểu ngày sinh:

```
struct Date{
    int day;
    int month;
    int year;
};
```

khi đó, cấu trúc Employee trở thành:

```
struct Employee{
    char name[20];    // Tên nhân viên
    Date birthDay;  // Ngày sinh của nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
};
```

Lưu ý:

- Trong định nghĩa các cấu trúc lồng nhau, cấu trúc con phải được định nghĩa trước cấu trúc cha để đảm bảo các kiểu dữ liệu của các thuộc tính của cấu trúc cha là tường minh tại thời điểm nó được định nghĩa.

3.1.3 Định nghĩa cấu trúc với từ khoá typedef

Để tránh phải dùng từ khoá **struct** mỗi khi khai báo biến cấu trúc, ta có thể dùng từ khoá **typedef** khi định nghĩa cấu trúc:

```
typedef struct {  
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;  
    <Kiểu dữ liệu 2> <Tên thuộc tính 2>;  
    ...  
    <Kiểu dữ liệu n> <Tên thuộc tính n>;  
} <Tên kiểu dữ liệu cấu trúc>;
```

Trong đó:

- **Tên kiểu dữ liệu cấu trúc:** là tên kiểu dữ liệu của cấu trúc vừa định nghĩa. Tên này sẽ được dùng như một kiểu dữ liệu thông thường khi khai báo biến cấu trúc.

Ví dụ, muốn có kiểu dữ liệu có cấu trúc nhân viên, có tên là Employee, ta dùng từ khoá **typedef** để định nghĩa cấu trúc như sau:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    int age;           // Tuổi nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;
```

Khi đó, muốn có hai biến là myEmployee1 và myEmployee2 có kiểu cấu trúc Employee, ta chỉ cần khai báo như sau mà không cần từ khoá **struct**:

```
Employee myEmployee1, myEmployee2;
```

Trong ví dụ khai báo lồng cấu trúc Employee, dùng từ khoá **typedef** cho kiểu Date:

```
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;
```

cấu trúc Employee trở thành:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    Date birthDay;    // Ngày sinh của nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;
```


Lưu ý:

- Khi không dùng từ khoá **typedef**, tên cấu trúc (nằm sau từ khoá **struct**) được dùng để khai báo biến. Trong khi đó, khi có từ khoá **typedef**, tên kiểu dữ liệu cấu trúc (dòng cuối cùng trong định nghĩa) mới được dùng để khai báo biến.
- Khi dùng từ khoá **typedef** thì không thể khai báo biến đồng thời với định nghĩa cấu trúc.

3.2 THAO TÁC TRÊN CẤU TRÚC

Các thao tác trên cấu trúc bao gồm:

- Khai báo và khởi tạo giá trị ban đầu cho biến cấu trúc
- Truy nhập đến các thuộc tính của cấu trúc

3.2.1 Khởi tạo giá trị ban đầu cho cấu trúc

Khởi tạo biến có cấu trúc đơn

Biến cấu trúc được khai báo theo các cách sau:

```
<Tên kiểu dữ liệu cấu trúc> <tên biến>;
```

Ngoài ra, ta có thể khởi tạo các giá trị cho các thuộc tính của cấu trúc ngay khi khai báo bằng các cú pháp sau:

```
<Tên kiểu dữ liệu cấu trúc> <tên biến> = {  
    <giá trị thuộc tính 1>,  
    <giá trị thuộc tính 2>,  
    ...  
    <giá trị thuộc tính n>  
};
```

Trong đó:

- **Giá trị thuộc tính:** là giá trị khởi đầu cho mỗi thuộc tính, có kiểu phù hợp với kiểu dữ liệu của thuộc tính. Mỗi giá trị của thuộc tính được phân cách bằng dấu phẩy “,”.

Ví dụ, với định nghĩa cấu trúc:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    int age;           // Tuổi nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;
```

thì có thể khai báo và khởi tạo cho một biến như sau:

```
Employee myEmployee1 = {  
    "Nguyen Van A",  
    27,  
    "Nhan vien",  
    300f
```



```
};
```

Khởi tạo các biến có cấu trúc lồng nhau

Trong trường hợp các cấu trúc lồng nhau, phép khởi tạo cũng thực hiện như thông thường với phép khởi tạo cho tất cả các cấu trúc con.

Ví dụ với khai báo cấu trúc như sau:

```
typedef struct {
    int day;
    int month;
    int year;
} Date;
```

và:

```
typedef struct {
    char name[20];    // Tên nhân viên
    Date birthDay;    // Ngày sinh của nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
} Employee;
```

Thì khai báo và khởi tạo một biến có kiểu `Employee` có thể thực hiện như sau:

```
Employee myEmployee1 = {
    "Nguyen Van A",
    {15, 05, 1980},    // Khởi tạo cấu trúc con
    "Nhan vien",
    300f
};
```

3.2.2 Truy nhập đến thuộc tính của cấu trúc

Việc truy nhập đến thuộc tính của cấu trúc được thực hiện bằng cú pháp:

<Tên biến cấu trúc>.<tên thuộc tính>

Ví dụ, với một biến cấu trúc kiểu `Employee` đơn:

```
Employee myEmployee1 = {
    "Nguyen Van A",
    27,
    "Nhan vien",
    300f
};
```

ta có thể truy xuất như sau:

```
cout << myEmployee1.name;    // hiển thị ra "Nguyen Van A"
myEmployee1.age += 1;        // Tăng số tuổi lên 1
```

Đối với kiểu cấu trúc lồng nhau, phép truy nhập đến thuộc tính được thực hiện lần lượt từ cấu trúc cha đến cấu trúc con.

Ví dụ, với một biến cấu trúc kiểu `Employee` lồng nhau:

```
Employee myEmployee1 = {  
    "Nguyen Van A",  
    {15, 05, 1980},  
    "Nhan vien",  
    300f  
};
```

ta có thể truy xuất như sau:

```
cout << myEmployee1.name;           // hiển thị ra "Nguyen Van A"  
myEmployee1.birthDay.day = 16;      // Sửa lại ngày sinh thành 16  
myEmployee1.birthDay.month = 07;    // Sửa lại tháng sinh thành 07
```

Chương trình 3.1a minh hoạ việc tạo lập và sử dụng cấu trúc Employee đơn, không dùng từ khoá typedef.

Chương trình 3.1a

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
struct Employee{  
    char name[20];    // Tên nhân viên  
    int age;          // Tuổi nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
};  
  
/* Khai báo khuôn mẫu hàm */  
void Display(Employee myEmployee);  
  
void Display(Employee myEmployee){  
    cout << "Name: " << myEmployee.name << endl;  
    cout << "Age: " << myEmployee.age << endl;  
    cout << "Role: " << myEmployee.role << endl;  
    cout << "Salary: " << myEmployee.salary << endl;  
    return;  
}  
  
void main(){  
    clrscr();  
    // Hiển thị giá trị mặc định  
    Employee myEmployee =
```

```
        {"Nguyen Van A", 27, "Nhan vien", 300f};
    cout << "Thông tin mặc định:" << endl;
    Display(myEmployee);

    // Thay đổi giá trị cho các thuộc tính
    cout << "Name: ";
    cin >> myEmployee.name;
    cout << "Age: ";
    cin >> myEmployee.age;
    cout << "Role: ";
    cin >> myEmployee.role;
    cout << "Salary: ";
    cin >> myEmployee.salary;

    cout << "Thông tin sau khi thay đổi:" << endl;
    Display(myEmployee);
    return;
}
```

Chương trình 3.1b minh họa việc tạo lập và sử dụng cấu trúc Employee lồng nhau, có dùng từ khoá **typedef**.

Chương trình 3.1b

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

typedef struct {
    int day;
    int month;
    int year;
} Date;

typedef struct {
    char name[20]; // Tên nhân viên
    Date birthDay; // Ngày sinh của nhân viên
    char role[20]; // Chức vụ của nhân viên
    float salary; // Lương của nhân viên
} Employee;

/* Khai báo khuôn mẫu hàm */
void Display(Employee myEmployee);
```

```
void Display(Employee myEmployee) {
    cout << "Name: " << myEmployee.name << endl;
    cout << "Birth day: " << myEmployee.birthDay.day << "/"
        << myEmployee.birthDay.month << "/"
        << myEmployee.birthDay.year << endl;
    cout << "Role: " << myEmployee.role << endl;
    cout << "Salary: " << myEmployee.salary << endl;
    return;
}

void main() {
    clrscr();
    // Hiển thị giá trị mặc định
    Employee myEmployee =
        {"Nguyen Van A", {15, 5, 1980}, "Nhan vien", 300f};
    cout << "Thông tin mặc định:" << endl;
    Display(myEmployee);

    // Thay đổi giá trị cho các thuộc tính
    cout << "Name: ";
    cin >> myEmployee.name;
    cout << "Day of birth: ";
    cin >> myEmployee.birthDay.day;
    cout << "Month of birth: ";
    cin >> myEmployee.birthDay.month;
    cout << "Year of birth: ";
    cin >> myEmployee.birthDay.year;
    cout << "Role: ";
    cin >> myEmployee.role;
    cout << "Salary: ";
    cin >> myEmployee.salary;

    cout << "Thông tin sau khi thay đổi:" << endl;
    Display(myEmployee);
    return;
}
```

3.3 CON TRỎ CẤU TRÚC VÀ MẢNG CẤU TRÚC

3.3.1 Con trỏ cấu trúc

Con trỏ cấu trúc là một con trỏ trỏ đến địa chỉ của một biến có kiểu cấu trúc. Cách khai báo và sử dụng con trỏ cấu trúc được thực hiện như con trỏ thông thường.

Khai báo con trỏ cấu trúc

Con trỏ cấu trúc được khai báo theo cú pháp:

```
<Tên kiểu cấu trúc> *(<Tên biến>);
```

Ví dụ, với kiểu khai báo cấu trúc:

```
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;
```

và:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    Date birthDay;    // Ngày sinh của nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;    // Lương của nhân viên  
} Employee;
```

thì ta có thể khai báo một con trỏ cấu trúc như sau:

```
Employee *ptrEmployee;
```

Lưu ý:

- Cũng như khai báo con trỏ thông thường, dấu con trỏ “*” có thể nằm ngay trước tên biến hoặc nằm ngay sau tên kiểu cấu trúc.

Cũng giống con trỏ thông thường, con trỏ cấu trúc được sử dụng khi:

- Cho nó trỏ đến địa chỉ của một biến cấu trúc
- Cấp phát cho nó một vùng nhớ xác định.

Gán địa chỉ cho con trỏ cấu trúc

Một con trỏ cấu trúc có thể trỏ đến địa chỉ của một biến cấu trúc có cùng kiểu thông qua phép gán:

```
<Tên biến con trỏ> = &<Tên biến thường>;
```

Ví dụ, khai báo và phép gán:

```
Employee *ptrEmployee, myEmployee;  
ptrEmployee = &myEmployee;
```

sẽ đưa con trỏ ptrEmployee trỏ đến địa chỉ của biến cấu trúc myEmployee.

Cấp phát bộ nhớ động cho con trỏ cấu trúc

Trong trường hợp ta muốn tạo ra một con trỏ cấu trúc mới, không trỏ vào một biến cấu trúc có sẵn nào, để sử dụng con trỏ mới này, ta phải cấp phát vùng nhớ cho nó. Cú pháp cấp phát vùng nhớ cho con trỏ cấu trúc:

```
<Tên biến con trỏ> = new <Kiểu cấu trúc>;
```

Ví dụ, cấu trúc `Employee` được khai báo bằng từ khoá **typedef**, ta có thể cấp phát vùng nhớ cho con trỏ cấu trúc như sau:

```
Employee *ptrEmployee;  
ptrEmployee = new Employee;
```

hoặc cấp phát ngay khi khai báo:

```
Employee *ptrEmployee = new Employee;
```

Sau khi cấp phát vùng nhớ cho con trỏ bằng thao tác **new**, khi con trỏ không được dùng nữa, hoặc cần trỏ sang một địa chỉ khác, ta phải giải phóng vùng nhớ vừa được cấp phát cho con trỏ bằng thao tác:

```
delete <Tên biến con trỏ>;
```

Ví dụ:

```
Employee *ptrEmployee = new Employee;  
...  
// Thực hiện các thao tác trên con trỏ  
...  
delete ptrEmployee;
```

Lưu ý:

- Thao tác **delete** chỉ được thực hiện đối với con trỏ mà trước đó, nó được cấp phát bộ nhớ động thông qua thao tác **new**:

```
Employee *ptrEmployee = new Employee;  
delete ptrEmployee; //đúng
```

mà không thể thực hiện với con trỏ chỉ trỏ đến địa chỉ của một biến cấu trúc khác:

```
Employee *ptrEmployee, myEmployee;  
ptrEmployee = &myEmployee;  
delete ptrEmployee; //lỗi
```

Truy nhập thuộc tính của con trỏ cấu trúc

Thuộc tính của con trỏ cấu trúc có thể được truy nhập thông qua hai cách:

Cách 1:

```
<Tên biến con trỏ> -> <Tên thuộc tính>;
```

Cách 2:

```
(*<Tên biến con trỏ>).<Tên thuộc tính>;
```

Ví dụ, thuộc tính tên nhân viên của cấu trúc `Employee` có thể được truy nhập thông qua hai cách:

```
Employee *ptrEmployee = new Employee;  
cin >> ptrEmployee -> name;
```

hoặc:

```
cin >> (*ptrEmployee).name;
```

Lưu ý:

- Trong cách truy nhập thứ hai, phải có dấu ngoặc đơn “()” quanh tên con trỏ vì phép toán truy nhập thuộc tính “.” có độ ưu tiên cao hơn phép toán lấy giá trị con trỏ “*”.
- Thông thường, ta dùng cách thứ nhất cho đơn giản và thuận tiện.

Chương trình 3.2 cài đặt việc khởi tạo và hiển thị nội dung của một con trỏ cấu trúc.

Chương trình 3.2

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

typedef struct {
    int day;
    int month;
    int year;
} Date;

typedef struct {
    char name[20];    // Tên nhân viên
    Date birthDay;    // Ngày sinh của nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
} Employee;

/* Khai báo khuôn mẫu hàm */
void InitStruct(Employee *myEmployee);
void Display(Employee *myEmployee);

void InitStruct(Employee *myEmployee){
    myEmployee = new Employee;
    cout << "Name: ";
    cin >> myEmployee->name;
    cout << "Day of birth: ";
    cin >> myEmployee->birthDay.day;
    cout << "Month of birth: ";
    cin >> myEmployee->birthDay.month;
    cout << "Year of birth: ";
    cin >> myEmployee->birthDay.year;
    cout << "Role: ";
    cin >> myEmployee->role;
    cout << "Salary: ";
```



```
    cin >> myEmployee->salary;
}

void Display(Employee myEmployee) {
    cout << "Name: " << myEmployee->name << endl;
    cout << "Birth day: " << myEmployee->birthDay.day << "/"
        << myEmployee->birthDay.month << "/"
        << myEmployee->birthDay.year << endl;
    cout << "Role: " << myEmployee->role << endl;
    cout << "Salary: " << myEmployee->salary << endl;
    return;
}

void main() {
    clrscr();
    Employee *myEmployee;
    InitStruct(myEmployee);
    Display(myEmployee);
    return;
}
```

3.3.2 Mảng cấu trúc

Khi cần xử lý nhiều đối tượng có dùng kiểu dữ liệu cấu trúc, ta có thể sử dụng mảng các cấu trúc. Vì một mảng một chiều là tương đương với một con trỏ có cùng kiểu. Do đó, có thể khai báo mảng theo hai cách: Khai báo mảng tĩnh như thông thường hoặc khai báo mảng động thông qua con trỏ.

Khai báo mảng tĩnh các cấu trúc

Khai báo mảng tĩnh các cấu trúc theo cú pháp:

<Tên kiểu cấu trúc> <Tên biến mảng>[<Số phần tử mảng>];

Ví dụ:

```
Employee employees[10];
```

là khai báo một mảng tên là employees gồm 10 phần tử có kiểu là cấu trúc Employee.

Khai báo mảng động các cấu trúc

Khai báo một mảng động các cấu trúc hoàn toàn tương tự khai báo một con trỏ cấu trúc cùng kiểu:

<Tên kiểu cấu trúc> *<Tên biến>;

Ví dụ, khai báo:

```
Employee *employees;
```

vừa có thể coi là khai báo một con trỏ thông thường có cấu trúc Employee, vừa có thể coi là khai báo một mảng động các cấu trúc có kiểu cấu trúc Employee.

Tuy nhiên, cách cấp phát bộ nhớ động cho mảng các cấu trúc khác với một con trỏ. Đây là cách để chương trình nhận biết ta đang dùng một con trỏ cấu trúc hay một mảng động có cấu trúc. Cách pháp cấp phát bộ nhớ cho mảng động như sau:

```
<Tên biến mảng> = new <Kiểu cấu trúc>[<Số lượng phần tử>];
```

Ví dụ, khai báo:

```
Employee *employees = new Employee[10];
```

sẽ cấp phát bộ nhớ cho một mảng động employees có 10 phần tử kiểu cấu trúc Employee.

Truy nhập đến phần tử của mảng cấu trúc

Việc truy nhập đến các phần tử của mảng cấu trúc được thực hiện như truy cập đến phần tử của mảng thông thường. Ví dụ muốn truy nhập đến thuộc tính tên nhân viên phần tử nhân viên thứ i trong mảng cấu trúc, ta viết như sau:

```
Employee *employees = new Employee[10];  
employees[i].name;
```

Chương trình 3.3 cài đặt việc khởi tạo một mảng các nhân viên của một phòng trong một công ty. Sau đó, chương trình sẽ tìm và in ra thông tin về nhân viên có lương cao nhất và nhân viên có lương thấp nhất trong phòng.

Chương trình 3.3

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;  
  
typedef struct {  
    char name[20];    // Tên nhân viên  
    Date birthDay;    // Ngày sinh của nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;  
  
/* Khai báo khuôn mẫu hàm */  
void InitArray(Employee *myEmployee, int length);  
Employee searchSalaryMax(Employee *myEmployee, int length);  
Employee searchSalaryMin(Employee *myEmployee, int length);  
void Display(Employee myEmployee);
```

```
void InitArray(Employee *myEmployee, int length){
    myEmployee = new Employee[length];
    for(int i=0; i<length; i++){
        cout << "Nhan vien thu " << i << endl;
        cout << "Name: ";
        cin >> myEmployee[i].name;
        cout << "Day of birth: ";
        cin >> myEmployee[i].birthDay.day;
        cout << "Month of birth: ";
        cin >> myEmployee[i].birthDay.month;
        cout << "Year of birth: ";
        cin >> myEmployee[i].birthDay.year;
        cout << "Role: ";
        cin >> myEmployee[i].role;
        cout << "Salary: ";
        cin >> myEmployee[i].salary;
    }
    return;
}

Employee searchSalaryMax(Employee *myEmployee, int length){
    int index = 0;
    int maxSalary = myEmployee[0].salary;
    for(int i=1; i<length; i++){
        if(myEmployee[i].salary > maxSalary){
            maxSalary = myEmployee[i].salary;
            index = i;
        }
    }
    return myEmployee[index];
}

Employee searchSalaryMin(Employee *myEmployee, int length){
    int index = 0;
    int minSalary = myEmployee[0].salary;
    for(int i=1; i<length; i++){
        if(myEmployee[i].salary < minSalary){
            minSalary = myEmployee[i].salary;
            index = i;
        }
    }
    return myEmployee[index];
}

void Display(Employee myEmployee){
```

```
cout << "Name: " << myEmployee.name << endl;
cout << "Birth day: " << myEmployee.birthDay.day << "/"
    << myEmployee.birthDay.month << "/"
    << myEmployee.birthDay.year << endl;
cout << "Role: " << myEmployee.role << endl;
cout << "Salary: " << myEmployee.salary << endl;
return;
}

void main(){
    clrscr();
    Employee *myEmployee, tmpEmployee;
    int length = 0;
    cout << "So luong nhan vien: ";
    cin >> length;

    // Khởi tạo danh sách nhân viên
    InitArray(myEmployee);

    // Nhân viên có lương cao nhất
    tmpEmployee = searchSalaryMax(myEmployee, length);
    Display(tmpEmployee);

    // Nhân viên có lương thấp nhất
    tmpEmployee = searchSalaryMin(myEmployee, length);
    Display(tmpEmployee);

    // Giải phóng vùng nhớ
    delete [] myEmployee;
    return;
}
```

3.4 MỘT SỐ KIỂU DỮ LIỆU TRỪU TƯỢNG

Nội dung phần này tập trung trình bày việc cài đặt một số cấu trúc dữ liệu trừu tượng, bao gồm:

- Ngăn xếp (stack)
- Hàng đợi (queue)
- Danh sách liên kết (list)

3.4.1 Ngăn xếp

Ngăn xếp (stack) là một kiểu danh sách cho phép thêm và bớt các phần tử ở một đầu danh sách, gọi là đỉnh của ngăn xếp. Ngăn xếp hoạt động theo nguyên lý: phần tử nào được đưa vào sau, sẽ được lấy ra trước.

Định nghĩa cấu trúc ngăn xếp

Vì ta chỉ cần quan tâm đến hai thuộc tính của ngăn xếp là:

- Danh sách các phần tử của ngăn xếp
- Vị trí đỉnh của ngăn xếp

nên ta có thể định nghĩa cấu trúc ngăn xếp như sau (các phần tử của ngăn xếp có kiểu int):

```
typedef SIZE 100;
typedef struct {
    int top;           // Vị trí của đỉnh
    int nodes[SIZE];  // Danh sách các phần tử
} Stack;
```

Tuy nhiên, định nghĩa này tồn tại một vấn đề, đó là kích thước (SIZE) của danh sách chứa các phần tử là tĩnh. Do đó:

- Nếu ta chọn SIZE lớn, nhưng khi gặp ứng dụng chỉ cần một số ít phần tử cho ngăn xếp thì rất tốn bộ nhớ.
- Nếu ta khai báo SIZE nhỏ, thì khi gặp bài toán cần ngăn xếp có nhiều phần tử, ta sẽ không thêm được các phần tử mới vào, chương trình sẽ có lỗi.

Để khắc phục hạn chế này, ta có thể sử dụng bộ nhớ động (mảng động thông qua con trỏ) để lưu danh sách các phần tử của ngăn xếp. Khi đó, định nghĩa cấu trúc ngăn xếp sẽ có dạng như sau:

```
typedef struct {
    int top;           // Vị trí của đỉnh
    int *nodes;        // Danh sách các phần tử
} Stack;
```

Ta sẽ sử dụng định nghĩa này trong các chương trình ứng dụng ngăn xếp.

Các thao tác trên ngăn xếp

Đối với các thao tác trên ngăn xếp, ta quan tâm đến hai thao tác cơ bản:

- Thêm một phần tử mới vào đỉnh ngăn xếp, gọi là **push**.
- Lấy ra một phần tử từ đỉnh ngăn xếp, gọi là **pop**.

Khi thêm một phần tử mới vào ngăn xếp, ta làm các bước như sau:

1. Số phần tử trong ngăn xếp cũ là (top+1). Do đó, ta cấp phát một vùng nhớ mới để lưu được (top+1+1) = (top+2) phần tử.
2. Sao chép (top+1) phần tử cũ sang vùng mới. Nếu danh sách ban đầu rỗng (top = -1) thì không cần thực hiện bước này.
3. Thêm phần tử mới vào cuối vùng nhớ mới
4. Giải phóng vùng nhớ của danh sách cũ

5. Cho danh sách nodes trở vào vùng nhớ mới.

Chương trình 3.4a cài đặt thủ tục thêm một phần tử mới vào ngăn xếp.

Chương trình 3.4a

```
void push(Stack *stack, int node){
    int *tmpNodes = new int[stack->top + 2]; // Cấp phát vùng nhớ mới
    stack->top++; // Tăng chỉ số của node đỉnh
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    tmpNodes[stack->top] = node; // Thêm node mới vào đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return;
}
```

Khi lấy ra một phần tử của ngăn xếp, ta làm các bước như sau:

- Kiểm tra xem ngăn xếp có rỗng ($top = -1$) hay không. Nếu không rỗng thì thực hiện các bước tiếp theo.
- Lấy phần tử ở đỉnh ngăn xếp ra
- Cấp phát một vùng nhớ mới có $(top+1) - 1 = top$ phần tử
- Sao chép top phần tử từ danh sách cũ sang vùng nhớ mới (trừ phần tử ở đỉnh).
- Giải phóng vùng nhớ cũ
- Cho con trỏ danh sách trở vào vùng nhớ mới.
- Trả về giá trị phần tử ở đỉnh đã lấy ra.

Chương trình 3.4b cài đặt thủ tục lấy một phần tử từ ngăn xếp.

Chương trình 3.4b

```
int pop(Stack *stack){
    if(stack->top < 0){ // Kiểm tra ngăn xếp rỗng
        cout << "Stack is empty!" << endl;
        return 0;
    }
    int result = stack->nodes[stack->top]; // Lưu giữ giá trị đỉnh
    int *tmpNodes = new int[stack->top]; // Cấp phát vùng nhớ mới
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    stack->top--; // Giảm chỉ số của node đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return result; // Trả về giá trị node đỉnh
}
```

```
}
```

Áp dụng

Ngăn xếp được sử dụng trong các ứng dụng thỏa mãn nguyên tắc: cái nào đặt vào trước sẽ được lấy ra sau. Chương trình 3.4c minh họa việc dùng ngăn xếp để đảo ngược một chuỗi ký tự được nhập vào từ bàn phím.

Chương trình 3.4c

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

typedef struct {
    int top;                // Vị trí node đỉnh
    int *nodes;             // Danh sách phần tử
} Stack;

/* Khai báo nguyên mẫu hàm */
void init(Stack *stack);
void push(Stack *stack, int node);
int pop(Stack *stack);
void release(Stack *stack);

void init(Stack *stack){
    stack = new Stack;     // Cấp phát vùng nhớ cho con trỏ
    stack->top = -1;        // Khởi tạo ngăn xếp rỗng
}

void push(Stack *stack, int node){
    int *tmpNodes = new int[stack->top + 2]; // Cấp phát vùng nhớ mới
    stack->top++;                // Tăng chỉ số của node đỉnh
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    tmpNodes[stack->top] = node; // Thêm node mới vào đỉnh
    delete [] stack->nodes;     // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes;    // Trỏ vào vùng nhớ mới
    return;
}

int pop(Stack *stack){
    if(stack->top < 0){         // Kiểm tra ngăn xếp rỗng
```



```

        cout << "Stack is empty!" << endl;
        return 0;
    }

    int result = stack->nodes[stack->top]; // Lưu giữ giá trị đỉnh
    int *tmpNodes = new int[stack->top]; // Cấp phát vùng nhớ mới
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    stack->top--; // Giảm chỉ số của node đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return result; // Trả về giá trị node đỉnh
}

void release(Stack *stack){
    delete [] stack->nodes; // Giải phóng vùng danh sách
    delete stack; // Giải phóng con trỏ
    return;
}

void main(){
    clrscr();
    Stack *stack;
    init(stack); // Khởi tạo ngăn xếp
    char strIn[250];
    // Nhập chuỗi kí tự từ bàn phím
    cout << "Nhap chuoi: ";
    cin >> strIn;
    for(int i=0; i<strlen(strIn); i++) // Đặt vào ngăn xếp
        push(stack, strIn[i]);
    while(stack->top > -1) // Lấy ra từ ngăn xếp
        cout << pop(stack);
    release(stack); // Giải phóng bộ nhớ
    return;
}

```

3.4.2 Hàng đợi

Hàng đợi (queue) cũng là một cấu trúc tuyến tính các phần tử. Trong đó, các phần tử luôn được thêm vào ở một đầu, gọi là đầu cuối hàng đợi, và việc lấy ra các phần tử luôn được thực hiện ở đầu còn lại, gọi là đầu mặt của hàng đợi. Hàng đợi hoạt động theo nguyên lý: phần tử nào được đưa vào trước, sẽ được lấy ra trước.

Định nghĩa cấu trúc hàng đợi

Hàng đợi có các thuộc tính:

- Một danh sách các phần tử có mặt trong hàng đợi.
- Chỉ số của phần tử đứng đầu của danh sách (front).
- Chỉ số phần tử cuối của danh sách (rear).

Nếu dùng cấu trúc tĩnh để định nghĩa, hàng đợi có cấu trúc như sau:

```
typedef SIZE 100;
typedef struct {
    int front, rear; // Vị trí của đỉnh đầu, đỉnh cuối
    int nodes[SIZE]; // Danh sách các phần tử
} Queue;
```

Nếu dùng bộ nhớ động để lưu giữ hàng đợi, thì phần tử front luôn là phần tử thứ 0 của danh sách. Và rear sẽ bằng độ dài danh sách trừ đi 1. Cấu trúc động của hàng đợi:

```
typedef struct {
    int front, rear; // Vị trí của đỉnh đầu, đỉnh cuối
    int *nodes; // Danh sách các phần tử
} Queue;
```

Thao tác trên hàng đợi

- Thêm một phần tử vào cuối hàng đợi
- Lấy một phần tử ở vị trí đầu của hàng đợi

Thao tác thêm một phần tử vào cuối hàng đợi với bộ nhớ động được thực hiện tương tự với ngăn xếp. Chương trình 3.5a cài đặt thủ tục thêm một phần tử vào cuối hàng đợi động.

Chương trình 3.5a

```
void insert(Queue *queue, int node){
    int *tmpNodes = new int[queue->rear + 2]; // Cấp phát vùng nhớ mới
    queue->rear++; // Tăng chỉ số của node đuôi
    if(queue->front == -1) // Nếu hàng đợi cũ rỗng
        queue->front = 0; // thì cập nhật front
    for(int i=0; i<queue->rear; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = queue->nodes[i];
    tmpNodes[queue->rear] = node; // Thêm node mới vào đuôi
    delete [] queue->nodes; // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes; // Trỏ vào vùng nhớ mới
    return;
}
```

Thao tác lấy ra một phần tử đầu của hàng đợi thực hiện theo các bước:

1. Kiểm tra tính rỗng (front = rear = -1) của hàng đợi. Nếu không rỗng mới thực hiện tiếp

2. Lấy phần tử nodes[0] ra.
3. Sao chép danh sách còn lại sang vùng nhớ mới
4. Giải phóng vùng nhớ cũ
5. Đưa danh sách trở vào vùng nhớ mới
6. Trả về giá trị phần tử lấy ra

Chương trình 3.5b cài đặt thủ tục lấy ra một phần tử của hàng đợi động.

Chương trình 3.5b

```
int remove(Queue *queue) {
    if((queue->front < 0) || (queue->rear < 0)) { // Kiểm tra hàng đợi rỗng
        cout << "Queue is empty!" << endl;
        return 0;
    }
    // Lưu giữ giá trị phần tử đầu
    int result = queue->nodes[queue->front];
    int *tmpNodes;
    if(queue->rear > 0) { // Nếu có hơn 1 phần tử
        tmpNodes = new int[queue->rear]; // Cấp phát vùng nhớ mới
        for(int i=0; i<queue->rear; i++) // Sao chép sang vùng nhớ mới
            tmpNodes[i] = queue->nodes[i];
    } else // Nếu chỉ có 1 phần tử
        queue->front --; // Hàng đợi thành rỗng
    queue->rear --; // Giảm chỉ số của node đuôi
    delete [] queue->nodes; // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return result; // Trả về giá trị node đầu
}
```

Áp dụng

Hàng đợi được áp dụng trong các bài toán cần cơ chế quản lý cái nào vào trước sẽ được lấy ra trước. Chương trình 3.5c minh họa cơ chế quản lý tiến trình đơn giản nhất của hệ điều hành: các tiến trình được quản lý theo mã tiến trình, khi xuất hiện, tiến trình được đưa vào cuối của một hàng đợi. Khi nào CPU rảnh thì sẽ lấy tiến trình đầu hàng đợi ra để thực hiện.

Chương trình 3.5c

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

typedef struct {
    int front, rear; // Vị trí của đỉnh đầu, đỉnh cuối
}
```

```

        int *nodes;                                // Danh sách các phần tử
    } Queue;

/* Khai báo các nguyên mẫu hàm */
void init(Queue *queue);
void insert(Queue *queue, int node);
int remove(Queue *queue);
void traverse(Queue *queue);
void release(Queue *queue);

void init(Queue *queue){
    queue = new Queue;                            // Cấp phát bộ nhớ cho con trỏ
    queue->front = -1;                             // Khởi tạo danh sách rỗng
    queue->rear = -1;
    return;
}

void insert(Queue *queue, int node){
    int *tmpNodes = new int[queue->rear + 2];      // Cấp phát vùng nhớ mới
    queue->rear++;                                  // Tăng chỉ số của node đuôi
    if(queue->front == -1)                          // Nếu hàng đợi cũ rỗng
        queue->front = 0;                          // thì cập nhật front
    for(int i=0; i<queue->rear; i++)               // Sao chép sang vùng nhớ mới
        tmpNodes[i] = queue->nodes[i];
    tmpNodes[queue->rear] = node;                  // Thêm node mới vào đuôi
    delete [] queue->nodes;                        // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes;                        // Trỏ vào vùng nhớ mới
    return;
}

int remove(Queue *queue){
    if((queue->front < 0) || (queue->rear < 0)){// Kiểm tra hàng đợi rỗng
        cout << "Queue is empty!" << endl;
        return 0;
    }
    // Lưu giữ giá trị phần tử đầu
    int result = queue->nodes[queue->front];
    int *tmpNodes;
    if(queue->rear > 0){                            // Nếu có hơn 1 phần tử
        tmpNodes = new int[queue->rear];           // Cấp phát vùng nhớ mới
        for(int i=0; i<queue->rear; i++)           // Sao chép sang vùng nhớ mới
            tmpNodes[i] = queue->nodes[i];
    }else                                           // Nếu chỉ có 1 phần tử

```

```

        queue->front --;                // Hàng đợi thành rỗng
queue->rear --;                        // Giảm chỉ số của node cuối
delete [] queue->nodes;                // Giải phóng vùng nhớ cũ
queue->nodes = tmpNodes;               // Trỏ vào vùng nhớ mới
return result;                        // Trả về giá trị node đầu
}

void traverse(Queue *queue){
    if(queue->front < 0){                // Khi danh sách rỗng
        cout << "Danh sach rong!" << endl;
        return;
    }
    for(int i=queue->front; i<=queue.rear; i++)
        cout << queue->nodes[i] << " "; // Liệt kê các phần tử
    cout << endl;
    return;
}

void release(Queue *queue){
    if(queue->front > -1)                //Nếu danh sách không rỗng thì
        delete [] queue->nodes; //giải phóng vùng nhớ của danh sách
    delete queue;                      //Giải phóng vùng nhớ của con trỏ
}

void main(){
    clrscr();
    Queue *queue;
    init(queue);                        // Khởi tạo hàng đợi
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot tien trinh vao hang doi" << endl;
        cout << "2: Dua mot tien trinh trinh vao thuc hien" << endl;
        cout<<"3: Xem tat ca cac tien trinh trong hang doi" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1':                    // Thêm vào hàng đợi
                int maso;
                cout << "Ma so tien trinh vao hang doi: ";

```

```

        cin >> maso;
        insert(queue, maso);
        break;
    case '2':                // Lấy ra khỏi hàng đợi
        cout << "Tien trinh duoc thuc hien: " <<
            remove(queue) << endl;
        break;
    case '3':                // Duyệt hàng đợi
        cout<<"Cac tien trinh dang o trong hang doi la:"
            <<endl;
        traverse(queue);
        break;
    }while(function != '5');
    release(queue);          // Giải phóng hàng đợi
    return;
}

```

3.4.3 Danh sách liên kết

Danh sách liên kết là một kiểu dữ liệu bao gồm một dãy các phần tử có thứ tự, các phần tử có cùng cấu trúc dữ liệu, ngoại trừ node đầu tiên của danh sách là node lưu thông tin về danh sách. Có hai loại danh sách liên kết:

- Danh sách liên kết đơn: mỗi node có một con trỏ trỏ đến node tiếp theo trong danh sách
- Danh sách liên kết kép: mỗi node có hai con trỏ, một trỏ vào node trước, một trỏ vào node tiếp theo trong danh sách.

Trong phần này sẽ trình bày danh sách liên kết đơn. Danh sách liên kết kép được coi như là một bài tập mở rộng từ danh sách liên kết đơn.

Định nghĩa danh sách đơn

Mỗi node của danh sách đơn chứa dữ liệu của nó, đồng thời trỏ đến node tiếp theo trong danh sách, cấu trúc một node như sau:

```

struct simple{
    Employee employee;    // Dữ liệu của node có kiểu Employee
    struct simple *next;  // Trỏ đến node kế tiếp
};
typedef struct simple SimpleNode;

```

Node đầu của danh sách đơn có cấu trúc riêng, nó không chứa dữ liệu như node thường mà chứa các thông tin:

- Số lượng node trong danh sách (không kể bản thân nó – node đầu)
- Con trỏ đến node đầu tiên của danh sách
- Con trỏ đến node cuối cùng của danh sách

Do vậy, cấu trúc node đầu của danh sách đơn là:

```
typedef struct{
    int nodeNumber;          // Số lượng các node
    SimpleNode *front, *rear;// Trỏ đến node đầu và cuối danh sách
} SimpleHeader;
```

Các thao tác trên danh sách liên kết đơn

Các thao tác cơ bản trên danh sách đơn bao gồm:

- Chèn thêm một node vào vị trí thứ n trong danh sách
- Loại ra một node ở vị trí thứ n trong danh sách

Việc chèn thêm một node vào vị trí thứ n trong danh sách được thực hiện theo các bước:

1. Nếu $n \leq 0$, chèn vào đầu. Nếu $n > \text{số phần tử của danh sách}$, chèn vào cuối. Trường hợp còn lại, chèn vào giữa.
2. Tìm node thứ n: giữ vết của hai node thứ n-1 và thứ n.
3. Tạo một node mới: cho node thứ n-1 trỏ tiếp vào node mới và node mới trỏ tiếp vào node thứ n.

Chương trình 3.6a cài đặt thủ tục chèn một node vào vị trí thứ n của danh sách.

Chương trình 3.6a

```
void insert(SimpleHeader *list, int position, int value){
    SimpleNode *newNode = new SimpleNode;
    newNode->value = value;
    if(position <= 0){                // Chèn vào đầu ds
        newNode->next = list->front; // Chèn vào trước node đầu
        list->front = newNode;        // Cập nhật lại node đầu ds
        if(list->nodeNumber == 0)     // Nếu ds ban đầu rỗng thì
            list->rear = newNode;     // node đầu trùng với node cuối
    }else if(position >= list->nodeNumber){ // Chèn vào cuối ds
        list->rear->next = newNode;   // Chèn vào sau node cuối
        list->rear = newNode;        // Cập nhật lại node cuối ds
        if(list->nodeNumber == 0)     // Nếu ds ban đầu rỗng thì
            list->front = newNode;    // node đầu trùng node cuối
    }else{                            // Chèn vào giữa ds
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){      // tìm node n-1 và n
            prev = curr;
            curr = curr->next;
            index++;
        }
        newNode->next = curr;         // chèn vào trước node n
        prev->next = newNode;         // và chèn vào sau node n-1
    }
}
```

```
list->nodeNumber++;           // Cập nhật số lượng node
return;
}
```

Việc xóa một node ở vị trí thứ n trong danh sách được thực hiện theo các bước:

1. Nếu $n < 0$ hoặc $n > \text{số phần tử của danh sách}$, không xóa node nào.
2. Tìm node thứ n: giữ vết của ba node thứ n-1, thứ n và thứ n+1.
3. Cho node thứ n-1 trở tiếp vào node thứ n+1, xóa con trỏ của node thứ n.
4. Trả về node thứ n.

Chương trình 3.6b cài đặt thủ tục xóa một node ở vị trí thứ n của danh sách.

Chương trình 3.6b

```
SimpleNode* remove(SimpleHeader *list, int position){
    if((position < 0) || (position >= list->nodeNumber))
        return NULL;           // Không xóa node nào cả
    SimpleNode* result;
    if(position == 0){           // Xóa node đầu
        result = list->front;     // Giữ node cần xóa
        list->front = list->front->next; // Cập nhật node đầu
        if(list->nodeNumber == 1) // Nếu ds chỉ có 1 node thì
            list->rear = list->front; // Cập nhật node cuối ds
        }else if(position == list->nodeNumber - 1){
            result = list->rear;   // Giữ node cần xóa
            SimpleNode *curr = list->front;
            while(curr->next != list->rear)
                curr = curr->next; // Tìm node trước của node cuối
            curr->next = NULL;     // Xóa node rear hiện tại
            list->rear = curr;     // Cập nhật node cuối ds
        }else{
            SimpleNode *prev = list->front, *curr = list->front;
            int index = 0;
            while(index < position){ // Tìm node n-1 và n
                prev = curr;
                curr = curr->next;
                index++;
            }
            result = curr;          // Giữ node cần xóa
            prev->next = curr->next; // Cho node n-1 trở đến node n+1
        }
        list->nodeNumber --;       // Cập nhật số lượng node
        return result;            // Trả về node cần xóa
    }
```

```
}
```

Áp dụng

Chương trình 3.6c minh họa việc dùng danh sách liên kết đơn để quản lý nhân viên văn phòng với các thông tin rất đơn giản: tên, tuổi và tiền lương của mỗi nhân viên.

Chương trình 3.6c

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

typedef struct{
    char name[25];           // Tên nhân viên
    int age;                 // Tuổi nhân viên
    float salary;           // Lương nhân viên
} Employee;

struct simple{
    Employee employee;       // Dữ liệu của node
    struct simple *next;     // Trỏ đến node kế tiếp
};

typedef struct simple SimpleNode;

typedef struct{
    int nodeNumber;         // Số lượng các node
    SimpleNode *front, *rear; // Trỏ đến node đầu và cuối ds
} SimpleHeader;

/* Khai báo các nguyên mẫu hàm */
void init(SimpleHeader *list);
void insert(SimpleHeader *list, int position, Employee employee);
SimpleNode* remove(SimpleHeader *list);
void traverse(SimpleHeader *list);
void release(SimpleHeader *list);

void init(SimpleHeader *list){
    list = new list;        // Cấp phát bộ nhớ cho con trỏ
    list->front = NULL;     // Khởi tạo danh sách rỗng
    list->rear = NULL;
```



```

    list->nodeNumber = 0;
    return;
}

void insert(SimpleHeader *list, int position, Employee employee){
    SimpleNode *newNode = new SimpleNode;
    newNode->employee = employee;
    if(position <= 0){
        // Chèn vào đầu ds
        newNode->next = list->front; // Chèn vào trước node đầu
        list->front = newNode;       // Cập nhật lại node đầu ds
        if(list->nodeNumber == 0)    // Nếu ds ban đầu rỗng thì
            list->rear = newNode;   // node đầu trùng với node đầu
    }else if(position >= list->nodeNumber){ // Chèn vào cuối ds
        list->rear->next = newNode; // Chèn vào sau node cuối
        list->rear = newNode;       // Cập nhật lại node cuối ds
        if(list->nodeNumber == 0) // Nếu ds ban đầu rỗng thì
            list->front = newNode; // node đầu trùng node cuối
    }else{
        // Chèn vào giữa ds
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){ // tìm node n-1 và n
            prev = curr;
            curr = curr->next;
            index++;
        }
        newNode->next = curr; // chèn vào trước node n
        prev->next = newNode; // và chèn vào sau node n-1
    }
    list->nodeNumber++; // Cập nhật số lượng node
    return;
}

SimpleNode* remove(SimpleHeader *list, int position){
    if((position < 0) || (position >= list->nodeNumber))
        return NULL; // Không xóa node nào cả
    SimpleNode* result;
    if(position == 0){
        // Xóa node đầu
        result = list->front; // Giữ node cần xóa
        list->front = list->front->next; // Cập nhật node đầu
        if(list->nodeNumber == 1) // Nếu ds chỉ có 1 node thì
            list->rear = list->front; // Cập nhật node cuối ds
    }else if(position == list->nodeNumber - 1){
        result = list->rear; // Giữ node cần xóa
    }
}

```



```

        SimpleNode *curr = list->front;
        while(curr->next != list->rear)
            curr = curr->next; // Tìm node trước của node cuối
        curr->next = NULL;      // Xoá node rear hiện tại
        list->rear = curr;      // Cập nhật node cuối ds
    }else{
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){ // Tìm node n-1 và n
            prev = curr;
            curr = curr->next;
            index++;
        }
        result = curr;          // Giữ node cần xoá
        prev->next = curr->next; // Cho node n-1 trở đến node n+1
    }
    list->nodeNumber--;         // Cập nhật số lượng node
    return result;              // Trả về node cần xoá
}

void traverse(SimpleHeader *list){
    if(list->nodeNumber <= 0){ // Khi danh sách rỗng
        cout << "Danh sach rong!" << endl;
        return;
    }
    SimpleNode *curr = list->front;
    while(curr != NULL){
        cout << curr->employee.name << " "
            << curr->employee.age << " "
            << curr->employee.salary << endl; // Liệt kê các phần tử
        curr = curr->next;
    }
    return;
}

void release(SimpleHeader *list){
    SimpleNode* curr = remove(list, 0);
    while(curr != NULL){
        delete curr;          //Giải phóng vùng nhớ của node
        curr = remove(list, 0);
    }
    delete list;              //Giải phóng vùng nhớ của con trỏ
}

```

```

void main() {
    clrscr();
    SimpleHeader *list;
    init(list);          // Khởi tạo ds
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot nhan vien" << endl;
        cout << "2: Xoa mot nhan vien" << endl;
        cout << "3: Xem tat ca cac nhan vien trong phong" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1':          // Thêm vào ds
                int position;
                Employee employee;
                cout << "Vi tri can chen: ";
                cin >> position;
                cout << "Ten nhan vien: ";
                cin >> employee.name;
                cout << "Tuoi nhan vien: ";
                cin >> employee.age;
                cout << "Luong nhan vien: ";
                cin >> employee.salary;
                insert(list, position, employee);
                break;
            case '2':          // Lấy ra khỏi ds
                int position;
                cout << "Vi tri can xoa: ";
                cin >> position;
                SimpleNode* result = remove(list, position);
                if(result != NULL){
                    cout << "Nhan vien bi loai: " << endl;
                    cout << "Ten: " << result->employee.name
                        << endl;
                    cout << "Tuoi: " << result->employee.age
                        << endl;
                    cout << "Luong: "
                        << result->employee.salary << endl;
                }
            }
        }
    } while(function != 5);
}

```

```
        }
        break;
    case '3':          // Duyệt ds
        cout<<"Cac nhan vien cua phong:"<<endl;
        traverse(list);
        break;
    }while(function != '5');
    release(list);      // Giải phóng ds
    return;
}
```

TỔNG KẾT CHƯƠNG 3

Nội dung chương 3 đã trình bày các vấn đề liên quan đến các kiểu dữ liệu có cấu trúc trong C++:

- Khai báo cấu trúc thông qua từ khoá **struct**
- Tự định nghĩa kiểu dữ liệu cấu trúc bằng từ khoá **typedef**.
- Khai báo một biến có kiểu dữ liệu cấu trúc
- Khai báo các cấu trúc lồng nhau.
- Truy nhập đến các thuộc tính của cấu trúc
- Khai báo con trỏ cấu trúc, cấp phát và giải phóng bộ nhớ động của con trỏ cấu trúc. Truy nhập đến các thuộc tính của con trỏ cấu trúc.
- Khai báo và sử dụng mảng cấu trúc
- Khai báo mảng cấu trúc bằng con trỏ cấu trúc. Cấp phát và giải phóng vùng nhớ của mảng động các cấu trúc.
- Cài đặt một số cấu trúc đặc biệt:
 - Ngăn xếp
 - Hàng đợi
 - Danh sách liên kết

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 3

1. Để định nghĩa một cấu trúc sinh viên có tên là Sinhvien, gồm có tên và tuổi sinh viên. Định nghĩa nào sau đây là đúng:

- a.

```
struct Sinhvien{
    char name[20];
    int age;
};
```
- b.

```
struct {
    char name[20];
    int age;
} Sinhvien;
```

```
c.    typedef struct Sinhvien{  
        char name[20];  
        int age;  
    };
```

2. Một cấu trúc được định nghĩa như sau:

```
struct Employee{  
    char name[20];  
    int age;  
};
```

Khi đó, cách khai báo biến nào sau đây là đúng:

- a. struct Employee myEmployee;
- b. struct employee myEmployee;
- c. Employee myEmployee;
- d. employee myEmployee;

3. Một cấu trúc được định nghĩa như sau:

```
typedef struct employee{  
    char name[20];  
    int age;  
} Employee;
```

Khi đó, cách khai báo biến nào sau đây là đúng:

- a. Employee myEmployee;
- b. employee myEmployee;
- c. struct Employee myEmployee;
- d. struct employee myEmployee;

4. Với cấu trúc được định nghĩa như trong bài 3. Khi đó, cách khởi tạo biến nào sau đây là đúng:

- a. Employee myEmployee = {'A', 27};
- b. Employee myEmployee = {"A", 27};
- c. Employee myEmployee = ('A', 27);
- d. Employee myEmployee = ("A", 27);

5. Với cấu trúc được định nghĩa như trong bài 3. Khi đó, các cách cấp phát bộ nhớ cho biến con trỏ nào sau đây là đúng:

- a. Employee *myEmployee = new Employee;
- b. Employee *myEmployee = new Employee();
- c. Employee *myEmployee = new Employee(10);
- d. Employee *myEmployee = new Employee[10];

6. Định nghĩa một cấu trúc về môn học của một học sinh có tên Subject, bao gồm các thông tin:

- Tên môn học, kiểu char[];
 - Điểm tổng kết môn học, kiểu float;
7. Định nghĩa cấu trúc về học sinh tên là Student bao gồm các thông tin sau:
- Tên học sinh, kiểu char[];
 - Tuổi học sinh, kiểu int;
 - Lớp học sinh, kiểu char[];
 - Danh sách điểm các môn học của học sinh, kiểu là một mảng các cấu trúc Subject đã được định nghĩa trong bài tập 6.
 - Xếp loại học lực, kiểu char[];
8. Khai báo một biến có cấu trúc là Student đã định nghĩa trong bài 7. Sau đó, thực hiện tính điểm trung bình của tất cả các môn học của học sinh đó, và viết một thủ tục xếp loại học sinh dựa vào điểm trung bình các môn học:
- Nếu điểm tb nhỏ hơn 5.0, xếp loại kém
 - Nếu điểm tb từ 5.0 đến dưới 6.5, xếp loại trung bình.
 - Nếu điểm tb từ 6.5 đến dưới 8.0, xếp loại khá
 - Nếu điểm tb từ 8.0 trở lên, xếp loại giỏi.
9. Viết một chương trình quản lý các học sinh của một lớp, là một dãy các cấu trúc có kiểu Stupid định nghĩa trong bài 7. Sử dụng thủ tục đã cài đặt trong bài 8 để thực hiện các thao tác sau:
- Khởi tạo danh sách và điểm của các học sinh trong lớp.
 - Tính điểm trung bình và xếp loại cho tất cả các học sinh.
 - Tìm tất cả các học sinh theo một loại nhất định
10. Sử dụng cấu trúc ngăn xếp đã định nghĩa trong bài để đổi một số từ kiểu thập phân sang kiểu nhị phân: Chi số nguyên cho 2, mãi cho đến khi thương < 2 , lưu các số dư vào ngăn xếp. Sau đó, đọc các giá trị dư từ ngăn xếp ra, ta sẽ thu được chuỗi nhị phân tương ứng.
11. Mở rộng cấu trúc hàng đợi đã định nghĩa trong bài để trở thành hàng đợi có độ ưu tiên:
- Cho mỗi node thêm một thuộc tính là độ ưu tiên của node đó
 - Khi thêm một node vào hàng đợi, thay vì thêm vào cuối hàng đợi như thông thường, ta tìm vị trí có độ ưu tiên phù hợp để chèn node vào, sao cho dãy các node trong hàng đợi là một danh sách có độ ưu tiên của các node là giảm dần.
 - Việc lấy ra là không thay đổi: lấy ra phần tử ở đầu hàng đợi, chính là phần tử có độ ưu tiên cao nhất.
12. Áp dụng hàng đợi có độ ưu tiên trong bài 11 để xây dựng chương trình quản lý tiến trình có độ ưu tiên của hệ điều hành, mở rộng ứng dụng trong bài ngăn xếp.
13. Mở rộng cấu trúc danh sách liên kết đơn trong bài thành danh sách liên kết kép:
- Mỗi node có thêm một con trỏ prev để trỏ đến node trước nó
 - Đối với node header, cũng cần 2 con trỏ: trỏ đến node đầu tiên và node cuối cùng của danh sách

- Riêng với node đầu tiên (front) của danh sách, con trỏ prev của nó sẽ trỏ đến NULL. Giống như con trỏ next của node rear.
14. Cài đặt lại hai thao tác thêm vào một node và xóa một node ở một vị trí xác định trong một cấu trúc danh sách liên kết kép định nghĩa trong bài 13.
 15. Áp dụng các định nghĩa và thao tác trong các bài 13 và 14. Cài đặt lại chương trình quản lý nhân viên ở chương trình 3.6c bằng danh sách liên kết kép.



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhkx@o-pit.edu.vn

CHƯƠNG 4

VÀO RA TRÊN TỆP

Nội dung chương này tập trung trình bày các vấn đề liên quan đến các thao tác trên tệp dữ liệu trong ngôn ngữ C++:

- Khái niệm tệp, tệp văn bản và tệp nhị phân
- Các thao tác vào ra trên tệp
- Phương thức truy nhập tệp trực tiếp

4.1 KHÁI NIỆM TỆP

4.1.1 Tệp dữ liệu

Trong C++, khi thao tác với một tệp dữ liệu, cần thực hiện tuần tự theo các bước như sau:

1. Mở tệp tin
2. Thực hiện các thao tác đọc, ghi trên tệp tin đang mở
3. Đóng tệp tin

Để thực hiện các thao tác liên quan đến tệp dữ liệu, C++ cung cấp một thư viện `<fstream.h>` chứa các lớp và các hàm phục vụ cho các thao tác này. Do vậy, trong các chương trình làm việc với tệp tin, ta cần khai báo chỉ thị dùng thư viện này ngay từ đầu chương trình:

```
#include<fstream.h>
```

Khai báo biến tệp

Trong C++, khi khai báo một biến tệp, đồng thời ta sẽ mở tệp tương ứng theo cú pháp tổng quát bằng cách dùng kiểu **`fstream`** như sau:

```
fstream <Tên biến tệp>(<Tên tệp>, <Chế độ mở tệp>);
```

Trong đó:

- **Tên biến tệp:** có tính chất như một tên biến thông thường, nó sẽ được dùng để thực hiện các thao tác với tệp gắn với nó. Tên biến tệp cũng phải tuân thủ theo quy tắc đặt tên biến trong C++.
- **Tên tệp:** là tên tệp dữ liệu mà ta cần thao tác trên nó.
- **Chế độ mở tệp:** là các hằng kiểu bit đã được định nghĩa sẵn bởi C++. Nó chỉ ra rằng ta đang mở tệp tin ở chế độ nào: đọc hoặc ghi, hoặc cả đọc lẫn ghi.

Ví dụ, khai báo:

```
fstream myFile("abc.txt", ios::in);
```

là khai báo một biến tệp, có tên là `myFile`, dùng để mở tệp tin có tên là `abc.txt` và tệp tin này được mở ở chế độ để đọc dữ liệu (bit chỉ thị `ios::in`).

Lưu ý:

- Tên tệp tin có dạng một chuỗi kí tự, nếu khai báo tên tệp có đường dẫn thư mục “\” thì mỗi dấu “\” phải được viết thành “\\” để tránh bị nhầm lẫn với các kí tự đặc biệt trong C như “\n”, “\d”...

Ví dụ, muốn mở một tệp tên là abc.txt trong thư mục myDir để đọc, ta phải khai báo như sau:

```
fstream myFile("myDir\\abc.txt", ios::in);
```

Các chế độ mở tệp tin

Các chế độ mở tệp tin được định nghĩa bởi các bit chỉ thị:

- **ios::in:** Mở một tệp tin để đọc.
- **ios::out:** Mở một tệp tin có sẵn để ghi.
- **ios::app:** Mở một tệp tin có sẵn để thêm dữ liệu vào cuối tệp.
- **ios::ate:** Mở tệp tin và đặt con trỏ tệp tin vào cuối tệp.
- **ios::trunc:** Nếu tệp tin đã có sẵn thì dữ liệu của nó sẽ bị mất.
- **ios::nocreate:** Mở một tệp tin, tệp tin này bắt buộc phải tồn tại.
- **ios::noreplace:** Chỉ mở tệp tin khi tệp tin chưa tồn tại.
- **ios::binary:** Mở một tệp tin ở chế độ nhị phân.
- **ios::text:** Mở một tệp tin ở chế độ văn bản.

Lưu ý:

- Khi muốn mở một tệp tin đồng thời ở nhiều chế độ khác nhau, ta kết hợp các bit chỉ thị tương ứng bằng phép toán hợp bit “|”.

Ví dụ, muốn mở một tệp tin abc.txt để đọc (ios::in) đồng thời với để ghi (ios::out) dưới chế độ văn bản (ios::text), ta khai báo như sau:

```
fstream myFile("abc.txt", ios::in|ios::out|ios::text);
```

4.1.2 Tệp văn bản

Để mở một tệp tin dưới chế độ văn bản, ta dùng cú pháp sau:

```
fstream <Tên biến tệp>(<Tên tệp>, ios::text);
```

Khi đó, các thao tác đọc, ghi trên biến tệp được thực hiện theo đơn vị là các từ, được phân cách bởi dấu trống (space bar) hoặc dấu xuống dòng (enter).

Ví dụ, muốn mở tệp tin baitho.txt dưới chế độ văn bản, ta khai báo như sau:

```
fstream myBaiTho("baitho.txt", ios::text);
```

4.1.3 Tệp nhị phân

Để mở một tệp tin dưới chế độ nhị phân, ta dùng cú pháp sau:

```
fstream <Tên biến tệp>(<Tên tệp>, ios::binary);
```

Khi đó, các thao tác đọc, ghi trên biến tệp được thực hiện theo đơn vị byte theo kích thước các bản ghi (cấu trúc) được ghi trong tệp.

Ví dụ, muốn mở tệp tin baitho.txt dưới chế độ nhị phân, ta khai báo như sau:

```
fstream myBaiTho("baitho.txt", ios::binary);
```


4.2 VÀO RA TRÊN TỆP

4.2.1 Vào ra tệp văn bản bằng “>>” và “<<”

Ghi tệp văn bản bằng “<<”

Các bước thực hiện để ghi dữ liệu vào một tệp tin như sau:

1. Mở tệp tin theo chế độ để ghi bằng đối tượng ofstream (mở tệp tin chỉ để ghi):

```
ofstream <Tên biến tệp>(<Tên tệp tin>, ios::out);
```

2. Ghi dữ liệu vào tệp bằng thao tác “<<”:

```
<Tên biến tệp> << <Dữ liệu>;
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Chương trình 4.1 minh họa việc ghi dữ liệu vào tệp tin:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ ghi vào tệp các kí tự do người dùng gõ vào từ bàn phím, mỗi kí tự được phân cách nhau bởi dấu trống (space bar).
- Chương trình dừng lại khi người dùng nhập kí tự ‘e’. Và tệp tin được kết thúc bằng một dấu xuống dòng “endl”.

Chương trình 4.1

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

const int length = 25;           // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], input;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName;    // Nhập tên tệp tin

    /* Mở tệp tin */
    ofstream fileOut(fileName, ios::out); // Khai báo và mở tệp tin
    if(!fileOut){                       // Không mở được tệp
        cout << "Khong the tao duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Ghi dữ liệu vào tệp tin */
```

```
do{
    cin >> input;                // Đọc kí tự từ bàn phím
    fileOut << input << ' ';    // Ghi kí tự vào tệp tin
}while((input != 'e') && (fileOut));
fileOut << endl;                // Xuống dòng cuối tệp tin

/* Đóng tệp tin */
fileOut.close();                // Đóng tệp tin
return;
}
```

Đọc dữ liệu từ tệp văn bản bằng ">>"

Các bước thực hiện để đọc dữ liệu từ một tệp tin như sau:

1. Mở tệp tin theo chế độ để đọc bằng đối tượng ifstream (mở tệp tin chỉ để đọc):

```
ifstream <Tên biến tệp>(<Tên tệp tin>, ios::in);
```

2. Đọc dữ liệu từ tệp bằng thao tác ">>":

```
<Tên biến tệp> >> <Biến dữ liệu>;
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Chương trình 4.2 minh họa việc đọc dữ liệu từ tệp tin vừa sử dụng trong chương trình 4.1 ra màn hình:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ đọc các kí tự trong tệp và hiển thị ra màn hình, mỗi kí tự được phân cách nhau bởi dấu trống (space bar).
- Chương trình dừng lại khi kết thúc tệp tin.

Chương trình 4.2

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

const int length = 25;                // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], output;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName;    // Nhập tên tệp tin

    /* Mở tệp tin */
```

```
ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin
if(!fileIn){                        // Không mở được tệp
    cout << "Khong the mo duoc tep tin " << fileName << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra màn hình */
while(fileIn){
    fileIn >> output;                // Đọc kí tự từ tệp tin
    cout << output;                  // Ghi kí tự ra màn hình
}
cout << endl;                        // Xuống dòng trên màn hình

/* Đóng tệp tin */
fileIn.close();                      // Đóng tệp tin
return;
}
```

Chương trình 4.3 minh họa việc copy toàn bộ nội dung của một tệp tin sang một tệp tin mới:

- Tên tệp tin nguồn và tệp tin đích được nhập từ bàn phím bởi người dùng.
- Tệp tin nguồn được mở ở chế độ đọc.
- Tệp tin đích được mở ở chế độ ghi.
- Đọc từng kí tự từ tệp tin nguồn và ghi ngay vào tệp tin đích.
- Đóng các tệp tin khi kết thúc.

Chương trình 4.3

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

const int length = 25;                // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char sourceFile[length], targetFile[length], data;
    cout << "Ten tep tin nguon: ";
    cin >> setw(length) >> sourceFile; // Nhập tên tệp tin nguồn

    cout << "Ten tep tin dich: ";
    cin >> setw(length) >> targetFile; // Nhập tên tệp tin đích
```

```

/* Mở tệp tin nguồn */
ifstream fileIn(sourceFile, ios::in); // Khai báo và mở tệp nguồn
if(!fileIn){                          // Không mở được tệp nguồn
    cout << "Khong the mo duoc tep tin nguon "
         << sourceFile << endl;
    exit(1);
}

/* Mở tệp tin đích */
ofstream fileOut(targetFile, ios::out); // Khai báo và mở tệp đích
if(!fileOut){                          // Không mở được tệp đích
    cout << "Khong the tao duoc tep tin dich "
         << targetFile << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra tệp đích */
while(fileIn){
    fileIn >> data;                    // Đọc kí tự từ tệp nguồn
    fileOut << data;                   // Ghi kí tự ra tệp đích
}

/* Đóng các tệp tin */
fileIn.close();                       // Đóng tệp tin nguồn
fileOut.close();                      // Đóng tệp tin đích
return;
}

```

Lưu ý:

- Tên biến tệp, sau khi dùng xong với một tệp xác định, có thể sử dụng để mở một tệp khác, với một chế độ mở tệp khác bằng phép toán **open()** của biến tệp.

`<Tên biến tệp>.open(<Tên tệp mới>, <chế độ mở mới>);`

Ví dụ, đoạn chương trình:

```

ofstream myFile("abc.txt", ios::out);
...    // ghi vào file abc.txt
myFile.close();
myFile.open("xyz.txt", ios::out|ios::app);
...    // Thêm vào cuối file xyz.txt
myFile.close();

```

sẽ dùng biến tệp myFile (có kiểu ofstream) hai lần: một lần là dùng với tệp tin abc.txt ở chế độ mở để ghi từ đầu. Một lần khác là với tệp tin xyz.txt ở chế độ mở để ghi thêm vào cuối.

4.2.2 Vào ra tệp nhị phân bằng read và write

Ghi vào tệp nhị phân bằng write

Các bước thực hiện để ghi dữ liệu vào một tệp nhị phân như sau:

1. Mở tệp tin theo chế độ để ghi nhị phân bằng đối tượng fstream:

```
fstream <Tên biến tệp>(<Tên tệp tin>, ios::out|ios::binary);
```

2. Ghi dữ liệu vào tệp bằng thao tác “**write()**”:

```
<Tên biến tệp>.write(char* <Dữ liệu>,  
int <Kích thước dữ liệu>);
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Trong đó, thao tác **write** nhận hai tham số đầu vào như sau:

- Tham số thứ nhất là con trỏ kiểu char trỏ đến vùng dữ liệu cần ghi vào tệp. Vì con trỏ bắt buộc có kiểu char nên khi muốn ghi dữ liệu có kiểu khác vào tệp, ta dùng hàm chuyển kiểu:

```
reinterpret_cast<char*>(<Dữ liệu>);
```

- Tham số thứ hai là kích cỡ dữ liệu được ghi vào tệp. Kích cỡ này được tính theo byte, nên thông thường ta dùng toán tử:

```
sizeof(<Kiểu dữ liệu>);
```

Lưu ý:

- Khi muốn đọc, ghi các dữ liệu có cấu trúc (struct) vào tệp thì ta phải dùng ở chế độ đọc/ghi tệp nhị phân mà không thể dùng chế độ đọc/ghi ở chế độ văn bản.
- Khi đọc/ghi dữ liệu có kiểu cấu trúc, để toán tử sizeof() thực hiện chính xác thì các thành viên của cấu trúc không được là kiểu con trỏ. Vì toán tử sizeof() đối với con trỏ chỉ cho kích cỡ của con trỏ mà không cho kích cỡ thật của vùng dữ liệu mà con trỏ trỏ tới.

Chương trình 4.4 minh họa việc ghi dữ liệu vào tệp tin nhị phân, dữ liệu là kiểu cấu trúc:

- Tên tệp tin và số lượng bản ghi được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ ghi vào tệp các bản ghi có cấu trúc do người dùng gõ vào từ bàn phím.

Chương trình 4.4

```
#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<type.h>

const int length = 25; // Độ dài tối đa tên tệp tin

typedef struct {
    int day; // Ngày
    int month; // Tháng
```

```

        int year;                                // Năm
    } Date;

typedef struct {
    char name[20];                                // Tên nhân viên
    Date birthDay;                                // Ngày sinh của nhân viên
    char role[20];                                // Chức vụ của nhân viên
    float salary;                                // Lương của nhân viên
} Employee;

void main(){
    clrscr();
    char fileName[length];                        // Tên tệp tin
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName;            // Nhập tên tệp tin

    int recordNumber;                             // Số lượng bản ghi
    cout << "So luong ban ghi: ";
    cin >> recordNumber;                         // Nhập số lượng bản ghi

    /* Mở tệp tin */
    // Khai báo và mở tệp tin
    fstream fileOut(fileName, ios::out|ios::binary);
    if(!fileOut){                                // Không mở được tệp
        cout << "Khong the tao duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Ghi dữ liệu vào tệp tin */
    Employee myEmployee;
    for(int i=0; i<recordNumber; i++){
        cout << "Ban ghi thu " << i+1 << endl;
        cout << "Name: ";
        cin >> myEmployee.name;                  // Nhập tên nhân viên
        cout << "Day of birth: ";
        cin >> myEmployee.birthDay.day;          // Nhập ngày sinh
        cout << "Month of birth: ";
        cin >> myEmployee.birthDay.month;        // Nhập tháng sinh
        cout << "Year of birth: ";
        cin >> myEmployee.birthDay.year;        // Nhập năm sinh
        cout << "Role: ";
        cin >> myEmployee.role;                  // Nhập chức vụ
        cout << "Salary: ";
    }
}

```

```
        cin >> myEmployee.salary;        // Nhập tiền lương

        // Ghi dữ liệu vào tệp
        fileOut.write(reinterpret_cast<char *>(&myEmployee),
                        sizeof(Employee));

    }

    /* Đóng tệp tin */
    fileOut.close();                        // Đóng tệp tin
    return;
}
```

Đọc dữ liệu từ tệp nhị phân bằng read

Các bước thực hiện để đọc dữ liệu từ một tệp tin nhị phân như sau:

1. Mở tệp tin theo chế độ để đọc nhị phân bằng đối tượng fstream (mở tệp tin chỉ để ghi):

```
fstream <Tên biến tệp>(<Tên tệp tin>, ios::in|ios::binary);
```

2. Đọc dữ liệu từ tệp bằng thao tác “read()”:

```
<Tên biến tệp>.read(char* <Dữ liệu ra>,
                    int <Kích thước dữ liệu>);
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Chương trình 4.5 minh họa việc đọc dữ liệu từ tệp tin vào biến có cấu trúc:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ đọc các cấu trúc nhân viên trong tệp và hiển thị ra màn hình.
- Chương trình dừng lại khi kết thúc tệp tin.

Chương trình 4.5

```
#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<type.h>

const int length = 25;                // Độ dài tối đa tên tệp tin

typedef struct {
    int day;                          // Ngày
    int month;                        // Tháng
    int year;                         // Năm
} Date;
```



```
typedef struct {
    char name[20];                // Tên nhân viên
    Date birthDay;                // Ngày sinh của nhân viên
    char role[20];                // Chức vụ của nhân viên
    float salary;                 // Lương của nhân viên
} Employee;

void main() {
    clrscr();
    char fileName[length];       // Tên tệp tin
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin

    /* Mở tệp tin */
    // Khai báo và mở tệp tin
    fstream fileIn(fileName, ios::in|ios::binary);
    if(!fileIn) {                // Không mở được tệp
        cout << "Khong the mo duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Đọc dữ liệu từ tệp tin ra màn hình */
    Employee myEmployee;
    while(fileIn) {
        fileIn.read(reinterpret_cast<char *>(&myEmployee),
                    sizeof(Employee)); // Đọc kí tự từ tệp tin
        cout << myEmployee.name << " "
             << myEmployee.birthDay.day << "/"
             << myEmployee.birthDay.month << "/"
             << myEmployee.birthDay.year << " "
             << myEmployee.role << " "
             << myEmployee.salary << endl; // Ghi kí tự ra màn hình
    }

    /* Đóng tệp tin */
    fileIn.close();              // Đóng tệp tin
    return;
}
```


4.3 TRUY NHẬP TỆP TRỰC TIẾP

4.3.1 Con trỏ tệp tin

Con trỏ tệp tin có vai trò như một đầu đọc trỏ vào một vị trí xác định của tệp và thao tác truy nhập tệp diễn ra tuần tự:

- Tại mỗi thời điểm, con trỏ tệp tin xác định một vị trí trên tệp mà tại đó, thao tác truy nhập tệp (đọc/ghi) được thực hiện.
- Sau thao tác truy nhập, con trỏ tệp tự động chuyển đến vị trí tiếp theo dựa vào kích thước đơn vị dữ liệu được truy nhập.

Cách truy nhập tệp tuần tự có nhược điểm là bao giờ cũng phải bắt đầu từ đầu tệp tin, đi tuần tự cho đến vị trí cần truy nhập. Khi tệp tin có kích thước lớn thì cách truy nhập này rất tốn thời gian.

Để tránh nhược điểm này, C++ cho phép truy nhập trực tiếp đến một vị trí xác định trên tệp tin bằng các phép toán:

- Truy nhập vị trí hiện tại của con trỏ tệp tin
- Dịch chuyển con trỏ tệp tin đến một vị trí xác định

4.3.2 Truy nhập vị trí hiện tại của con trỏ tệp

Cú pháp truy nhập đến vị trí hiện thời của con trỏ tệp phụ thuộc vào kiểu biến tệp đang dùng là để đọc hay ghi.

- Nếu biến tệp là kiểu mở tệp để đọc **ifstream** thì cú pháp là:

```
<Tên biến tệp>.tellg();
```

- Nếu biến tệp là kiểu mở tệp để ghi **ofstream** thì cú pháp là:

```
<Tên biến tệp>.tellp();
```

Chương trình 4.6a minh họa việc xác định vị trí hiện thời của con trỏ tệp sau một số thao tác đọc tệp trước đó.

Chương trình 4.6a

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

const int length = 25; // Độ dài tối đa tên tệp tin

void main() {
    clrscr();
    char fileName[length], output;
    cout << "Tên tệp tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin
```

```
/* Mở tệp tin */
ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin
if(!fileIn){                        // Không mở được tệp
    cout << "Khong the mo duoc tep tin " << fileName << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra màn hình
 * Ghi vị trí con trỏ tệp ra màn hình cứ sau 5 lần đọc kí tự */
int index = 0;
while(fileIn){
    fileIn >> output;                // Đọc kí tự từ tệp tin
    cout << output;                  // Ghi kí tự ra màn hình
    if(index % 5 == 0)               // Ghi ra vị trí con trỏ tệp
        cout<< endl << "Vi tri con tro tep: "
            << fileIn.tellg() << endl;
    index ++;
}
cout << endl;                        // Xuống dòng trên màn hình

/* Đóng tệp tin */
fileIn.close();                      // Đóng tệp tin
return;
}
```

Chương trình 4.6b minh họa việc xác định vị trí hiện thời của con trỏ tệp sau một số thao tác ghi vào tệp trước đó.

Chương trình 4.6b

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

const int length = 25;                // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], input;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin
```

```

/* Mở tệp tin */
ofstream fileOut(fileName, ios::out); // Khai báo và mở tệp tin
if(!fileOut){                          // Không mở được tệp
    cout << "Khong the tao duoc tep tin " << fileName << endl;
    exit(1);
}

/* Ghi dữ liệu vào tệp tin
* Hiện ra màn hình vị trí con trỏ tệp sau khi ghi được 5 kí tự*/
int index = 0;
do{
    cin >> input;                      // Đọc kí tự từ bàn phím
    fileOut << input << ' ';           // Ghi kí tự vào tệp tin
    if(index%5 == 0)                   // Hiện thị vị trí con trỏ tệp
        cout << "Vi tri con tro tep: "
              << fileOut.tellp() << endl;
    index++;
}while((input != 'e') && (fileOut));
fileOut << endl;                       // Xuống dòng cuối tệp tin

/* Đóng tệp tin */
fileOut.close();                       // Đóng tệp tin
return;
}

```

4.3.3 Dịch chuyển con trỏ tệp

Ngoài việc xác định vị trí hiện thời của con trỏ tệp, C++ còn cho phép dịch chuyển con trỏ tệp đến một vị trí bất kì trên tệp. Cú pháp dịch chuyển phụ thuộc vào kiểu biến tệp là đọc hay ghi.

- Nếu biến tệp có kiểu là mở tệp tin để đọc **ifstream**, cú pháp sẽ là:
`<Tên biến tệp>.seekg(<Kích thước>, <Mốc dịch chuyển>);`
- Nếu biến tệp có kiểu là mở tệp để ghi **ofstream**, cú pháp sẽ là:
`<Tên biến tệp>.seekp(<Kích thước>, <Mốc dịch chuyển>);`

Trong đó:

- **Kích thước:** là tham số mô tả khoảng cách dịch chuyển so với vị trí mốc dịch chuyển. Đơn vị tính của kích thước là byte, có kiểu là số nguyên.
- **Mốc dịch chuyển:** là vị trí gốc để xác định khoảng cách dịch chuyển của con trỏ tệp. Có ba tham số hằng về kiểu mốc dịch chuyển:
 - **ios::beg:** Mốc dịch chuyển là đầu tệp tin.
 - **ios::cur:** Mốc dịch chuyển là vị trí hiện thời của con trỏ tệp.
 - **ios::end:** Mốc dịch chuyển là vị trí cuối cùng của tệp tin.

Ví dụ:

```
ifstream fileIn("abc.txt", ios::in);
fileIn.seekg(sizeof(char)*7, ios::beg);
```

sẽ dịch chuyển con trỏ tệp tin đến kí tự (kiểu char) thứ $7+1 = 8$ trong tệp tin abc.txt để đọc (giả sử tệp tin abc.txt lưu các kí tự kiểu char).

Lưu ý:

- Vì khoảng cách cách dịch chuyển có kiểu số nguyên (int) cho nên có thể nhận giá trị âm hoặc dương. Nếu giá trị dương, dịch chuyển về phía sau vị trí làm mốc, nếu giá trị âm, dịch chuyển về phía trước vị trí làm mốc.
- Nếu vị trí dịch chuyển đến nằm ngoài phạm vi tệp tin (phía sau vị trí cuối cùng của tệp hoặc phía trước vị trí đầu tiên của tệp) sẽ nảy sinh lỗi, khi đó <Tên biến tệp> = false.

Chương trình 4.7 cài đặt chương trình truy nhập tệp tin trực tiếp để đọc giá trị kí tự (kiểu char) trong tệp:

- Tên tệp tin (chứa dữ liệu kiểu char) do người dùng nhập vào từ bàn phím.
- Sau đó, mỗi khi người dùng nhập vào một số nguyên, chương trình sẽ dịch chuyển đến vị trí mới, cách vị trí cũ đúng bằng từng ấy kí tự, tính từ vị trí hiện thời của con trỏ tệp.
- Chương trình sẽ kết thúc khi người dùng nhập vào số 0.

Chương trình 4.7

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

const int length = 25;           // Độ dài tối đa tên tệp tin

void main() {
    clrscr();
    char fileName[length], output;
    cout << "Tên tệp tin: ";
    cin >> setw(length) >> fileName;    // Nhập tên tệp tin

    /* Mở tệp tin */
    ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin
    if(!fileIn){                       // Không mở được tệp
        cout << "Khong the mo duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Đọc dữ liệu từ tệp tin ra màn hình
    * Ghi vị trí con trỏ tệp ra màn hình cứ sau 5 lần đọc kí tự */
```

```
int index = 1;
do{
    cout << "Số kí tự dịch chuyển: ";
    cin >> index;

    // Dịch chuyển con trỏ tệp từ vị trí hiện thời
    fileIn.seekg(sizeof(char)*index, ios::cur);
    if(fileIn){ // Đúng
        fileIn >> output; // Đọc kí tự từ tệp tin

        // Ghi kí tự ra màn hình
        cout << "Vị trí: " << fileIn.tellg() << output;
    }else{ // Ra khỏi phạm vi tệp
        fileIn.clear(); // Về vị trí đầu tệp
    }
}while(index);

/* Đóng tệp tin */
fileIn.close(); // Đóng tệp tin
return;
}
```

TỔNG KẾT CHƯƠNG 4

Nội dung chương 4 đã tập trung trình bày các vấn đề liên quan đến các thao tác trên tệp tin trong ngôn ngữ C++. Bao gồm:

- Các bước tuần tự khi thao tác với một tệp tin:
 - Mở tệp tin
 - Đọc/ghi dữ liệu trên tệp tin
 - Đóng tệp tin
- Thao tác mở tệp tin với nhiều chế độ bằng kiểu fstream.
- Thao tác mở tệp tin chỉ để đọc với kiểu ifstream
- Thao tác mở tệp tin chỉ để ghi với thao tác ofstream.
- Đọc dữ liệu từ tệp tin văn bản với thao tác ">>".
- Ghi dữ liệu vào tệp tin văn bản bằng thao tác "<<".
- Đọc tệp tin nhị phân bằng thao tác read().
- Ghi vào tệp tin nhị phân bằng thao tác write().
- Xác định vị trí hiện thời của con trỏ tệp tin với các thao tác tellg() và tellp().
- Dịch chuyển vị trí của con trỏ tệp tin với các thao tác seekg() và seekp().
- Thiết lập lại trạng thái cho con trỏ tệp tin bằng thao tác clear().

- Đóng tệp tin đã sử dụng bằng thao tác `close()`.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 4

1. Muốn mở một tệp tin tên là `abc.txt` để đọc dữ liệu, lệnh mở tệp nào sau đây là đúng:

- a. `fstream myFile("abc.txt", ios::in);`
- b. `fstream myFile("abc.txt", ios::out);`
- c. `fstream myFile("abc.txt", ios::app);`
- d. `fstream myFile("abc.txt", ios::ate);`

2. Muốn mở một tệp tin `abc.txt` nằm trong thư mục `xyz` để ghi dữ liệu vào. Lệnh mở nào sau đây là đúng:

- a. `fstream myFile("xyz\\abc.txt", ios::out);`
- b. `fstream myFile("xyz\\abc.txt", ios::out);`
- c. `fstream myFile("xyz/abc.txt", ios::out);`
- d. `fstream myFile("xyz//abc.txt", ios::out);`

3. Muốn mở một tệp tin `abc.txt` để ghi thêm dữ liệu vào cuối tệp, lệnh nào sau đây là đúng:

- a. `fstream myFile("abc.txt", ios::out);`
- b. `fstream myFile("abc.txt", ios::app);`
- c. `fstream myFile("abc.txt", ios::out|ios::app);`
- d. `fstream myFile("abc.txt", ios::out||ios::app);`

4. Xét hai lệnh khai báo sau:

```
fstream myFile1("abc.txt", ios::out);  
ofstream myFile2("abc.txt", ios::out);
```

Nhận định nào sau đây là đúng:

- a. `myFile1` và `myFile2` có chức năng giống nhau.
- b. `myFile1` và `myFile2` có chức năng khác nhau

5. Xét hai lệnh khai báo sau:

```
fstream myFile1("abc.txt", ios::in);  
ifstream myFile2("abc.txt", ios::in);
```

Nhận định nào sau đây là đúng:

- a. `myFile1` và `myFile2` có chức năng giống nhau.
- b. `myFile1` và `myFile2` có chức năng khác nhau

6. Xét đoạn chương trình sau:

```
ofstream myFile("abc.txt", ios::out);  
if(myFile) myFile << "abc.txt";
```

Chương trình sẽ làm gì?

- a. Ghi ra màn hình dòng chữ "abc.txt"
- b. Ghi vào tệp tin `abc.txt` dòng chữ "abc.txt"
- c. Đọc từ tệp tin `abc.txt` dòng chữ "abc.txt"

d. Chương trình sẽ báo lỗi.

7. Xét đoạn chương trình sau:

```
ifstream myFile("abc.txt", ios::in);
char text[20];
if(myFile) myFile >> text;
```

Chương trình sẽ làm gì, nếu tệp tin abc.txt có nội dung là dòng chữ "abc.txt"?

- a. Ghi ra màn hình dòng chữ "abc.txt"
- b. Ghi vào tệp tin abc.txt dòng chữ "abc.txt"
- c. Đọc từ tệp tin abc.txt dòng chữ "abc.txt"
- d. Chương trình sẽ báo lỗi.

8. Xét đoạn chương trình sau:

```
fstream myFile("abc.txt", ios::out);
if(myFile) myFile << "abc.txt";
myFile.close();
myFile.open("abc.txt", ios::in);
char text[20];
if(myFile) myFile >> text;
cout << text;
```

Chương trình sẽ làm gì, nếu tệp tin abc.txt có nội dung là dòng chữ "abc.txt"?

- a. Ghi vào tệp tin abc.txt dòng chữ "abc.txt"
- b. Đọc từ tệp tin abc.txt dòng chữ "abc.txt"
- c. Ghi ra màn hình dòng chữ "abc.txt"
- d. Cả ba đáp án trên.
- e. Chương trình sẽ báo lỗi.

9. Xét đoạn chương trình sau:

```
ifstream myFile("abc.txt", ios::in);
if(myFile) cout << myFile.tellg();
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0
- b. 1
- c. 8
- d. 16

10. Xét đoạn chương trình sau, nếu tệp abc.txt chứa một số lượng kí tự đủ lớn:

```
ifstream myFile("abc.txt", ios::in);
if(myFile) {
    char c;
    myFile >> c;
    cout << myFile.tellg();
}
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0

- b. 1
- c. 8
- d. 16

11. Xét đoạn chương trình sau, nếu tệp abc.txt chứa một số lượng kí tự đủ lớn:

```
ifstream myFile("abc.txt", ios::in);  
if(myFile) {  
    myFile.seekg(sizeof(char)*5, ios::beg);  
    myFile.seekg(sizeof(char)*5, ios::cur);  
    cout << myFile.tellg();  
}
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0
 - b. 5
 - c. 10
 - d. 80
12. Viết một chương trình gộp nội dung của hai tệp tin có sẵn vào một tệp tin thứ ba. Tên các tệp tin được nhập vào từ bàn phím.
13. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có tên là X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
14. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có năm sinh là X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
15. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có lương cao hơn hoặc bằng một giá trị X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
16. Viết một chương trình sao chép một đoạn đầu nội dung của một tệp tin vào một tệp tin thứ hai. Tên các tệp tin và độ dài đoạn nội dung cần sao chép được nhập từ bàn phím.
17. Viết một chương trình sao chép một đoạn cuối nội dung của một tệp tin vào một tệp tin thứ hai. Tên các tệp tin và độ dài đoạn nội dung cần sao chép được nhập từ bàn phím.

CHƯƠNG 5

LỚP

Nội dung chương này tập trung trình bày các vấn đề liên quan đến lớp đối tượng trong C++:

- Khái niệm, khai báo và sử dụng lớp
- Khai báo và sử dụng các thành phần của lớp: các thuộc tính và các phương thức của lớp
- Phạm vi truy nhập lớp
- Khai báo và sử dụng các phương thức khởi tạo và hủy bỏ của lớp
- Sử dụng lớp thông qua con trỏ đối tượng, mảng các đối tượng.

5.1 KHÁI NIỆM LỚP ĐỐI TƯỢNG

C++ coi lớp là sự trừu tượng hóa các đối tượng, là một khuôn mẫu để biểu diễn các đối tượng thông qua các thuộc tính và các hành động đặc trưng của đối tượng.

5.1.1 Định nghĩa lớp đối tượng

Để định nghĩa một lớp trong C++, ta dùng từ khóa **class** với cú pháp:

```
class <Tên lớp>{  
    };  
};
```

Trong đó:

- **class**: là tên từ khóa bắt buộc để định nghĩa một lớp đối tượng trong C++.
- **Tên lớp**: do người dùng tự định nghĩa. Tên lớp có tính chất như tên kiểu dữ liệu để sử dụng sau này. Cách đặt tên lớp phải tuân thủ theo quy tắc đặt tên biến trong C++.

Ví dụ:

```
class Car{  
    };  
};
```

là định nghĩa một lớp xe ô tô (Car). Lớp này chưa có bất kì một thành phần nào, việc định nghĩa các thành phần cho lớp sẽ được trình bày trong mục 5.2.

Lưu ý:

- Từ khóa **class** là bắt buộc để định nghĩa một lớp đối tượng trong C++. Hơn nữa, C++ có phân biệt chữ hoa chữ thường trong khai báo cho nên chữ **class** phải được viết bằng chữ thường.

Ví dụ:

```
class Car{ // Định nghĩa đúng  
    };  
};
```

nhưng:

```
Class Car{ // Lỗi từ khóa  
    };  
};
```

- Bắt buộc phải có dấu chấm phẩy “;” ở cuối định nghĩa lớp vì C++ coi định nghĩa một lớp như định nghĩa một kiểu dữ liệu, cho nên phải có dấu chấm phẩy cuối định nghĩa (tương tự định nghĩa kiểu dữ liệu kiểu cấu trúc).
- Để phân biệt với tên biến thông thường, ta nên (nhưng không bắt buộc) đặt *tên lớp bắt đầu bằng một chữ in hoa và các tên biến bắt đầu bằng một chữ in thường*.

5.1.2 Sử dụng lớp đối tượng

Lớp đối tượng được sử dụng khi ta khai báo các thể hiện của lớp đó. Một thể hiện của một lớp chính là một đối tượng cụ thể của lớp đó. Việc khai báo một thể hiện của một lớp được thực hiện như cú pháp khai báo một biến có kiểu lớp:

```
<Tên lớp> <Tên biến lớp>;
```

Trong đó:

- **Tên lớp:** là tên lớp đối tượng đã được định nghĩa trước khi khai báo biến.
- **Tên biến lớp:** là tên đối tượng cụ thể. Tên biến lớp sẽ được sử dụng như các biến thông thường trong C++, ngoại trừ việc nó có kiểu lớp đối tượng.

Ví dụ, muốn khai báo một thể hiện (biến) của lớp Car đã được định nghĩa trong mục 5.1.1, ta khai báo như sau:

```
Car myCar;
```

Sau đó, ta có thể sử dụng biến myCar trong chương trình như các biến thông thường: truyền tham số cho hàm, gán cho biến khác ...

Lưu ý:

- Khi khai báo biến lớp, ta không dùng lại từ khóa **class** nữa. Từ khóa **class** chỉ được sử dụng khi định nghĩa lớp mà không dùng khi khai báo biến lớp.

Ví dụ, khai báo:

```
Car myCar; // đúng
```

là đúng, nhưng khai báo:

```
class Car myCar; // Lỗi cú pháp
```

là sai cú pháp.

5.2 CÁC THÀNH PHẦN CỦA LỚP

Việc khai báo các thành phần của lớp có dạng như sau:

```
class <Tên lớp>{  
private:  
    <Khai báo các thành phần riêng>  
protected:  
    <Khai báo các thành phần được bảo vệ>  
public:  
    <Khai báo các thành phần công cộng>  
};
```

Trong đó:

- **private:** là từ khóa chỉ tính chất của C++ để chỉ ra rằng các thành phần được khai báo trong phạm vi từ khóa này là riêng tư đối với lớp đối tượng. Các đối tượng của các lớp khác không truy nhập được các thành phần này.
- **protected:** các thành phần được khai báo trong phạm vi từ khóa này đều được bảo vệ. Qui định loại đối tượng nào được truy nhập đến các thành phần được bảo vệ sẽ được mô tả chi tiết trong mục 5.3.
- **public:** các thành phần công cộng. Các đối tượng của các lớp khác đều có thể truy nhập đến các thành phần công cộng của một đối tượng bất kì.

Các thành phần của lớp được chia làm hai loại:

- Các thành phần chỉ dữ liệu của lớp, được gọi là *thuộc tính* của lớp
- Các thành phần chỉ hành động của lớp, được gọi là *phương thức* của lớp.

5.2.1 Thuộc tính của lớp

Khai báo thuộc tính

Thuộc tính của lớp là thành phần chứa dữ liệu, đặc trưng cho các tính chất của lớp. Thuộc tính của lớp được khai báo theo cú pháp sau:

<Kiểu dữ liệu> <Tên thuộc tính>;

Trong đó:

- **Kiểu dữ liệu:** có thể là các kiểu dữ liệu cơ bản của C++, cũng có thể là các kiểu dữ liệu phức tạp do người dùng tự định nghĩa như struct, hoặc kiểu là một lớp đã được định nghĩa trước đó.
- **Tên thuộc tính:** là tên thuộc tính của lớp, có tính chất như một biến thông thường. Tên thuộc tính phải tuân theo quy tắc đặt tên biến của C++.

Ví dụ, khai báo:

```
class Car{
private:
    int speed;
public:
    char mark[20];
};
```

là khai báo một lớp xe ô tô (Car), có hai thuộc tính: thuộc tính tốc độ (speed) có tính chất private, thuộc tính nhãn hiệu xe (mark) có tính chất public.

Lưu ý:

- Không được khởi tạo giá trị ban đầu cho các thuộc tính ngay trong lớp. Vì các thuộc tính chỉ có giá trị khi nó gắn với một đối tượng cụ thể, là một thể hiện (biến) của lớp.

Ví dụ:

```
class Car{
private:
    int speed;           // đúng
    int weight = 500;    // lỗi
```

};

- Khả năng truy nhập thuộc tính của lớp là phụ thuộc vào thuộc tính ấy được khai báo trong phạm vi của từ khóa nào: private, protected hay public.
- Thông thường, do yêu cầu đóng gói dữ liệu của hướng đối tượng, ta nên khai báo các thuộc tính có tính chất riêng tư (private). Nếu muốn các đối tượng khác truy nhập được vào các thuộc tính này, ta xây dựng các hàm public truy nhập (get / set) đến thuộc tính đó.

Sử dụng thuộc tính

Thuộc tính có thể được sử dụng cho các chương trình nằm ngoài lớp thông qua tên biến lớp hoặc sử dụng ngay trong lớp bởi các phương thức của lớp.

- Nếu thuộc tính được dùng bên ngoài phạm vi lớp, cú pháp phải thông qua tên biến lớp (cách này chỉ sử dụng được với các biến có tính chất public):

<Tên biến lớp>.<tên thuộc tính>;

- Nếu thuộc tính được dùng bên trong lớp, cú pháp đơn giản hơn:

<Tên thuộc tính>;

Ví dụ, với định nghĩa lớp:

```
class Car{
private:
    int speed;
public:
    char mark[20];
};
```

ta khai báo một biến lớp:

```
Car myCar;
```

Thì có thể sử dụng thuộc tính nhãn hiệu xe khi in ra màn hình như sau:

```
cout << myCar.mark;
```

Lưu ý:

- Khi dùng thuộc tính bên trong các phương thức của lớp, mà tên thuộc tính lại bị trùng với tên biến toàn cục (tự do) của chương trình, ta phải chỉ rõ việc dùng tên thuộc tính của lớp (mà không phải tên biến toàn cục) bằng cách dùng chỉ thị phạm vi lớp "::" với cú pháp:

<Tên lớp>::<Tên thuộc tính>;

5.2.2 Phương thức của lớp

Khai báo khuôn mẫu phương thức

Một phương thức là một thao tác thực hiện một số hành động đặc trưng của lớp đối tượng. Phương thức được khai báo tương tự như các hàm trong C++:

<Kiểu trả về> <Tên phương thức>([<Các tham số>]);

Trong đó:

- **Kiểu trả về:** là kiểu dữ liệu trả về của phương thức. Kiểu có thể là các kiểu dữ liệu cơ bản của C++, cũng có thể là kiểu do người dùng định nghĩa, hoặc kiểu lớp đã được định nghĩa.

- **Tên phương thức:** do người dùng tự đặt tên, tuân theo quy tắc đặt tên biến của C++.
- **Các tham số:** Các tham số đầu vào của phương thức, được biểu diễn bằng kiểu dữ liệu tương ứng. Các tham số được phân cách bởi dấu phẩy “,”. Các tham số là tùy chọn (Phần trong dấu ngoặc vuông “[]” là tùy chọn).

Ví dụ, khai báo:

```
class Car{
private:
    int speed;
    char mark[20];
public:
    void show();
};
```

là định nghĩa một lớp Car có hai thuộc tính cục bộ là speed và mark, và khai báo một phương thức show() để mô tả đối tượng xe tương ứng. Show() là một phương thức không cần tham số và kiểu trả về là void.

Lưu ý:

- Khả năng truy nhập phương thức từ bên ngoài là phụ thuộc vào phương thức được khai báo trong phạm vi của từ khóa nào: private, protected hay public.

Định nghĩa phương thức

Trong C++, việc cài đặt chi tiết nội dung của phương thức có thể tiến hành ngay trong phạm vi lớp hoặc bên ngoài phạm vi định nghĩa lớp. Cú pháp chỉ khác nhau ở dòng khai báo tên phương thức.

- Nếu cài đặt phương thức ngay trong phạm vi định nghĩa lớp, cú pháp là:

```
<Kiểu trả về> <Tên phương thức>(<[Các tham số]>){
    ... // Cài đặt chi tiết
}
```
- Nếu cài đặt phương thức bên ngoài phạm vi định nghĩa lớp, ta phải dùng chỉ thị phạm vi “::” để chỉ ra rằng đây là một phương thức của lớp mà không phải là một hàm tự do trong chương trình:

```
<Kiểu trả về> <Tên lớp>::<Tên phương thức>(<[Các tham số]>){
    ... // Cài đặt chi tiết
}
```

Ví dụ, nếu cài đặt phương thức show() của lớp Car ngay trong phạm vi định nghĩa lớp, ta cài đặt như sau:

```
class Car{
private:
    int    speed;        // Tốc độ
    char   mark[20];     // Nhãn hiệu
public:
    void show(){         // Khai báo phương thức ngay trong lớp
        cout << "This is a " << mark << " having a speed of "
```

```

        << speed << "km/h!" << endl;
    return;
}

};

```

Nếu muốn cài đặt bên ngoài lớp, ta cài đặt như sau:

```

class Car{
private:
    int    speed;        // Tốc độ
    char   mark[20];     // Nhãn hiệu
public:
    void show();         // Giới thiệu xe
};
/* Khai báo phương thức bên ngoài lớp */
void Car::show() {
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h!" << endl;
    return;
}

```

Lưu ý:

- Nếu phương thức được cài đặt ngay trong lớp thì các tham số phải tường minh, nghĩa là mỗi tham số phải được biểu diễn bằng một cặp <Kiểu dữ liệu> <Tên tham số> như khi cài đặt chi tiết một hàm tự do trong chương trình.
- Thông thường, chỉ các phương thức ngắn (trên một dòng) là nên cài đặt ngay trong lớp. Còn lại nên cài đặt các phương thức bên ngoài lớp để chương trình được sáng sủa, rõ ràng và dễ theo dõi.

Sử dụng phương thức

Cũng tương tự như các thuộc tính của lớp, các phương thức cũng có thể được sử dụng bên ngoài lớp thông qua tên biến lớp, hoặc có thể được dùng ngay trong lớp bởi các phương thức khác của lớp định nghĩa nó.

- Nếu phương thức được dùng bên ngoài phạm vi lớp, cú pháp phải thông qua tên biến lớp (cách này chỉ sử dụng được với các phương thức có tính chất public):

```
<Tên biến lớp>.<Tên phương thức>([<Các đối số>]);
```

- Nếu thuộc tính được dùng bên trong lớp, cú pháp đơn giản hơn:

```
<Tên phương thức>([<Các đối số>]);
```

Ví dụ, với định nghĩa lớp:

```

class Car{
private:
    int    speed;        // Tốc độ
    char   mark[20];     // Nhãn hiệu
public:
    void show();         // Giới thiệu xe

```



```
};  
/* Khai báo phương thức bên ngoài lớp */  
void Car::show() {  
    cout << "This is a " << mark << " having a speed of "  
        << speed << "km/h!" << endl;  
    return;  
}
```

ta khai báo một biến lớp:

```
Car myCar;
```

Thì có thể sử dụng phương thức giới thiệu xe như sau:

```
myCar.show();
```

Lưu ý:

- Khi dùng phương thức bên trong các phương thức khác của lớp, mà phương thức lại bị trùng với các phương thức tự do của chương trình, ta phải chỉ rõ việc dùng phương thức của lớp (mà không phải dùng phương thức tự do) bằng cách dùng chỉ thị phạm vi lớp "::" với cú pháp:

```
<Tên lớp>::<Tên phương thức>([<Các đối số>]);
```

Chương trình 5.1 cài đặt đầy đủ một lớp xe ô tô (Car) với các thuộc tính có tính chất cục bộ:

- Tốc độ xe (speed)
- Nhân hiệu xe (mark)
- Giá xe (price)

Và các phương thức có tính chất public:

- Khởi tạo các tham số (init)
- Giới thiệu xe (show)
- Các phương thức truy nhập (get/set) các thuộc tính

Sau đó, chương trình main sẽ sử dụng lớp Car này để định nghĩa các đối tượng cụ thể và sử dụng các phương thức của lớp này.

Chương trình 5.1

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp */  
class Car{  
    private:  
        int    speed;           // Tốc độ  
        char   mark[20];       // Nhân hiệu  
        float  price;          // Giá xe  
    public:  
        void   setSpeed(int);   // Gán tốc độ cho xe
```

```

        int    getSpeed();           // Đọc tốc độ xe
        void   setMark(char);       // Gán nhãn cho xe
        char[] getMark();           // Đọc nhãn xe
        void   setPrice(float);     // Gán giá cho xe
        float  getPrice();          // Đọc giá xe
        void   init(int, char[], float); // Khởi tạo thông tin về xe
        void   show();              // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
void Car::setSpeed(int speedIn){    // Gán tốc độ cho xe
    speed = speedIn;
}
int Car::getSpeed(){               // Đọc tốc độ xe
    return speed;
}
void Car::setMark(char markIn){    // Gán nhãn cho xe
    strcpy(mark, markIn);
}
char[] Car::getMark(){            // Đọc nhãn xe
    return mark;
}
void Car::setPrice(float priceIn){ // Gán giá cho xe
    price = priceIn;
}
float Car::getPrice(){            // Đọc giá xe
    return price;
}

void Car::init(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
    return;
}

void Car::show(){                 // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính

```



```
void main(){
    clrscr();
    Car myCar;                                // Khai báo biến lớp

    // Khởi tạo lần thứ nhất
    cout << "Xe thu nhât: " << endl;
    myCar.init(100, "Ford", 3000);
    cout << "Toc do (km/h): " << myCar.getSpeed() << endl;
    cout << "Nhan hieu : " << myCar.getMark() << endl;
    cout << "Gia ($) : " << myCar.getPrice() << endl;

    // Thay đổi thuộc tính xe
    cout << "Xe thu hai: " << endl;
    myCar.setSpeed(150);
    myCar.setMark("Mercedes");
    myCar.setPrice(5000);
    myCar.show();
    return;
}
```

Chương trình 5.1 sẽ in ra kết quả như sau:

```
Xe thu nhât:
Toc do (km/h): 100
Nhan hieu: Ford
Gia ($) : 3000
Xe thu hai:
This is a Mercedes having a speed of 150km/h and its price is $5000
```

5.3 PHẠM VI TRUY NHẬP LỚP

5.3.1 Phạm vi truy nhập lớp

Trong C++, có một số khái niệm về phạm vi, xếp từ bé đến lớn như sau:

- **Phạm vi khối lệnh:** Trong phạm vi giữa hai dấu giới hạn “{}” của một khối lệnh. Ví dụ các lệnh trong khối lệnh lặp while(){} sẽ có cùng phạm vi khối lệnh.
- **Phạm vi hàm:** Các lệnh trong cùng một hàm có cùng mức phạm vi hàm.
- **Phạm vi lớp:** Các thành phần của cùng một lớp có cùng phạm vi lớp với nhau: các thuộc tính và các phương thức của cùng một lớp.
- **Phạm vi chương trình (còn gọi là phạm vi tệp):** Các lớp, các hàm, các biến được khai báo và định nghĩa trong cùng một tệp chương trình thì có cùng phạm vi chương trình.

Trong phạm vi truy nhập lớp, ta chỉ quan tâm đến hai phạm vi lớn nhất, đó là phạm vi lớp và phạm vi chương trình. Trong C++, phạm vi truy nhập lớp được quy định bởi các từ khóa về thuộc tính truy nhập:

- **private**: Các thành phần của lớp có thuộc tính private thì chỉ có thể được truy nhập trong phạm vi lớp.
- **protected**: Trong cùng một lớp, thuộc tính protected cũng có ảnh hưởng tương tự như thuộc tính private: các thành phần lớp có thuộc tính protected chỉ có thể được truy nhập trong phạm vi lớp. Ngoài ra nó còn có thể được truy nhập trong các lớp con khi có kế thừa (sẽ được trình bày trong chương 6).
- **public**: các thành phần lớp có thuộc tính public thì có thể được truy nhập trong phạm vi chương trình, có nghĩa là nó có thể được truy nhập trong các hàm tự do, các phương thức bên trong các lớp khác...

Ví dụ, thuộc tính price của lớp Car có tính chất private nên chỉ có thể truy nhập bởi các phương thức của lớp Car. Không thể truy nhập từ bên ngoài lớp (phạm vi chương trình), chẳng hạn trong một hàm tự do ngoài lớp Car.

```
void Car::setPrice(float priceIn) {  
    price = priceIn; // Đúng, vì setPrice là một phương thức  
                    // của lớp Car  
}
```

nhưng:

```
void freeFunction(Car myCar) {  
    myCar.price = 3000; // Lỗi, vì freeFunction là một hàm tự do  
                      // nằm ngoài phạm vi lớp Car  
}
```

Khi đó, hàm freeFunction phải truy nhập gián tiếp đến thuộc tính price thông qua phương thức truy nhập có tính chất public như sau:

```
void freeFunction(Car myCar) {  
    myCar.setPrice(3000); // Đúng, vì setPrice là một phương thức  
                          // của  
                          // lớp Car có thuộc tính public  
}
```

Tuy nhiên, C++ cho phép một cách đặc biệt để truy nhập đến các thành phần private và protected của một lớp bằng khái niệm *hàm bạn* và *lớp bạn* của một lớp: trong các hàm bạn và lớp bạn của một lớp, có thể truy nhập đến các thành phần private và protected như bên trong phạm vi lớp đó.

5.3.2 Hàm bạn

Có hai kiểu hàm bạn cơ bản trong C++:

- Một hàm tự do là hàm bạn của một lớp
- Một hàm thành phần (phương thức) của một lớp là bạn của một lớp khác

Ngoài ra còn có một số kiểu hàm bạn mở rộng từ hai kiểu này:

- Một hàm là bạn của nhiều lớp

- Tất cả các hàm của một lớp là bạn của lớp khác (lớp bạn)

Hàm tự do bạn của một lớp

Một hàm bạn của một lớp được khai báo bằng từ khóa **friend** khi khai báo khuôn mẫu hàm trong lớp tương ứng.

```
class <Tên lớp>{
...    // Khai báo các thành phần lớp như thông thường
// Khai báo hàm bạn
friend <Kiểu trả về> <Tên hàm bạn>([<Các tham số>]);
};
```

Khi đó, định nghĩa chi tiết hàm bạn được thực hiện như định nghĩa một hàm tự do thông thường:

```
<Kiểu trả về> <Tên hàm bạn>([<Các tham số>]){
...    // Có thể truy nhập trực tiếp các thành phần private
        // của lớp đã khai báo
}
```

Lưu ý:

- Mặc dù hàm bạn được khai báo khuôn mẫu hàm trong phạm vi lớp, nhưng hàm bạn tự do lại không phải là một phương thức của lớp. Nó là hàm tự do, việc định nghĩa và sử dụng hàm này hoàn toàn tương tự như các hàm tự do khác.
- Việc khai báo khuôn mẫu hàm bạn trong phạm vi lớp ở vị trí nào cũng được: hàm bạn không bị ảnh hưởng bởi các từ khóa private, protected hay public trong lớp.
- Trong hàm bạn, có thể truy nhập trực tiếp đến các thành phần private và protected của đối tượng có kiểu lớp mà nó làm bạn (truy nhập thông qua đối tượng cụ thể).

Chương trình 5.2 minh họa việc định nghĩa một hàm bạn của lớp Car, hàm này so sánh xem hai chiếc xe, chiếc nào đắt hơn.

Chương trình 5.2

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

/* Định nghĩa lớp */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        void   init(int, char[], float); // Khởi tạo thông tin về xe
        // Khai báo hàm bạn của lớp
        friend void moreExpensive(Car, Car);
};
```

```

/* Khai báo phương thức bên ngoài lớp */
void Car::init(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
    return;
}

/* Định nghĩa hàm bạn tự do */
void moreExpensive(Car car1, Car car2){
    if(car1.price > car2.price)//Truy nhập đến các thuộc tính private
        cout << "xe thu nhat dat hon" << endl;
    else if(car1.price < car2.price)
        cout << "xe thu nhat dat hon" << endl;
    else
        cout << "hai xe dat nhu nhau" << endl;
    return;
}

// Hàm main, chương trình chính
void main(){
    clrscr();
    Car car1, car2;                                // Khai báo biến lớp

    // Khởi tạo xe thứ nhất, thứ hai
    car1.init(100, "Ford", 3000);
    car2.init(150, "Mercedes", 3500);

    // So sánh giá hai xe
    moreExpensive(car1, car2);                      // Sử dụng hàm bạn tự do
    return;
}

```

Chương trình 5.2 sẽ in ra thông báo:

xe thu hai dat hon

vì xe thứ hai có giá \$3500, trong khi xe thứ nhất được khởi tạo giá là \$3000.

Phương thức lớp là bạn của một lớp khác

Trong C++, một phương thức của lớp này cũng có thể làm bạn của một lớp kia. Để khai báo một phương thức f của lớp B là bạn của lớp A và f nhận một tham số có kiểu lớp A, ta phải khai báo tuần tự như sau (trong cùng một chương trình):

- Khai báo khuôn mẫu lớp A, để làm tham số cho hàm f của lớp B:

```
class A;
```

- Khai báo lớp B với hàm f như khai báo các lớp thông thường:

```
class B{
    ...           // Khai báo các thành phần khác của lớp B
    void f(A);
};
```

- Khai báo chi tiết lớp A với hàm f của lớp B là bạn

```
class A{
    ...           // Khai báo các thành phần khác của lớp A
    friend void B::f(A);
};
```

- Định nghĩa chi tiết hàm f của lớp B:

```
void B::f(A) {
    ...           // Định nghĩa chi tiết hàm f
}
```

Lưu ý:

- Trong trường hợp này, hàm f chỉ được định nghĩa chi tiết một khi lớp A đã được định nghĩa chi tiết. Do vậy, chỉ có thể định nghĩa chi tiết hàm f ngay trong lớp A (ở bước 3) hoặc sau khi định nghĩa lớp A (ở bước 4), mà không thể định nghĩa chi tiết hàm f ngay trong lớp B (ở bước 2).
- Hàm f có thể truy nhập đến các thành phần private và protected của cả hai lớp A và B. Tuy nhiên, muốn f truy nhập đến các thành phần của lớp A thì phải thông qua một đối tượng cụ thể có kiểu lớp A.

Chương trình 5.3 minh họa việc cài đặt và sử dụng một hàm permission() của lớp Person, là hàm bạn của lớp Car. Hàm này thực hiện việc kiểm tra xem một người có đủ quyền điều khiển xe hay không, theo luật sau:

- Với các loại xe thông thường, người điều khiển phải đủ 18 tuổi.
- Với các loại xe có tốc độ cao hơn 150km/h, người điều khiển phải đủ 21 tuổi.

Chương trình 5.3

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

class Car;                               // Khai báo nguyên mẫu lớp

/* Định nghĩa lớp Person */
class Person{
    private:
        char   name[25];                 // Tên
        int    age;                      // Tuổi
```

```

        public:
            void init(char[], int);          // Khởi tạo thông tin về người
            int permission(Car);             // Xác định quyền điều khiển xe
    };

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;                      // Tốc độ
        char   mark[20];                   // Nhãn hiệu
        float  price;                      // Giá xe
    public:
        void  init(int, char[], float);    // Khởi tạo thông tin về xe
        // Khai báo hàm bạn của lớp
        friend int Person::permission(Car);
};

/* Khai báo phương thức bên ngoài lớp */
void Person::init(char nameIn[], int ageIn){
    strcpy(name, nameIn);
    age = ageIn;
    return;
}

void Car::init(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
    return;
}

/* Định nghĩa hàm bạn */
int Person::permission(Car car){
    if(age < 18)
        return 0;

    //Truy nhập thuộc tính private thông qua đối tượng car
    if((age < 21)&&(car.speed > 150))
        return 0;
    return 1;
}

// Hàm main, chương trình chính
void main(){

```



```
clrscr();
// Khai báo các biến lớp
Car car;
Person person;

// Khởi tạo các đối tượng
car.init(100, "Ford", 3000);
person.init("Vinh", 20);

// Xác định quyền điều khiển xe
if(person.permission(car))           // Sử dụng hàm bạn
    cout << "Co quyen dieu khien" << endl;
else
    cout << "Khong co quyen dieu khien" << endl;
return;
}
```

Chương trình 5.3 sẽ hiển thị thông báo:

Co quyen dieu khien

Vì người chủ xe đã 20 tuổi và xe chỉ có tốc độ 100km/h.

5.3.3 Lớp bạn

Khi tất cả các phương thức của một lớp là bạn của một lớp khác, thì lớp của các phương thức đó cũng trở thành lớp bạn của lớp kia. Muốn khai báo một lớp B là lớp bạn của lớp A, ta khai báo theo tuần tự sau:

- Khai báo khuôn mẫu lớp B:

```
class B;
```

- Định nghĩa lớp A, với khai báo B là lớp bạn:

```
class A{
    ...           // Khai báo các thành phần của lớp A
    // Khai báo lớp bạn B
    friend class B;
};
```

- Định nghĩa chi tiết lớp B:

```
class B{
    ...           // Khai báo các thành phần của lớp B
};
```

Lưu ý:

- Trong trường hợp này, lớp B là lớp bạn của lớp A nhưng không có nghĩa là lớp A cũng là bạn của lớp B: tất cả các phương thức của lớp B có thể truy nhập các thành phần private của lớp A (thông qua các đối tượng có kiểu lớp A) nhưng các phương thức của lớp A lại không thể truy nhập đến các thành phần private của lớp B.

- Muốn các phương thức của lớp A cũng truy nhập được đến các thành phần private của lớp B, thì phải khai báo thêm là lớp A cũng là lớp bạn của lớp B.

5.4 HÀM KHỞI TẠO VÀ HUỖ BỎ

5.4.1 Hàm khởi tạo

Hàm khởi tạo được gọi mỗi khi khai báo một đối tượng của lớp. Ngay cả khi không được khai báo tường minh, C++ cũng gọi hàm khởi tạo ngầm định khi đối tượng được khai báo.

Khai báo hàm khởi tạo

Hàm khởi tạo của một lớp được khai báo tường minh theo cú pháp sau:

```
class <Tên lớp>{  
    public:  
        <Tên lớp>([<Các tham số>]);    // Khai báo hàm khởi tạo  
};
```

Ví dụ:

```
class Car{  
    int    speed;  
    char   mark[20];  
    float  price;  
    public:  
        Car(int speedIn, char markIn[], float priceIn){  
            speed = speedIn;  
            strcpy(mark, markIn);  
            price = priceIn;  
        }  
};
```

là khai báo một hàm khởi tạo với ba tham số của lớp Car.

Lưu ý:

- Hàm khởi tạo phải có tên trùng với tên của lớp
- Hàm khởi tạo không có giá trị trả về
- Hàm khởi tạo có tính chất public
- Có thể có nhiều hàm khởi tạo của cùng một lớp

Sử dụng hàm khởi tạo của lớp

Hàm khởi tạo được sử dụng khi khai báo biến lớp. Khi đó, ta có thể vừa khai báo, vừa khởi tạo giá trị các thuộc tính của đối tượng lớp theo cú pháp sau:

```
<Tên lớp> <Tên đối tượng>([<Các đối số khởi tạo>]);
```

Trong đó:

- **Tên lớp:** là tên kiểu lớp đã được định nghĩa
- **Tên đối tượng:** là tên biến có kiểu lớp, tuân thủ theo quy tắc đặt tên biến của C++

- **Các đối số khởi tạo:** Là các đối số tương ứng với hàm khởi tạo của lớp tương ứng. Tương tự như việc truyền đối số khi dùng các lời gọi hàm thông thường.

Ví dụ, nếu lớp Car có hàm khởi tạo Car(int, char[], float) thì khi khai báo biến có kiểu lớp Car, ta có thể sử dụng hàm khởi tạo này như sau:

```
Car myCar(100, "Ford", 3000);
```

Lưu ý:

- Khi sử dụng hàm khởi tạo, phải truyền đúng số lượng và đúng kiểu của các tham số của các hàm khởi tạo đã được định nghĩa của lớp.
- Khi một lớp đã có ít nhất một hàm khởi tạo tường minh, thì không được sử dụng hàm khởi tạo ngầm định của C++. Do đó, khi đã khai báo ít nhất một hàm khởi tạo, nếu muốn khai báo biến mà không cần tham số, lớp tương ứng phải có ít nhất một hàm khởi tạo không có tham số.

Ví dụ, nếu lớp Car chỉ có duy nhất một hàm khởi tạo như sau:

```
class Car{
public:
    Car(int, char[], float);
};
```

thì không thể khai báo một biến như sau:

```
Car myCar; // Khai báo lỗi
```

- Trong trường hợp dùng hàm khởi tạo không có tham số, ta không cần phải sử dụng cặp dấu ngoặc đơn “()” sau tên biến đối tượng. Việc khai báo trở thành cách khai báo thông thường.

Chương trình 5.4a minh họa việc định nghĩa và sử dụng lớp Car với hai hàm khởi tạo khác nhau.

Chương trình 5.4a

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

/* Định nghĩa lớp */
class Car{
private:
    int    speed;           // Tốc độ
    char   mark[20];        // Nhãn hiệu
    float  price;           // Giá xe
public:
    Car();                  // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đầy đủ tham số
    void   show();          // Giới thiệu xe
};
```

```
/* Khai báo phương thức bên ngoài lớp */
Car::Car() {                                // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo có đầy đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn) {
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::show() {                          // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
         << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main() {
    clrscr();
    Car myCar1;                             // Sử dụng hàm khởi tạo không tham số
    Car myCar2(150, "Mercedes", 5000);      // Dùng hàm khởi tạo đủ tham số

    // Giới thiệu xe thứ nhất
    cout << "Xe thu nhât: " << endl;
    myCar1.show();

    // Giới thiệu xe thứ hai
    cout << "Xe thu hai: " << endl;
    myCar2.show();
    return;
}
```

Chương trình 5.4a sẽ hiển thị các thông tin như sau:

Xe thu nhât:

This is a having a speed of 0km/h and its price is \$0

Xe thu hai:

This is a Mercedes having a speed of 150km/h and its price is \$5000

Lí do là xe thứ nhất sử dụng hàm khởi tạo không có tham số nên xe không có tên, tốc độ và giá đều là mặc định (0). Trong khi đó, xe thứ hai được khởi tạo đầy đủ cả ba tham số nên thông tin giới thiệu xe được đầy đủ.

Tuy nhiên, khi đối tượng có nhiều thuộc tính riêng, để tránh trường hợp phải định nghĩa nhiều hàm khởi tạo cho các trường hợp thiếu vắng một vài tham số khác nhau. Ta có thể sử dụng hàm khởi tạo với các giá trị khởi đầu ngầm định. Chương trình 5.4b cho kết quả hoàn toàn giống chương trình 5.4a, nhưng đơn giản hơn vì chỉ cần định nghĩa một hàm khởi tạo với các tham số có giá trị ngầm định.

Chương trình 5.4b

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

/* Định nghĩa lớp */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhân hiệu
        float  price;           // Giá xe
    public:
        // Khởi tạo với các giá trị ngầm định cho các tham số
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void  show();           // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::show() {                // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
         << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main(){
    clrscr();
    Car myCar1;                   // Các tham số nhận giá trị mặc định
```

```
Car myCar2(150, "Mercedes", 5000); // Dùng hàm khởi tạo đủ tham số

// Giới thiệu xe thứ nhất
cout << "Xe thu nhât: " << endl;
myCar1.show();

// Giới thiệu xe thứ hai
cout << "Xe thu hai: " << endl;
myCar2.show();
return;
}
```

5.4.2 Hàm hủy bỏ

Hàm hủy bỏ được tự động gọi đến khi mà đối tượng được giải phóng khỏi bộ nhớ. Nhiệm vụ của hàm hủy bỏ là dọn dẹp bộ nhớ trước khi đối tượng bị giải phóng. Cú pháp khai báo hàm hủy bỏ như sau:

```
class <Tên lớp>{
    public:
        ~<Tên lớp>(<[Các tham số]>); // Khai báo hàm khởi tạo
};
```

Ví dụ:

```
class Car{
    int    speed;
    char   *mark;
    float  price;
    public:
        ~Car(){
            delete [] mark;
        };
};
```

là khai báo một hàm hủy bỏ của lớp Car với thuộc tính mark có kiểu con trỏ. Hàm này sẽ giải phóng vùng nhớ đã cấp phát cho con trỏ kiểu char của thuộc tính nhãn hiệu xe.

Lưu ý:

- Hàm hủy bỏ phải có tên bắt đầu bằng dấu “~”, theo sau là tên của lớp tương ứng.
- Hàm hủy bỏ không có giá trị trả về.
- Hàm hủy bỏ phải có tính chất public
- Mỗi lớp chỉ có nhiều nhất một hàm hủy bỏ. Trong trường hợp không khai báo tường minh hàm hủy bỏ, C++ sẽ sử dụng hàm hủy bỏ ngầm định.

- Nói chung, khi có ít nhất một trong các thuộc tính của lớp là con trỏ, thì nên dùng hàm hủy bỏ tường minh để giải phóng triệt để các vùng nhớ của các thuộc tính, trước khi đối tượng bị giải phóng khỏi bộ nhớ.

Chương trình 5.5 minh họa việc định nghĩa lớp Car với một hàm khởi tạo có các tham số với giá trị mặc định và một hàm hủy bỏ tường minh.

Chương trình 5.5

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

/* Định nghĩa lớp */
class Car{
    private:
        int    speed;           // Tốc độ
        char   *mark;           // Nhãn hiệu
        float  price;           // Giá xe
    public:
        // Khởi tạo với các giá trị ngầm định cho các tham số
        Car(int speedIn=0, char *markIn=NULL, float priceIn=0);
        void  show();           // Giới thiệu xe
        ~Car();                 // Hàm hủy bỏ tường minh
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char *markIn, float priceIn){
    speed = speedIn;
    mark = markIn;
    price = priceIn;
}

void Car::show(){              // Phương thức giới thiệu xe
    cout << "This is a " << *mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

Car::~~Car(){                  // Hàm hủy bỏ tường minh
    delete [] mark;
    cout << "The object has been destroyed!" << endl;
}
```

```
// Hàm main, chương trình chính
void main() {
    clrscr();
    Car myCar(150, "Mercedes", 5000); // Dùng hàm khởi tạo đủ tham số

    // Giới thiệu xe
    cout << "Gioi thieu xe: " << endl;
    myCar.show();
    return;
}
```

Chương trình 5.5 sẽ in ra thông báo như sau:

```
Gioi thieu xe:
This is a Mercedes having a speed of 150km/h and its price is $5000
The object has been destroyed!
```

Dòng cuối cùng là của hàm hủy bỏ, mặc dù ta không gọi hàm hủy bỏ trực tiếp, nhưng khi thoát khỏi hàm main() của chương trình chính, đối tượng myCar bị giải phóng khỏi bộ nhớ và khi đó, C++ tự động gọi đến hàm hủy bỏ mà ta đã định nghĩa trước. Do vậy, mà có dòng thông báo cuối cùng này.

5.5 CON TRỎ ĐỐI TƯỢNG VÀ MẢNG ĐỐI TƯỢNG

5.5.1 Con trỏ đối tượng

Con trỏ đối tượng là con trỏ trỏ đến địa chỉ của một đối tượng có kiểu lớp. Các thao tác liên quan đến con trỏ đối tượng bao gồm:

- Khai báo con trỏ đối tượng
- Cấp phát bộ nhớ cho con trỏ đối tượng
- Sử dụng con trỏ đối tượng
- Giải phóng bộ nhớ cho con trỏ đối tượng

Khai báo con trỏ đối tượng

Con trỏ đối tượng được khai báo tương tự như khai báo các con trỏ có kiểu thông thường:

```
<Tên lớp> *<Tên con trỏ đối tượng>;
```

Ví dụ, muốn khai báo một con trỏ đối tượng có kiểu của lớp Car, ta khai báo như sau:

```
Car *myCar;
```

Khi đó, myCar là một con trỏ đối tượng có kiểu lớp Car.

Cấp phát bộ nhớ cho con trỏ đối tượng

Con trỏ đối tượng cũng cần phải cấp phát bộ nhớ hoặc trỏ vào một địa chỉ của một đối tượng lớp xác định trước khi được sử dụng. Cấp phát bộ nhớ cho con trỏ đối tượng cũng bằng thao tác **new**:

```
<Tên con trỏ đối tượng> = new <Tên lớp>([<Các đối số>]);
```

Ví dụ, nếu lớp Car có hai hàm khởi tạo như sau:

```
class Car{
public:
    Car();
    Car(int, char[], float);
};
```

thì ta có thể cấp phát bộ nhớ theo hai cách, tương ứng với hai hàm khởi tạo của lớp:

```
myCar = new Car(); // Khởi tạo không tham số
myCar = new Car(100, "Ford", 3000); // Khởi tạo đủ tham số
```

Lưu ý:

- Các đối số truyền phải tương ứng với ít nhất một trong các hàm khởi tạo của lớp.
- Khi sử dụng hàm khởi tạo không có tham số, ta vẫn phải sử dụng cặp ngoặc đơn “()” trong thao tác **new**.
- Khi lớp không có một hàm khởi tạo tường minh nào, sẽ dùng hàm khởi tạo ngầm định của C++ và cú pháp tương tự như sử dụng hàm khởi tạo tường minh không có tham số.
- Có thể vừa khai báo, vừa cấp phát bộ nhớ cho con trỏ đối tượng.

Ví dụ:

```
Car myCar = new Car(); // Khởi tạo không tham số
```

Sử dụng con trỏ đối tượng

Con trỏ đối tượng được sử dụng qua các thao tác:

- Trỏ đến địa chỉ của một đối tượng cùng lớp
- Truy nhập đến các phương thức của lớp

Con trỏ đối tượng có thể trỏ đến địa chỉ của một đối tượng có sẵn, cùng lớp theo cú pháp sau:

```
<Tên con trỏ đối tượng> = &<Tên đối tượng có sẵn>;
```

Ví dụ, ta có một con trỏ và một đối tượng của lớp Car:

```
Car *ptrCar, myCar(100, "Ford", 3000);
```

Khi đó, có thể cho con trỏ ptrCar trỏ vào địa chỉ của đối tượng myCar như sau:

```
ptrCar = &myCar;
```

Khi muốn truy nhập đến các thành phần của con trỏ đối tượng, ta dùng cú pháp sau:

```
<Tên con trỏ đối tượng> -> <Tên thành phần lớp>([<Các đối số>]);
```

Ví dụ, đoạn chương trình sau sẽ thực hiện phương thức giới thiệu xe của lớp Car thông qua con trỏ ptrCar:

```
Car *ptrCar = new Car(100, "Ford", 3000);
ptrCar->show();
```

Lưu ý:

- Danh sách các đối số phải tương thích với tên phương thức tương ứng.
- Các quy tắc phạm vi truy nhập vẫn áp dụng trong truy nhập các thành phần lớp thông qua con trỏ.

Giải phóng bộ nhớ cho con trỏ đối tượng

Con trỏ đối tượng cũng được giải phóng thông qua thao tác **delete**:

```
delete <Tên con trỏ đối tượng>;
```

Ví dụ:

```
Car *ptrCar = new Car(); // Khai báo và cấp phát bộ nhớ
...                      // Sử dụng con trỏ ptrCar
delete ptrCar;           // Giải phóng bộ nhớ.
```

Lưu ý:

- Thao tác **delete** chỉ được dùng khi trước đó, con trỏ được cấp phát bộ nhớ qua thao tác **new**:

```
Car *ptrCar = new Car();
delete ptrCar;           // Đúng.
```

Nhưng không được dùng **delete** khi trước đó, con trỏ chỉ trỏ vào một địa chỉ của đối tượng có sẵn (tĩnh):

```
Car *ptrCar, myCar(100, "Ford", 3000);
ptrCar = &myCar;
delete ptrCar;           // Không được
```

Chương trình 5.6 minh họa việc dùng con trỏ đối tượng có kiểu lớp là Car.

Chương trình 5.6

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

/* Định nghĩa lớp */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        // Khởi tạo với các giá trị ngầm định cho các tham số
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void  show();           // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
```



```

}

void Car::show() {
    // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main() {
    clrscr();
    // Khai báo con trỏ, cấp phát bộ nhớ dùng hàm khởi tạo đủ tham số
    Car *myCar = new Car(150, "Mercedes", 5000);

    // Giới thiệu xe
    cout << "Gioi thieu xe: " << endl;
    myCar->show();

    // Giải phóng con trỏ
    delete myCar;
    return;
}

```

Chương trình 5.6 hiển thị ra thông báo là một lời giới thiệu xe;

```

Gioi thieu xe:
This is a Mercedes having a speed of 150km/h and its price is $5000

```

5.5.2 Mảng các đối tượng

Mảng các đối tượng cũng có thể được khai báo và sử dụng như mảng của các biến có kiểu thông thường.

Khai báo mảng tĩnh các đối tượng

Mảng các đối tượng được khai báo theo cú pháp:

```
<Tên lớp> <Tên biến mảng>[<Số lượng đối tượng>];
```

Ví dụ:

```
Car cars[10];
```

Là khai báo một mảng có 10 đối tượng có cùng kiểu lớp Car.

Lưu ý:

- Có thể khai báo mảng tĩnh các đối tượng mà chưa cần khai báo độ dài mảng, cách này thường dùng khi chưa biết chính xác độ dài mảng:

```
Car cars[];
```

- Muốn khai báo được mảng tĩnh các đối tượng, lớp tương ứng phải có hàm khởi tạo không có tham số. Vì khi khai báo mảng, tương đương với khai báo một dãy các đối tượng với hàm khởi tạo không có tham số.

Khai báo mảng động với con trỏ

Một mảng các đối tượng cũng có thể được khai báo và cấp phát động thông qua con trỏ đối tượng như sau:

```
<Tên lớp> *<Tên biến mảng động> = new <Tên lớp>[<Độ dài mảng>];
```

Ví dụ:

```
Car *cars = new Car[10];
```

Sau khi được sử dụng, mảng động các đối tượng cũng cần phải giải phóng bộ nhớ:

```
delete [] <Tên biến mảng động>;
```

Ví dụ:

```
Car *cars = new Car[10]; // Khai báo và cấp phát động
...                      // Sử dụng biến mảng động
delete [] cars;          // Giải phóng bộ nhớ của mảng động
```

Sử dụng mảng đối tượng

Khi truy nhập vào các thành phần của một đối tượng có chỉ số xác định trong mảng đã khai báo, ta có thể sử dụng cú pháp:

```
<Tên biến mảng>[<Chỉ số đối tượng>].<Tên thành phần>([<Các đối số>]);
```

Ví dụ:

```
Car cars[10];
cars[5].show();
```

sẽ thực hiện phương thức show() của đối tượng có chỉ số thứ 5 (tính từ chỉ số 0) trong mảng cars.

Chương trình 5.7 sẽ cài đặt một chương trình, trong đó nhập vào độ dài mảng, sau đó yêu cầu người dùng nhập thông tin về mảng các xe. Cuối cùng, chương trình sẽ tìm kiếm và hiển thị thông tin về chiếc xe có giá đắt nhất trong mảng.

Chương trình 5.7

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        void   setSpeed(int);    // Gán tốc độ cho xe
```

```

        int    getSpeed();           // Đọc tốc độ xe
        void   setMark(char);        // Gán nhãn cho xe
        char[] getMark();           // Đọc nhãn xe
        void   setPrice(float);      // Gán giá cho xe
        float  getPrice();          // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void   show();              // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::setSpeed(int speedIn){    // Gán tốc độ cho xe
    speed = speedIn;
}

int Car::getSpeed(){               // Đọc tốc độ xe
    return speed;
}

void Car::setMark(char markIn){    // Gán nhãn cho xe
    strcpy(mark, markIn);
}

char[] Car::getMark(){            // Đọc nhãn xe
    return mark;
}

void Car::setPrice(float priceIn){ // Gán giá cho xe
    price = priceIn;
}

float Car::getPrice(){            // Đọc giá xe
    return price;
}

void Car::show(){                 // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
         << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính

```

```

void main() {
    clrscr();
    int length;           // Chiều dài mảng
    float maxPrice = 0;   // Giá đắt nhất
    int index = 0;        // Chỉ số của xe đắt nhất
    Car *cars;            // Khai báo mảng đối tượng

    // Nhập số lượng xe, tức là chiều dài mảng
    cout << "So luong xe: ";
    cin >> length;

    // Cấp phát bộ nhớ động cho mảng
    cars = new Car[length];

    // Khởi tạo các đối tượng trong mảng
    for(int i=0; i<length; i++){
        int speed;        // (Biến tạm) tốc độ
        char mark[20];     // (Biến tạm) nhãn hiệu
        float price;      // (Biến tạm) giá xe
        cout << "Xe thu " << i << ": " << endl;
        cout << "Toc do (km/h): ";
        cin >> speed; cars[i].setSpeed(speed);    // Nhập tốc độ
        cout << "Nhan hieu : ";
        cin >> mark; cars[i].setMark(mark);       // Nhập nhãn xe
        cout << "Gia ($): ";
        cin >> price; cars[i].setPrice(price);    // Nhập giá xe

        if(maxPrice < price){
            maxPrice = price;
            index = i;
        }
    }

    // Tìm xe đắt nhất
    for(int i=0; i<length; i++){
        if(i == index){
            cars[i].show(); // Giới thiệu xe đắt nhất
            break;
        }
    }

    // Giải phóng bộ nhớ của mảng
    delete [] cars;
    return;
}

```

```
}
```

TỔNG KẾT CHƯƠNG 5

Nội dung chương 5 đã tập trung trình bày các vấn đề cơ bản về lớp đối tượng trong ngôn ngữ C++:

- Khai báo, định nghĩa lớp bằng từ khóa `class`
- Sử dụng biến lớp như những đối tượng cụ thể
- Khai báo, định nghĩa các thuộc tính và các phương thức của lớp: định nghĩa các phương thức trong hoặc ngoài phạm vi khai báo lớp
- Khai báo phạm vi truy nhập lớp bằng các từ khóa chỉ phạm vi: `private`, `protected` và `public`. Giới thiệu các kiểu hàm có thể truy nhập phạm vi bất quy tắc: hàm bạn và lớp bạn với từ khóa `friend`.
- Khai báo, định nghĩa tường minh hàm khởi tạo của lớp và sử dụng khi khai báo biến lớp
- Khai báo, định nghĩa tường minh hàm hủy bỏ của lớp, nhằm dọn dẹp, giải phóng bộ nhớ trước khi đối tượng bị giải phóng khỏi bộ nhớ.
- Khai báo, cấp phát bộ nhớ, sử dụng và giải phóng bộ nhớ cho con trỏ đối tượng
- Khai báo, cấp phát bộ nhớ động, sử dụng và giải phóng vùng nhớ của mảng các đối tượng.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 5

1. Trong các khai báo lớp sau, những khai báo nào là đúng:
 - a. `class MyClass;`
 - b. `Class MyClass;`
 - c. `class MyClass{...};`
 - d. `Class MyClass{...};`
2. Giả sử ta đã định nghĩa lớp `MyClass`, bây giờ ta khai báo một đối tượng thuộc kiểu lớp này. Khai báo nào là đúng:
 - a. `class MyClass me;`
 - b. `MyClass me;`
 - c. `MyClass me();`
 - d. `MyClass me{};`
3. Trong các khai báo thuộc tính ngay trong phạm vi của khai báo lớp như sau, những khai báo nào là đúng:
 - a. `int myAge;`
 - b. `private int myAge;`
 - c. `public int myAge;`
 - d. `private: int myAge;`

4. Trong các khai báo phương thức ngay trong phạm vi của khai báo lớp MyClass như sau, những khai báo nào là đúng:

a. `public: void show();`
 b. `void show();`
 c. `void show(){cout << "hello!";};`
 d. `void MyClass::show(){cout << "hello!";}`

5. Xét đoạn chương trình sau:

```
class MyClass{
    int age;
    public:
        int getAge();
};
MyClass me;
```

Khi đó, trong các lệnh sau, lệnh nào có thể thêm vào cuối đoạn chương trình trên:

a. `cout << MyClass::age;`
 b. `cout << me.age;`
 c. `cout << me.getAge();`
 d. `cin >> me.age;`

6. Trong các khai báo hàm khởi tạo cho lớp MyClass như sau, những khai báo nào là đúng:

a. `myClass();`
 b. `MyClass();`
 c. `MyClass(MyClass);`
 d. `void MyClas();`
 e. `MyClass MyClass();`

7. Trong các khai báo hàm hủy bỏ cho lớp MyClass như sau, những khai báo nào là đúng:

a. `~myClass();`
 b. `~MyClass();`
 c. `~MyClass(MyClass);`
 d. `void ~MyClas();`
 e. `MyClass ~MyClass();`

8. Giả sử ta khai báo lớp MyClass có một thuộc tính age và một hàm khởi tạo:

```
class MyClass{
    int age;
    public:
        MyClass(MyClass me){
            ...
        };
        int getAge(){return age;}
};
```

Trong các dòng lệnh sau, những dòng nào có thể xuất hiện trong hàm khởi tạo trên:

- a. `age = MyClass.age;`
- b. `age = me.age;`
- c. `age = MyClass.getAge();`
- d. `age = me.getAge();`
- e. `age = 10;`

9. Xét khai báo lớp như sau:

```
class MyClass{
    public:
        MyClass(MyClass);
        MyClass(int);
};
```

Khi đó, trong số các khai báo đối tượng sau, những khai báo nào là đúng:

- a. `MyClass me;`
- b. `MyClass me();`
- c. `MyClass me(10);`
- d. `MyClass me1(10), me2(me1);`

10. Xét khai báo lớp như sau:

```
class MyClass{
    public:
        MyClass();
};
```

Khi đó, trong số các khai báo con trỏ đối tượng sau, những khai báo nào là đúng:

- a. `MyClass *me;`
- b. `MyClass *me();`
- c. `MyClass *me = new me();`
- d. `MyClass *me = new MyClass();`

11. Xét khai báo lớp như sau:

```
class MyClass{
    public:
        MyClass(int i=0, int j=0, float k=0);
};
```

Khi đó, trong số các khai báo con trỏ đối tượng sau, những khai báo nào là đúng:

- a. `MyClass *me;`
- b. `MyClass *me = new MyClass();`
- c. `MyClass *me = new MyClass(1, 20);`
- d. `MyClass *me = new MyClass(1, 20, 30);`

12. Khai báo một lớp nhân viên, có tên là `Employee`, với ba thuộc tính có tính chất `private`:

- Tên nhân viên, có dạng một con trỏ kiểu `char`

- Tuổi nhân viên, có kiểu int
 - Lương nhân viên, có kiểu float.
13. Thêm vào lớp Employee trong bài 12 các phương thức có tính chất public:
- set/get giá trị thuộc tính tên nhân viên
 - set/get giá trị thuộc tính tuổi nhân viên
 - set/get giá trị thuộc tính lương nhân viên.
14. Viết một hàm khởi tạo không có tham số cho lớp Employee trong bài 13, các thuộc tính của lớp nhận giá trị mặc định:
- giá trị thuộc tính tên nhân viên, mặc định là chuỗi kí tự rỗng ""
 - giá trị thuộc tính tuổi nhân viên, mặc định là 18
 - giá trị thuộc tính lương nhân viên, mặc định là \$100.
15. Viết thêm một hàm khởi tạo với đầy đủ ba tham số tương ứng với ba thuộc tính của lớp Employee trong bài 14.
16. Thêm vào lớp Employee một phương thức show() để giới thiệu về tên, tuổi và lương của đối tượng nhân viên.
17. Viết một hàm hủy bỏ tường minh cho lớp Employee nhằm giải phóng vùng nhớ của con trỏ char, là kiểu của thuộc tính tên nhân viên.
18. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo không tham số
 - Dùng hàm show() để giới thiệu về đối tượng đó
19. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo không tham số
 - Nhập từ bàn phím giá trị các thuộc tính tên, tuổi, lương nhân viên.
 - Gán các giá trị này cho các thuộc tính của đối tượng đã khai báo, dùng các hàm set
 - Dùng hàm show() để giới thiệu về đối tượng đó
20. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo với đủ 3 tham số
 - Dùng hàm show() để giới thiệu về đối tượng đó
21. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một con trỏ đối tượng kiểu Employee
 - Cấp phát bộ nhớ, dùng hàm khởi tạo với đủ 3 tham số
 - Dùng hàm show() để giới thiệu về đối tượng mà con trỏ này đang trỏ tới
22. Viết một chương trình nhập dữ liệu cho một mảng động các đối tượng của lớp Employee trong bài 17. Chiều dài mảng động cũng được nhập từ bàn phím.

23. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên trẻ nhất và nhân viên già nhất trong mảng đó.
24. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên có lương cao nhất và nhân viên có lương thấp nhất trong mảng đó.
25. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên có tên xác định, do người dùng nhập từ bàn phím.



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhkx@pit.edu.vn

CHƯƠNG 6

TÍNH KẾ THỪA VÀ ĐA HÌNH

Nội dung chương này tập trung trình bày các vấn đề liên quan đến tính kế thừa và tương ứng bội (đa hình) trong ngôn ngữ C++:

- Khái niệm kế thừa, dẫn xuất và các kiểu dẫn xuất
- Khai báo, định nghĩa các hàm khởi tạo và hàm hủy bỏ trong lớp dẫn xuất
- Truy nhập tới các thành phần của lớp cơ sở và các lớp dẫn xuất
- Việc một lớp được kế thừa từ nhiều lớp cơ sở khác nhau
- Khai báo và sử dụng các lớp cơ sở trừu tượng trong kế thừa
- Tính đa hình trong kế thừa

6.1 KHÁI NIỆM KẾ THỪA

Lập trình hướng đối tượng có hai đặc trưng cơ bản:

- Đóng gói dữ liệu, được thể hiện bằng cách dùng khái niệm lớp để biểu diễn đối tượng với các thuộc tính private, chỉ cho phép bên ngoài truy nhập vào thông qua các phương thức get/set.
- Dùng lại mã, thể hiện bằng việc thừa kế giữa các lớp. Việc thừa kế cho phép các lớp thừa kế (gọi là lớp dẫn xuất) sử dụng lại các phương thức đã được định nghĩa trong các lớp gốc (gọi là lớp cơ sở).

6.1.1 Khai báo thừa kế

Cú pháp khai báo một lớp kế thừa từ một lớp khác như sau:

```
class <Tên lớp dẫn xuất>: <Từ khóa dẫn xuất> <Tên lớp cơ sở>{  
    ...           // Khai báo các thành phần lớp  
};
```

Trong đó:

- **Tên lớp dẫn xuất:** là tên lớp được cho kế thừa từ lớp khác. Tên lớp này tuân thủ theo quy tắc đặt tên biến trong C++.
- **Tên lớp cơ sở:** là tên lớp đã được định nghĩa trước đó để cho lớp khác kế thừa. Tên lớp này cũng tuân thủ theo quy tắc đặt tên biến của C++.
- **Từ khóa dẫn xuất:** là từ khóa quy định tính chất của sự kế thừa. Có ba từ khóa dẫn xuất là private, protected và public. Mục tiếp theo sẽ trình bày ý nghĩa của các từ khóa dẫn xuất này.

Ví dụ:

```
class Bus: public Car{  
    ...           // Khai báo các thành phần  
};
```

là khai báo một lớp Bus (xe buýt) kế thừa từ lớp Car (xe ô tô) với tính chất kế thừa là public.

6.1.2 Tính chất dẫn xuất

Sự kế thừa cho phép trong lớp dẫn xuất có thể sử dụng lại một số mã nguồn của các phương thức và thuộc tính đã được định nghĩa trong lớp cơ sở. Nghĩa là lớp dẫn xuất có thể truy nhập trực tiếp đến một số thành phần của lớp cơ sở. Tuy nhiên, phạm vi truy nhập từ lớp dẫn xuất đến lớp cơ sở không phải bao giờ cũng giống nhau: chúng được quy định bởi các từ khóa dẫn xuất private, protected và public.

Dẫn xuất private

Dẫn xuất private quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần private của lớp dẫn xuất
- Các thành phần public của lớp cơ sở cũng trở thành các thành phần private của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Dẫn xuất protected

Dẫn xuất protected quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần protected của lớp dẫn xuất
- Các thành phần public của lớp cơ sở cũng trở thành các thành phần protected của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Dẫn xuất public

Dẫn xuất public quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần protected của lớp dẫn xuất.
- Các thành phần public của lớp cơ sở vẫn là các thành phần public của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Bảng 6.1 tóm tắt lại các quy tắc truy nhập được quy định bởi các từ khóa dẫn xuất.

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất
private	private	Không truy nhập được

	protected public	private private
protected	private protected public	Không truy nhập được protected protected
public	private protected public	Không truy nhập được protected public

6.2 HÀM KHỞI TẠO VÀ HUỖ BỎ TRONG KẾ THỪA

6.2.1 Hàm khởi tạo trong kế thừa

Khi khai báo một đối tượng có kiểu lớp được dẫn xuất từ một lớp cơ sở khác. Chương trình sẽ tự động gọi tới hàm khởi tạo của lớp dẫn xuất. Tuy nhiên, thứ tự được gọi sẽ bắt đầu từ hàm khởi tạo tương ứng của lớp cơ sở, sau đó đến hàm khởi tạo của lớp dẫn xuất. Do đó, thông thường, trong hàm khởi tạo của lớp dẫn xuất phải có hàm khởi tạo của lớp cơ sở.

Cú pháp khai báo hàm khởi tạo như sau:

```
<Tên hàm khởi tạo dẫn xuất>([<Các tham số>]):
    <Tên hàm khởi tạo cơ sở>([<Các đối số>]){
    ...    // Khởi tạo các thuộc tính mới bổ sung của lớp dẫn xuất
    };
```

Vì tên hàm khởi tạo là trùng với tên lớp, nên có thể viết lại thành:

```
<Tên lớp dẫn xuất>([<Các tham số>]):
    <Tên lớp cơ sở>([<Các đối số>]){
    ...    // Khởi tạo các thuộc tính mới bổ sung của lớp dẫn xuất
    };
```

Ví dụ:

```
Bus() : Car() {
    ...    // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
}
```

là một định nghĩa một hàm khởi tạo của lớp Bus kế thừa từ lớp Car. Định nghĩa này được thực hiện trong phạm vi khai báo lớp Bus. Đây là một hàm khởi tạo không tham số, nó gọi tới hàm khởi tạo không tham số của lớp Car.

Lưu ý:

- Nếu định nghĩa hàm khởi tạo bên ngoài phạm vi lớp thì phải thêm tên lớp dẫn xuất và toán tử phạm vi "::" trước tên hàm khởi tạo.
- Giữa tên hàm khởi tạo của lớp dẫn xuất và hàm khởi tạo của lớp cơ sở, chỉ có một dấu hai chấm ".", nếu là hai dấu "::" thì trở thành toán tử phạm vi lớp.

- Nếu không chỉ rõ hàm khởi tạo của lớp cơ sở sau dấu hai chấm “:” chương trình sẽ tự động gọi hàm khởi tạo ngầm định hoặc hàm khởi tạo không có tham số của lớp cơ sở nếu hàm đó được định nghĩa tường minh trong lớp cơ sở.

Ví dụ, định nghĩa hàm khởi tạo:

```
Bus():Car() {
    ...           // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
};
```

Có thể thay bằng:

```
Bus() {           // Gọi hàm khởi tạo không tham số của lớp Car
    ...           // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
};
```

Chương trình 6.1 định nghĩa lớp Car có 3 thuộc tính với hai hàm khởi tạo, sau đó định nghĩa lớp Bus có thêm thuộc tính label là số hiệu của tuyến xe buýt. Lớp Bus sẽ được cài đặt hai hàm khởi tạo tường minh, gọi đến hai hàm khởi tạo tương ứng của lớp Car.

Chương trình 6.1

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    int    speed;           // Tốc độ
    char   mark[20];        // Nhân hiệu
    float  price;           // Giá xe
public:
    Car();                 // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đủ tham số
};

Car::Car() {               // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
```

```
class Bus: public Car{
    int label; // Số hiệu tuyến xe
public:
    Bus(); // Khởi tạo không tham số
    Bus(int, char[], float, int); // Khởi tạo đủ tham số
};

Bus::Bus():Car(){ // Khởi tạo không tham số
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}
```

Trong hàm khởi tạo của lớp Bus, muốn khởi tạo các thuộc tính của lớp Car, ta phải khởi tạo gián tiếp thông qua hàm khởi tạo của lớp Car mà không thể gán giá trị trực tiếp cho các thuộc tính speed, mark và price. Lí do là các thuộc tính này có tính chất private, nên lớp dẫn xuất không thể truy nhập trực tiếp đến chúng.

6.2.2 Hàm hủy bỏ trong kế thừa

Khi một đối tượng lớp dẫn xuất bị giải phóng khỏi bộ nhớ, thứ tự gọi các hàm hủy bỏ ngược với thứ tự gọi hàm thiết lập: gọi hàm hủy bỏ của lớp dẫn xuất trước khi gọi hàm hủy bỏ của lớp cơ sở. Vì mỗi lớp chỉ có nhiều nhất là một hàm hủy bỏ, nên ta không cần phải chỉ ra hàm hủy bỏ nào của lớp cơ sở sẽ được gọi sau khi hủy bỏ lớp dẫn xuất. Do vậy, hàm hủy bỏ trong lớp dẫn xuất được khai báo và định nghĩa hoàn toàn giống với các lớp thông thường:

```
<Tên lớp>::~~<Tên lớp>([<Các tham số>]){
    ... // giải phóng phần bộ nhớ cấp phát cho các thuộc tính bổ sung
}
```

Lưu ý:

- Hàm hủy bỏ của lớp dẫn xuất chỉ giải phóng phần bộ nhớ được cấp phát động cho các thuộc tính mới bổ sung trong lớp dẫn xuất, nếu có, mà không được giải phóng bộ nhớ được cấp cho các thuộc tính trong lớp cơ sở (phần này là do hàm hủy bỏ của lớp cơ sở đảm nhiệm).
- Không phải gọi tường minh hàm hủy bỏ của lớp cơ sở trong hàm hủy bỏ của lớp dẫn xuất.
- Ngay cả khi lớp dẫn xuất không định nghĩa tường minh hàm hủy bỏ (do không cần thiết) mà lớp cơ sở lại có định nghĩa tường minh. Chương trình vẫn gọi hàm hủy bỏ ngầm định của lớp dẫn xuất, sau đó vẫn gọi hàm hủy bỏ tường minh của lớp cơ sở.

Chương trình 6.2 cài đặt lớp Bus kế thừa từ lớp Car: lớp Car có một thuộc tính có dạng con trỏ nên cần giải phóng bằng hàm hủy bỏ tường minh. Lớp Bus có thêm một thuộc tính có dạng con

trở là danh sách các đường phố mà xe buýt đi qua (mảng động các chuỗi kí tự `*char[]`) nên cũng cần giải phóng bằng hàm hủy bỏ tường minh.

Chương trình 6.2

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    char *mark;                // Nhãn hiệu xe
public:
    ~Car();                    // Hủy bỏ tường minh
};

Car::~Car() {                  // Hủy bỏ tường minh
    delete [] mark;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    char *voyage[];            // Hành trình tuyến xe
public:
    ~Bus();                    // Hủy bỏ tường minh
};

Bus::~Bus() {                  // Hủy bỏ tường minh
    delete [] voyage;
}
```

Trong hàm hủy bỏ của lớp Bus, ta chỉ được giải phóng vùng nhớ được cấp phát cho thuộc tính voyage (hành trình của xe buýt), là thuộc tính được bổ sung thêm của lớp Bus. Mà không được giải phóng vùng nhớ cấp phát cho thuộc tính mark (nhãn hiệu xe), việc này là thuộc trách nhiệm của hàm hủy bỏ của lớp Car vì thuộc tính mark được khai báo trong lớp Car.

6.3 TRUY NHẬP TỚI CÁC THÀNH PHẦN TRONG KẾ THỪA LỚP

6.3.1 Phạm vi truy nhập

Mối quan hệ giữa các thành phần của lớp cơ sở và lớp dẫn xuất được quy định bởi các từ khóa dẫn xuất, như đã trình bày trong mục 6.1.2, được tóm tắt trong bảng 6.2

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất
private	private protected	Không truy nhập được private

	public	private
protected	private protected public	Không truy nhập được protected protected
public	private protected public	Không truy nhập được protected public

Ta xét phạm vi truy nhập theo hai loại:

- Phạm vi truy nhập từ các hàm bạn, lớp bạn của lớp dẫn xuất
- Phạm vi truy nhập từ các đối tượng có kiểu lớp dẫn xuất

Truy nhập từ các hàm bạn và lớp bạn của lớp dẫn xuất

Nhìn vào bảng tổng kết 6.2, phạm vi truy nhập của hàm bạn, lớp bạn của lớp dẫn xuất vào lớp cơ sở như sau:

- Với dẫn xuất private, hàm bạn có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần private của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Với dẫn xuất protected, hàm bạn cũng có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần protected của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Với dẫn xuất public, hàm bạn cũng có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần protected và public của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Đối với cả ba loại dẫn xuất, hàm bạn đều không truy nhập được các thành phần private của lớp cơ sở, vì các thành phần này cũng không truy nhập được từ lớp dẫn xuất.

Truy nhập từ các đối tượng tạo bởi lớp dẫn xuất

Nhìn vào bảng tổng kết 6.2, phạm vi truy nhập của các đối tượng của lớp dẫn xuất vào lớp cơ sở như sau:

- Với dẫn xuất private, đối tượng của lớp dẫn xuất không truy nhập được bất cứ thành phần nào của lớp cơ sở vì chúng trở thành các thành phần private của lớp dẫn xuất, không truy nhập được từ bên ngoài.
- Với dẫn xuất protected, đối tượng của lớp dẫn xuất không truy nhập được bất cứ thành phần nào của lớp cơ sở vì chúng trở thành các thành phần protected của lớp dẫn xuất, không truy nhập được từ bên ngoài.
- Với dẫn xuất public, đối tượng của lớp dẫn xuất có thể truy nhập được các thành phần public của lớp cơ sở vì chúng trở thành các thành phần public của lớp dẫn xuất, có thể truy nhập được từ bên ngoài.

Bảng 6.3 tổng kết phạm vi truy nhập từ hàm bạn và đối tượng của lớp dẫn xuất vào các thành phần của lớp cơ sở, được quy định bởi các từ khóa dẫn xuất.

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất	Truy nhập từ hàm bạn của lớp dẫn xuất	Truy nhập từ đối tượng của lớp dẫn xuất
private	private protected public	--- private private	--- ok ok	--- --- ---
protected	private protected public	--- protected protected	--- ok ok	--- --- ---
public	private protected public	--- protected public	--- ok ok	--- --- ok

6.3.2 Sử dụng các thành phần của lớp cơ sở từ lớp dẫn xuất

Từ bảng tổng kết phạm vi truy nhập, ta thấy rằng chỉ có dẫn xuất theo kiểu public thì đối tượng của lớp dẫn xuất mới có thể truy nhập đến các thành phần (thuộc loại public) của lớp cơ sở. Khi đó, việc gọi đến các thành phần của lớp cơ sở cũng tương tự như gọi các thành phần lớp thông thường:

- Đối với biến đối tượng thông thường:
`<Tên đối tượng>.<Tên thành phần>([Các đối số]);`
- Đối với con trỏ đối tượng:
`<Tên đối tượng>-><Tên thành phần>([Các đối số]);`

Lưu ý:

- Cách gọi hàm thành phần này được áp dụng khi trong lớp dẫn xuất, ta không định nghĩa lại các hàm thành phần của lớp cơ sở. Trường hợp định nghĩa lại hàm thành phần của lớp cơ sở sẽ được trình bày trong mục 6.3.3.

Chương trình 6.3 minh họa việc sử dụng các thành phần lớp cơ sở từ đối tượng lớp dẫn xuất: lớp Bus kế thừa từ lớp Car. Lớp Bus có định nghĩa bổ sung một số phương thức và thuộc tính mới. Khi đó, đối tượng của lớp Bus có thể gọi các hàm public của lớp Bus cũng như của lớp Car.

Chương trình 6.3

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;                // Tốc độ
```

```

        char mark[20];                // Nhãn hiệu
        float price;                  // Giá xe

    public:
        void setSpeed(int);           // Gán tốc độ cho xe
        int getSpeed();               // Đọc tốc độ xe
        void setMark(char);           // Gán nhãn cho xe
        char[] getMark();             // Đọc nhãn xe
        void setPrice(float);         // Gán giá cho xe
        float getPrice();             // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void show();                  // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::setSpeed(int speedIn){      // Gán tốc độ cho xe
    speed = speedIn;
}

int Car::getSpeed(){                 // Đọc tốc độ xe
    return speed;
}

void Car::setMark(char markIn){       // Gán nhãn cho xe
    strcpy(mark, markIn);
}

char[] Car::getMark(){               // Đọc nhãn xe
    return mark;
}

void Car::setPrice(float priceIn){    // Gán giá cho xe
    price = priceIn;
}

float Car::getPrice(){               // Đọc giá xe
    return price;
}

void Car::show(){                    // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
}

```

```

        return;
    }

    /* Định nghĩa lớp Bus kế thừa từ lớp Car */
    class Bus: public Car{
        int label;                // Số hiệu tuyến xe

    public:
        // Khởi tạo đủ tham số
        Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
        void setLabel(int);       // Gán số hiệu tuyến xe
        int getLabel();           // Đọc số hiệu tuyến xe
    };

    // Khởi tạo đủ tham số
    Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
        label = lIn;
    }

    void Bus::setLabel(int labelIn){ // Gán số hiệu tuyến xe
        label = labelIn;
    }

    int Bus::getLabel(){             // Đọc số hiệu tuyến xe
        return label;
    }

    // Chương trình chính
    void main(){
        clrscr();
        Bus myBus;                   // Biến đối tượng của lớp Bus
        int speedIn, labelIn;
        float priceIn;
        char markIn[20];

        // Nhập giá trị cho các thuộc tính
        cout << "Tốc độ xe bus:";
        cin >> speedIn;
        cout << "Nhập hiệu xe bus:";
        cin >> markIn;
        cout << "Giá xe bus:";
        cin >> priceIn;
        cout << "Số hiệu tuyến xe bus:";
        cin >> labelIn;

        myBus.setSpeed(speedIn);     // Phương thức của lớp Car
    }

```

```

myBus.setMark(markIn);          // Phương thức của lớp Car
myBus.setPrice(priceIn);        // Phương thức của lớp Car
myBus.setLabel(labelIn);        // Phương thức của lớp Bus
myBus.show();                    // Phương thức của lớp Car
return;
}

```

Trong chương trình 6.3, đối tượng myBus có kiểu lớp Bus, là lớp dẫn xuất của lớp cơ sở Car, có thể sử dụng các phương thức của lớp Car và lớp Bus một cách bình đẳng. Khi đó, lệnh myBus.show() sẽ gọi đến phương thức show() của lớp Car, do vậy, chương trình trên sẽ in ra màn hình kết quả như sau (tùy theo dữ liệu nhập vào ở 4 dòng đầu):

```

Toc do xe bus: 80
Nhan hieu xe bus: Mercedes
Gia xe bus: 5000
So hieu tuyen xe bus: 27
This is a Mercedes having a speed of 80km/h and its price is $5000

```

Trong dòng giới thiệu xe bus (vì ta đang dùng đối tượng myBus của lớp Bus), không có giới thiệu số hiệu tuyến xe. Lí do là vì ta đang dùng hàm show của lớp Car. Muốn có thêm phần giới thiệu về số hiệu tuyến xe buýt, ta phải định nghĩa lại hàm show trong lớp Bus. Mục 6.3.3 sẽ trình bày nội dung này.

6.3.3 Định nghĩa chồng các phương thức của lớp cơ sở

Định nghĩa chồng phương thức của lớp cơ sở

Một phương thức của lớp cơ sở bị coi là nạp chồng nếu ở lớp dẫn xuất cũng định nghĩa một phương thức có cùng tên.

Ví dụ, trong lớp Car, đã có phương thức show(), bây giờ, trong lớp Bus kế thừa từ lớp Car, ta cũng định nghĩa lại phương thức show():

```

class Car{
    ...
    public:
    ...
    void show();          // Phương thức của lớp cơ sở
};

class Bus: public Car{
    ...
    public:
    ...
    void show();          // Phương thức nạp chồng
};

```

khi đó, phương thức show() của lớp Bus được coi là phương thức nạp chồng từ phương thức show() của lớp Car.

Sử dụng các phương thức nạp chồng

Từ một đối tượng của lớp dẫn xuất, việc truy nhập đến phương thức đã được định nghĩa lại trong lớp dẫn xuất được thực hiện như lời gọi một phương thức thông thường:

- Đối với biến đối tượng thông thường:
`<Tên đối tượng>.<Tên thành phần>([Các đối số]);`
- Đối với con trỏ đối tượng:
`<Tên đối tượng>-><Tên thành phần>([Các đối số]);`

Ví dụ:

```
Bus myBus;  
myBus.show();
```

sẽ gọi đến phương thức `show()` được định nghĩa trong lớp `Bus`.

Trong trường hợp, từ một đối tượng của lớp dẫn xuất, muốn truy nhập đến một phương thức của lớp cơ sở (đã bị định nghĩa lại ở lớp dẫn xuất) thì phải sử dụng chỉ thị phạm vi lớp trước phương thức được gọi:

- Đối với biến đối tượng thông thường:
`<Tên đối tượng>.<Tên lớp cơ sở>::<Tên thành phần>([Các đối số]);`
- Đối với con trỏ đối tượng:
`<Tên đối tượng>-><Tên lớp cơ sở>::<Tên thành phần>([Các đối số]);`

Ví dụ:

```
Bus myBus;  
myBus.Car::show();
```

sẽ gọi đến phương thức `show()` được định nghĩa trong lớp `Car` từ một đối tượng của lớp `Bus`.

Chương trình 6.4 minh họa việc định nghĩa chồng hàm `show()` trong lớp `Bus` và việc sử dụng hai phương thức `show()` của hai lớp từ một đối tượng của lớp dẫn xuất.

Chương trình 6.4

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp Car */  
class Car{  
    private:  
        int    speed;           // Tốc độ  
        char   mark[20];        // Nhãn hiệu  
        float  price;           // Giá xe  
    public:  
        int    getSpeed();       // Đọc tốc độ xe  
        char[] getMark();        // Đọc nhãn xe  
        float  getPrice();       // Đọc giá xe  
        // Khởi tạo thông tin về xe
```

```

        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void show(); // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::setSpeed(int speedIn){ // Gán tốc độ cho xe
    speed = speedIn;
}

int Car::getSpeed(){ // Đọc tốc độ xe
    return speed;
}

char[] Car::getMark(){ // Đọc nhãn xe
    return mark;
}

float Car::getPrice(){ // Đọc giá xe
    return price;
}

void Car::show(){ // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label; // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void show(); // Giới thiệu xe bus
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

```

```
// Định nghĩa nạp chồng phương thức
void Bus::show() { // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl;
    return;
}

// Chương trình chính
void main() {
    clrscr();
    Bus myBus(80, "Mercedes", 5000, 27); // Biến đối tượng của lớp Bus

    cout << "Gioi thieu xe:" << endl;
    myBus.Car::show(); // Phương thức của lớp Car
    cout << "Gioi thieu xe bus:" << endl;
    myBus.show(); // Phương thức của lớp Bus
    return;
}
```

Chương trình 6.4 sẽ hiển thị các thông báo như sau:

```
Gioi thieu xe:
This is a Mercedes having a speed of 80km/h and its price is $5000
Gioi thieu xe bus:
This is a bus of type Mercedes, on the line 27, having a speed of
80km/h and its price is $5000
```

Lưu ý:

- Trong phương thức show() của lớp Bus, ta phải dùng các hàm get để truy nhập đến các thuộc tính của lớp Car. Không được truy nhập trực tiếp đến tên các thuộc tính (speed, mark và price) vì chúng có dạng private của lớp Car.

6.3.4 Chuyển đổi kiểu giữa lớp cơ sở và lớp dẫn xuất

Về mặt dữ liệu, một lớp dẫn xuất bao giờ cũng chứa toàn bộ dữ liệu của lớp cơ sở. Ta luôn tìm thấy lớp cơ sở trong lớp dẫn xuất, nhưng không phải bao giờ cũng tìm thấy lớp dẫn xuất trong lớp cơ sở. Do vậy:

- Có thể gán một đối tượng lớp dẫn xuất cho một đối tượng lớp cơ sở:
`<Đối tượng lớp cơ sở> = <Đối tượng lớp dẫn xuất>; // Đúng`
- Nhưng không thể gán một đối tượng lớp cơ sở cho một đối tượng lớp dẫn xuất:
`<Đối tượng lớp dẫn xuất> = <Đối tượng lớp cơ sở>; // Không được`

Ví dụ, ta có lớp Bus kế thừa từ lớp Car và:

```
Bus myBus;
```



```
Car myCar;
```

khi đó, phép gán:

```
myCar = myBus;    // đúng
```

thì chấp nhận được, nhưng phép gán:

```
myBus = myCar;    // không được
```

thì không chấp nhận được.

Lưu ý:

- Nguyên tắc chuyển kiểu này cũng đúng với các phép gán con trỏ: một con trỏ đối tượng lớp cơ sở có thể trỏ đến địa chỉ của một đối tượng lớp dẫn xuất. Nhưng một con trỏ đối tượng lớp dẫn xuất không thể trỏ đến địa chỉ một đối tượng lớp cơ sở.
- Nguyên tắc chuyển kiểu này cũng đúng với truyền đối số cho hàm: có thể truyền một đối tượng lớp dẫn xuất vào vị trí của tham số có kiểu lớp cơ sở. Nhưng không thể truyền một đối tượng lớp cơ sở vào vị trí một tham số có kiểu lớp dẫn xuất.

Chương trình 6.5 minh họa việc chuyển kiểu giữa các đối tượng của lớp cơ sở và lớp dẫn xuất.

Chương trình 6.5

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        int    getSpeed();       // Đọc tốc độ xe
        char[] getMark();        // Đọc nhãn xe
        float  getPrice();       // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void   show();           // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```



```

int Car::getSpeed(){                                // Đọc tốc độ xe
    return speed;
}
char[] Car::getMark(){                              // Đọc nhãn xe
    return mark;
}
float Car::getPrice(){                              // Đọc giá xe
    return price;
}

void Car::show(){                                    // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label;                                       // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void show();                                    // Định nghĩa chồng phương thức
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

// Định nghĩa nạp chồng phương thức
void Bus::show(){                                  // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl;
    return;
}

// Chương trình chính
void main(){
    clrscr();
    Car myCar(100, "Ford", 3000); // Biến đối tượng lớp Car
    Bus myBus(80, "Mercedes", 5000, 27); // Biến đối tượng lớp Bus
}

```

```

        cout << "Gioi thieu xe o to lan 1:" << endl;
        myCar.show();
        cout << "Gioi thieu xe o to lan 2:" << endl;
        myCar = myBus;
        myCar.show();
        cout << "Gioi thieu xe bus:" << endl;
        myBus.show();
        return;
    }

```

Chương trình 6.5 sẽ in ra kết quả thông báo như sau:

Goi thieu xe o to lan 1:

This is a Ford having a speed of 100km/h and its price is \$3000

Goi thieu xe lan o to lan 2:

This is a Mercedes having a speed of 80km/h and its price is \$5000

Goi thieu xe bus:

This is a bus of type Mercedes, on the line 27, having a speed of 80km/h and its price is \$5000

Ở thông báo thứ nhất, đối tượng myCar gọi phương thức show() của lớp Car với các dữ liệu được khởi đầu cho myCar: (100, "Ford", 3000). Ở thông báo thứ hai, myCar gọi phương thức show() của lớp Car, nhưng với dữ liệu vừa được gán từ đối tượng myBus: (80, "Mercedes", 5000). Ở thông báo thứ ba, myBus gọi phương thức show() của lớp Bus với các dữ liệu của myBus: (80, "Mercedes", 5000, 27).

6.4 ĐA KẾ THỪA

C++ cho phép đa kế thừa, tức là một lớp có thể được dẫn xuất từ nhiều lớp cơ sở khác nhau, với những kiểu dẫn xuất khác nhau.

6.4.1 Khai báo đa kế thừa

Đa kế thừa được khai báo theo cú pháp:

```

class <Tên lớp dẫn xuất>: <Từ khoá dẫn xuất> <Tên lớp cơ sở 1>,
    <Từ khoá dẫn xuất> <Tên lớp cơ sở 2>,
    ...
    <Từ khoá dẫn xuất> <Tên lớp cơ sở n>{
    ...    // Khai báo thêm các thành phần lớp dẫn xuất
};

```

Ví dụ:

```

class Bus: public Car, public PublicTransport{
    ...    // Khai báo các thành phần bổ sung
};

```

là khai báo lớp Bus (xe buýt) kế thừa từ hai lớp xe Car (ô tô) và PublicTransport (phương tiện giao thông công cộng) theo cùng một kiểu dẫn xuất là public.

Lưu ý:

- Trong đa kế thừa, mỗi lớp cơ sở được phân cách nhau bởi dấu phẩy “,”.
- Mỗi lớp cơ sở cơ thể có một kiểu dẫn xuất bởi một từ khoá dẫn xuất khác nhau.
- Nguyên tắc truy nhập vào các thành phần lớp cơ sở cũng hoàn toàn tương tự như trong kế thừa đơn.

6.4.2 Hàm khởi tạo và hàm huỷ bỏ trong đa kế thừa

Hàm khởi tạo trong đa kế thừa

Hàm khởi tạo trong đa kế thừa được khai báo tương tự như trong đơn kế thừa, ngoại trừ việc phải sắp xếp thứ tự gọi tới hàm khởi tạo của các lớp cơ sở: thông thường, thứ tự gọi đến hàm khởi tạo của các lớp cơ sở nên tuân theo thứ tự dẫn xuất từ các lớp cơ sở trong đa kế thừa.

Chương trình 6.6 minh hoạ việc định nghĩa hàm khởi tạo tường minh trong đa kế thừa: thứ tự gọi hàm khởi tạo của các lớp cơ sở trong hàm khởi tạo của lớp Bus là tương tự thứ tự dẫn xuất: hàm khởi tạo của lớp Car trước hàm khởi tạo của lớp PublicTransport.

Chương trình 6.6

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    int    speed;           // Tốc độ
    char   mark[20];        // Nhãn hiệu
    float  price;           // Giá xe
public:
    Car();                 // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đủ tham số
};

Car::Car() {               // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```

```

}

/* Định nghĩa lớp PublicTransport */
class PublicTransport{
    float ticket;                // Giá vé phương tiện

public:
    PublicTransport();           // Khởi tạo không tham số
    PublicTransport(float);      // Khởi tạo đủ tham số
};

PublicTransport::PublicTransport(){ // Khởi tạo không tham số
    ticket = 0;
}

// Khởi tạo đủ tham số
PublicTransport::PublicTransport(float ticketIn){
    ticket = ticketIn;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */
class Bus: public Car, public PublicTransport{ // Thứ tự khai báo
    int label;                                // Số hiệu tuyến xe

public:
    Bus();                                    // Khởi tạo không tham số
    Bus(int, char[], float, float, int);     // Khởi tạo đủ tham số
};

// Khởi tạo không tham số
Bus::Bus(): Car(), Transport(){              // Theo thứ tự dẫn xuất
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, float tIn, int lIn):
    Car(sIn, mIn, pIn), PublicTransport(tIn){ // Theo thứ tự dẫn xuất
    label = lIn;
}

```

Lưu ý:

- Trong trường hợp dùng hàm khởi tạo ngầm định hoặc không có tham số, ta có thể không cần gọi tường minh các hàm khởi tạo của các lớp cơ sở, trình biên dịch sẽ tự động gọi tới chúng theo đúng thứ tự dẫn xuất

Ví dụ, trong chương trình 6.6, hai cách định nghĩa hàm khởi tạo không tham số của lớp Bus sau là tương đương:

```
Bus::Bus(): Car(), Transport() { // Theo thứ tự dẫn xuất
    label = 0;
}
```

là tương đương với:

```
Bus::Bus() { // Theo thứ tự dẫn xuất ngầm định
    label = 0;
}
```

Hàm hủy bỏ trong đa kế thừa

Vì hàm hủy bỏ là duy nhất của mỗi lớp, hơn nữa hàm hủy bỏ của lớp cơ sở sẽ được tự động gọi đến khi giải phóng đối tượng của lớp dẫn xuất. Cho nên hàm hủy bỏ trong đa kế thừa hoàn toàn tương tự hàm hủy bỏ trong đơn kế thừa:

- Hàm hủy bỏ của lớp dẫn xuất chỉ giải phóng bộ nhớ cho các thành phần bổ sung, nếu có, của lớp dẫn xuất.
- Hàm hủy bỏ của lớp dẫn xuất sẽ được gọi đến sớm nhất. Sau đó các hàm hủy bỏ của các lớp cơ sở sẽ được gọi đến.
- Quá trình này được trình biên dịch thực hiện tự động.

6.4.3 Truy nhập các thành phần lớp trong đa kế thừa

Việc truy nhập đến các thành phần của các lớp trong đa kế thừa được dựa trên các nguyên tắc sau:

- Việc truy nhập từ đối tượng lớp dẫn xuất đến các thành phần của mỗi lớp cơ sở được tuân theo quy tắc phạm vi tương tự như trong đơn kế thừa.
- Trong trường hợp các lớp cơ sở đều có các thành phần cùng tên, việc truy xuất đến thành phần của lớp nào phải được chỉ rõ bằng toán tử phạm vi: “<Tên lớp>::” đối với thành phần lớp cơ sở đó.

Ví dụ, ta định nghĩa lớp Bus kế thừa từ hai lớp cơ sở: Car và PublicTransport. Nhưng cả ba lớp này đều định nghĩa một phương thức show() để tự giới thiệu:

```
class Car{
public:
    void show();
};
class PublicTransport{
public:
    void show();
};
class Bus: public Car, public PublicTransport{
public:
    void show();
};
```

Khi đó, khai báo:

```
Bus myBus;
```

và lời gọi hàm:

```
myBus.show(); // Gọi đến hàm của lớp Bus
myBus.Car::show(); // Gọi đến hàm của lớp Car
myBus.PublicTransport::show(); // Gọi đến hàm của lớp PublicTransport
```

Chương trình 6.7 minh họa việc truy nhập đến các thành phần trùng nhau trong các lớp cơ sở và được định nghĩa lại trong lớp dẫn xuất.

Chương trình 6.7

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

/* Định nghĩa lớp Car */
class Car{
    int    speed;           // Tốc độ
    char   mark[20];        // Nhãn hiệu
    float  price;           // Giá xe
public:
    Car();                 // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đủ tham số
    void show();           // Giới thiệu
    float getSpeed(){return speed;};
    char[] getMark(){return mark;};
    float getPrice(){return price;};
};

Car::Car() {               // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```

```
// Giới thiệu
void Car::show(){
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

/* Định nghĩa lớp PublicTransport */
class PublicTransport{
    float ticket;                // Giá vé phương tiện
public:
    PublicTransport();           // Khởi tạo không tham số
    PublicTransport(float);      // Khởi tạo đủ tham số
    void show();                // Giới thiệu
    float getTicket(){return ticket;};
};

PublicTransport::PublicTransport(){ // Khởi tạo không tham số
    ticket = 0;
}

// Khởi tạo đủ tham số
PublicTransport::PublicTransport(float ticketIn){
    ticket = ticketIn;
}

// Giới thiệu
void PublicTransport::show(){
    cout << "This public transport had a ticket of $"
        << ticket << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */
class Bus: public Car, public PublicTransport{ // Thứ tự khai báo
    int label;                                // Số hiệu tuyến xe
public:
    Bus();                                    // Khởi tạo không tham số
    Bus(int, char[], float, float, int);    // Khởi tạo đủ tham số
    void show();                             // Giới thiệu
};

// Khởi tạo không tham số
```



```

Bus::Bus(): Car(), Transport() { // Theo thứ tự dẫn xuất
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, float tIn, int lIn):
    Car(sIn, mIn, pIn), PublicTransport(tIn) { // Theo thứ tự dẫn xuất
    label = lIn;
}

// Giới thiệu
void Bus::show() {
    cout << "This is a bus on the line " << label
        << ", its speed is " << getSpeed()
        << "km/h, mark is " << getMark()
        << ", price is $" << getPrice()
        << " and ticket is " << getTicket() << endl;
    return;
}

// phương thức main
void main() {
    clrscr();
    Bus myBus(100, "Mercedes", 3000, 1.5, 27);

    myBus.Car::show(); // Hàm của lớp Car
    myBus.PublicTransport::show(); // Hàm của lớp PublicTransport
    myBus.show(); // Hàm của lớp Bus
    return;
}

```

Chương trình 6.7 sẽ in ra thông báo như sau:

```

This is a Mercedes having a speed of 100km/h and its price is $3000
This public transport had a ticket of $1.5
This is a bus on the line 27, its speed is 100km/h, mark is Mercedes,
price is $3000 and ticket is $1.5

```

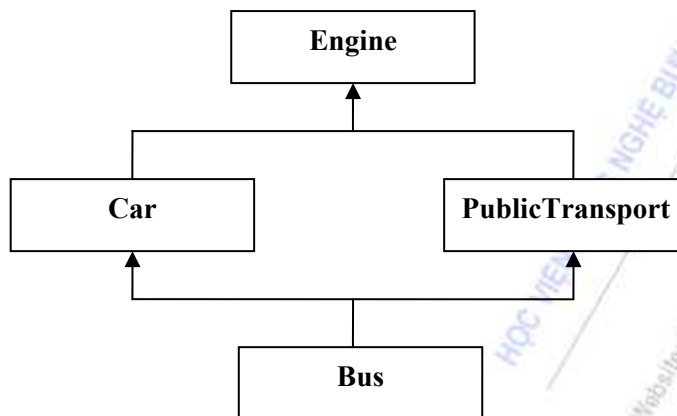
Dòng thứ nhất là kết quả của phương thức show() của lớp Car, dòng thứ hai, tương ứng là kết quả phương thức show() của lớp PublicTransport, dòng thứ ba là kết quả phương thức show() của lớp Bus.

6.5 LỚP CƠ SỞ TRỪU TƯỢNG

6.5.1 Đặt vấn đề

Sự cho phép đa kế thừa trong C++ dẫn đến một số hậu quả xấu, đó là sự đụng độ giữa các thành phần của các lớp cơ sở, khi có ít nhất hai lớp cơ sở lại cùng được kế thừa từ một lớp cơ sở khác. Xét trường hợp:

- Lớp Bus kế thừa từ lớp Car và lớp PublicTransport.
- Nhưng lớp Car và lớp PublicTransport lại cùng được thừa kế từ lớp Engine (động cơ).
Lớp Engine có một thuộc tính là power (công suất của động cơ).



Khi đó, nảy sinh một số vấn đề như sau:

- Các thành phần dữ liệu của lớp Engine bị lặp lại trong lớp Bus hai lần: một lần do kế thừa theo đường Bus::Car::Engine, một lần theo đường Bus::PublicTransport::Engine. Điều này là không an toàn.
- Khi khai báo một đối tượng của lớp Bus, hàm khởi tạo của lớp Engine cũng được gọi hai lần: một lần do gọi truy hồi từ hàm khởi tạo lớp Car, một lần do gọi truy hồi từ hàm khởi tạo lớp PublicTransport.
- Khi giải phóng một đối tượng của lớp Bus, hàm huỷ bỏ của lớp Engine cũng sẽ bị gọi tới hai lần.

Để tránh các vấn đề này, C++ cung cấp một khái niệm là kế thừa từ *lớp cơ sở trừu tượng*. Khi đó, ta cho các lớp Car và PublicTransport kế thừa trừu tượng từ lớp Engine. Bằng cách này, các thành phần của lớp Engine chỉ xuất hiện trong lớp Bus đúng một lần. Lớp Engine được gọi là lớp cơ sở trừu tượng của các lớp Car và PublicTransport.

6.5.2 Khai báo lớp cơ sở trừu tượng

Việc chỉ ra một sự kế thừa trừu tượng được thực hiện bằng từ khoá **virtual** khi khai báo lớp cơ sở:

```

class <Tên lớp cơ sở>: <Từ khoá dẫn xuất> virtual <Tên lớp cơ sở>{
    ...          // Khai báo các thành phần bổ sung
};
    
```

Ví dụ:

```

class Engine{
    ...          // Các thành phần lớp Engine
};
    
```

```
};
class Car: public virtual Engine{
    ...    // Khai báo các thành phần bổ sung
};
```

là khai báo lớp Car, kế thừa từ lớp cơ sở trừu tượng Engine, theo kiểu dẫn xuất public.

Lưu ý:

- Từ khoá virtual được viết bằng chữ thường.
- Từ khoá virtual không ảnh hưởng đến phạm vi truy nhập thành phần lớp cơ sở, phạm vi này vẫn được quy định bởi từ khoá dẫn xuất như thông thường.
- Từ khoá virtual chỉ ra một lớp cơ sở là trừu tượng nhưng lại được viết trong khi khai báo lớp dẫn xuất.
- Một lớp dẫn xuất có thể được kế thừa từ nhiều lớp cơ sở trừu tượng

6.5.3 Hàm khởi tạo lớp cơ sở trừu tượng

Khác với các lớp cơ sở thông thường, khi có một lớp dẫn xuất từ một lớp cơ sở trừu tượng, lại được lấy làm cơ sở cho một lớp dẫn xuất khác thì trong hàm khởi tạo của lớp dẫn xuất cuối cùng, vẫn phải gọi hàm khởi tạo tường minh của lớp cơ sở trừu tượng. Hơn nữa, hàm khởi tạo của lớp cơ sở trừu tượng phải được gọi sớm nhất.

Ví dụ, khi lớp Car và lớp PublicTransport được kế thừa từ lớp cơ sở trừu tượng Engine. Sau đó, lớp Bus được kế thừa từ hai lớp Car và PublicTransport. Khi đó, hàm khởi tạo của lớp Bus cũng phải gọi tường minh hàm khởi tạo của lớp Engine, theo thứ tự sớm nhất, sau đó mới gọi đến hàm khởi tạo của các lớp Car và PublicTransport.

```
class Engine{
    public:
        Engine() {... };
};
class Car: public virtual Engine{                //Lớp cơ sở virtual
    public:
        Car(): Engine() {... };
};
class PublicTransport: public virtual Engine{    //Lớp cơ sở virtual
    public:
        PublicTransport():Engine() {... };
};
class Bus: public Car, public PublicTransport{
    public:
        // Gọi hàm khởi tạo tường minh của lớp cơ sở trừu tượng
        Bus():Engine(), Car(), PublicTransport() {... };
};
```

Lưu ý:

- Trong trường hợp lớp Engine không phải là lớp cơ sở trừu tượng của các lớp Car và PublicTransport, thì trong hàm khởi tạo của lớp Bus không cần gọi hàm khởi tạo của lớp

Engine, mà chỉ cần gọi tới các hàm khởi tạo của các lớp cơ sở trực tiếp của lớp Bus là lớp Car và lớp PublicTransport.

Chương trình 6.8 minh họa việc khai báo và sử dụng lớp cơ sở trừu tượng: lớp Engine là lớp cơ sở trừu tượng của các lớp Car và lớp PublicTransport. Hai lớp này, sau đó, lại làm lớp cơ sở của lớp Bus.

Chương trình 6.8

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

/* Định nghĩa lớp Engine */
class Engine{
    int    power;                // Công suất
public:
    Engine(){power = 0;};        // Khởi tạo không tham số
    Engine(int pIn){power = pIn;}; // Khởi tạo đủ tham số
    void show();                // Giới thiệu
    float getPower(){return power;};
};

// Giới thiệu
void Engine::show(){
    cout << "This is an engine having a power of "
        << power << "KWH" << endl;
    return;
}

/* Định nghĩa lớp Car dẫn xuất từ lớp cơ sở trừu tượng Engine*/
class Car: public virtual Engine{
    int    speed;                // Tốc độ
    char    mark[20];            // Nhãn hiệu
    float price;                // Giá xe
public:
    Car();                      // Khởi tạo không tham số
    Car(int, int, char[], float); // Khởi tạo đủ tham số
    void show();                // Giới thiệu
    float getSpeed(){return speed;};
    char[] getMark(){return mark;};
    float getPrice(){return price;};
};
```

```

Car::Car(): Engine() {                                     // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int pwIn, int sIn, char mIn[], float prIn): Engine(pwIn){
    speed = sIn;
    strcpy(mark, mIn);
    price = prIn;
}

// Giới thiệu
void Car::show() {
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h, its power is" << getPower()
        << "KWh and price is $" << price << endl;
    return;
}

/* Định nghĩa lớp PublicTransport dẫn xuất trừu tượng từ lớp Engine */
class PublicTransport: public virtual Engine{
    float ticket;                                           // Giá vé phương tiện
public:
    PublicTransport();                                     // Khởi tạo không tham số
    PublicTransport(int, float);                          // Khởi tạo đủ tham số
    void show();                                           // Giới thiệu
    float getTicket(){return ticket;};
};

// Khởi tạo không tham số
PublicTransport::PublicTransport(): Engine(){
    ticket = 0;
}

// Khởi tạo đủ tham số
PublicTransport::PublicTransport(int pwIn, float tIn): Engine(pwIn){
    ticket = tIn;
}

// Giới thiệu
void PublicTransport::show() {

```

```

        cout << "This is a public transport havin a ticket of $"
            << ticket << " and its power is " << getPower()
            << "KWh" << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */
class Bus: public Car, public PublicTransport{ // Thứ tự khai báo
    int label; // Số hiệu tuyến xe
public:
    Bus(); // Khởi tạo không tham số
    Bus(int,int,char[],float,float,int); // Khởi tạo đủ tham số
    void show(); // Giới thiệu
};

// Khởi tạo không tham số
Bus::Bus(): Engine(), Car(), Transport(){ // Theo thứ tự dẫn xuất
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int pwIn, int sIn, char mIn[], float prIn, float tIn, int lIn):
    Engine(pwIn), Car(sIn, mIn, prIn), PublicTransport(tIn){
    label = lIn;
}

// Giới thiệu
void Bus::show(){
    cout << "This is a bus on the line " << label
        << ", its speed is " << getSpeed()
        << "km/h, power is" << Car::getPower()
        << "KWh, mark is " << getMark()
        << ", price is $" << getPrice()
        << " and ticket is " << getTicket() << endl;
    return;
}

// phương thức main
void main(){
    clrscr();
    Bus myBus(250, 100, "Mercedes", 3000, 1.5, 27);

    myBus.Car::Engine::show(); // Hàm của lớp Engine
}

```

```
myBus.PublicTransport::Engine::show();// Hàm của lớp Engine
myBus.Car::show();                      // Hàm của lớp Car
myBus.PublicTransport:: show();         // Hàm của lớp PublicTransport
myBus.show();                           // Hàm của lớp Bus
return;
}
```

Chương trình 6.8 sẽ in ra thông báo như sau:

```
This is an engine having a power of 250KWh
This is an engine having a power of 250KWh
This is a Mercedes having a speed of 100km/h, its power is 250KWh and
price is $3000
This is a public transport having a ticket of $1.5 and its power is
250KWh
This is a bus on the line 27, its speed is 100km/h, power is 250KWh,
mark is Mercedes, price is $3000 and ticket is $1.5
```

Hai dòng đầu là kết quả của phương thức show() của lớp Engine: một lần gọi qua lớp Car, một lần gọi qua lớp PublicTransport, chúng cho kết quả như nhau. Dòng thứ ba là kết quả phương thức show() của lớp Car. Dòng thứ tư, tương ứng là kết quả phương thức show() của lớp PublicTransport. Dòng thứ năm là kết quả phương thức show() của lớp Bus.

6.6 ĐA HÌNH

6.6.1 Đặt vấn đề

Sự kế thừa trong C++ cho phép có sự tương ứng giữa lớp cơ sở và các lớp dẫn xuất trong sơ đồ thừa kế:

- Một con trỏ có kiểu lớp cơ sở luôn có thể trỏ đến địa chỉ của một đối tượng của lớp dẫn xuất.
- Tuy nhiên, khi thực hiện lời gọi một phương thức của lớp, trình biên dịch sẽ quan tâm đến kiểu của con trỏ chứ không phải đối tượng mà con trỏ đang trỏ tới: phương thức của lớp mà con trỏ có kiểu được gọi chứ không phải phương thức của đối tượng mà con trỏ đang trỏ tới được gọi.

Ví dụ, lớp Bus kế thừa từ lớp Car, cả hai lớp này đều định nghĩa phương thức show():

```
class Car{
public:
    void show();
};
class Bus: public Car{
public:
    void show();
};
```

khi đó, nếu ta khai báo một con trỏ lớp Bus, nhưng lại trỏ vào địa chỉ của một đối tượng lớp Car:


```
Bus myBus;  
Car *ptrCar = &myBus;    // đúng
```

nhưng khi gọi:

```
ptrCar->show();
```

thì chương trình sẽ gọi đến phương thức `show()` của lớp `Car` (là kiểu của con trỏ `ptrCar`), mà không gọi tới phương thức `show()` của lớp `Bus` (là kiểu của đối tượng `myBus` mà con trỏ `ptrCar` đang trỏ tới).

Để giải quyết vấn đề này, C++ đưa ra một khái niệm là phương thức trừu tượng. Bằng cách sử dụng phương thức trừu tượng. Khi gọi một phương thức từ một con trỏ đối tượng, trình biên dịch sẽ xác định kiểu của đối tượng mà con trỏ đang trỏ đến, sau đó nó sẽ gọi phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới.

6.6.2 Khai báo phương thức trừu tượng

Phương thức trừu tượng (còn gọi là phương thức ảo, hàm ảo) được khai báo với từ khoá **virtual**:

- Nếu khai báo trong phạm vi lớp:

```
virtual <Kiểu trả về> <Tên phương thức>([<Các tham số>]);
```

- Nếu định nghĩa ngoài phạm vi lớp:

```
virtual <Kiểu trả về> <Tên lớp>::<Tên phương thức>([<Các tham số>]) {...}
```

Ví dụ:

```
class Car{  
    public:  
        virtual void show();  
};
```

là khai báo phương thức trừu tượng `show()` của lớp `Car`: phương thức không có tham số và không cần giá trị trả về (`void`).

Lưu ý:

- Từ khoá `virtual` có thể đặt trước hay sau kiểu trả về của phương thức.
- Với cùng một phương thức được khai báo ở lớp cơ sở lẫn lớp dẫn xuất, chỉ cần dùng từ khoá `virtual` ở một trong hai lần định nghĩa phương thức đó là đủ: hoặc ở lớp cơ sở, hoặc ở lớp dẫn xuất.
- Trong trường hợp cây kế thừa có nhiều mức, cũng chỉ cần khai báo phương thức là trừu tượng (`virtual`) ở một mức bất kì. Khi đó, tất cả các phương thức trùng tên với phương thức đó ở tất cả các mức đều được coi là trừu tượng.
- Đôi khi không cần thiết phải định nghĩa chồng (trong lớp dẫn xuất) một phương thức đã được khai báo trừu tượng trong lớp cơ sở.

6.6.3 Sử dụng phương thức trừu tượng – đa hình

Một khi phương thức được khai báo là trừu tượng thì khi một con trỏ gọi đến phương thức đó, chương trình sẽ thực hiện phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới, thay vì

thực hiện phương thức của lớp cùng kiểu với con trỏ. Đây được gọi là hiện tượng đa hình (tương ứng bội) trong C++.

Chương trình 6.9 minh họa việc sử dụng phương thức trừu tượng: lớp Bus kế thừa từ lớp Car, hai lớp này cùng định nghĩa phương thức trừu tượng show().

- Khi ta dùng một con trỏ có kiểu lớp Car trỏ vào địa chỉ của một đối tượng kiểu Car, nó sẽ gọi phương thức show() của lớp Car.
- Khi ta dùng cùng con trỏ đó, trỏ vào địa chỉ của một đối tượng kiểu Bus, nó sẽ gọi phương thức show() của lớp Bus.

Chương trình 6.9

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        int    getSpeed(){return speed;}// Đọc tốc độ xe
        char[] getMark(){return mark;}// Đọc nhãn xe
        float  getPrice(){return price;}// Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        virtual void show();      // Giới thiệu xe, trừu tượng
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

// Phương thức trừu tượng giới thiệu xe
virtual void Car::show(){
    cout << "This is a " << mark << " having a speed of "
         << speed << "km/h and its price is $" << price << endl;
    return;
}
```



```
/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label;                // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void show();              // Giới thiệu xe
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

// Định nghĩa nạp chồng phương thức trừu tượng
void Bus::show() {           // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl;
    return;
}

// Chương trình chính
void main(){
    clrscr();
    Car *ptrCar, myCar(100, "Ford", 3000);
    Bus myBus(150, "Mercedes", 5000, 27); // Biến đổi tượng của lớp Bus

    ptrCar = &myCar;          // Trỏ đến đối tượng lớp Car
    ptrCar->show();            // Phương thức của lớp Car

    ptrCar = &myBus;          // Trỏ đến đối tượng lớp Bus
    ptrCar->show();            // Phương thức của lớp Bus
    return;
}
```

Chương trình 6.9 hiển thị kết quả thông báo như sau:

```
This is a Ford having a speed of 100km/h and its price is $3000
This is a bus of type Mercedes, on the line 27, having a speed of
150km/h and its price is $5000
```

Dòng thứ nhất là kết quả khi con trỏ ptrCar trỏ đến địa chỉ của đối tượng myCar, thuộc lớp Car nên sẽ gọi phương thức show() của lớp Car với các dữ liệu của đối tượng myCar: (100, Ford,

3000). Dòng thứ hai tương ứng là kết quả khi con trỏ ptrCar trỏ đến địa chỉ của đối tượng myBus, thuộc lớp Bus nên sẽ gọi phương thức show() của lớp Bus, cùng với các tham số của đối tượng myBus: (150, Mercedes, 5000, 27).

Lưu ý:

- Trong trường hợp ở lớp dẫn xuất không định nghĩa lại phương thức trừu tượng, thì chương trình sẽ gọi phương thức của lớp cơ sở, nhưng với dữ liệu của lớp dẫn xuất.

Ví dụ, nếu trong chương trình 6.9, lớp Bus không định nghĩa chồng phương thức trừu tượng show() thì kết quả hiển thị sẽ là hai dòng thông báo giống nhau, chỉ khác nhau ở dữ liệu của hai đối tượng khác nhau:

```
This is a Ford having a speed of 100km/h and its price is $3000
```

```
This is a Mercedes having a speed of 150km/h and its price is $5000
```

TỔNG KẾT CHƯƠNG 6

Nội dung chương 6 đã trình bày các vấn đề cơ bản liên quan đến thừa kế và tương ứng bội trong C++ như sau:

- Khai báo một lớp dẫn xuất kế thừa từ một lớp cơ sở bằng khai báo kế thừa “:” đi kèm với một từ khoá dẫn xuất.
- Có ba loại dẫn xuất khác nhau, được quy định bởi ba từ khoá dẫn xuất khác nhau: private, protected và public. Kiểu dẫn xuất phổ biến là dẫn xuất public.
- Sự kế thừa tạo ra mối quan hệ tương ứng giữa lớp cơ sở và lớp dẫn xuất: có thể chuyển kiểu ngầm định (trong phép gán, phép truyền đối số, phép trỏ địa chỉ) từ một đối tượng lớp cơ sở đến một đối tượng lớp dẫn xuất. Nhưng không thể chuyển kiểu ngược lại từ lớp dẫn xuất vào lớp cơ sở.
- Hàm khởi tạo của lớp dẫn xuất có thể gọi tường minh hoặc gọi ngầm định hàm khởi tạo của lớp cơ sở. Hàm khởi tạo lớp cơ sở bao giờ cũng được thực hiện trước hàm khởi tạo lớp dẫn xuất.
- Hàm huỷ bỏ của lớp dẫn xuất luôn gọi ngầm định hàm huỷ bỏ của lớp cơ sở. Trái với hàm khởi tạo, hàm huỷ bỏ lớp cơ sở luôn được thực hiện sau hàm huỷ bỏ của lớp dẫn xuất.
- Có thể truy nhập các phương thức của lớp cơ sở từ lớp dẫn xuất, phạm vi truy nhập là phụ thuộc vào kiểu dẫn xuất: private, protected hoặc public. Điều này cho phép sử dụng lại mã nguồn của lớp cơ sở, mà không cần định nghĩa lại ở lớp dẫn xuất.
- Trong lớp dẫn xuất, có thể định nghĩa chồng một số phương thức của lớp cơ sở. Khi có định nghĩa chồng, muốn truy nhập vào phương thức lớp cơ sở, phải dùng toán tử phạm vi lớp “<Tên lớp>::”.
- C++ còn cho phép một lớp có thể được dẫn xuất từ nhiều lớp cơ sở khác nhau, gọi là đa kế thừa. Trong đa kế thừa, quan hệ giữa lớp dẫn xuất với mỗi lớp cơ sở là tương tự như trong đơn kế thừa.
- Trong đa kế thừa, hàm khởi tạo lớp dẫn xuất sẽ gọi tường minh (hoặc ngầm định) hàm khởi tạo các lớp cơ sở, theo thứ tự khai báo kế thừa. Hàm huỷ bỏ lớp dẫn xuất lại gọi ngầm định các hàm huỷ bỏ của các lớp cơ sở.

- C++ cung cấp khái niệm kế thừa từ lớp cơ sở trừu tượng để tránh trường hợp trùng lặp dữ liệu ở lớp dẫn xuất, khi các lớp cơ sở lại cùng được dẫn xuất từ một lớp khác.
- C++ cũng cho phép cơ chế tương ứng bội (đa hình) bằng cách định nghĩa một phương thức là trừu tượng trong sơ đồ thừa kế. Khi đó, một con trỏ lớp cơ sở có thể trỏ đến địa chỉ của một đối tượng lớp dẫn xuất, và phương thức được thực hiện là tùy thuộc vào kiểu của đối tượng mà con trỏ đang trỏ tới.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 6

1. Trong các khai báo sau, khai báo nào là đúng cú pháp kế thừa lớp:
 - a. `class A: public class B{...};`
 - b. `class A: public B{...};`
 - c. `class A: class B{...};`
 - d. `class A:: public B{...};`
2. Trong các kiểu dẫn xuất sau, từ các phương thức lớp dẫn xuất, không thể truy nhập đến các thành phần private của lớp cơ sở:
 - a. `private`
 - b. `protected`
 - c. `public`
 - d. Cả ba kiểu trên
3. Trong các kiểu dẫn xuất sau, từ đối tượng của lớp dẫn xuất, có thể truy nhập đến các thành phần của lớp cơ sở:
 - a. `private`
 - b. `protected`
 - c. `public`
 - d. Cả ba kiểu trên
4. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử có các kiểu khai báo:
`A myA, *ptrA;`
`B myB, *ptrB;`
Khi đó, các lệnh nào sau đây là không có lỗi:
 - a. `myA = myB;`
 - b. `myB = myA;`
 - c. `ptrA = &myB;`
 - d. `ptrB = &myA;`
 - e. `ptrA = ptrB;`
 - f. `ptrB = ptrA;`
5. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử có các kiểu khai báo và nguyên mẫu hàm:
`A myA;`
`B myB;`
`void show(A, B);`

Khi đó, các lệnh gọi hàm nào sau đây là không có lỗi:

- a. `show(myA, myA);`
- b. `show(myA, myB);`
- c. `show(myB, myA);`
- d. `show(myB, myB);`

6. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có một hàm khởi tạo:

```
B(int, float);
```

Khi đó, định nghĩa hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

- a. `A::A() {...};`
- b. `A::A(): B() {...};`
- c. `A::A(int x, float y): B() {...};`
- d. `A::A(int x, float y): B(x, y) {...};`

7. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có hai hàm khởi tạo:

```
B();  
B(int, float);
```

Khi đó, những định nghĩa hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

- a. `A::A() {...};`
- b. `A::A(): B() {...};`
- c. `A::A(int x, float y): B() {...};`
- d. `A::A(int x, float y): B(x, y) {...};`

8. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có hàm hủy bỏ tường minh:

```
~B();
```

Khi đó, những định nghĩa hàm hủy bỏ nào sau đây của lớp A là chấp nhận được:

- a. `A::~~A() {...};`
- b. `A::~~A(): ~B() {...};`
- c. `A::~~A(int x) {...};`
- d. `A::~~A(int x): ~B() {...};`

9. Giả sử B là một lớp được khai báo:

```
class B{  
    int x;  
    public: int getx();  
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu private:

```
class A: private B{  
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. `myA.x;`
- b. `myA.getx();`
- c. Cả hai lệnh trên.

d. Không lệnh nào cả.

10. Giả sử B là một lớp được khai báo:

```
class B{
    int x;
    public: int getX();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu protected:

```
class A: protected B{
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. myA.x;
- b. myA.getX();
- c. Cả hai lệnh trên.
- d. Không lệnh nào cả.

11. Giả sử B là một lớp được khai báo:

```
class B{
    int x;
    public: int getX();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu public:

```
class A: public B{
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. myA.x;
- b. myA.getX();
- c. Cả hai lệnh trên.
- d. Không lệnh nào cả.

12. Giả sử B là một lớp được khai báo:

```
class B{
    public: void show();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu public, có định nghĩa chồng hàm show():

```
class A: public B{
    public: void show();
};
```

khi đó, nếu myA là một đối tượng lớp A, muốn thực hiện phương thức show() của lớp B thì lệnh nào sau đây là chấp nhận được:

- a. myA.show();
- b. myA.B::show();
- c. B::myA.show();

d. `A::B::show();`

13. Muốn khai báo một lớp A kế thừa từ hai lớp cơ sở B và C, những lệnh nào là đúng:

a. `class A: B, C{...};`

b. `class A: public B, C{...};`

c. `class A: public B, protected C{...};`

d. `class A: public B, public C{...};`

14. B là một lớp có hai hàm khởi tạo:

`B();`

`B(int);`

C cũng là một lớp có hai hàm khởi tạo:

`C();`

`C(int, int);`

Và A là một lớp kế thừa từ B và C:

`class A: public B, public C{...};`

Khi đó, hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

a. `A::A() {...};`

b. `A::A(): B(), C() {...};`

c. `A::A(int x, int y): C(x, y) {...};`

d. `A::A(int x, int y, int z): B(x), C(y, z) {...};`

e. `A::A(int x, int y, int z): C(x, y), B(z) {...};`

15. Muốn khai báo lớp A kế thừa từ lớp cơ sở trừu tượng B, những khai báo nào sau đây là đúng:

a. `virtual class A: public B{...};`

b. `class virtual A: public B{...};`

c. `class A: virtual public B{...};`

d. `class A: public virtual B{...};`

16. Lớp A là một lớp dẫn xuất, được kế thừa từ lớp cơ sở B. Hai lớp này đều định nghĩa hàm `show()`. Muốn hàm này trở thành trừu tượng thì những định nghĩa nào sau đây là đúng:

a. `void A::show() {...}` và `void B::show() {...}`

b. `virtual void A::show() {...}` và `void B::show() {...}`

c. `void A::show() {...}` và `virtual void B::show() {...}`

d. `virtual void A::show() {...}` và `virtual void B::show() {...}`

17. Khai báo lớp người (Human) bao gồm các thuộc tính sau:

- Tên người (name)
- Tuổi của người đó (age)
- Giới tính của người đó (sex)

Sau đó khai báo lớp Cá nhân (Person) kế thừa từ lớp Human vừa được định nghĩa ở trên.

18. Bổ sung các phương thức truy nhập các thuộc tính của lớp Human, các phương thức này có tính chất public.
19. Bổ sung thêm các thuộc tính của lớp Person: địa chỉ và số điện thoại. Thêm các phương thức truy nhập các thuộc tính này trong lớp Person.
20. Xây dựng hai hàm khởi tạo cho lớp Human: một hàm không tham số, một hàm với đủ ba tham số tương ứng với ba thuộc tính của nó. Sau đó, xây dựng hai hàm khởi tạo cho lớp Person có sử dụng các hàm khởi tạo của lớp Human: một hàm không tham số, một hàm đủ năm tham số (ứng với hai thuộc tính của lớp Person và ba thuộc tính của lớp Human).
21. Xây dựng một hàm main, trong đó có yêu cầu nhập các thuộc tính để tạo một đối tượng có kiểu Human và một đối tượng có kiểu Person, thông qua các hàm set thuộc tính đã xây dựng.
22. Xây dựng hàm show() cho hai lớp Human và Person. Thay đổi hàm main: dùng một đối tượng có kiểu lớp Person, gọi hàm show() của lớp Person, sau đó lại gọi hàm show() của lớp Human từ chính đối tượng đó.
23. Khai báo thêm một lớp người lao động (Worker), kế thừa từ lớp Human, có thêm thuộc tính là số giờ làm việc trong một tháng (hour) và tiền lương của người đó (salary). Sau đó, khai báo thêm một lớp nhân viên (Employee) kế thừa đồng thời từ hai lớp: Person và Worker. Lớp Employee có bổ sung thêm một thuộc tính là chức vụ (position).
24. Chuyển lớp Human thành lớp cơ sở trừu tượng của hai lớp Person và Worker. Xây dựng thêm hai hàm show() của lớp Worker và lớp Employee. Trong hàm main, khai báo một đối tượng lớp Employee, sau đó gọi đến các hàm show() của các lớp Employee, Person, Worker và Human.
25. Chuyển hàm show() trong các lớp trên thành phương thức trừu tượng. Trong hàm main, khai báo một con trỏ kiểu Human, sau đó, cho nó trỏ đến lần lượt các đối tượng của các lớp Human, Person, Worker và Employee, mỗi lần đều gọi phương thức show() để hiển thị thông báo ra màn hình.

CHƯƠNG 7

MỘT SỐ LỚP QUAN TRỌNG

Nội dung chương này tập trung trình bày một số lớp cơ bản trong C++:

- Lớp vật chứa (Container)
- Lớp tập hợp (Set)
- Lớp chuỗi kí tự (String)
- Lớp ngăn xếp (Stack) và hàng đợi (Queue)
- Lớp danh sách liên kết (Lists)

7.1 LỚP VẬT CHỨA

Lớp vật chứa (Container) bao gồm nhiều lớp cơ bản của C++: lớp Vector, lớp danh sách (List) và các kiểu hàng đợi (Stack và Queue), lớp tập hợp (Set) và lớp ánh xạ (Map). Trong chương này sẽ trình bày một số lớp cơ bản của Container là: Set, Stack, Queue và List

7.1.1 Giao diện của lớp Container

Các lớp cơ bản của Container có một số toán tử và phương thức có chức năng giống nhau, bao gồm:

- **==**: Toán tử so sánh bằng
- **<**: Toán tử so sánh nhỏ hơn
- **begin()**: Giá trị khởi đầu của con chạy iterator
- **end()**: Giá trị kết thúc của con chạy iterator
- **size()**: Số lượng phần tử đối tượng của vật chứa
- **empty()**: Vật chứa là rỗng
- **front()**: Phần tử thứ nhất của vật chứa
- **back()**: Phần tử cuối của vật chứa
- **[]**: Toán tử truy nhập đến phần tử của vật chứa
- **insert()**: Thêm vào vật chứa một (hoặc một số) phần tử
- **push_back()**: Thêm một phần tử vào cuối vật chứa
- **push_front()**: Thêm một phần tử vào đầu vật chứa
- **erase()**: Loại bỏ một (hoặc một số) phần tử khỏi vật chứa
- **pop_back()**: Loại bỏ phần tử cuối của vật chứa
- **pop_front()**: Loại bỏ phần tử đầu của vật chứa.

Ngoài ra, tùy vào các lớp cụ thể mà có một số toán tử và phương thức đặc trưng của lớp đó. Các toán tử và phương thức này sẽ được trình bày chi tiết trong nội dung từng lớp tiếp theo.

7.1.2 Con chạy Iterator

Iterator là một con trỏ trỏ đến các phần tử của vật chứa. Nó đóng vai trò là một con chạy cho phép người dùng di chuyển qua từng phần tử có mặt trong vật chứa. Mỗi phép tăng giảm một đơn vị của con chạy này tương ứng với một phép dịch đến phần tử tiếp theo hay phần tử trước của phần tử hiện tại mà con chạy đang trỏ tới.

Khai báo con chạy

Cú pháp chung để khai báo một biến con chạy iterator như sau:

```
Tên_lớp<T>::iterator Tên_con_chạy;
```

Trong đó:

- **Tên lớp:** là tên của lớp cơ bản ta đang dùng, ví dụ lớp Set, lớp List...
- **T:** là tên kiểu lớp của các phần tử chứa trong vật chứa. Kiểu có thể là các kiểu cơ bản trong C++, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.
- **Tên con chạy:** là tên biến sẽ được sử dụng làm biến chạy trong vật chứa.

Ví dụ:

```
Set<int>::iterator iter;
```

là khai báo một biến con chạy iter cho lớp tập hợp (Set), trong đó, các phần tử của lớp vật chứa có kiểu cơ bản int. Hoặc:

```
List<Person>::iterator iter;
```

là khai báo một biến con chạy iter cho lớp danh sách (List), trong đó, các phần tử có kiểu lớp do người dùng tự định nghĩa là Person.

Sử dụng con chạy

Con chạy được sử dụng khi cần duyệt lần lượt các phần tử có mặt trong vật chứa. Ví dụ sau sẽ in ra các phần tử có kiểu int của một tập hợp:

```
Set<int> mySet; // Khai báo một đối tượng của lớp Set,
               // các phần tử có kiểu int
Set<int>::iterator i; // Khai báo con chạy của lớp Set,
                   // các phần tử có kiểu int
... // Thêm phần tử vào
for(i=mySet.begin(); i<mySet.end(); i++)
    cout << mySet[i] << " ";
```

Lưu ý:

- Biến con chạy phải được khởi đầu bằng phương thức begin() và kết thúc bằng phương thức end() của đối tượng cần duyệt tương ứng.
- Một con chạy có thể được sử dụng nhiều lần cho nhiều đối tượng nếu chúng có cùng kiểu lớp vật chứa và các phần tử của chúng cũng cùng kiểu.

7.2 LỚP TẬP HỢP

Lớp tập hợp (Set) chứa các phần tử có cùng kiểu, không phân biệt thứ tự giữa các phần tử nhưng lại phân biệt giữa các phần tử: các phần tử là khác nhau từng đôi một. Muốn sử dụng lớp tập hợp, phải có chỉ thị đầu tệp:

```
#include<set.h>    // Thư viện riêng của lớp tập hợp
```

hoặc:

```
#include<stl.h>    // Thư viện chung cho các lớp vật chứa
```

7.2.1 Hàm khởi tạo

Lớp tập hợp có ba kiểu khởi tạo chính:

- Khởi tạo không tham số:

```
Set<T> Tên_đối_tượng;
```

- Khởi tạo bằng một mảng các đối tượng phần tử:

```
Set<T> Tên_đối_tượng(T*, chiều_dài_mảng);
```

- Khởi tạo bằng một đối tượng thuộc lớp tập hợp khác:

```
Set<T> Tên_đối_tượng(Set<T>);
```

Trong đó:

- **T**: là tên kiểu của các phần tử của tập hợp. Kiểu này có thể là kiểu cơ bản của C++, cũng có thể là các kiểu cấu trúc (struct) hoặc lớp (class) do người dùng tự định nghĩa.

Ví dụ:

```
Set<int> mySet;
```

là khai báo một đối tượng mySet của lớp tập hợp, mySet khởi đầu chưa có phần tử nào, các phần tử của đối tượng này có kiểu cơ bản int. Hoặc:

```
Person *myPerson = new Person[10];    // Khởi tạo mảng đối tượng
Set<Person> mySet(myPerson, 10);        // Khai báo tập hợp
```

là khai báo một đối tượng mySet của lớp tập hợp, có 10 phần tử tương ứng với các phần tử trong mảng động myPerson, các phần tử có kiểu lớp Person.

7.2.2 Toán tử

Các toán tử trên lớp tập hợp là tương ứng với các toán tử số học trên tập hợp thông thường: OR “|”, AND “&”, XOR “^” và phép trừ “-”.

Phép toán “|” và “|= ”

Phép toán này trả về phép hợp (OR) của hai đối tượng của lớp tập hợp, kết quả cũng là một đối tượng của lớp tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> | <Đối_tượng_3>;
<Đối_tượng_1> |= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 | mySet2;
```

thì mySet3 sẽ chứa các phần tử kiểu int là {1,2,3,4}.

Phép toán "&" và "&="

Phép toán này trả về phép giao (AND) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> & <Đối_tượng_3>;  
<Đối_tượng_1> &= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 & mySet2;
```

thì mySet3 sẽ chứa các phần tử có kiểu int là {2}.

Phép toán "^" và "^="

Phép toán này trả về phép hợp ngoại (XOR) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> ^ <Đối_tượng_3>;  
<Đối_tượng_1> ^= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 ^ mySet2;
```

thì mySet3 sẽ chứa các phần tử có kiểu int là {1,3,4}.

Phép toán "-" và "-="

Phép toán này trả về phép hiệu (loại trừ) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> - <Đối_tượng_3>;  
<Đối_tượng_1> -= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 - mySet2;
```

thì mySet3 sẽ chứa các phần tử có kiểu int là {1}.

7.2.3 Phương thức

Lớp tập hợp có một số phương thức cơ bản sau:

- Thêm một phần tử vào tập hợp
- Loại một phần tử khỏi tập hợp
- Tìm kiếm một phần tử trong tập hợp

Thêm một phần tử vào tập hợp

Có hai cú pháp để thêm một phần tử vào tập hợp:

```
pair<iterator, bool> insert(T&);  
iterator insert(<Vị trí con chạy>, T&);
```

Trong đó:

- Phương thức thứ nhất thêm một phần tử vào tập hợp, nếu phần tử đã có mặt trong tập hợp, trả về false và vị trí con chạy của phần tử đó. Nếu phần tử chưa tồn tại, trả về true và vị trí con chạy của phần tử mới thêm vào.
- Phương thức thứ hai cũng thêm vào một phần tử, nhưng chỉ kiểm tra xem phần tử đã tồn tại hay chưa bắt đầu từ vị trí con chạy được chỉ ra trong biến <vị trí con chạy>. Phương thức này trả về vị trí con chạy của phần tử mới thêm vào (nếu thành công) hoặc vị trí của phần tử đã có mặt.

Ví dụ, mySet là một tập hợp kiểu int:

```
mySet.insert(10);
```

sẽ thêm một phần tử có giá trị 10 vào tập hợp.

Lưu ý:

- Do tuân thủ theo lý thuyết tập hợp, nên khi chèn vào tập hợp nhiều lần với cùng một giá trị phần tử. Trong tập hợp chỉ tồn tại duy nhất một giá trị phần tử đó, không được trùng lặp.

Loại một phần tử khỏi tập hợp

Có ba cú pháp để loại bỏ phần tử khỏi tập hợp:

```
int erase(T&);  
void erase(<vị trí con chạy>);  
void erase(<vị trí bắt đầu>, <vị trí kết thúc>);
```

Trong đó:

- T: là tên kiểu các phần tử của tập hợp.
- Phương thức thứ nhất xoá phần tử có giá trị được chỉ rõ trong tham số đầu vào.
- Phương thức thứ hai loại bỏ phần tử ở vị trí của con chạy được xác định bởi tham số đầu vào.
- Phương thức thứ ba loại bỏ một số phần tử nằm trong phạm vi từ <vị trí bắt đầu> cho đến <vị trí kết thúc> của con chạy.

Ví dụ, mySet là một tập hợp các phần tử có kiểu int:

```
mySet.erase(10);
```

sẽ xoá phần tử có giá trị 10, hoặc:

```
mySet.erase((Set<int>::iterator)10);
```

sẽ xoá phần tử ở vị trí thứ 10 trong tập hợp, hoặc:

```
mySet.erase(mySet.begin(), mySet.end());
```

sẽ xoá toàn bộ các phần tử hiện có của tập hợp mySet.

Tìm kiếm một phần tử trong tập hợp

Có hai cú pháp để tìm kiếm một phần tử trong tập hợp:

```
iterator find(T&);  
int count(T&);
```

Trong đó:

- Phương thức thứ nhất tìm phần tử có giá trị xác định bởi tham số đầu vào, kết quả là vị trí con chạy của phần tử đó.
- Phương thức thứ hai chỉ để kiểm tra xem phần tử có xuất hiện trong tập hợp hay không: trả về 1 nếu có mặt, trả về 0 nếu không có mặt.

Ví dụ, mySet là một tập hợp các phần tử có kiểu int:

```
Set<int>::iterator index = mySet.find(10);  
cout << index;
```

sẽ hiển thị vị trí con chạy của phần tử có giá trị 10 trong tập hợp mySet.

7.2.4 Áp dụng

Chương trình 7.1 minh họa một số thao tác trên một tập hợp các phần tử có kiểu char: thêm phần tử, loại bỏ phần tử, duyệt các phần tử.

Chương trình 7.1

```
#include<stdio.h>  
#include<conio.h>  
#include<set.h>  
void main() {  
    clrscr();  
    Set<char> mySet;  
    int function;  
    do{  
        clrscr();  
        cout << "CAC CHUC NANG:" << endl;  
        cout << "1: Them mot phan tu vao tap hop" << endl;  
        cout << "2: Loai bo mot phan tu khoi tap hop" << endl;  
        cout << "3: Xem tat ca cac phan tu cua tap hop" << endl;  
        cout << "5: Thoat!" << endl;  
        cout << "======" << endl;  
        cout << "Chon chuc nang: " << endl;  
        cin >> function;  
        switch(function){  
            case '1':                // Thêm vào  
                char phantu;  
                cout << "Ki tu them vao: ";  
                cin >> phantu;  
                mySet.insert(phantu);  
                break;  
            case '2':                // Loại ra  
                char phantu;  
                cout << "Loai bo ki tu: " << endl;  
                cin >> phantu;
```

```

        mySet.erase(phantu);
        break;
    case '3': // Duyệt
        cout<<"Cac phan tu cua tap hop la:"<<endl;
        Set<char>::iterator i;
        for(i=mySet.begin(); i<mySet.end(); i++)
            cout << mySet[i] << " ";
        break;
    }while(function != '5');
return;
}

```

7.3 LỚP CHUỖI

Lớp chuỗi (String) cũng là một loại lớp chứa, nó chứa một tập các phần tử là một dãy các kí tự có phân biệt thứ tự, các phần tử không nhất thiết phải phân biệt nhau. Muốn sử dụng lớp String, cần thêm vào chỉ thị đầu tệp:

```
#include <string.h>
```

7.3.1 Hàm khởi tạo

Lớp String có ba hàm khởi tạo chính:

- Hàm khởi tạo không tham số:
`String <tên biến>;`
- Hàm khởi tạo từ một string khác:
`String <tên biến>(const String &);`
- Hàm khởi tạo từ một mảng các kí tự:
`String <tên biến>(char*, <chiều dài mảng>);`

Ví dụ:

```
String myStr;
```

là khai báo một chuỗi myStr chưa có phần tử nào (rỗng). Hoặc:

```
String myStr("hello!");
```

là khai báo một chuỗi myStr có các phần tử theo thứ tự là {'h', 'e', 'l', 'l', 'o', '!'}. Hoặc:

```
char* myChar = new char[10];
String myStr(myChar, 10);
```

là khai báo một chuỗi myStr có 10 phần tử tương ứng với các kí tự trong mảng myChar.

Lưu ý:

- Trong trường hợp khởi tạo bằng một chuỗi khác, ta có thể truyền vào một đối số có kiểu cơ bản khác, trình biên dịch sẽ tự động chuyển đổi số đó sang dạng string.

Ví dụ:

```
String myStr(12);
```


thì chuỗi myStr sẽ chứa hai phần tử kí tự {'1', '2'}. Hoặc:

```
String myStr(15.55);
```

thì chuỗi myStr sẽ chứa năm phần tử kí tự {'1', '5', '.', '5', '5'}.

7.3.2 Toán tử

Lớp String có các toán tử cơ bản là:

- Phép gán chuỗi
- Phép cộng chuỗi
- Phép so sánh chuỗi
- Phép vào/ra

Phép gán chuỗi “=”

Cú pháp phép gán chuỗi là tương tự cú pháp gán các đối tượng cơ bản:

```
<Tên biến 1> = <Tên biến 2>;
```

Ví dụ:

```
String s1(12), s2;  
s2 = s1;
```

thì chuỗi s2 cũng chứa hai phần tử như s1 {'1', '2'}.

Lưu ý:

- Có thể gán trực tiếp các đối tượng cơ bản cho chuỗi:

```
String myStr = 12; // myStr có hai phần tử {'1', '2'}
```

- Nhưng phép gán lại có độ ưu tiên thấp hơn phép toán học:

```
String myStr = 12+1.5; // tương đương myStr = 13.5
```

Phép cộng chuỗi “+” và “+=”

Phép cộng chuỗi sẽ nối chuỗi thứ hai vào sau chuỗi thứ nhất, kết quả cũng là một chuỗi:

```
<Tên biến 1> = <Tên biến 2> + <Tên biến 3>;  
<Tên biến 1> += <Tên biến 2>;
```

Ví dụ:

```
String s1(12), s2(3);  
s1+= s2;
```

thì s1 sẽ có ba phần tử kí tự {'1', '2', '3'}.

Phép so sánh chuỗi

Các phép so sánh chuỗi đều là các phép toán hai ngôi, trả về kết quả ở dạng bool (true/false):

- Phép so sánh lớn hơn “>”: chuỗi_1 > chuỗi_2;
- Phép so sánh lớn hơn hoặc bằng “>=”: chuỗi_1 >= chuỗi_2;
- Phép so sánh nhỏ hơn “<”: chuỗi_1 < chuỗi_2;
- Phép so sánh nhỏ hơn hoặc bằng “<=”: chuỗi_1 <= chuỗi_2;

- Phép so sánh bằng “==”: chuỗi_1 == chuỗi_2;
- Phép so sánh khác (không bằng) “!=”: chuỗi_1 != chuỗi_2;

Lưu ý:

- Phép so sánh chuỗi thực hiện so sánh mã ASCII của từng ký tự ở hai chuỗi theo thứ tự tương ứng cho đến khi có sự khác nhau đầu tiên giữa hai ký tự.
- Phép so sánh là phép so sánh dựa trên từ điển, có phân biệt chữ hoa và chữ thường.

Ví dụ:

```
"12" < "a";           // có giá trị đúng
"a" <= "A";           // có giá trị sai
```

Phép vào/ra

- Phép xuất ra “<<”: cout << biến_chuỗi;
- Phép nhập vào “>>”: cin >> biến_chuỗi;

Ví dụ:

```
String s("hello!");
cout << s;
```

sẽ in ra màn hình dòng chữ “hello!”.

7.3.3 Phương thức

Lớp chuỗi có một số phương thức cơ bản:

- Lấy chiều dài chuỗi
- Tìm một chuỗi con
- Thêm một chuỗi con
- Xoá một chuỗi con
- Chuyển kiểu ký tự

Lấy chiều dài chuỗi

Cú pháp:

```
<biến_chuỗi>.length();
```

trả về chiều dài của chuỗi (số lượng phần tử ký tự trong chuỗi).

Ví dụ:

```
String s("hello!");
cout << s.length();
```

sẽ in ra màn hình độ dài chuỗi s là 6.

Tìm một chuỗi con

Cú pháp:

```
<biến_chuỗi>.find(<Chuỗi con>, <Vị trí bắt đầu>, <Kiểu so khớp>);
```

Trong đó:

- Tham số thứ nhất là chuỗi con cần tìm.
- Tham số thứ hai là vị trí để bắt đầu tìm, mặc định là bắt đầu tìm từ phần tử có chỉ số 0.
- Tham số thứ ba chỉ ra cách so khớp có phân biệt chữ hoa với chữ thường: SM_IGNORE là không phân biệt, SM_SENSITIVE là có phân biệt.
- Phương thức này trả về kết quả dạng bool, tương ứng là có tìm thấy hay không.

Ví dụ:

```
s.find("12", 0, SM_IGNORE);
```

sẽ tìm trong chuỗi s xem có sự xuất hiện của chuỗi "12" hay không, vị trí bắt đầu tìm là 0, với cách tìm không phân biệt chữ hoa chữ thường.

Thêm một chuỗi con

Cú pháp:

```
<biến_chuỗi>.insert(<vị trí chèn>, <chuỗi con>);
```

Trong đó:

- Tham số thứ nhất là vị trí chỉ số mà tại đó, chuỗi con sẽ được chèn vào
- Tham số thứ hai là chuỗi con cần chèn, chuỗi con này cũng có thể là một kí tự.

Ví dụ:

```
s.insert(0, "12");
```

sẽ chèn vào đầu chuỗi s một chuỗi con có hai phân tử "12".

Xoá một chuỗi con

Cú pháp:

```
<biến_chuỗi>.delete(<vị trí bắt đầu>, <độ dài chuỗi xoá>);
```

Trong đó:

- Tham số thứ nhất là vị trí bắt đầu xoá chuỗi con
- Tham số thứ hai là độ dài chuỗi con bị xoá, giá trị mặc định là 1.

Ví dụ:

```
s.delete(0, 2);
```

sẽ xoá hai kí tự đầu của chuỗi s.

Chuyển kiểu kí tự

- Đổi chuỗi thành các kí tự hoa: `<biến_chuỗi>.toUpperCase();`
- Đổi chuỗi thành các kí tự thường: `<biến_chuỗi>.toLowerCase();`

Ví dụ:

```
s.toUpperCase(); // chuyển chuỗi s thành kí tự hoa  
s.toLowerCase(); // chuyển chuỗi s thành kí tự thường
```

7.3.4 Áp dụng

Chương trình 7.2 minh họa một số thao tác cơ bản trên lớp chuỗi, có sử dụng thư viện lớp chuỗi chuẩn của C++: cộng thêm một chuỗi, chèn thêm một chuỗi con, xóa một chuỗi con, tìm một chuỗi con...

Chương trình 7.2

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main() {
    clrscr();
    String myStr;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Cong them mot chuoi" << endl;
        cout << "2: Chen them mot chuoi" << endl;
        cout << "3: Xoa di mot chuoi" << endl;
        cout << "4: Tim mot chuoi con" << endl;
        cout << "5: Chuyen thanh chu hoa" << endl;
        cout << "6: Chuyen thanh chu thuong" << endl;
        cout << "7: Xem noi dung chuoi" << endl;
        cout << "8: Xem chieu dai chuoi" << endl;
        cout << "9: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1': // Thêm vào cuối
                String subStr;
                cout << "Chuoi them vao: ";
                cin >> subStr;
                myStr += subStr;
                break;
            case '2': // Chèn vào chuỗi
                String subStr;
                int position;
                cout << "Chuoi them vao: ";
                cin >> subStr;
                cout << "Vi tri chen:";
                cin >> position;
```

```

        myStr.insert(position, subStr);
        break;
    case '3': // Xoá đi một chuỗi con
        int position, count;
        cout << "Vi tri bat dau xoa:";
        cin >> position;
        cout << "Do dai xoa:";
        cin >> count;
        myStr.delete(position, count);
        break;
    case '4': // Tìm chuỗi con
        String subStr;
        int position;
        cout << "Chuoi con can tim:";
        cin >> subStr;
        cout << "Vi tri bat dau tim:";
        cin >> position;
        if(myStr.find(position, subStr))
            cout << "Co xuất hiện!" << endl;
        else
            cout << "Không xuất hiện!" << endl;
        break;
    case '5': // Chuyển thành chữ hoa
        myStr.toUpperCase();
        cout << myStr << endl;
        break;
    case '6': // Chuyển thành chữ thường
        myStr.toLowerCase();
        cout << myStr << endl;
        break;
    case '7': // Duyệt
        cout << "Nội dung chuỗi:" << endl;
        cout << myStr << endl;
        break;
    case '8': // Duyệt
        cout << "Chiều dài chuỗi:"
            << myStr.length() << endl;
        break;
    }while(function != '9');
return;
}

```

7.4 LỚP NGĂN XẾP VÀ HÀNG ĐOỊ

7.4.1 Lớp ngăn xếp

Lớp ngăn xếp (stack) cũng là một loại lớp vật chứa, nó chứa các phần tử cùng kiểu, không bắt buộc phải phân biệt nhau nhưng có phân biệt về thứ tự: các thao tác thêm phần tử và lấy phần tử ra đều được thực hiện ở một đầu ngăn xếp. Phần tử nào được thêm vào trước thì sẽ bị lấy ra sau.

Muốn dùng lớp Stack phải dùng chỉ thị đầu tệp:

```
#include<stack.h>
```

Hàm khởi tạo

Lớp Stack có hai cách khởi tạo:

- Khởi tạo không tham số:

```
Stack<T> biến_ngăn_xếp;
```

- Khởi tạo bằng một ngăn xếp khác, có cùng kiểu phần tử:

```
Stack<T> biến_ngăn_xếp(Stack<T>);
```

Trong đó:

- **T**: là kiểu của các phần tử chứa trong ngăn xếp. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
Stack<int> myStack;
```

là khai báo một biến myStack, chứa các phần tử có kiểu cơ bản int.

Toán tử

Lớp Stack chỉ dùng đến các toán tử gán "=" và toán tử so sánh bằng "==":

- Phép gán "=":

```
<ngăn xếp 1> = <ngăn xếp 2>;
```

Dùng để gán hai đối tượng ngăn xếp.

- Phép so sánh bằng "==":

```
<ngăn xếp 1> == <ngăn xếp 2>;
```

Dùng để kiểm tra xem hai đối tượng ngăn xếp có bằng nhau hay không. Kết quả trả về có kiểu bool (true/false).

Phương thức

- Thêm một phần tử:

```
<biến ngăn xếp>.push(T);
```

sẽ thêm một phần tử có kiểu T vào đỉnh ngăn xếp.

- Loại một phần tử:

```
<biến ngăn xếp>.pop();
```

sẽ trả về phần tử đang nằm ở đỉnh ngăn xếp.

- Kiểm tra tính rỗng:

```
<biên ngăn xếp>.empty();
```

sẽ trả về kết quả kiểu bool, tương ứng với trạng thái của ngăn xếp có rỗng hay không.

- Kích thước ngăn xếp:

```
<biên ngăn xếp>.size();
```

sẽ trả về số lượng các phần tử hiện đang có mặt trong ngăn xếp.

Áp dụng

Trong phần này, ta sẽ viết lại chương trình 3.4c đã được minh họa trong phần viết về cấu trúc ngăn xếp (chương 3). Nhưng thay vì phải định nghĩa cấu trúc ngăn xếp, ta dùng lớp ngăn xếp của thư viện C++: đảo ngược một chuỗi ký tự được nhập vào từ bàn phím.

Chương trình 7.3

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stack.h>

void main() {
    clrscr();
    Stack<char> myStack;
    char strIn[250];
    cout << "Nhập chuỗi: ";
    cin >> strIn;        // Nhập chuỗi ký tự từ bàn phím
    for(int i=0; i<strlen(strIn); i++) // Đặt vào ngăn xếp
        myStack.push(strIn[i]);
    while(!myStack.empty()) // Lấy ra từ ngăn xếp
        cout << myStack.pop();
    return;
}
```

7.4.2 Lớp hàng đợi

Lớp hàng đợi (queue) cũng là một loại lớp vật chứa, nó chứa các phần tử cùng kiểu, không bắt buộc phải phân biệt nhau nhưng có phân biệt về thứ tự: các thao tác thêm phần tử được thực hiện ở một đầu, các thao tác lấy phần tử ra được thực hiện ở một đầu còn lại của hàng đợi. Phần tử nào được thêm vào trước thì sẽ bị lấy ra trước.

Muốn dùng lớp Queue phải dùng chỉ thị đầu tệp:

```
#include<queue.h>
```

Hàm khởi tạo

Lớp Queue có hai cách khởi tạo:

- Khởi tạo không tham số:

```
Queue<T> biến_hàng_đợi;
```

- Khởi tạo bằng một hàng đợi khác, có cùng kiểu phần tử:

```
Queue<T> biến_hàng_đợi (Queue<T>);
```

Trong đó:

- **T**: là kiểu của các phần tử chứa trong hàng đợi. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
Queue<int> myQueue;
```

là khai báo một biến myQueue, chứa các phần tử có kiểu cơ bản int.

Toán tử

Lớp Queue chỉ dùng đến các toán tử gán “=” và toán tử so sánh bằng “==”:

- Phép gán “=”:

```
<hàng_đợi_1> = <hàng_đợi_2>;
```

Dùng để gán hai đối tượng hàng đợi.

- Phép so sánh bằng “==”:

```
<hàng_đợi_1> == <hàng_đợi_2>;
```

Dùng để kiểm tra xem hai đối tượng hàng đợi có bằng nhau hay không. Kết quả trả về có kiểu bool (true/false).

Phương thức

- Thêm một phần tử:

```
<biến_hàng_đợi>.push(T);
```

sẽ thêm một phần tử có kiểu T vào cuối hàng đợi.

- Loại một phần tử:

```
<biến_hàng_đợi>.pop();
```

sẽ trả về phần tử đang nằm ở đỉnh đầu hàng đợi.

- Kiểm tra tính rỗng:

```
<biến_hàng_đợi>.empty();
```

sẽ trả về kết quả kiểu bool, tương ứng với trạng thái của hàng đợi có rỗng hay không.

- Kích thước hàng đợi:

```
<biến_hàng_đợi>.size();
```

sẽ trả về số lượng các phần tử hiện đang có mặt trong hàng đợi.

Áp dụng

Trong phần này, ta sẽ cài đặt lại chương trình trong phần cấu trúc hàng đợi (chương 3). Nhưng thay vì phải tự định nghĩa cấu trúc hàng đợi, ta dùng lớp Queue của thư viện C++ để mô phỏng chương trình quản lý tiến trình của hệ điều hành.

Chương trình 7.4

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

void main(){
    clrscr();
    Queue<int> myQueue;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot tien trinh vao hang doi" << endl;
        cout << "2: Dua mot tien trinh tinh vao thuc hien" << endl;
        cout<<"3: Xem tat ca cac tien trinh trong hang doi"<< endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1':                // Thêm vào hàng đợi
                int maso;
                cout << "Ma so tien trinh vao hang doi: ";
                cin >> maso;
                myQueue.push(maso);
                break;
            case '2':                // Lấy ra khỏi hàng đợi
                cout << "Tien trinh duoc thuc hien: " <<
                    myQueue.pop() << endl;
                break;
            case '3':                // Duyệt hàng đợi
                Queue<int>::iterator i;
                for(i=myQueue.begin(); i<myQueue.end(); i++)
                    cout << myQueue[i] << " ";
                break;
        }while(function != '5');
    }
    return;
}
```


7.5 LỚP DANH SÁCH LIÊN KẾT

Lớp danh sách liên kết (List) cũng là một kiểu lớp vật chứa, nó chứa các phần tử cùng kiểu, có tính đến thứ tự. Muốn sử dụng lớp List của thư viện C++, phải khai báo chỉ thị đầu tệp:

```
#include<list.h> // Dành riêng cho lớp List
```

hoặc:

```
#include<stl.h> // Dùng chung cho các lớp vật chứa
```

7.5.1 Hàm khởi tạo

Lớp List có ba kiểu khởi tạo:

- Khởi tạo không tham số:
`List<T> bienn_danh_sach;`
- Khởi tạo từ một danh sách cùng kiểu:
`List<T> bienn_danh_sach(List<T>);`
- Khởi tạo từ một mảng các phần tử:
`List<T> bienn_danh_sach(T* <Mảng phần tử>, int <chiều dài mảng>);`

Trong đó:

- **T**: là kiểu của các phần tử chứa trong danh sách liên kết. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
List<int> myList;
```

sẽ khai báo một danh sách liên kết myList, các phần tử của nó có kiểu cơ bản int.

7.5.2 Toán tử

Toán tử gán “=”

Cú pháp:

```
<danh sách 1> = <danh sách 2>;
```

sẽ copy toàn bộ các phần tử của <danh sách 2> vào <danh sách 1>.

Toán tử so sánh bằng “==”

Cú pháp:

```
<danh sách 1> == <danh sách 2>;
```

sẽ trả về một giá trị kiểu bool, tương ứng với việc hai danh sách này có bằng nhau hay không. Việc so sánh được tiến hành trên từng phần tử ở vị trí tương ứng nhau.

Lưu ý:

- Ngoài ra còn có các phép toán so sánh khác cũng có thể thực hiện trên danh sách: <, >, <=, >=, !=.

7.5.3 Phương thức

Thêm một phần tử vào danh sách

Cú pháp:

```
<biến danh sách>.insert(<vị trí chèn>, <phần tử>);  
<biến danh sách>.push_front(<phần tử>);  
<biến danh sách>.push_back(<phần tử>);
```

Trong đó:

- Phương thức thứ nhất chèn một phần tử vào một vị trí bất kì của danh sách, vị trí chèn được chỉ ra bởi tham số thứ nhất.
- Phương thức thứ hai chèn một phần tử vào đầu danh sách
- Phương thức thứ ba chèn một phần tử vào cuối danh sách.

Ví dụ:

```
List<int> myList;  
myList.push_front(7);
```

sẽ chèn vào đầu danh sách myList một phần tử có giá trị là 7.

Xoá đi một phần tử

Cú pháp:

```
<biến danh sách>.erase(<vị trí xoá>);  
<biến danh sách>.pop_front();  
<biến danh sách>.pop_back();
```

Trong đó:

- Phương thức thứ nhất xoá một phần tử ở một vị trí bất kì của danh sách, vị trí xoá được chỉ ra bởi tham số thứ nhất.
- Phương thức thứ hai xoá một phần tử ở đầu danh sách
- Phương thức thứ ba xoá một phần tử ở cuối danh sách.

Ví dụ:

```
List<int> myList;  
cout << myList.pop_front();
```

sẽ in ra màn hình phần tử đầu của danh sách myList.

Kiểm tra tính rỗng của danh sách

Cú pháp:

```
<biến danh sách>.empty();
```

trả về giá trị bool, tương ứng với trạng thái hiện tại của biến danh sách là rỗng hay không.

Xem kích thước danh sách

Cú pháp:

```
<biến danh sách>.size();
```

Ví dụ:

```
cout << myList.size();
```

sẽ in ra màn hình kích cỡ (số lượng các phần tử) của danh sách.

Xem nội dung một phần tử

Cú pháp:

```
<biến danh sách>.get();  
<biến danh sách>.next();
```

Trong đó:

- Phương thức thứ nhất trả về phần tử ở vị trí hiện tại của con chạy
- Phương thức thứ hai trả về phần tử tiếp theo, và con chạy cũng di chuyển sang phần tử đó.

Ví dụ:

```
List<int> myList;  
cout << myList.get();
```

sẽ in ra màn hình nội dung phần tử thứ nhất của danh sách myList.

7.5.4 Áp dụng

Trong phần này, ta cài đặt lại chương trình 3.6c đã được trình bày trong phần cấu trúc danh sách liên kết (chương 3). Nhưng thay vì tự tạo danh sách liên kết, ta dùng lớp List trong thư viện của C++ để quản lý nhân viên văn phòng.

Chương trình 7.5

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
#include<list.h>  
  
typedef struct{  
    char name[25];           // Tên nhân viên  
    int age;                 // Tuổi nhân viên  
    float salary;            // Lương nhân viên  
} Employee;  
void main(){  
    clrscr();  
    List<Employee> myList;  
    int function;  
    do{  
        clrscr();  
        cout << "CAC CHUC NANG:" << endl;  
        cout << "1: Them mot nhan vien" << endl;  
        cout << "2: Xoa mot nhan vien" << endl;  
        cout << "3: Xem tat ca cac nhan vien trong phong" << endl;
```

```

cout << "5: Thoat!" << endl;
cout << "===== " << endl;
cout << "Chon chuc nang: " << endl;
cin >> function;
switch(function){
    case '1':                // Thêm vào ds
        int position;
        Employee employee;
        cout << "Vi tri can chen: ";
        cin >> position;
        cout << "Ten nhan vien: ";
        cin >> employee.name;
        cout << "Tuoi nhan vien: ";
        cin >> employee.age;
        cout << "Luong nhan vien: ";
        cin >> employee.salary;
        myList.insert(position, employee);
        break;
    case '2':                // Lấy ra khỏi ds
        int position;
        cout << "Vi tri can xoa: ";
        cin >> position;
        Employee result = myList.erase(position);
        if(result != NULL){
            cout << "Nhan vien bi loai: " endl;
            cout << "Ten: " << result.name << endl;
            cout << "Tuoi: " << result.age << endl;
            cout << "Luong: " << result.salary << endl;
        }
        break;
    case '3':                // Duyệt ds
        cout << "Cac nhan vien cua phong:" << endl;
        Employee result = myList.front();
        do{
            cout << result.name << " "
                << result.age << " "
                << result.salary << endl;
            result = myList.next();
        }while(result != NULL);
        break;
    }while(function != '5');
return;
}

```

TỔNG KẾT CHƯƠNG 7

Nội dung chương 7 đã trình bày một số lớp cơ bản của lớp vật chứa (Container) trong thư viện của C++:

- Lớp tập hợp (Set)
- Lớp chuỗi (String)
- Lớp ngăn xếp (Stack)
- Lớp hàng đợi (Queue)
- Lớp danh sách liên kết (List)

Hầu hết các lớp vật chứa này đều có thể chứa các phần tử với kiểu bất kì: có thể là kiểu cơ bản, cũng có thể là kiểu phức tạp do người dùng tự định nghĩa. Ngoại trừ lớp chuỗi, chỉ chứa các phần tử có kiểu char.

Các lớp vật chứa này có một số phương thức tương tự nhau, nhưng cách thực hiện lại khác nhau:

- Thêm vào một phần tử
- Lấy ra một phần tử
- Kiểm tra tính rỗng
- Kiểm tra số lượng phần tử.

Người dùng có thể áp dụng các lớp có sẵn trong thư viện này của C++ để giải quyết các bài toán khác nhau mà không phải tự định nghĩa lại các lớp.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 7

1. Dùng thư viện lớp stack để kiểm tra một số tự nhiên nhập vào từ bàn phím có phải là một số palindrom hay không: Một số tự nhiên là palindrom nếu đảo ngược thứ tự các chữ số, ta vẫn thu được chính số đó, ví dụ, 87278 là một số như vậy.
2. Viết lại chương trình 7.1 dùng lớp tập hợp, nhưng dùng kiểu của các phần tử là lớp Car đã được định nghĩa trong chương 5.
3. Dùng thư viện lớp String để viết chương trình chia nhỏ một chuỗi ban đầu thành một số chuỗi con, ranh giới phân chia là một chuỗi con được nhập vào từ bàn phím. Ví dụ “abc acb”, mà chia nhỏ theo chuỗi con “c” sẽ thu được các chuỗi: “ab”, “ca”, “cb”.
4. Dùng thư viện lớp String để viết chương trình thay thế các chuỗi con của một chuỗi ban đầu bằng một chuỗi con khác. Các chuỗi con được nhập từ bàn phím.
5. Viết lại chương trình 7.5, vẫn dùng thư viện lớp List, nhưng thay thế kiểu phần tử bằng kiểu lớp Car đã được định nghĩa trong chương 5.

HƯỚNG DẪN TRẢ LỜI CÂU HỎI VÀ BÀI TẬP

Chương 1

Chương 2

1. c và e.
2. a và d.
3. c.
4. b.
5. c.
6. c.
7. d.
8. b và c.
9. b.
10. c.
11. d.
12. c.
13. d.

Chương 3

1. a.
2. c.
3. a.
4. b.
5. b và d.
6. Gợi ý:

```
struct Subject{  
    char subject[20];  
    float note;  
};
```

Hoặc:

```
typedef struct {  
    char subject[20];  
    float note;  
}Subject;
```

7. Gợi ý:

```
typedef struct {  
    char name[20];
```

```
int age;
char class[20];
Subject* subjects;
char type[20];
}Student;
```

Chương 4

1. a.
2. b.
3. c.
4. a.
5. a.
6. b.
7. c.
8. d.
9. a.
10. b.
11. c.

Chương 5

1. a và c.
2. b.
3. d.
4. a, b và c.
5. c.
6. b và c.
7. b và c.
8. b, d và e.
9. c và d.
10. a và d.
11. a, b, c và d.
12. Gợi ý (từ bài 12 đến bài 17):

```
class Employee{
    char* name;
    int age;
    float salary;
public:
    Employee();
    Employee(char* nameIn="", int ageIn=18, float salaryIn=100);
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhdx@o-pit.edu.vn

CHƯƠNG TRÌNH
ĐÀO TẠO ĐẠI HỌC TỪ XA
PTIT

```
void setName(char*);  
char* getName();  
void setAge(int);  
int getAge();  
void setSalary(float);  
float getSalary();  
void show();  
~Employee();  
  
};
```

Chương 6

1. b.
2. d.
3. c.
4. b, d và f.
5. b và d.
6. d.
7. a, b, c và d.
8. a và c.
9. d.
10. d.
11. b.
12. b.
13. c và d.
14. a, b, c và d.
15. c và d.
16. b và c.
17. Gợi ý (từ bài 17 đến bài 25):

```
class Human{  
    char* name;  
    int age;  
    int sex;  
public:  
    Human();  
    Human(char*, int, int);  
    void setName(char*);  
    char* getName();  
    void setAge(int);  
    int getAge();  
    void setSex(int);  
};
```

```
        int getSex();
        void show();
        ~Human();
};

class Person: public virtual Human{
    char* address;
    char* phone;
public:
    Person();
    Person(char*, int, int, char*, char*);
    void setAddress(char*);
    char* getAddress();
    void setPhone(char*);
    char* getPhone();
    void show();
    ~Person();
};

class Worker: public virtual Human{
    int hour;
    float salary;
public:
    Worker();
    Worker(int, float);
    void setHour(int);
    int getHour();
    void setSalary(float);
    float getSalary();
    void show();
};

class Employee: public Person, public Worker{
    char* position;
public:
    Employee();
    Employee(char*, int, int, char*, char*, int, float,
char*);
    void setPosition(char*);
    char* getPosition();
    void virtual show();
    ~Employee();
};
```

Chương 7

2. Gợi ý


```
void main() {
    clrscr();
    Set<Car> mySet;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot phan tu vao tap hop" << endl;
        cout << "2: Loai bo mot phan tu khoi tap hop" << endl;
        cout << "3: Xem tat ca cac phan tu cua tap hop" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1': // Thêm vào
                Car car = new Car();
                cout << "O to them vao: " << endl;
                cout << "Toc do: ";
                int speed;
                cin >> speed;
                car.setSpeed(speed);
                cout << "Nhan hieu: ";
                char[] mark;
                cin >> mark;
                car.setMark(mark);
                float price;
                cout << "Gia xe: ";
                cin >> price;
                car.setPrice(price);
                mySet.insert(car);
                break;
            case '2': // Loại ra
                Set<Car>::iterator index;
                cout << "Loai bo xe o vi tri: ";
                cin >> index;
                if(index >= mySet.begin()) && (index < mySet.end())
                    mySet.erase(mySet[index]);
                break;
            case '3': // Duyệt
                cout<<"Cac phan tu cua tap hop la:"<<endl;
                Set<char>::iterator i;
                for(i=mySet.begin(); i<mySet.end(); i++){
                    cout << mySet[i].getSpeed() << " ";
                    cout << mySet[i].getMark() << " ";
                    cout << mySet[i].getPrice() << " ";
                }
        }
    }
}
```

```
        break;
    }while(function != '5');
    return;
}
```

5. Gợi ý

```
void main() {
    clrscr();
    List<Car> myList;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot xe o to" << endl;
        cout << "2: Xoa mot xe o to" << endl;
        cout << "3: Xem tat ca cac o to" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1': // Thêm vào ds
                int possition;
                Car car = new Car();
                cout << "Vi tri can chen: ";
                cin >> possition;
                cout << "Toc do: ";
                int speed;
                cin >> speed;
                car.setSpeed(speed);
                cout << "Nhan hieu:";
                char[] mark;
                cin >> mark;
                car.setMark(mark);
                float price;
                cout << "Gia xe:";
                cin >> price;
                car.setPrice(price);
                myList.insert(possition, car);
                break;
            case '2': // Lấy ra khỏi ds
                int possition;
                cout << "Vi tri can xoa: ";
                cin >> possition;
                Car result = myList.erase(possition);
                if(result != NULL){
                    cout << "O to bi loai: " << endl;
                    cout<<"Toc do:"<<result.getSpeed()<<endl;
                }
            }
        }
    }while(function != '5');
```

```
        cout << "Nhan hieu: " << result.getMark()
            << endl;
        cout<<"Gia: "<<result.getPrice()<<endl;
    }
    break;
case '3':
    // Duyệt ds
    cout << "Cac o to hien co:" << endl;
    Car result = myList.front();
    do{
        cout << result.getSpeed() << " "
            << result.getMark() << " "
            << result.getPrice()<< endl;
        result = myList.next();
    }while(result != NULL);
    break;
}while(function != '5');
return;
}
```

TÀI LIỆU THAM KHẢO

Tài liệu tiếng Anh

- [1] James P. Cohoon and Jack W. Davidson, *C++ Program Design – An Introduction to Programming and Object-Oriented Design*, 2nd edition, WCB McGraw-Hill, 1999.
- [2] Nell Dale, Chip Weems and Mark Headington, *Programming and Problem Solving with C++*, John & Barlett Publisher, 1996.
- [3] Michael T. Goodrich, Roberto Tamassia and David Mount, *Data Structures and Algorithms in C++*, John Wiley & Sons Inc, 2004.
- [4] Scott R. Ladd, *C++ Components and Algorithms*, 2nd edition, M&T Books, 1994.
- [5] Scott R. Ladd, *C++ Templates and Tools*, M&T Books, 1994
- [6] Robert Lafore, *Object – Oriented Programming in C++*, Fourth edition, SAMS, 2001.

Tài liệu tiếng Việt

- [1] Lê Đ. Hưng, Tạ T. Anh, Nguyễn H. Đức và Nguyễn T. Thuý, *Lập trình hướng đối tượng với C++*, NXB Khoa học và Kỹ thuật, 2005.
- [2] Nguyễn T. Thuý, Tạ T. Anh, Nguyễn Q. Huy và Nguyễn H. Đức, *Bài tập lập trình hướng đối tượng với C++*, NXB Khoa học và Kỹ thuật, 2004.

Các địa chỉ web

- 1. <http://www.angelfire.com/country/aldev0/cpphowto/>
- 2. <http://www.gnacademy.org/text/cc/Tutorial/tutorial.html>
- 3. <http://sophia.dtp.fmph.uniba.sk/cpptut/tutorial.us.html>
- 4. <http://www.brpreiss.com/books/opus4/html/book.html>
- 5. <http://www.fredosaurus.com/notes-cpp/index.html>

MỤC LỤC

GIỚI THIỆU	3
CHƯƠNG 1.....	5
GIỚI THIỆU VỀ CÁC PHƯƠNG PHÁP LẬP TRÌNH	5
1.1 LẬP TRÌNH TUYẾN TÍNH.....	5
1.2 LẬP TRÌNH HƯỚNG CẤU TRÚC	5
1.2.1 Đặc trưng của lập trình hướng cấu trúc.....	5
1.2.2 Phương pháp thiết kế trên xuống (top-down)	6
1.3 LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG	7
1.3.1 Lập trình hướng đối tượng.....	7
1.3.2 Một số khái niệm cơ bản.....	8
1.3.3 Lập trình hướng đối tượng trong C++	9
TỔNG KẾT CHƯƠNG 1.....	10
CHƯƠNG 2.....	11
CON TRỎ VÀ MẢNG	11
2.1 KHÁI NIỆM CON TRỎ	11
2.1.1 Khai báo con trỏ.....	11
2.1.2 Sử dụng con trỏ.....	11
2.2 CON TRỎ VÀ MẢNG	14
2.2.1 Con trỏ và mảng một chiều.....	14
2.2.2 Con trỏ và mảng nhiều chiều	17
2.3 CON TRỎ HÀM.....	18
2.4 CẤP PHÁT BỘ NHỚ ĐỘNG.....	20
2.4.1 Cấp phát bộ nhớ động cho biến	21
2.4.2 Cấp phát bộ nhớ cho mảng động một chiều.....	22
2.4.3 Cấp phát bộ nhớ cho mảng động nhiều chiều.....	23
TỔNG KẾT CHƯƠNG 2.....	25
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 2.....	26
CHƯƠNG 3.....	30
KIỂU DỮ LIỆU CẤU TRÚC	30
3.1 ĐỊNH NGHĨA CẤU TRÚC.....	30
3.1.1 Khai báo cấu trúc	30
3.1.2 Cấu trúc lồng nhau.....	31
3.1.3 Định nghĩa cấu trúc với từ khoá typedef.....	32
3.2 THAO TÁC TRÊN CẤU TRÚC	33
3.2.1 Khởi tạo giá trị ban đầu cho cấu trúc	33
3.2.2 Truy nhập đến thuộc tính của cấu trúc.....	34
3.3 CON TRỎ CẤU TRÚC VÀ MẢNG CẤU TRÚC	38
3.3.1 Con trỏ cấu trúc.....	38
3.3.2 Mảng cấu trúc	41
3.4 MỘT SỐ KIỂU DỮ LIỆU TRỪU TƯỢNG.....	44
3.4.1 Ngăn xếp.....	45
3.4.2 Hàng đợi.....	48
3.4.3 Danh sách liên kết.....	53
TỔNG KẾT CHƯƠNG 3.....	60
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 3.....	60
CHƯƠNG 4.....	64

VÀO RA TRÊN TỆP	64
4.1 KHÁI NIỆM TỆP	64
4.1.1 Tệp dữ liệu	64
4.1.2 Tệp văn bản	65
4.1.3 Tệp nhị phân	65
4.2 VÀO RA TRÊN TỆP	66
4.2.1 Vào ra tệp văn bản bằng ">>" và "<<"	66
4.2.2 Vào ra tệp nhị phân bằng read và write	70
4.3 TRUY NHẬP TỆP TRỰC TIẾP	74
4.3.1 Con trỏ tệp tin	74
4.3.2 Truy nhập vị trí hiện tại của con trỏ tệp	74
4.3.3 Dịch chuyển con trỏ tệp	76
TỔNG KẾT CHƯƠNG 4	78
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 4	79
CHƯƠNG 5	82
LỚP	82
5.1 KHÁI NIỆM LỚP ĐỐI TƯỢNG	82
5.1.1 Định nghĩa lớp đối tượng	82
5.1.2 Sử dụng lớp đối tượng	83
5.2 CÁC THÀNH PHẦN CỦA LỚP	83
5.2.1 Thuộc tính của lớp	84
5.2.2 Phương thức của lớp	85
5.3 PHẠM VI TRUY NHẬP LỚP	90
5.3.1 Phạm vi truy nhập lớp	90
5.3.2 Hàm bạn	91
5.3.3 Lớp bạn	96
5.4 HÀM KHỞI TẠO VÀ HUỖY BỎ	97
5.4.1 Hàm khởi tạo	97
5.4.2 Hàm huỷ bỏ	101
5.5 CON TRỎ ĐỐI TƯỢNG VÀ MẢNG ĐỐI TƯỢNG	103
5.5.1 Con trỏ đối tượng	103
5.5.2 Mảng các đối tượng	106
TỔNG KẾT CHƯƠNG 5	110
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 5	110
CHƯƠNG 6	115
TÍNH KẾ THỪA VÀ ĐA HÌNH	115
6.1 KHÁI NIỆM KẾ THỪA	115
6.1.1 Khai báo thừa kế	115
6.1.2 Tính chất dẫn xuất	116
6.2 HÀM KHỞI TẠO VÀ HUỖY BỎ TRONG KẾ THỪA	117
6.2.1 Hàm khởi tạo trong kế thừa	117
6.2.2 Hàm huỷ bỏ trong kế thừa	119
6.3 TRUY NHẬP TỚI CÁC THÀNH PHẦN TRONG KẾ THỪA LỚP	120
6.3.1 Phạm vi truy nhập	120
6.3.2 Sử dụng các thành phần của lớp cơ sở từ lớp dẫn xuất	122
6.3.3 Định nghĩa chồng các phương thức của lớp cơ sở	125
6.3.4 Chuyển đổi kiểu giữa lớp cơ sở và lớp dẫn xuất	128
6.4 ĐA KẾ THỪA	131
6.4.1 Khai báo đa kế thừa	131
6.4.2 Hàm khởi tạo và hàm huỷ bỏ trong đa kế thừa	132
6.4.3 Truy nhập các thành phần lớp trong đa kế thừa	134
	183

6.5 LỚP CƠ SỞ TRỪU TƯỢNG	138
6.5.1 Đặt vấn đề	138
6.5.2 Khai báo lớp cơ sở trừu tượng	138
6.5.3 Hàm khởi tạo lớp cơ sở trừu tượng	139
6.6 ĐA HÌNH	143
6.6.1 Đặt vấn đề	143
6.6.2 Khai báo phương thức trừu tượng	144
6.6.3 Sử dụng phương thức trừu tượng – đa hình	144
TỔNG KẾT CHƯƠNG 6	147
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 6	148
CHƯƠNG 7	153
MỘT SỐ LỚP QUAN TRỌNG	153
7.1 LỚP VẬT CHỨA	153
7.1.1 Giao diện của lớp Container	153
7.1.2 Con chạy Iterator	154
7.2 LỚP TẬP HỢP	155
7.2.1 Hàm khởi tạo	155
7.2.2 Toán tử	155
7.2.3 Phương thức	156
7.2.4 Áp dụng	158
7.3 LỚP CHUỖI	159
7.3.1 Hàm khởi tạo	159
7.3.2 Toán tử	160
7.3.3 Phương thức	161
7.3.4 Áp dụng	163
7.4 LỚP NGĂN XẾP VÀ HÀNG ĐỢI	165
7.4.1 Lớp ngăn xếp	165
7.4.2 Lớp hàng đợi	166
7.5 LỚP DANH SÁCH LIÊN KẾT	169
7.5.1 Hàm khởi tạo	169
7.5.2 Toán tử	169
7.5.3 Phương thức	170
7.5.4 Áp dụng	171
TỔNG KẾT CHƯƠNG 7	173
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 7	173
HƯỚNG DẪN TRẢ LỜI CÂU HỎI VÀ BÀI TẬP	174
Chương 1	174
Chương 2	174
Chương 3	174
Chương 4	175
Chương 5	175
Chương 6	176
Chương 7	177
TÀI LIỆU THAM KHẢO	181
MỤC LỤC	182

NGÔN NGỮ LẬP TRÌNH C++

Mã số : 412LTC340

Chịu trách nhiệm bản thảo

TRUNG TÂM ĐÀO TẠO BƯU CHÍNH VIỄN THÔNG 1

Lời nói đầu

Lập trình cấu trúc là phương pháp tổ chức, phân chia chương trình thành các hàm, thủ tục, chúng được dùng để xử lý dữ liệu nhưng lại tách rời các cấu trúc dữ liệu. Thông qua các ngôn ngữ Foxpro, Pascal, C đã số những người làm Tin học đã khá quen biết với phương pháp lập trình này.

Lập trình hướng đối tượng dựa trên việc tổ chức chương trình thành các lớp. Khác với hàm và thủ tục, lớp là một đơn vị bao gồm cả dữ liệu và các phương thức xử lý. Vì vậy lớp có thể mô tả các thực thể một cách chân thực, đầy đủ cả phần dữ liệu và yêu cầu quản lý. Tư tưởng lập trình hướng đối tượng được áp dụng cho hầu hết các ngôn ngữ mới chạy trên môi trường Windows như Microsoft Access, Visual Basic, Visual C. Vì vậy việc nghiên cứu phương pháp lập trình mới này là rất cần thiết đối với tất cả những người quan tâm, yêu thích Tin học.

C ra đời năm 1973 với mục đích ban đầu là để viết hệ điều hành Unix trên máy tính mini PDP. Sau đó C đã được sử dụng rộng rãi trên nhiều loại máy tính khác nhau và đã trở thành một ngôn ngữ lập trình cấu trúc rất được ưa chuộng.

Để đưa C vào thế giới hướng hướng đối tượng, năm 1980 nhà khoa học người Mỹ B. Stroustrup đã cho ra đời một ngôn ngữ C mới có tên ban đầu là “C có lớp”, sau đó đến năm 1983 thì gọi là C++. Ngôn ngữ C++ là một sự phát triển mạnh mẽ của C. Trong C++ chẳng những đưa vào tất cả các khái niệm, công cụ của lập trình hướng đối tượng mà còn đưa vào nhiều khả năng mới mẻ cho hàm. Như vậy C++ là một ngôn ngữ lai cho phép tổ chức chương trình theo các lớp và các hàm. Có thể nói C++ đã thúc đẩy ngôn ngữ C vốn đã rất thuyết phục đi vào thế giới lập trình hướng đối tượng và C++ đã trở thành ngôn ngữ hướng đối tượng nổi bật trong những năm 90.

Cuốn sách này sẽ trình bày một cách hệ thống các khái niệm của lập trình hướng đối tượng được cài đặt trong C++ như lớp, đối tượng, sự thừa kế, tính tương ứng bội và các khả năng mới trong xây dựng, sử dụng hàm như: đối tham chiếu, đối mặc định, hàm trùng tên, hàm toán tử. Có một số vấn đề còn ít được biết đến như cách xây dựng hàm với số đối bất định trong C cũng sẽ được giới thiệu. Các chương từ 1 đến 10 với cách giải thích tỉ mỉ và với gần 100 chương trình minh họa sẽ cung cấp cho bạn đọc các khái niệm, phương pháp và kinh nghiệm lập trình hướng đối tượng trên C++. Mục lục cuối sách sẽ hệ thống ngắn gọn phương pháp phân tích, thiết kế và lập trình hướng đối tượng trên bình diện chung.

Cuốn sách gồm 10 chương và 6 phụ lục

Chương 1 hướng dẫn cách làm việc với phần mềm TC++ 3.0 để thử nghiệm các chương trình, trình bày sơ lược về các phương pháp lập trình và giới thiệu một số mở rộng đơn giản của C++ .

Chương 2 trình bày các khả năng mới trong việc xây dựng và sử dụng hàm trong C++ như biến tham chiếu, đối có kiểu tham chiếu, đối có giá trị mặc định, hàm trực tuyến, hàm trùng tên, hàm toán tử.

Chương 3 nói về một khái niệm trung tâm của lập trình hướng đối tượng là lớp gồm: Định nghĩa lớp, khai báo các biến, mảng đối tượng (kiểu lớp), phương thức, dùng con trỏ this trong phương thức, phạm vi truy xuất của các thành phần, các phương thức toán tử.

Chương 4 trình bày các vấn đề tạo dựng, sao chép, huỷ bỏ các đối tượng và các vấn đề khác có liên quan như: Hàm tạo, hàm tạo sao chép, hàm huỷ, toán tử gán, cấp phát bộ nhớ cho đối tượng, hàm bạn, lớp bạn.

Chương 5 trình bày một khái niệm quan trọng tạo nên khả năng mạnh của lập trình hướng đối tượng trong việc phát triển, mở rộng phần mềm, đó là khả năng thừa kế của các lớp.

Chương 6 trình bày một khái niệm quan trọng khác cho phép xử lý các vấn đề khác nhau, các thực thể khác nhau, các thuật toán khác nhau theo cùng một lược đồ thống nhất, đó là tính tương ứng bội và phương thức ảo. Các công cụ này cho phép dễ dàng tổ chức chương trình quản lý nhiều dạng đối tượng khác nhau.

Chương 7 nói về việc tổ chức vào - ra trong C++. C++ đưa vào một khái niệm mới gọi là các dòng tin (Stream). Các thao tác vào - ra sẽ thực hiện trao đổi dữ liệu giữa bộ nhớ với dòng tin: Vào là chuyển dữ liệu từ dòng nhập vào bộ nhớ, ra là chuyển dữ liệu từ bộ nhớ lên dòng xuất. Để nhập xuất dữ liệu trên một thiết bị cụ thể nào, ta chỉ cần gắn dòng nhập xuất với thiết bị đó. Việc tổ chức vào ra theo cách như vậy là rất khoa học và tiện lợi vì nó có tính độc lập thiết bị.

Chương 8 trình bày các hàm đồ họa sử dụng trong C và C++. Các hàm này được sử dụng rải rác trong toàn bộ cuốn sách để xây dựng các đối tượng đồ họa.

Chương 9 trình bày các hàm truy xuất trực tiếp vào bộ nhớ của máy tính, trong đó có bộ nhớ màn hình. Các hàm này sẽ được sử dụng trong chương 10 để xây dựng các lớp menu và cửa sổ.

Chương 10 giới thiệu 5 chương trình tương đối hoàn chỉnh nhằm minh họa thêm khả năng và kỹ thuật lập trình hướng đối tượng trên C++

Phụ lục 1 trình bày các phép toán trong C++ và thứ tự ưu của chúng.

Phụ lục 2 liệt kê một danh sách các từ khoá của C++.

Phụ lục 3 trình bày bảng mã ASCII và mã quét của các ký tự.

Phụ lục 4 trình bày một vấn đề quan trọng nhưng còn ít được nói đến trong các tài liệu, đó là cách sử dụng con trỏ void để xây dựng các hàm với số đối không cố định giống như các hàm printf và scanf của C.

Vì trong C++ vẫn sử dụng các hàm của C, nên trong **phụ lục 5** sẽ giới thiệu tóm tắt hơn 200 hàm để bạn đọc tiện việc tra cứu.

Cuối cùng, **phụ lục 6** trình bày một cách ngắn gọn phương pháp phân tích, thiết kế và lập trình hướng đối tượng trên bình diện chung.

Khi viết chúng tôi đã hết sức cố gắng để cuốn sách được hoàn chỉnh, song chắc chắn không tránh khỏi thiếu sót, vì vậy rất mong nhận được sự góp ý của độc giả.

Nhân dịp này chúng tôi xin chân thành cảm ơn cử nhân Nguyễn Văn Phác đã tận tình giúp đỡ trong việc hiệu đính và biên tập cuốn sách này.

Tác giả

Chương 1

C++ và lập trình hướng đối tượng

Trong chương này trình bày các vấn đề sau:

- Cách sử dụng phần mềm TC++ 3.0
- Những sửa đổi cần thiết một chương trình C để biến nó thành một chương trình C++ (chạy được trong môi trường C++)
- Tóm lược về các phương pháp lập trình cấu trúc và lập trình hướng đối tượng
- Những mở rộng của C++ so với C

Bài 1. Làm việc với TC++ 3.0

Các ví dụ trong cuốn sách này sẽ viết và thực hiện trên môi trường TC++ 3.0. Bộ cài đặt TC++ 3.0 gồm 5 đĩa. Sau khi cài đặt (giả sử vào thư mục C:\TC) thì trong thư mục TC sẽ gồm các thư mục con sau:

C:\TC\BGI chứa các tệp đuôi BGI và CHR

C:\TC\BIN chứa các tệp chương trình (đuôi EXE) như TC, TCC, TLIB, TLINK

C:\TC\INCLUDE chứa các tệp tiêu đề đuôi H

C:\TC\LIB chứa các tệp đuôi LIB, OBJ

Để vào môi trường của TC++ chỉ cần thực hiện tệp chương trình TC trong thư mục C:\TC\BIN. Kết quả nhận được hệ menu chính của TC++ với màu nền xanh gần giống như hệ menu quen thuộc của TC (Turbo C). Hệ menu của TC++ gồm các menu: File, Edit, Search, Run, Compile, Debug, Project, Options, Window, Help.

Cách soạn thảo, biên dịch và chạy chương trình trong TC++ cũng giống như trong TC, ngoại trừ điểm sau: Tệp chương trình trong hệ soạn thảo của TC++ có đuôi mặc định là CPP cũng trong TC thì tệp chương trình luôn có đuôi C.

Trong TC++ có thể thực hiện cả chương trình C và C++. Để thực hiện chương trình C cần dựng đuôi C để đặt tên cho tệp chương trình, để thực hiện chương trình C++ cần dựng đuôi CPP để đặt tên cho tệp chương trình.

Bài 2. C và C++

- Có thể nói C++ là sự mở rộng (đáng kể) của C. Điều đó có nghĩa là mọi khả năng, mọi khái niệm trong C đều dùng được trong C++.

- Vì trong C++ sử dụng gần như toàn bộ các khái niệm, định nghĩa, các kiểu dữ liệu, các cấu trúc lệnh, các hàm và các công cụ khác của C, nên yêu cầu bắt buộc đối với các đọc giả C++ là phải biết sử dụng tương đối thành thạo ngôn ngữ C.

- Vì C++ là sự mở rộng của C, nên bản thân một chương trình C đó là chương trình C++ (chỉ cần thay đuôi C bằng đuôi CPP). Tuy nhiên Trình biên dịch TC++ yêu cầu mọi hàm chuẩn dùng trong chương trình đều phải khai báo nguyên mẫu bằng một câu lệnh #include, trong khi điều này không bắt buộc đối với Trình biên dịch của TC.

Trong C có thể dùng một hàm chuẩn mà bỏ qua câu lệnh #include để khai báo nguyên mẫu của hàm được dùng. Điều này không báo lỗi khi biên dịch, nhưng có thể dẫn đến kết quả sai khi chạy chương trình.

Ví dụ khi biên dịch chương trình sau trong môi trường C sẽ không gặp các dòng cảnh báo (Warning) và thông báo lỗi (error). Nhưng khi chạy sẽ nhận được kết quả sai.

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

```
void main()
```

```
{  
    float a,b,c,p,s;  
    printf("\nNhap a, b, c ");  
    scanf("%f%f%f",&a,&b,&c);  
    p=(a+b+c)/2;  
    s= sqrt(p*(p-a)*(p-b)*(p-c));  
    printf("\nDien tich = %0.2f",s);  
    getch();  
}
```

Nếu biên dịch chương trình này trong TC++ sẽ nhận được các thông báo lỗi sau:

Error: Funtion 'sqrt' should have a prototype

Error: Funtion 'getch' should have a prototype

Để biến chương trình trên thành một chương trình C++ cần:

+ Đặt tên chương chương với đuôi CPP

+ Thêm 2 câu lệnh #include để khai báo nguyên mẫu cho các hàm sqrt, getch:

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

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

Bài 3. Lập trình cấu trúc và lập trình hướng đối tượng

3.1. Phương pháp lập trình cấu trúc

- Tư tưởng chính của lập trình cấu trúc là tổ chức chương trình thành các chương trình con. Trong PASCAL có 2 kiểu chương trình con là thủ tục và hàm. Trong C chỉ có một loại chương trình con là hàm.

Hàm là một đơn vị chương trình độc lập dùng để thực hiện một phần việc nào đó như: Nhập số liệu, in kết quả hay thực hiện một số tính toán. Hàm cần có đối và các biến, mảng cục bộ dùng riêng cho hàm.

Việc trao đổi dữ liệu giữa các hàm thực hiện thông qua các đối và các biến toàn bộ.

Các ngôn ngữ như C, PASCAL, FOXPRO là các ngôn ngữ cho phép triển khai phương pháp lập trình cấu trúc.

Một chương trình cấu trúc gồm các cấu trúc dữ liệu (như biến, mảng, bản ghi) và các hàm, thủ tục.

Nhiệm vụ chính của việc tổ chức thiết kế chương trình cấu trúc là tổ chức chương trình thành các hàm, thủ tục: Chương trình sẽ bao gồm các hàm, thủ tục nào.

Ví dụ xét yêu cầu sau: Viết chương trình nhập tọa độ (x,y) của một dãy điểm, sau đó tìm một cặp điểm cách xa nhau nhất.

Trên tư tưởng của lập trình cấu trúc có thể tổ chức chương trình như sau:

+ Sử dụng 2 mảng thực toàn bộ x và y để chứa tọa độ dãy điểm

+ Xây dựng 2 hàm:

Hàm nhapsl dùng để nhập tọa độ n điểm, hàm này có một đối là biến nguyên n và được khai báo như sau:

```
void nhapsl(int n);
```

Hàm `do_dai` dùng để tính độ dài đoạn thẳng đi qua 2 điểm có chỉ số là `i` và `j`, nó được khai báo như sau:

```
float do_dai(int i, int j);
```

Chương trình C cho bài toán trên được viết như sau:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
float x[100],y[100];
float do_dai(int i, int j)
{
    return sqrt(pow(x[i]-x[j],2)+pow(y[i]-y[j],2));
}
void nhapsl(int n)
{
    int i;
    for (i=1;i<=n;++i)
    {
        printf("\nNhap toa do x, y cua diem thu %d : ",i);
        scanf("%f%f",&x[i],&y[i]);
    }
}
void main()
{
    int n,i,j,imax,jmax;
    float d,dmax;
    printf("\nSo diem N= ");
    scanf("%d",&n);
    nhapsl(n);
    dmax=do_dai(1,2); imax=1;jmax=2;
    for (i=1;i<=n-1;++i)
        for (j=i+1;j<=n;++j)
        {
            d=do_dai(i,j);
            if (d>dmax)
            {
                dmax=d;
                imax=i;
                jmax=j;
            }
        }
    printf("\nDoan thang lon nhat co do dai bang: %0.2f",dmax);
    printf("\n Di qua 2 diem co chi so la %d va %d",imax,jmax);
```

```
    getch();  
}
```

3.2. Phương pháp lập trình hướng đối tượng

+ Khái niệm trung tâm của lập trình hướng đối tượng là lớp (class). Có thể xem lớp là sự kết hợp các thành phần dữ liệu và các hàm. Cũng có thể xem lớp là sự mở rộng của cấu trúc trong C (struct) bằng cách đưa thêm vào các phương thức (method) hay cũng gọi là hàm thành viên (member function). Một lớp được định nghĩa như sau:

```
Class Tên_Lớp  
{  
    // Khai báo các thành phần dữ liệu  
    // Khai báo các phương thức  
};
```

+ Các phương thức có thể được viết (xây dựng) bên trong hoặc bên ngoài (phía dưới) phần định nghĩa lớp. Cấu trúc (cách viết) phương thức tương tự như hàm ngoại trừ quy tắc sau: Khi xây dựng một phương thức bên ngoài định nghĩa lớp thì trong dòng đầu tiên cần dùng tên lớp và 2 dấu : đặt trước tên phương thức để chỉ rõ phương thức thuộc lớp nào (xem ví dụ bên dưới).

+ Sử dụng các thành phần dữ liệu trong phương thức: Vì phương thức và các thành phần dữ liệu thuộc cùng một lớp và vì phương thức được lập lên cốt để xử lý các thành phần dữ liệu, nên trong thân của phương thức có quyền truy nhập đến các thành phần dữ liệu (của cùng lớp).

+ Biến lớp: Sau khi định nghĩa một lớp, có thể dùng tên lớp để khai báo các biến kiểu lớp hay cũng gọi là đối tượng. Mỗi đối tượng sẽ có các thành phần dữ liệu và các phương thức. Lời gọi một phương thức cần chứa tên đối tượng để xác định phương thức thực hiện từ đối tượng nào.

+ Một chương trình hướng đối tượng sẽ bao gồm các lớp có quan hệ với nhau.

+ Việc phân tích, thiết kế chương trình theo phương pháp hướng đối tượng nhằm thiết kế, xây dựng các lớp.

+ Từ khái niệm lớp nảy sinh hàng loạt khái niệm khác như: Thành phần dữ liệu, phương thức, phạm vi, sự đóng gói, hàm tạo, hàm hủy, sự thừa kế, lớp cơ sở, lớp dẫn xuất, tương ứng bội, phương thức ảo, ...

+ Ưu điểm của việc thiết kế hướng đối tượng là tập trung xác định các lớp để mô tả các thực thể của bài toán. Mỗi lớp đưa vào các thành phần dữ liệu của thực thể và xây dựng luôn các phương thức để xử lý dữ liệu. Như vậy việc thiết kế chương trình xuất phát từ các nội dung, các vấn đề của bài toán.

+ Các ngôn ngữ thuần túy hướng đối tượng (như Smalltalk) chỉ hỗ trợ các khái niệm về lớp, không có các khái niệm hàm.

+ C++ là ngôn ngữ lai, nó cho phép sử dụng cả các công cụ của lớp và hàm.

Để minh họa các khái niệm vừa nêu về lập trình hướng đối tượng ta trở lại xét bài toán tìm độ dài lớn nhất đi qua 2 điểm. Trong bài toán này ta gặp một thực thể là dãy điểm. Các thành phần dữ liệu của lớp dãy điểm gồm:

- Biến nguyên n là số điểm của dãy
- Con trỏ x kiểu thực trỏ đến vùng nhớ chứa dãy hoành độ
- Con trỏ y kiểu thực trỏ đến vùng nhớ chứa dãy tung độ

Các phương thức cần đưa vào theo yêu cầu bài toán gồm:

- Nhập tọa độ một điểm

- Tính độ dài đoạn thẳng đi qua 2 điểm

Dưới đây là chương trình viết theo thiết kế hướng đối tượng. Để thực hiện chương trình này nhớ đặt tên tệp có đuôi CPP. Xem chương trình ta thấy thêm một điều mới trong C++ là:

Các khai báo biến, mảng có thể viết bất kỳ chỗ nào trong chương trình (tất nhiên phải trước khi sử dụng biến, mảng).

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <alloc.h>
class daydiem
{
public:
    int n;
    float *x,*y;
    float do_dai(int i, int j)
    {
        return sqrt(pow(x[i]-x[j],2)+pow(y[i]-y[j],2));
    }
    void nhapsl(void);
};
void daydiem::nhapsl(void)
{
    int i;
    printf("\nSo diem N= ");
    scanf("%d",&n);
    x=(float*)malloc((n+1)*sizeof(float));
    y=(float*)malloc((n+1)*sizeof(float));
    for (i=1;i<=n;++i)
    {
        printf("\nNhap toa do x, y cua diem thu %d : ",i);
        scanf("%f%f",&x[i],&y[i]);
    }
}
void main()
{
    daydiem p;
    p.nhapsl();
    int n,i,j,imax,jmax;
    float d,dmax;
    n=p.n;
    dmax=p.do_dai(1,2); imax=1;jmax=2;
    for (i=1;i<=n-1;++i)
```

```
for (j=i+1;j<=n;++j)
{
    d=p.do_dai(i,j);
    if (d>dmax)
    {
        dmax=d;
        imax=i;
        jmax=j;
    }
}
printf("\nDoan thang lon nhat co do dai bang: %0.2f",dmax);
printf("\n Di qua 2 diem co chi so la %d va %d",imax,jmax);
getch();
}
```

Bài 4. Một số mở rộng đơn giản của C++ so với C

Trong mục này trình bày một số mở rộng của C++ , tuy đơn giản, ngắn gọn nhưng đem lại rất nhiều tiện lợi.

4.1. Viết các dòng ghi chú

Trong C++ vẫn có thể viết các dòng ghi chú trong các dấu /* và */ như trong C. Cách này cho phép viết các ghi chú trên nhiều dòng hoặc trên một dòng. Ngoài ra trong C++ cũng cho phép viết ghi chú trên một dòng sau 2 dấu gạch chéo, ví dụ:

```
int x,y ; // Khai báo 2 biến thực
```

4.2. Khai báo linh hoạt

Trong C tất cả các câu lệnh khai báo biến, mảng cục bộ phải đặt tại đầu khối. Do vậy nhiều khi, vị trí khai báo và vị trí sử dụng của biến khá xa nhau, gây khó khăn trong việc kiểm soát chương trình. C++ đã khắc phục nhược điểm này bằng cách cho phép các lệnh khai báo biến, mảng có thể đặt bất kỳ chỗ nào trong chương trình trước khi các biến, mảng được sử dụng. Ví dụ chương trình nhập một dãy số thực rồi sắp xếp theo thứ tự tăng dần có thể viết trong C++ như sau:

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
void main()
{
    int n;
    printf("\n So phan tu cua day N= ");
    scanf("%d",&n);
    float *x= (float*)malloc((n+1)*sizeof(float));
    for (int i=1;i<=n;++i)
    {
        printf("\nX[%d]= ",i);
```



```
scanf("%f",x+i);
}
for (i=1;i<=n-1;++i)
for (int j=i+1;j<=n;++j)
if (x[i]>x[j])
{
float tg=x[i];
x[i]=x[j];
x[j]=tg;
}
printf("\nDay sau khi sap xep\n");
for (i=1;i<=n;++i)
printf("%0.2f ",x[i]);
getch();
}
```

4.3. Toán tử ép kiểu

Toán tử này được viết trong C như sau:

(Kiểu) biểu thức

Trong C++ vẫn có thể dùng cách viết này. Ngoài ra C++ cho phép viết một cách khác tiện lợi hơn như sau:

Kiểu(biểu thức)

Ví dụ chương trình tính cộng thức

$$S = 2/1 + 3/2 + \dots + (n+1)/n$$

với n là một số nguyên dương nhập từ bàn phím, có thể viết như sau:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int n;
printf("\n So phan tu cua day N= ");
scanf("%d",&n);
float s=0.0;
for (int i=1;i<=n;++i)
s += float(i+1)/float(i) ; // Ep kieu theo C++
printf("S= %0.2f",s);
getch();
}
```

4.4. Hằng có kiểu

Để tạo ra một hằng có kiểu, ta sử dụng từ khoá const đặt trước một khai báo có khởi gán giá trị. Sau đây là một số ví dụ.

+ Hằng nguyên:

```
const int maxsize = 1000;
```

```
int a[maxsize] ;
```

+ Cấu trúc hằng:

```
typedef struct
```

```
{
```

```
    int x, y ; // Toạ độ của điểm
```

```
    int mau ; // Mỏ màu của điểm
```

```
} DIEM ;
```

```
const DIEM d = {320, 240, 15};
```

Chương trình dưới đây minh họa cách dùng hằng có kiểu. Chương trình tạo một cấu trúc hằng (kiểu DIEM) mô tả điểm giữa màn hình đồ họa với màu trắng. Điểm này được hiển thị trên màn hình đồ họa.

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

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

```
#include <graphics.h>
```

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

```
typedef struct
```

```
{
```

```
    int x,y;
```

```
    int mau;
```

```
} DIEM;
```

```
void main()
```

```
{
```

```
    int mh=0,mode=0;
```

```
    initgraph(&mh,&mode,"");
```

```
    int loi=graphresult();
```

```
    if (loi)
```

```
    {
```

```
        printf("\nLoi do hoa: %s",grapherrormsg(loi));
```

```
        getch(); exit(0);
```

```
    }
```

```
    const DIEM gmh = {getmaxx()/2,getmaxy()/2,WHITE};
```

```
    putpixel(gmh.x,gmh.y,gmh.mau);
```

```
    getch();
```

```
    closegraph();
```

```
}
```

Chú ý:

a. Có thể dùng các hàm để gán giá trị cho các hằng có kiểu (trong chương trình trên dùng các hàm getmax và getmaxy).

b. Mọi câu lệnh nhằm thay đổi giá trị hằng có kiểu đều bị báo lỗi khi biên dịch chương trình. Ví dụ nếu trong chương trình đưa vào câu lệnh:

gmh.x=200;

thì khi dịch chương trình sẽ nhận được thông báo lỗi như sau:

Cannot modify a const object

4.5. Các kiểu char và int

Trong C một hằng ký tự được xem là nguyên do đó nó có kích thước 2 byte, ví dụ trong C:

sizeof('A') = sizeof(int) = 2

Cũng trong C++ một hằng ký tự được xem là giá trị kiểu char và có kích thước một byte. Như vậy trong C++ thì:

sizeof('A') = sizeof(char) = 1

4.6. Lấy địa chỉ các phần tử mảng thực 2 chiều

Trong Turbo C 2.0 không cho phép dùng phép & để lấy địa chỉ các phần tử mảng thực 2 chiều. Vì vậy khi nhập một ma trận thực (dùng scanf) ta phải nhập qua một biến trung gian sau đó mới gán cho các phần tử mảng.

Trong TC ++ 3.0 cho phép lấy địa chỉ các phần tử mảng thực 2 chiều, do đó có thể dùng scanf để nhập trực tiếp vào các phần tử mảng.

Chương trình C++ dưới đây sẽ minh họa điều này. Chương trình nhập một ma trận thực cấp mxn và xác định phần tử có giá trị lớn nhất.

```
#include <conio.h>
#include <stdio.h>
void main()
{
    float a[20][20], smax;
    int m,n,i,j, imax, jmax;
    clrscr();
    puts( "Cho biet so hang va so cot cua ma tran: " );
    scanf("%d%d",&m,&n) ;
    for (i=1;i<=m;++i)
        for (j=1;j<=n;++j)
        {
            printf("\na[%d][%d]= ",i,j);
            scanf("%f",&a[i][j]); // Lấy địa chỉ phần tử mảng thực
                                // 2 chiều
        }
    smax = a[1][1]; imax=1; jmax=1;
    for (i=1;i<=m;++i)
        for (j=1;j<=n;++j)
            if (smax<a[i][j])
            {
                smax = a[i][j];
                imax=i ; jmax = j;
            }
}
```

```
puts( "\n\n Ma tran" );
for (i=1;i<=m;++i)
    for (j=1;j<=n;++j)
    {
        if (j==1) puts("");
        printf("%6.1f", a[i][j]);
    }
puts( "\n\nPhan tu max:" );
printf("\nco gia tri = %6.1f", smax);
printf("\nTai hang %d cot %d ",imax, jmax) ;
getch();
}
```

Bài 5. Vào ra trong C++

5.1. Các toán tử và phương thức xuất nhập

Để in dữ liệu ra màn hình và nhập dữ liệu từ bàn phím , trong C++ vẫn có thể dùng các hàm printf và scanf (như chỉ ra trong các chương trình C++ ở các mục trên).

Ngoài ra trong C++ cũng dùng toán tử xuất:

cout << biểu thức << ... << biểu thức ;

để đưa giá trị các biểu thức ra màn hình, dùng toán tử nhập:

cin >> biến >> ... >> biến

để nhập các giá trị số (nguyên thực) từ bàn phím và gán cho các biến.

Để nhập một dãy không quá n ký tự và chứa vào mảng h (kiểu char) có thể dùng phương thức cin.get như sau:

cin.get(h,n);

Chú ý 1: Toán tử nhập cin >> sẽ để lại ký tự chuyển dòng ‘\n’ trong bộ đệm, ký tự này có thể làm trôi phương thức cin.get. Để khắc phục tình trạng trên cần dùng phương thức cin.ignore để bỏ qua một ký tự chuyển dòng như sau:

cin.ignore(1);

Chú ý 2: Để sử dụng các toán tử và phương thức nói trên cần khai báo tệp tiêu đề:

#include <iostream.h>

Chương trình sau minh họa việc sử dụng các dụng cụ vào ra mới của C++ để nhập một danh sách n thí sinh. Dữ liệu mỗi thí sinh gồm họ tên, các điểm toán, lý, hoá. Sau đó in danh sách thí sinh theo thứ tự giảm của tổng điểm.

```
#include <iostream.h>
#include <conio.h>
void main()
{
    struct
    {
        char ht[25];
```

```
float t,l,h,td;
} ts[50],tg;
int n,i,j;
clrscr();
cout << " So thi sinh: " ;
cin >> n ;
for (i=1;i<=n;++i)
{
    cout << "\n Thi sinh " << i ;
    cout << "\n Ho ten: " ;
    cin.ignore(1);
    cin.get(ts[i].ht,25) ;
    cout << "Cac diem toan, ly, hoa: ";
    cin >> ts[i].t >> ts[i].l >> ts[i].h ;
    ts[i].td = ts[i].t + ts[i].l + ts[i].h ;
}
for (i=1;i<=n-1;++i)
    for (j=i+1;j<=n;++j)
        if (ts[i].td < ts[j].td )
        {
            tg=ts[i];
            ts[i]=ts[j];
            ts[j]=tg;
        }
cout << "\nDanh sach thi sinh sau khi sap xep " ;
for (i=1;i<=n;++i)
{
    cout << "\n Ho ten: " << ts[i].ht;
    cout << " Tong diem: " << ts[i].td;
}
getch();
}
```

5.2. Định dạng khi in ra màn hình

+ Để quy định số thực (float, double) được in ra có đúng p chữ số sau dấu chấm thập phân, ta sử dụng đồng thời các hàm sau:

setiosflags(ios::showpoint); // Bật cờ hiệu showpoint

setprecision(p);

Các hàm này cần đặt trong toán tử xuất như sau:

cout << setiosflags(ios::showpoint) << setprecision(p) ;

Câu lệnh trên sẽ có hiệu lực đối với tất cả các toán tử xuất tiếp theo cho đến khi gặp một câu lệnh định dạng mới.

+ Để quy định độ rộng tối thiểu là w vị trí cho giá trị (nguyên, thực, chuỗi) được in trong các toán tử xuất, ta dùng hàm

setw(w)

Hàm này cần đặt trong toán tử xuất và nó chỉ có hiệu lực cho một giá trị được in gần nhất. Các giá trị in ra tiếp theo sẽ có độ rộng tối thiểu mặc định là 0. Như vậy câu lệnh:

```
cout << setw(3) << "AB" << "CD"
```

Sẽ in ra 5 ký tự là: một dấu cách và 4 chữ cái A, B, C và D.

Chú ý: Muốn sử dụng các hàm trên cần đưa vào câu lệnh #include sau:

#include <iomanip.h>

Trở lại chương trình trên ta thấy danh sách thứ sinh in ra sẽ không thẳng cột. Để khắc phục điều này cần viết lại đoạn chương trình in như sau:

```
cout << "\nDanh sach thi sinh sau khi sap xep " ;
cout << setiosflags(ios::showpoint) << setprecision(1) ;
for(i=1;i<=n;++i)
{
    cout << "\n Ho ten: " << setw(25) << ts[i].ht;
    cout << " Tong diem: " << setw(5) << ts[i].td;
}
getch();
```

Chương trình dưới đây là một minh họa khác về việc sử dụng các toán tử nhập xuất và cách định dạng trong C++ . Chương trình nhập một ma trận thực cấp mxn. Sau đó in ma trận dưới dạng bảng và tìm một phần tử lớn nhất.

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
void main()
{
    float a[20][20], smax;
    int m,n,i,j, imax, jmax;
    clrscr();
    cout << " Cho biet so hang va so cot cua ma tran: " ;
    cin >> m >> n ;
    for (i=1;i<=m;++i)
        for (j=1;j<=n;++j)
        {
            cout << "a[" << i << ", " << j << "] = " ;
            cin >> a[i][j] ;
        }
    smax = a[1][1]; imax=1; jmax=1;
    for (i=1;i<=m;++i)
        for (j=1;j<=n;++j)
            if (smax<a[i][j])
```

```
{
    smax = a[i][j];
    imax=i ; jmax = j;
}
cout << "\n\n Ma tran" ;
cout << setiosflags(ios::showpoint) << setprecision(1) ;
for (i=1;i<=m;++i)
    for (j=1;j<=n;++j)
    {
        if (j==1) cout << 'n' ;
        cout << setw(6) << a[i][j];
    }
cout << "\n\n" << "Phan tu max:" << 'n' ;
cout << "co gia tri = " << setw(6) << smax;
cout << "\nTai hang " << imax << " cot " << jmax ;
getch();
}
```

Bài 6. Cấu trúc, hợp và kiểu liệt kê

6.1. Tên sau từ khoá struct được xem như tên kiểu cấu trúc

Trong C++ một kiểu cấu trúc cũng được định nghĩa như C theo mẫu:

```
struct Tên_kiểu_ct
{
    // Khai báo các thành phần của cấu trúc
};
```

Sau đó để khai báo các biến, mảng cấu trúc, trong C dùng mẫu sau:

```
struct Tên_kiểu_ct danh sách biến, mảng cấu trúc ;
```

Như vậy trong C, tên viết sau từ khoá struct chưa phải là tên kiểu và chưa có thể dùng để khai báo.

Trong C++ xem tên viết sau từ khoá struct là tên kiểu cấu trúc và có thể dùng nó để khai báo. Như vậy để khai báo các biến, mảng cấu trúc trong C++ , ta có thể dùng mẫu sau:

```
Tên_kiểu_ct danh sách biến, mảng cấu trúc ;
```

Ví dụ sau sẽ: Định nghĩa kiểu cấu trúc TS (thí sinh) gồm các thành phần : ht (họ tên), sobd (số báo danh), dt (điểm toán), dl (điểm lý), dh (điểm hoá) và td (tổng điểm), sau đó khai báo biến cấu trúc h và mảng cấu trúc ts.

```
struct TS
{
    char ht [25];
    long sobd;
    float dt, dl, dh, td;
};
TS h, ts[1000] ;
```

6.2. Tên sau từ khoá union được xem như tên kiểu hợp

Trong C++ một kiểu hợp (union) cũng được định nghĩa như C theo mẫu:

```
union Tên_kiểu_hợp
{
    // Khai báo các thành phần của hợp
};
```

Sau đó để khai báo các biến, mảng kiểu hợp, trong C dùng mẫu sau:

```
union Tên_kiểu_hợp danh sách biến, mảng kiểu hợp ;
```

Như vậy trong C, tên viết sau từ khoá union chưa phải là tên kiểu và chưa có thể dùng để khai báo.

Trong C++ xem tên viết sau từ khoá union là tên kiểu hợp và có thể dùng nó để khai báo. Như vậy để khai báo các biến, mảng kiểu hợp, trong C++ có thể dùng mẫu sau:

```
Tên_kiểu_hợp danh sách biến, mảng kiểu hợp ;
```

6.3. Các union không tên

Trong C++ cho phép dùng các union không tên dạng:

```
union
{
    // Khai báo các thành phần
};
```

Khi đó các thành phần (khai báo trong union) sẽ dùng chung một vùng nhớ. Điều này cho phép tiết kiệm bộ nhớ và cho phép dễ dàng tách các byte của một vùng nhớ.

Ví dụ nếu các biến nguyên i, biến ký tự ch và biến thực x không đồng thời sử dụng thì có thể khai báo chúng trong một union không tên như sau:

```
union
{
    int i;
    char ch;
    float x;
};
```

Khi đó các biến i, ch và f sử dụng chung một vùng nhớ 4 byte.

Xét ví dụ khác, để tách các byte của một biến unsigned long ta dùng union không tên sau:

```
union
{
    unsigned long u;
    unsigned char b[4];
};
```

Khi đó nếu gán

```
u = 0xDDCCBBAA; // Số hệ 16
```

thì :

```
b[0] = 0xAA
b[1] = 0xBB
b[2] = 0xCC
b[3] = 0xDD
```


6.4. Kiểu liệt kê (enum)

+ Cũng giống như cấu trúc và hợp, tên viết sau từ khoá enum được xem là kiểu liệt kê và có thể dùng để khai báo, ví dụ:

```
enum MAU { xanh, do, tim, vang } ; // Định nghĩa kiểu MAU
```

```
MAU m, dsm[10] ; // Khai báo các biến, mảng kiểu MAU
```

+ Các giá trị kiểu liệt kê (enum) là các số nguyên. Do đó có thể thực hiện các phép tính trên các giá trị enum, có thể in các giá trị enum, có thể gán giá trị enum cho biến nguyên, ví dụ:

```
MAU m1, m2 ;
```

```
int n1, n2 ;
```

```
m1 = tim ;
```

```
m2 = vang ;
```

```
n1 = m1 ; // n1 = 2
```

```
n2 = m1 + m2 ; // n2 = 5
```

```
printf ("n %d ", m2 ); // in ra số 3
```

+ Không thể gán trực tiếp một giá trị nguyên cho một biến enum mà phải dùng phép ép kiểu, ví dụ:

```
m1 = 2 ; // lỗi
```

```
m1 = MAU(2) ; // đúng
```

Bài 7. Cấp phát bộ nhớ

7.1. Trong C++ có thể sử dụng các hàm cấp phát bộ nhớ động của C như: hàm malloc để cấp phát bộ nhớ, hàm free để giải phóng bộ nhớ được cấp phát.

7.2. Ngoài ra trong C++ cũng đưa thêm toán tử new để cấp phát bộ nhớ và toán tử delete để giải phóng bộ nhớ được cấp phát bởi new

7.3. Cách dùng toán tử new để cấp phát bộ nhớ như sau:

+ Trước hết cần khai báo một con trỏ để chứa địa chỉ vùng nhớ sẽ được cấp phát:

```
Kiểu *p;
```

ở đây Kiểu có thể là:

- các kiểu dữ liệu chuẩn của C++ như int, long, float, double, char, ...

- các kiểu do lập trình viên định nghĩa như: mảng, hợp, cấu trúc, lớp, ...

+ Sau đó dùng toán tử new theo mẫu:

```
p = new Kiểu ; // Cấp phát bộ nhớ cho một biến (một phần tử)
```

```
p = new Kiểu[n] ; // Cấp phát bộ nhớ cho n phần tử
```

Ví dụ để cấp phát bộ nhớ cho một biến thực ta dùng câu lệnh sau:

```
float *px = new float ;
```

Để cấp phát bộ nhớ cho 100 phần tử nguyên ta dùng các câu lệnh:

```
int *pn = new int[100] ;
```

```
for (int i=0 ; i < 100 ; ++i )
```

*pn[i] = 20*i ; // Gán cho phần tử thứ i*

7.4. Hai cách kiểm tra sự thành công của new

Khi dùng câu lệnh:

Kiểu *p = new Kiểu[n] ;

hoặc câu lệnh:

Kiểu *p = new Kiểu ;

để cấp phát bộ nhớ sẽ xuất hiện một trong 2 trường hợp: thành công hoặc không thành công.

Nếu thành công thì p sẽ chứa địa chỉ đầu vùng nhớ được cấp phát.

Nếu không thành công thì p = NULL.

Đoạn chương trình sau minh họa cách kiểm tra lỗi cấp phát bộ nhớ:

```
double *pd ;
int n ;
cout << "\n Số phần tử : " ;
cin >> n ;
pd = new double[n] ;
if (pd==NULL)
{
    cout << " Lỗi cấp phát bộ nhớ "
    exit (0) ;
}
```

Cách thứ 2 để kiểm tra sự thành công của toán tử new là dùng con trỏ hàm:

_new_handler

được định nghĩa trong tệp "new.h". Khi gặp lỗi trong toán tử new (cấp phát không thành công) thì chương trình sẽ thực hiện một hàm nào đó do con trỏ _new_handler trỏ tới. Cách dùng con trỏ này như sau:

+ Xây dựng một hàm dùng để kiểm tra sự thành công của new

+ Gán tên hàm này cho con trỏ _new_handler

Như vậy hàm kiểm tra sẽ được gọi mỗi khi có lỗi xảy ra trong toán tử new.

Đoạn chương trình kiểm tra theo cách thứ nhất có thể viết theo cách thứ hai như sau:

```
void kiem_tra_new(void) // Lập hàm kiểm tra
{
    cout << " Lỗi cấp phát bộ nhớ "
    exit (0) ;
}
_new_handler = kiem_tra_new // Gán tên hàm cho con trỏ
double *pd ;
int n ;
cout << "\n Số phần tử : " ;
cin >> n ;
pd = new double[n] ; // Khi xảy ra lỗi sẽ gọi hàm kiem_tra_new
```

Chú ý: Có thể dùng lệnh gán để gán tên hàm xử lý lỗi cho con trỏ `_new_handler` như trong đoạn chương trình trên, hoặc dùng hàm:

set_new_handler(Tên hàm) ;

(xem các chương trình minh hoạ bên dưới)

7.5. Toán tử delete dùng để giải phóng vùng nhớ được cấp phát bởi new

Cách dùng như sau:

delete p ; // p là con trỏ dùng trong new

Ví dụ:

*float *px ;*

px = new float[2000] ; // Cấp phát bộ nhớ cho 2000 phần tử thực

// Sử dụng bộ nhớ được cấp phát

delete px ; // giải phóng bộ nhớ

7.6. Hai chương trình minh hoạ

Chương trình thứ nhất minh hoạ cách dùng new để cấp phát bộ nhớ chứa n thí sinh. Mỗi thí sinh là một cấu trúc gồm các trường ht (họ tên), sobd (số báo danh) và td (tổng điểm). Chương trình sẽ nhập n, cấp phát bộ nhớ chứa n thí sinh, kiểm tra lỗi cấp phát bộ nhớ (dùng cách 1), nhập n thí sinh, sắp xếp thí sinh theo thứ tự giảm của tổng điểm, in danh sách thí sinh sau khi sắp xếp, và cuối cùng là giải phóng bộ nhớ đó cấp phát.

```
#include <iomanip.h>
```

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

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

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

```
struct TS
```

```
{
```

```
    char ht[20];
```

```
    long sobd;
```

```
    float td;
```

```
};
```

```
void main(void)
```

```
{
```

```
    TS*ts ;
```

```
    int n;
```

```
    cout << "\n So thi sinh n = " ;
```

```
    cin >> n;
```

```
    ts = new TS[n+1];
```

```
    if(ts==NULL)
```

```
    {
```

```
        cout << "\nLỗi cấp phát bộ nhớ " ;
```

```
        getch();
```

```
        exit(0);
```

```
    }
```

```
for (int i=1;i<=n;++i)
{
    cout << "\nThi sinh thu " << i;
    cout << "\nHo ten: ";
    cin.ignore(1) ;
    cin.get(ts[i].ht,20);
    cout << "So bao danh: ";
    cin >> ts[i].sobd ;
    cout << "Tong diem: ";
    cin >> ts[i].td ;
}
for (i=1;i<=n-1;++i)
    for (int j=i+1;j<=n;++j)
        if (ts[i].td < ts[j].td)
        {
            TS tg=ts[i];
            ts[i]=ts[j];
            ts[j]=tg;
        }
cout << setiosflags(ios::showpoint) << setprecision(1) ;
for (i=1;i<=n;++i)
    cout << "\n" << setw(20) << ts[i].ht <<
        setw(6)<< ts[i].sobd << setw(6)<< ts[i].td;
delete ts;
getch();
}
```

Chương trình thứ hai minh họa cách dựng con trỏ `_new_handler` để kiểm tra sự thành công của toán tử `new`. Chương trình sẽ cấp phát bộ nhớ cho một mảng con trỏ và sẽ theo dõi khi nào thì không đủ bộ nhớ để cấp phát.

```
#include <new.h>
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
int k;
void loi_bo_nho(void)
{
    cout << "\nLoi bo nho khi cap phat bo nho cho q[" << k << "]";
    getch();
    exit(0);
}
void main()
```

```
{  
    double *q[100] ; long n;  
    clrscr();  
    set_new_handler(loi_bo_nho) ;  
    // _new_handler=loi_bo_nho;  
    n=10000;  
    for ( k=0;k<100;++k)  
        q[k] = new double[n];  
    cout << "Khong loi";  
    getch();  
}
```

Bài 8. Các hàm trong C++

Trong C++ có rất nhiều mở rộng, cải tiến về hàm làm cho việc xây dựng và sử dụng hàm rất tiện lợi. Điều này sẽ trình bày kỹ trong chương sau. Trong mục này chỉ thống kê một số điểm mới về hàm mà C++ đưa vào.

8.1. Đối kiểu tham chiếu

Trong C, để nhận kết quả của hàm cần dùng đối con trỏ, làm cho việc xây dựng cũng như sử dụng hàm khá phiền phức. Trong C++ đưa vào đối kiểu tham chiếu (giống như PASCAL) dùng để chứa kết quả của hàm, khiến cho việc tạo lập cũng như sử dụng hàm đơn giản hơn.

8.2. Đối tham chiếu const

Đối tham chiếu có đặc điểm là các câu lệnh trong thân hàm có thể truy nhập tới và dễ dàng làm cho giá trị của nó thay đổi. Nhiều khi ta muốn dùng đối kiểu tham chiếu chỉ để tăng tốc độ trao đổi dữ liệu giữa các hàm, không muốn dùng nó để chứa kết quả của hàm. Khi đó có thể dùng đối tham chiếu const để bảo toàn giá trị của đối trong thân hàm.

8.3. Đối có giá trị mặc định

Trong nhiều trường hợp người dùng viết một lời gọi hàm nhưng cũng chưa biết nên chọn giá trị nào cho các đối. Để khắc phục khó khăn này, C++ đưa ra giải pháp đối có giá trị mặc định. Khi xây dựng hàm, ta gán giá trị mặc định cho một số đối. Người dùng nếu không cung cấp giá trị cho các đối này, thì hàm sẽ dùng giá trị mặc định.

8.4. Hàm on line

Đối với một đoạn chương trình nhỏ (số lệnh không lớn) thì việc thay các đoạn chương trình này bằng các lời gọi hàm sẽ làm cho chương trình gọn nhẹ đôi chút nhưng làm tăng thời gian máy. Trong các trường hợp này có thể dùng hàm trực tuyến (on line) vừa giảm kích thước chương trình nguồn, vừa không làm tăng thời gian chạy máy.

8.5. Các hàm trùng tên (định nghĩa chồng các hàm)

Để lấy giá trị tuyệt đối của một số, trong C cần lập ra nhiều hàm với tên khác nhau, ví dụ abs cho số nguyên, fabs cho số thực, labs cho số nguyên dài, cabs cho số phức. Điều này rườ rẻ gây phiền toái cho người sử dụng. Trong C++ cho phép xây dựng các hàm trùng tên nhưng khác nhau về kiểu đối. Như vậy chỉ cần lập một hàm để lấy giá trị tuyệt đối cho nhiều kiểu dữ liệu khác nhau.

8.6. Định nghĩa chồng toán tử

Việc dùng các phép toán thay cho một lời gọi hàm rườ rằn làm cho chương trình ngắn gọn, sáng sủa hơn nhiều. Ví dụ để thực hiện phép cộng 2 ma trận nếu dùng phép cộng và viết:

$$C = A + B ;$$

thì rất gần với toán học. Trong C++ cho phép dùng các phép toán chuẩn để đặt tên cho các hàm (gọi là định nghĩa chồng toán tử). Sau đó có thể thay lời gọi hàm bằng các phép toán như nói ở trên. Như vậy một phép toán mang nhiều ý nghĩa, ví dụ phép + có thể hiểu là cộng 2 số nguyên, 2 số thực hoặc 2 ma trận. C++ sẽ căn cứ vào kiểu của các số hạng mà quyết định chọn phép cộng cụ thể.

chương 2

Hàm trong C++

Chương này trình bày những khả năng mới của C++ trong việc xây dựng và sử dụng hàm. Đó là:

- + Kiểu tham chiếu và việc truyền dữ liệu cho hàm bằng tham chiếu.
- + Đối tham chiếu hằng (const)
- + Đối có giá trị mặc định
- + Hàm trực tuyến
- + Việc định nghĩa chồng các hàm
- + Việc định nghĩa chồng các toán tử

1. Biến tham chiếu (Reference variable)

1.1. Hai loại biến dùng trong C

Trước khi nói đến biến tham chiếu, chúng ta nhắc lại 2 loại biến gặp trong C là:

Biến giá trị dùng để chứa dữ liệu (nguyên, thực, ký tự, ...)

Biến con trỏ dùng để chứa địa chỉ

Các biến này đều được cung cấp bộ nhớ và có địa chỉ. Ví dụ câu lệnh khai báo:

```
double x , *px;
```

sẽ tạo ra biến giá trị kiểu double x và biến con trỏ kiểu double px. Biến x có vùng nhớ 8 byte, biến px có vùng nhớ 4 byte (nếu dùng mô hình Large). Biến x dùng để chứa giá trị kiểu double, ví dụ lệnh gán:

```
x = 3.14;
```

sẽ chứa giá trị 3.14 vào biến x. Biến px dùng để chứa địa chỉ của một biến thực, ví dụ câu lệnh:

```
px = &x ;
```

sẽ lưu trữ địa chỉ của biến x vào con trỏ px.

1.2. Biến tham chiếu

Trong C++ cho phép sử dụng loại biến thứ ba là biến tham chiếu. So với 2 loại biến quen biết nói trên, thì biến này có những đặc điểm sau:

- + Biến tham chiếu không được cấp phát bộ nhớ, không có địa chỉ riêng.
- + Nó dùng làm bí danh cho một biến (kiểu giá trị) nào đó và nó sử dụng vùng nhớ của biến này.

Ví dụ câu lệnh:

```
float u, v, &r = u ;
```

tạo ra các biến thực u, v và biến tham chiếu thực r. Biến r không được cấp phát bộ nhớ, nó là một tên khác (bí danh) của u và nó dùng chung vùng nhớ của biến u.

Thuật ngữ: Khi r là bí danh (alias) của u thì ta nói r tham chiếu đến biến u. Như vậy 2 thuật ngữ trên được hiểu như nhau.

ý nghĩa: Khi r là bí danh của u thì r dùng chung vùng nhớ của u, đó đó :

- + Trong mọi câu lệnh, viết u hay viết r đều có ý nghĩa như nhau, vì đều truy nhập đến cùng một vùng nhớ.
- + Có thể dùng biến tham chiếu để truy nhập đến một biến kiểu giá trị.

Ví dụ:

```
int u, v, &r = u;
r = 10; // u=10
cout << u; // in ra số 10
r++; // u = 11
++u; // r = 12
cout << r; // in ra số 12
v = r; // v=12
&r; // Cho địa chỉ của u
```

Công dụng: Biến tham chiếu thường được sử dụng làm đối của hàm để cho phép hàm truy nhập đến các tham số biến trong lời gọi hàm.

Vài chú ý về biến tham chiếu:

a. Vì biến tham chiếu không có địa chỉ riêng, nó chỉ là bí danh của một biến kiểu giá trị nên trong khai báo phải chỉ rõ nó tham chiếu đến biến nào. Ví dụ nếu khai báo:

```
double &x;
```

thì Trình biên dịch sẽ báo lỗi:

```
Reference variable 'x' must be initialized
```

b. Biến tham chiếu có thể tham chiếu đến một phần tử mảng, ví dụ:

```
int a[10], &r = a[5];
r = 25; // a[5] = 25
```

c. Không cho phép khai báo mảng tham chiếu

d. Biến tham chiếu có thể tham chiếu đến một hằng. Khi đó nó sẽ sử dụng vùng nhớ của hằng và nó có thể làm thay đổi giá trị chứa trong vùng nhớ này.

Ví dụ nếu khai báo:

```
int &s = 23;
```

thì Trình biên dịch đưa ra cảnh báo (warning):

```
Temporary used to initialize 's'
```

Tuy nhiên chương trình vẫn làm việc. Các câu lệnh dưới đây vẫn thực hiện và cho kết quả như sau:

```
s++;
cout << "\ns= " << s; // In ra s=24
```

Chương trình dưới đây minh họa cách dùng biến tham chiếu đến một phần tử mảng cấu trúc để nhập dữ liệu và thực hiện các phép tính trên các trường của phần tử mảng cấu trúc.

```
#include <iostream.h>
#include <conio.h>
struct TS
{
    char ht[25];
    float t,l,h,td;
};
void main()
```



```
{
    TS ts[10], &h=ts[1]; // h tham chiếu đến ts[1]
    cout << "\n Ho ten: " ;
    cin.get(h.ht, 25) ;
    cout << "Cac diem toan, ly, hoa: ";
    cin >> h.t >> h.l >> h.h ;
    h.td = h.t + h.l + h.h ;
    cout << "\n Ho ten: " << ts[1].ht;
    cout << "\n Tong diem: " << ts[1].td;
    getch();
}
```

1.3. Hằng tham chiếu (const)

Hằng tham chiếu được khai báo theo mẫu:

```
int n = 10 ;
const int &r = n;
```

Cũng giống như biến tham chiếu, hằng tham chiếu có thể tham chiếu đến một biến hoặc một hằng. Ví dụ:

```
int n = 10 ;
const int &r = n ; // Hằng tham chiếu r tham chiếu đến biến n
const int &s=123 ; //Hằng tham chiếu s tham chiếu đến hằng 123
```

Sự khác nhau giữa biến và hằng tham chiếu ở chỗ: Không cho phép dùng hằng tham chiếu để làm thay đổi giá trị của vùng nhớ mà nó tham chiếu.

Ví dụ:

```
int y = 12, z ;
const int &py=y; // Hằng tham chiếu py tham chiếu đến biến y
y++; // Đúng
z = 2*py ; // Đúng z = 26
cout << y << " " << py; // In ra: 13 13
py=py+1; // Sai, Trình biên dịch thông báo lỗi:
// Cannot modify a const object
```

Cách dùng: Hằng tham chiếu cho phép sử dụng giá trị chứa trong một vùng nhớ, nhưng không cho phép thay đổi giá trị này.

Hằng tham chiếu thường được sử dụng làm đối của hàm để cho phép hàm sử dụng giá trị của các tham số trong lời gọi hàm, nhưng tránh không làm thay đổi giá trị của các tham số.

2. Truyền giá trị cho hàm theo tham chiếu

2.1. Hàm trong C

Trong C chỉ có một cách truyền dữ liệu cho hàm theo giá trị :

+ Cấp phát vùng nhớ cho các đối.

+ Gán giá trị các tham số trong lời gọi hàm cho các đối sau đó hàm làm việc trên vùng nhớ của các đối chứ không liên quan gì đến các tham số.

Như vậy chương trình sẽ tạo ra các bản sao (các đối) của các tham số và hàm sẽ thao tác trên các bản sao này, chứ không làm việc trực tiếp với các tham số. Phương pháp này có 2 nhược điểm chính:

Tốn kém về thời gian và bộ nhớ vì phải tạo ra các bản sao. Không thao tác trực tiếp trên các tham số, vì vậy không làm thay đổi được giá trị các tham số.

2.2. Truyền giá trị cho hàm theo tham chiếu

Trong C++ cung cấp thêm cách truyền dữ liệu cho hàm theo tham chiếu bằng cách dùng đối là biến tham chiếu hoặc đối là hằng tham chiếu. Cách này có ưu điểm:

Không cần tạo ra các bản sao của các tham số, do đó tiết kiệm bộ nhớ và thời gian chạy máy.

Hàm sẽ thao tác trực tiếp trên vùng nhớ của các tham số, do đó dễ dàng thay đổi giá trị các tham số khi cần.

2.3. Môi quan hệ giữa đối và tham số trong lời gọi hàm

Nếu đối là biến hoặc hằng tham chiếu kiểu K thì tham số (trong lời gọi hàm) phải là biến hoặc phân tử mảng kiểu K. Ví dụ:

+ Đối là biến hoặc hằng tham chiếu kiểu double, thì tham số là biến hoặc phân tử mảng kiểu double

+ Đối là biến hoặc hằng tham chiếu kiểu cấu trúc, thì tham số là biến hoặc phân tử mảng kiểu cấu trúc

2.4. Các chương trình minh họa

/*

Chương trình sau được tổ chức thành 3 hàm:

Nhập dãy số double

Hoán vị 2 biến double

Sắp xếp dãy số double theo thứ tự tăng dần

Chương trình sẽ nhập một dãy số và in dãy sau khi sắp xếp

*/

#include <iostream.h>

#include <conio.h>

#include <stdio.h>

void nhapds(double *a, int n)

{

for (int i=1; i<= n ; ++i)

{

cout << "nPhan tu thu " << i << " : " ;

cin >> a[i] ;

}

}

void hv(double &x, double &y)

{

double tg=x; x=y; y= tg;

}

void sapxep(double * a, int n)

```
{
    for (int i=1; i <= n-1 ; ++i)
        for (int j=i+1 ; j<=n ; ++j)
            if (a[i] > a[j])
                hv(a[i],a[j]);
}
void main()
```

```
{
    double x[100];
    int i, n;
    cout << "\n N= ";
    cin >> n;
    nhapds(x,n);
    sapxep(x,n);
    for (i=1;i<=n; ++i)
        printf("\n%0.1lf",x[i]);
    getch();
}
```

/*

Chương trình sau gồm các hàm:

- Nhập dãy cấu trúc (mỗi cấu trúc chứa dữ liệu một thí sinh)
- Hoán vị 2 biến cấu trúc
- Sắp xếp dãy thí sinh theo thứ tự giảm của tổng điểm
- In một cấu trúc (in họ tên và tổng điểm)

Chương trình sẽ nhập dữ liệu một danh sách thí sinh, nhập điểm chuẩn và in danh sách thí sinh trúng tuyển

*/

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

```
#include <iomanip.h>
```

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

```
struct TS
```

```
{
    char ht[20];
```

```
    float t,l,h,td;
```

```
};
```

```
void ints(const TS &ts)
```

```
{
    cout << setiosflags(ios::showpoint) << setprecision(1) ;
    cout << "\nHo ten: " << setw(20) << ts.ht << setw(6) << ts.td ;
}
```

```
void nhapsl(TS *ts,int n)
{
    for (int i=1;i<=n;++i)
    {
        cout << "\n Thi sinh " << i ;
        cout << "\n Ho ten: " ;
        cin.ignore(1);
        cin.get(ts[i].ht,25) ;
        cout << "Cac diem toan, ly, hoa: ";
        cin >> ts[i].t >> ts[i].l >> ts[i].h ;
        ts[i].td = ts[i].t + ts[i].l + ts[i].h ;
    }
}

void hvts(TS &ts1, TS &ts2)
{
    TS tg=ts1;
    ts1=ts2;
    ts2=tg;
}

void sapxep(TS *ts,int n)
{
    for (int i=1;i<=n-1;++i)
        for (int j=i+1;j<=n;++j)
            if (ts[i].td < ts[j].td)
                hvts(ts[i],ts[j]);
}

void main()
{
    TS ts[100];
    int n,i;
    clrscr();
    cout << " So thi sinh: " ;
    cin >> n ;
    nhapsl(ts,n);
    sapxep(ts,n) ;
    float dc;
    cout << " Diem chuan: " ;
    cin >> dc;
    cout << "\n\nDanh sach trung tuyen\n" ;
    for (i=1;i<=n;++i)
        if (ts[i].td >= dc)
```

```
        ints(ts[i]);
    else
        break;
    getch();
}

/*
Chương trình sau gồm các hàm:
Nhập một ma trận thực cấp mxn
In một ma trận thực dưới dạng bảng
Tìm phần tử lớn nhất và phần tử nhỏ nhất của dãy số thực;
Chương trình sẽ nhập một ma trận, in ma trận vừa nhập và in các phần tử lớn nhất và nhỏ nhất
trên mỗi hàng của ma trận
*/
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <stdio.h>
void nhapmt(float a[20][20], int m, int n)
{
    for (int i=1 ; i<= m ; ++i)
        for (int j=1; j<= n ; ++j)
        {
            cout << "na[" << i << ", " << j << "] = ";
            cin >> a[i][j] ;
        }
}
void inmt(float a[20][20], int m, int n)
{
    cout << setiosflags(ios::showpoint) << setprecision(1);
    for (int i=1 ; i<= m ; ++i)
        for (int j=1; j<= n ; ++j)
        {
            if (j==1) cout << "n";
            cout << setw(6) << a[i][j] ;
        }
}
void maxmins(float *x, int n,int &vtmax, int &vtmin)
{
    vtmax = vtmin = 1 ;
    for (int i=2; i<=n ; ++i)
```

```
{
    if (x[i] > x[vtmax]) vtmax = i;
    if (x[i] < x[vtmin]) vtmin = i;
}
}

void main()
{
    float a[20][20];
    int m, n;
    cout << "\n So hang va so cot ma tran: ";
    cin >> m >> n;
    nhapmt(a,m,n);
    clrscr();
    inmt(a,m,n);
    float *p = (float*)a;
    int vtmax, vtmin;
    for (int i=1;i<=m;++i)
    {
        p = ((float*)a) + i*20 ;
        maxminds(p , n, vtmax, vtmin) ;
        printf("\nHang %d Phan tu max= %6.1f tai cot
        %d",i,p[vtmax],vtmax);
        printf("\n Phan tu min= %6.1f tai cot %d", p[vtmin],vtmin);
    }
    getch();
}
```

3. Hàm trả về các tham chiếu

Hàm có thể có kiểu tham chiếu và trả về giá trị tham chiếu. Khi đó có thể dùng hàm để truy nhập đến một biến hoặc một phần tử mảng nào đó. Dưới đây là một số ví dụ.

Ví dụ 1 trình bày một hàm trả về một tham chiếu đến một biến toàn bộ. Do đó có thể dùng hàm để truy nhập đến biến này.

```
#include <iostream.h>
#include <conio.h>
int z ;
int &f() // Hàm trả về một bí danh của biến toàn bộ z
{
    return z;
}

void main(void)
```

```
{  
    f()=50; // z = 50  
    cout << "\nz= " << z;  
    getch();  
}
```

Ví dụ 2 trình bày một hàm trả về bí danh của một biến cấu trúc toàn bộ. Khác với ví dụ trên, ở đây không dùng hàm một cách trực tiếp mà gán hàm cho một biến tham chiếu, sau đó dùng biến tham chiếu này để truy nhập đến biến cấu trúc toàn bộ.

```
#include <iostream.h>  
#include <conio.h>  
struct TS  
{  
    char ht[25];  
    float t,l,h,td;  
};  
TS ts;  
TS &f()  
{  
    return ts;  
}  
void main()  
{  
    TS &h=f(); // h tham chiếu đến biến ts  
    cout << "\n Ho ten: " ;  
    cin.get(h.ht,25) ;  
    cout << "Cac diem toan, ly, hoa: ";  
    cin >> h.t >> h.l >> h.h ;  
    h.td = h.t + h.l + h.h ;  
    cout << "\n Ho ten: " << ts.ht;  
    cout << "\n Tong diem: " << ts.td;  
    getch();  
}
```

Ví dụ 3 trình bày một hàm trả về bí danh của một phần tử mảng cấu trúc toàn bộ.

Hàm sẽ kiểm tra xem chỉ số mảng có vượt ra ngoài miền quy định hay không. Sau đó dùng hàm này để truy nhập đến các phần tử mảng cấu trúc.

```
#include <iostream.h>  
#include <conio.h>  
#include <stdlib.h>  
struct TS
```

```
{
    char ht[25];
    float t,l,h,td;
};
TS *ts;
void cap_phat_bo_nho_nhapsl(int n)
{
    ts = new TS[n+1] ;
    if (ts==NULL)
    {
        cout << "Loi cap phat bo nho " ;
        exit(1);
    }
    for (int i=1;i<=n;++i)
    {
        TS &h=ts[i];
        cout << "\nThi sinh thu " << i ;
        cout << "\n Ho ten: " ;
        cin.ignore(1);
        cin.get(h.ht,25) ;
        cout << "Cac diem toan, ly, hoa: ";
        cin >> h.t >> h.l >> h.h ;
        h.td = h.t + h.l + h.h ;
    }
}
TS &f(int i, int n) // Cho bi danh ts[i]
{
    if (i<1 || i>n)
    {
        cout << "Chi so mang khong hop le " ;
        exit(1);
    }
    return ts[i];
}
void main()
{
    int n, i ;
    cout << "\n So thi sinh : " ;
    cin >> n;
    cap_phat_bo_nho_nhapsl(n);
    while (1)
    {
        cout << "\nCan xem thi sinh thu may: " ;
```



```
cout << "\nChon so tu 1 den " << n << " (bam sai ket thuc CT) ";
cin >> i;
TS &h=f(i,n);
cout << "\n Ho ten: " << h.ht;
cout << "\n Tong diem: " << h.td;
}
}
```

4. Đối có giá trị mặc định

4.1. Thế nào là đối mặc định

Một trong các khả năng mạnh của C++ là nó cho phép xây dựng hàm với các đối có giá trị mặc định. Thông thường số tham số trong lời gọi hàm phải bằng số đối của hàm. Mỗi đối sẽ được khởi gán giá trị theo tham số tương ứng của nó. Trong C++ cho phép tạo giá trị mặc định cho các đối. Các đối này có thể có hoặc không có tham số tương ứng trong lời gọi hàm. Khi không có tham số tương ứng, đối được khởi gán bởi giá trị mặc định.

Ví dụ hàm delay với đối số mặc định được viết theo một trong 2 cách sau:

Cách 1 (Không khai báo nguyên mẫu):

```
void delay(int n=1000)
{
    for (int i=0 ; i<n ; ++i)
        ;
}
```

Cách 2 (Có khai báo nguyên mẫu):

```
void delay(int n=1000) ;
void delay(int n)
{
    for (int i=0 ; i<n ; ++i)
        ;
}
```

Cách dùng:

- + Cung cấp giá trị cho đối n (Có tham số trong lời gọi hàm)
 delay(5000) ; // Đối n = 5000
- + Sử dụng giá trị mặc định của đối (Không có tham số trong lời gọi)
 delay() ; // Đối n = 1000

4.2. Quy tắc xây dựng hàm với đối mặc định

+ Các đối mặc định cần phải là các đối cuối cùng tính từ trái sang phải. Giả sử có 5 đối theo thứ tự từ trái sang phải là

d1, d2, d3, d4, d5

Khi đó:

nếu một đối mặc định thì phải là d5

nếu hai đối mặc định thì phải là d4, d5

nếu ba đối mặc định thì phải là d3, d4, d5

...

Các ví dụ sai:

d3 và d5 mặc định (khi đó d4 cũng phải mặc định)

d3 và d4 mặc định (khi đó d5 cũng phải mặc định)

+ Khi xây dựng hàm, nếu sử dụng khai báo nguyên mẫu, thì các đối mặc định cần được khởi gán trong nguyên mẫu, ví dụ:

```
// Khởi gán giá trị cho 3 đối mặc định d3, d4 và d5)
```

```
void f(int d1, float d2, char *d3="HA NOI",
```

```
    int d4 = 100, double d5=3.14) ;
```

```
void f(int d1, float d2, char *d3, int d4, double d5)
```

```
{
```

```
    // Các câu lệnh trong thân hàm
```

```
}
```

Không được khởi gán lại cho các đối mặc định trong dòng đầu của định nghĩa hàm. Nếu vi phạm điều này thì Chương trình dịch sẽ thông báo lỗi.

+ Khi xây dựng hàm, nếu không khai báo nguyên mẫu, thì các đối mặc định được khởi gán trong dòng đầu của định nghĩa hàm, ví dụ:

```
// Khởi gán giá trị cho 3 đối mặc định d3, d4 và d5)
```

```
void f(int d1, float d2, char *d3="HA NOI",
```

```
    int d4 = 100, double d5=3.14)
```

```
{
```

```
    // Các câu lệnh trong thân hàm
```

```
}
```

+ Giá trị dùng để khởi gán cho đối mặc định

Có thể dùng các hằng, các biến toàn bộ, các hàm để khởi gán cho đối mặc định, ví dụ:

```
int MAX = 10000;
```

```
void f(int n, int m = MAX, int xmax = getmaxxx(),
```

```
    int ymax = getmaxy() ) ;
```

4.3. Cách sử dụng hàm có đối mặc định

Lời gọi hàm cần viết theo quy định sau:

Các tham số thiếu vắng trong lời gọi hàm phải tương ứng với các đối mặc định cuối cùng (tính từ trái sang phải).

Nói cách khác: Đã dùng giá trị mặc định cho một đối (tất nhiên phải là đối mặc định) thì cũng phải sử dụng giá trị mặc định cho các đối còn lại.

Ví dụ với hàm có 3 đối mặc định:

```
void f(int d1, float d2, char *d3="HA NOI",
```

```
    int d4 = 100, double d5=3.14) ;
```

Thì các lời gọi sau là đúng:

```
f(3,3.4,"ABC",10,1.0) ; // Đầy đủ tham số
f(3,3.4,"ABC") ;       // Thiếu 2 tham số cuối
f(3,3.4) ;              // Thiếu 3 tham số cuối
```

Các lời gọi sau là sai:

```
f(3) ;                  // Thiếu tham số cho đối không mặc định d2
f(3,3.4, ,10) ;         // Đã dùng giá trị mặc định cho d3, thì cũng
                        // phải dùng giá trị mặc định cho d4 và d5
```

4.4. Các ví dụ

Hàm ht (bên dưới) dùng để hiển thị chuỗi ký tự dc trên n dòng màn hình. Các đối dc và n đều có giá trị mặc định.

```
#include <conio.h>
#include <iostream.h>
void ht(char *dc="HA NOI",int n=10) ;
void ht(char *dc , int n )
{
    for (int i=0;i<n;++i)
        cout << "\n" << dc;
}
void main()
{
    ht(); // In dòng chữ "HA NOI" trên 10 dòng
    ht("ABC",3); // In dòng chữ "ABC" trên 3 dòng
    ht("DEF"); // In dòng chữ "DEF" trên 10 dòng
    getch();
}
```

Ví dụ dưới đây trình bày hàm hiển thị một chuỗi str trên màn hình đồ hoạ, tại vị trí (x,y) và có màu m. Các đối x, y và m là mặc định. Dùng các hàm getmaxx() và getmaxy() để khởi gán cho x, y. Dùng hằng RED gán cho m.

```
#include <conio.h>
#include <graphics.h>
void hiendc(char *str, int x=getmaxx()/2,
            int y = getmaxy()/2, int m=RED);
void hiendc(char *str, int x,int y, int m)
{
    int mau_ht = getcolor(); // Lưu màu hiện tại
    setcolor(m);
    outtextxy(x,y,str) ;
    setcolor(mau_ht); // Khôi phục màu hiện tại
```

```
}  
void main()  
{  
    int mh=0, mode=0;  
    initgraph(&mh,&mode,"");  
    setbkcolor(BLUE);  
    hiendc("HELLO"); // HELLO màu đỏ giữa màn hình  
    hiendc("CHUC MUNG",1,1); // CHUC MUNG màu đỏ tại vị  
                           // trí (1,1)  
    hiendc("CHAO",1,400,YELLOW); // CHAO màu vàng tại vị  
                           // trí (1,400)  
    getch();  
}
```

Ví dụ dưới đây trình bày hàm tính tích phân xác định gồm 3 đối: f là hàm cần tính tích phân, a và b là các cận dưới và trên ($a < b$). Cả 3 đối f, a và b đều mặc định. Giá trị mặc định của con trỏ hàm f là địa chỉ của hàm bp (bình phương), của a bằng 0, của b bằng 1.

```
#include <conio.h>  
#include <iostream.h>  
#include <iomanip.h>  
#include <math.h>  
double bp(double x);  
double tp( double (*f)(double)=bp,double a=0.0, double b=1.0 );  
double bp(double x)  
{  
    return x*x;  
}  
double tp(double (*f)(double), double a, double b )  
{  
    int n=1000;  
    double s=0.0, h=(b-a)/n;  
    for (int i=0; i<n ; ++i)  
        s+= f(a+i*h + h) + f(a+i*h ) ;  
    return s*h/2;  
}  
void main()  
{  
    clrscr();  
    cout << setiosflags(ios::showpoint) << setprecision(2);  
    cout << "\nTích phân từ 0 đến 1 của  $x^2$ = " << tp() ;  
    cout << "\nTích phân từ 0 đến 1 của  $\exp(x)$ = " << tp(exp);
```

```
cout << "\nTích phân từ 0 đến PI/2 của sin(x) " <<
                                     tp(sin,0,3.14/2);

getch();
}
```

5. Các hàm trực tuyến (inline)

5.1. Ưu, nhược điểm của hàm

Việc tổ chức chương trình thành các hàm có 2 ưu điểm rõ rệt : Thứ nhất là chia chương trình thành các đơn vị độc lập, làm cho chương trình được tổ chức một cách khoa học dễ kiểm soát dễ phát hiện lỗi, dễ phát triển, mở rộng.

Thứ hai là giảm được kích thước chương trình, vì mỗi đoạn chương trình thực hiện nhiệm vụ của hàm được thay bằng một lời gọi hàm.

Tuy nhiên hàm cũng có nhược điểm là làm chậm tốc độ chương trình do phải thực hiện một số thao tác có tính thủ tục mỗi khi gọi hàm như: Cấp phát vùng nhớ cho các đối và biến cục bộ, truyền dữ liệu của các tham số cho các đối, giải phóng vùng nhớ trước khi thoát khỏi hàm.

Các hàm trực tuyến trong C++ cho khả năng khắc phục được nhược điểm nói trên.

5.2. Các hàm trực tuyến

Để biến một hàm thành trực tuyến ta viết thêm từ khoá

inline

vào trước khai báo nguyên mẫu hàm. Nếu không dùng nguyên mẫu thì viết từ khoá này trước dòng đầu tiên của định nghĩa hàm. Ví dụ:

```
inline float f(int n, float x);
float f(int n, float x)
{
    // Các câu lệnh trong thân hàm
}
```

hoặc

```
inline float f(int n, float x)
{
    // Các câu lệnh trong thân hàm
}
```

Chú ý: Trong mọi trường hợp, từ khoá inline phải xuất hiện trước các lời gọi hàm thì Trình biên dịch mới biết cần xử lý hàm theo kiểu inline.

Ví dụ hàm f trong chương trình sau sẽ không phải là hàm trực tuyến vì từ khoá inline viết sau lời gọi hàm:

```
#include <conio.h>
#include <iostream.h>
void main()
{
    int s ;
    s = f(5,6);
}
```

```
cout << s ;
getch();
}
inline int f(int a, int b)
{
    return a*b;
}
```

Chú ý: Trong C++ , nếu hàm được xây dựng sau lời gọi hàm thì bắt buộc phải khai báo nguyên mẫu hàm trước lời gọi. Trong ví dụ trên, Trình biên dịch C++ sẽ bắt lỗi vì thiếu khai báo nguyên mẫu hàm f .

5.3. Cách biên dịch hàm trực tuyến

Chương trình dịch xử lý các hàm inline như các macro (được định nghĩa trong lệnh #define), nghĩa là nó sẽ thay mỗi lời gọi hàm bằng một đoạn chương trình thực hiện nhiệm vụ của hàm. Cách này làm cho chương trình dài ra, nhưng tốc độ chương trình tăng lên do không phải thực hiện các thao tác có tính thủ tục khi gọi hàm.

5.4. So sánh macro và hàm trực tuyến

Dùng macro và hàm trực tuyến đều dẫn đến hiệu quả tương tự, tuy nhiên người ta thích dùng hàm trực tuyến hơn, vì cách này đảm bảo tính cấu trúc của chương trình, dễ sử dụng và tránh được các sai sót lặt vặt thường gặp khi dùng #define (như thiếu các dấu ngoặc, dấu chấm phẩy)

5.5. Khi nào thì nên dùng hàm trực tuyến

Phương án dùng hàm trực tuyến rút ngắn được thời gian chạy máy nhưng lại làm tăng khối lượng bộ nhớ chương trình (nhất là đối với các hàm trực tuyến có nhiều câu lệnh). Vì vậy chỉ nên dùng phương án trực tuyến đối với các hàm nhỏ.

5.6. Sự hạn chế của Trình biên dịch

58

tuyến.

Chú ý rằng từ khoá inline chỉ là một sự gợi ý cho Trình biên dịch chứ không phải là một mệnh lệnh bắt buộc.

Có một số hàm mà các Trình biên dịch thường không xử lý theo cách inline như các hàm chứa biến static, hàm chứa các lệnh chu trình hoặc lệnh goto hoặc lệnh switch, hàm đệ quy. Trong trường hợp này từ khoá inline lẽ dĩ nhiên bị bỏ qua.

Thậm chí từ khoá inline vẫn bị bỏ qua ngay cả đối với các hàm không có những hạn chế nêu trên nếu như Trình biên dịch thấy cần thiết (ví dụ đã có quá nhiều hàm inline làm cho bộ nhớ chương trình quá lớn)

Ví dụ: Chương trình sau sử dụng hàm inline tính chu vi và diện tích của hình chữ nhật:

Phương án 1: Không khai báo nguyên mẫu. Khi đó hàm dtcvhn phải đặt trên hàm main.

```
#include <conio.h>
#include <iostream.h>
inline void dtcvhn(int a, int b, int &dt, int &cv)
{
    dt=a*b;
    cv=2*(a+b);
}
```

```
}
void main()
{
    int a[20],b[20],cv[20],dt[20],n;
    cout << "\n So hình chu hat: " ;
    cin >> n;
    for (int i=1;i<=n;++i)
    {
        cout << "\nNhập 2 cạnh của hình chu nhật thu " <<i<< ": ";
        cin >> a[i] >> b[i] ;
        dtcvhcn(a[i],b[i],dt[i],cv[i]);
    }
    clrscr();
    for (i=1;i<=n;++i)
    {
        cout << "\n Hình chu nhật thu " << i << " : ";
        cout << "\nDo dài 2 cạnh= " << a[i] << " và " << b[i] ;
        cout << "\nDiện tích= " << dt[i] ;
        cout << "\nChu vi= " << cv[i] ;
    }
    getch();
}
```

Phương án 2: Sử dụng khai báo nguyên mẫu. Khi đó từ khoá inline đặt trước nguyên mẫu.

Chú ý: Không được đặt inline trước định nghĩa hàm. Trong chương trình dưới đây nếu đặt inline trước định nghĩa hàm thì hậu quả như sau: Chương trình vẫn dịch thông, nhưng khi chạy thì chương trình bị quẩn, không thoát được.

```
#include <conio.h>
#include <iostream.h>
inline void dtcvhcn(int a, int b, int &dt, int &cv) ;
void main()
{
    int a[20],b[20],cv[20],dt[20],n;
    cout << "\n So hình chu hat: " ;
    cin >> n;
    for (int i=1;i<=n;++i)
    {
        cout << "\nNhập 2 cạnh của hình chu nhật thu " <<i<< ": ";
        cin >> a[i] >> b[i] ;
        dtcvhcn(a[i],b[i],dt[i],cv[i]);
    }
    clrscr();
}
```

```
for (i=1;i<=n;++i)
{
    cout << "\n Hình chu nhật thu " << i << " : ";
    cout << "\nDo dai 2 canh= " << a[i] << " va " << b[i] ;
    cout << "\nDien tích= " << dt[i] ;
    cout << "\nChu vi= " << cv[i] ;
}
getch();
}

void dtcvhcn(int a, int b, int &dt, int &cv)
{
    dt=a*b;
    cv=2*(a+b);
}
```

6. Định nghĩa chồng các hàm (Overloading)

6.1. Khái niệm về định nghĩa chồng

Định nghĩa chồng (hay còn gọi sự tải bội) các hàm là dùng cùng một tên để định nghĩa các hàm khác nhau. Đây là một mở rộng rất có ý nghĩa của C++.

Như đã biết, trong C và các ngôn ngữ khác (như PASCAL, FORTRAN,...) mỗi hàm đều phải có một tên phân biệt. Đôi khi đây là một sự hạn chế lớn, vì phải dùng nhiều hàm khác nhau để thực hiện cùng một công việc. Ví dụ để lấy giá trị tuyệt đối trong C cần dùng tới 3 hàm khác nhau:

```
int abs(int i); // Lấy giá trị tuyệt đối giá trị kiểu int
long labs(long l); // Lấy giá trị tuyệt đối giá trị kiểu long
double fabs(double d); // Lấy giá trị tuyệt đối giá trị kiểu double
```

Nhờ khả năng định nghĩa chồng, trong C++ có thể dùng chung một tên cho cả 3 hàm trên như sau:

```
int abs(int i) ; // Lấy giá trị tuyệt đối giá trị kiểu int
long abs(long l) ; // Lấy giá trị tuyệt đối giá trị kiểu long
double abs(double d) ; // Lấy giá trị tuyệt đối giá trị kiểu double
```

6.2. Yêu cầu về các hàm định nghĩa chồng

Khi dùng cùng một tên để định nghĩa nhiều hàm, Trình biên dịch C++ sẽ dựa vào sự khác nhau về tập đối của các hàm này để đổi tên các hàm. Như vậy, sau khi biên dịch mỗi hàm sẽ có một tên khác nhau.

Từ đó cho thấy: các hàm được định nghĩa trùng tên phải có tập đối khác nhau (về số lượng hoặc kiểu). Nếu 2 hàm hoàn toàn trùng tên và trùng đối thì Trình biên dịch sẽ không có cách nào phân biệt được. Ngay cả khi 2 hàm này có kiểu khác nhau thì Trình biên dịch vẫn báo lỗi. Ví dụ sau xây dựng 2 hàm cùng có tên là f và cùng có một đối nguyên a, nhưng kiểu hàm khác nhau. Hàm thứ nhất kiểu nguyên (trả về a*a), hàm thứ hai kiểu void (in giá trị a). Chương trình sẽ bị thông báo lỗi khi biên dịch (bạn hãy thử xem sao)

```
#include <conio.h>
#include <iostream.h>
int f(int a);
```



```
void f(int a);
int f(int a)
{
    return a*a;
}
void f(int a)
{
    cout << "\n " << a ;
}
void main()
{
    int b=f(5);
    f(b);
    getch();
}
```

6.3. Sử dụng các hàm định nghĩa chồng

Khi gặp một lời gọi, Trình biên dịch sẽ căn cứ vào số lượng và kiểu của các tham số để gọi hàm có đúng tên và đúng bộ đối số tương ứng. Ví dụ:

```
abs(123); // Tham số kiểu int, gọi hàm int abs(int i) ;
abs(123L); // Tham số kiểu long, gọi hàm long abs(long l);
abs(3.14); //Tham số kiểu double, gọi hàm double abs(double d);
```

Khi không có hàm nào có bộ đối cùng kiểu với bộ tham số (trong lời gọi), thì Trình biên dịch sẽ chọn hàm nào có bộ đối gần kiểu nhất (phép chuyển kiểu dễ dàng nhất). Ví dụ:

```
abs('A') ; // Tham số kiểu char, gọi hàm int abs(int i) ;
abs(3.14F); // Tham số kiểu float, gọi hàm double abs(double d);
```

6.4. Nên sử dụng phép định nghĩa chồng các hàm như thế nào

Như đã nói ở trên, khi xây dựng cũng như sử dụng các hàm trùng tên, Trình biên dịch C++ đã phải suy đoán và giải quyết nhiều trường hợp khá nhập nhằng. Vì vậy không nên lạm dụng quá đáng khả năng định nghĩa chồng, vì điều đó làm cho chương trình khó kiểm soát và dễ dẫn đến sai sót. Việc định nghĩa chồng sẽ hiệu quả hơn nếu được sử dụng theo các lời khuyên sau:

- + Chỉ nên định nghĩa chồng các hàm thực hiện những công việc như nhau nhưng trên các đối tượng có kiểu khác nhau. Ví dụ trong chương trình cần xây dựng các hàm: cộng 2 ma trận vuông kiểu double, cộng 2 ma trận vuông kiểu int, cộng 2 ma trận chữ nhật kiểu double, cộng 2 ma trận chữ nhật kiểu int, thì 4 hàm trên nên định nghĩa chồng (đặt cùng tên).

- + Nên dùng các phép chuyển kiểu (nếu cần) để bộ tham số trong lời gọi hoàn toàn trùng kiểu với bộ đối số của một hàm được định nghĩa chồng. Vì như thế mới tránh được sự nhập nhằng cho Trình biên dịch và Trình biên dịch sẽ chọn đúng hàm cần gọi.

6.5. Lấy địa chỉ các hàm trùng tên

Giả sử có 4 hàm đều có tên là `tinh_max` được khai báo như sau:

```
int tinh_max(int a, int b, int c) ; // Max của 3 số nguyên
double tinh_max(double a, double b, double c); // Max của 3 số // thực
```

```
int tinh_max(int *a, int n) ; // Max của một dãy số nguyên
double tinh_max(double *a, int n) ; //Max của một dãy số thực
```

Vấn đề đặt ra là làm thế nào lấy được địa chỉ của mỗi hàm. Câu trả lời như sau:

Để lấy địa chỉ của một hàm, ta khai báo một con trỏ hàm có kiểu và bộ đối như hàm cần lấy địa chỉ. Sau đó gán tên hàm cho con trỏ hàm. Ví dụ:

```
int (*f1)(int , int, int );
f1 = tinh_max ;           // Lấy địa chỉ của hàm thứ nhất
double (*f2)(double , double, double);
f2 = tinh_max ;           // Lấy địa chỉ của hàm thứ hai
int (*f3)(int *, int );
f3 = tinh_max ;           // Lấy địa chỉ của hàm thứ ba
double (*f4)(double *, int );
f4 = tinh_max ;           // Lấy địa chỉ của hàm thứ tư
```

6.6. Các ví dụ

Ví dụ 1: Chương trình giải bài toán tìm max của một dãy số nguyên và max của một dãy số thực. Trong chương trình có 6 hàm. Hai hàm dùng để nhập dãy số nguyên và dãy số thực có tên chung là nhapds. Bốn hàm: tính max 2 số nguyên, tính max 2 số thực, tính max của dãy số nguyên, tính max của dãy số thực được đặt chung một tên là max.

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
void nhapds(int *x, int n);
void nhapds(double *x, int n);
int max(int x, int y);
double max(double x, double y);
int max(int *x, int n);
double max(double *x, int n);
void nhapds(int *x, int n)
{
    for (int i=1;i<=n;++i)
    {
        cout << "Phan tu " << i << " = ";
        cin >> x[i] ;
    }
}
void nhapds(double *x, int n)
{
    for (int i=1;i<=n;++i)
    {
        cout << "Phan tu " << i << " = ";
        cin >> x[i] ;
    }
}
```

```
int max(int x, int y)
{
    return x>y?x:y ;
}
double max(double x, double y)
{
    return x>y?x:y ;
}
int max(int *x, int n)
{
    int s=x[1];
    for (int i=2;i<=n;++i)
        s = max(s,x[i]);
    return s;
}
double max(double *x, int n)
{
    double s=x[1];
    for (int i=2;i<=n;++i)
        s = max(s,x[i]);
    return s;
}
void main()
{
    int a[20] , n , ni, nd, maxi ;
    double x[20] , maxd ;
    clrscr();
    cout << "\nSo phan tu nguyen ni = " ;
    cin >> ni ;
    cout << "Nhap day so nguyen\n " ;
    nhapds(a,ni);
    cout << "\nSo phan tu thuc nd = " ;
    cin >> nd ;
    cout << "Nhap day so thuc\n " ;
    nhapds(x,nd);
    maxi = max(a,ni);
    maxd = max(x,nd);
    cout << "\nMax cua day nguyen = " << maxi ;
    cout << "\nMax cua day thuc  = " << maxd ;
    getch();
}
```

}

Ví dụ 2:

Chương trình sau thực hiện phép nhân ma trận:

$$D = A * B * C$$

trong đó A, B là các ma trận vuông, C là ma trận chữ nhật. Trong chương trình có 3 cặp hàm trùng tên để thực hiện 3 nhiệm vụ (nhưng trên 2 đối tượng khác nhau là ma trận vuông và chữ nhật): Nhập ma trận, nhân 2 ma trận và in ma trận.

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
typedef int MT[20][20];
void nhapmt(MT a, char *ten, int m, int n);
void inmt(MT a, char *ten, int m, int n);
void nhanmt(MT a, MT b, MT c, int m, int n, int p);
void nhapmt(MT a, char *ten, int n);
void inmt(MT a, char *ten, int n);
void nhanmt(MT a, MT b, MT c, int n);
void nhapmt(MT a, char *ten, int m, int n)
{
    for (int i=1; i<=m; ++i)
        for (int j=1; j<=n; ++j)
        {
            cout << "\n" << ten << "[" << i << ", " << j << "] = ";
            cin >> a[i][j];
        }
}

void nhapmt(MT a, char *ten, int n)
{
    nhapmt(a, ten, n, n);
}

void inmt(MT a, char *ten, int m, int n)
{
    cout << "\nMa tran: " << ten;
    for (int i=1; i<=m; ++i)
    {
        cout << "\n";
        for (int j=1; j<=n; ++j)
            cout << setw(6) << a[i][j];
    }
}

void inmt(MT a, char *ten, int n)
```

```
{
    inmt(a,ten,n,n) ;
}
void nhanmt(MT a,MT b, MT c, int m, int n, int p)
{
    for (int i=1;i<=m;++i)
        for (int j=1;j<=p;++j)
        {
            c[i][j]=0;
            for (int k=1;k<=n;++k)
                c[i][j] += a[i][k] * b[k][j];
        }
}
void nhanmt(MT a,MT b, MT c, int n)
{
    nhanmt(a,b,c,n,n, n) ;
}
void main()
{
    MT a,b,c,d; // d= abc
    MT u;
    clrscr();
    nhapmt(a,"A",2);
    nhapmt(b,"B",2);
    nhapmt(c,"C",2,3);
    nhanmt(a,b,u,2);
    nhanmt(u,c,d,2,2,3);
    inmt(a,"A",2);
    inmt(b,"B",2);
    inmt(u,"U = A*B",2);
    inmt(c,"C",2,3);
    inmt(d,"D = U*C",2,3);
    getch();
}
```

7. Định nghĩa chồng các toán tử

7.1. Các phép toán trong C và C++

Trong C và C++ có khá nhiều các phép toán dùng để thực hiện các thao tác trên các kiểu dữ liệu chuẩn. Ví dụ các phép số học: + - * / áp dụng cho các kiểu dữ liệu nguyên, thực. Phép lấy phần dư % áp dụng đối với kiểu nguyên.

7.2. Thực hiện các phép toán trên các kiểu dữ liệu không chuẩn trong C

Việc thực hiện các phép toán trên các đối tượng tự định nghĩa (như mảng, cấu trúc) là nhu cầu bắt buộc của thực tế. Chẳng hạn cần thực hiện các phép số học trên số phức, trên phân số, trên đa

thức, trên véc tơ, trên ma trận. Để đáp ứng yêu cầu này, ta sử dụng các hàm trong C. Ví dụ sau đây là một chương trình C gồm các hàm nhập phân số, in phân số và thực hiện các phép cộng trừ nhân chia phân số. Chương trình sẽ nhập 5 phân số: p, q, z, u, v và tính phân số s theo công thức:

$$s = (p - q * z) / (u + v)$$

```
#include <conio.h>
#include <stdio.h>
#include <math.h>
typedef struct
{
    int a,b;
} PS;
void nhap(PS *p);
void in(PS p);
int uscln(int x, int y);
PS rutgon(PS p);
PS cong(PS p1, PS p2);
PS tru(PS p1, PS p2);
PS nhan(PS p1, PS p2);
PS chia(PS p1, PS p2);
void nhap(PS *p)
{
    int t, m;
    printf("\nTu va mau: ");
    scanf("%d%d", &t, &m);
    p->a = t; p->b = m;
}
void in(PS p)
{
    printf(" %d/%d",p.a,p.b);
}
int uscln(int x, int y)
{
    x=abs(x); y=abs(y);
    if (x*y==0) return 1;
    while (x!=y)
        if (x>y) x-=y;
        else y-=x;
    return x;
}
PS rutgon(PS p)
{

```

```
    PS q;
    int x;
    x=uscln(p.a,p.b);
    q.a = p.a / x ;
    q.b = p.b / x ;
    return q;
}
PS cong(PS p1, PS p2)
{
    PS q;
    q.a = p1.a*p2.b + p2.a*p1.b;
    q.b = p1.b * p2.b ;
    return rutgon(q);
}
PS tru(PS p1, PS p2)
{
    PS q;
    q.a = p1.a*p2.b - p2.a*p1.b;
    q.b = p1.b * p2.b ;
    return rutgon(q);
}
PS nhan(PS p1, PS p2)
{
    PS q;
    q.a = p1.a * p2.a ;
    q.b = p1.b * p2.b ;
    return rutgon(q);
}
PS chia(PS p1, PS p2)
{
    PS q;
    q.a = p1.a * p2.b ;
    q.b = p1.b * p2.a ;
    return rutgon(q);
}
void main()
{
    PS p, q, z, u, v ;
    PS tu, mau, s;
```

```
printf("\n Nhap phan so p: "); nhap(&p);
printf("\n Nhap phan so q: ");nhap(&q);
printf("\n Nhap phan so z: ");nhap(&z);
printf("\n Nhap phan so u: ");nhap(&u);
printf("\n Nhap phan so v: ");nhap(&v);
tu = nhan(q,z);
tu = tru(p,tu) ;
mau = cong(u,v) ;
s = chia(tu,mau);
printf("\n Phan so s = "); in(s);
getch();
}
```

Nhận xét: Việc sử dụng các hàm để thực hiện các phép tính không được tự nhiên và tỏ ra dài dòng. Ví dụ để thực hiện một công thức

$$s = (p - q * z) / (u + v)$$

phải dùng 2 biến trung gian và 4 lời gọi hàm. Câu hỏi đặt ra là có cách nào để chỉ cần viết đúng công thức toán học, mà vẫn nhận được kết quả mong muốn hay không?

Trong C++ có thể đáp ứng được mong muốn này bằng cách sử dụng các phép toán chuẩn của nó cho các kiểu dữ liệu tự định nghĩa (mảng, cấu trúc, ...). Nói cách khác C++ cho phép dùng các phép toán để định nghĩa các hàm, mà ta thường gọi là định nghĩa chồng các toán tử (hay còn gọi: Sự tải bội các toán tử).

7.3. Cách định nghĩa chồng các toán tử

7.3.1. Tên hàm toán tử: Gồm từ khoá operator và tên phép toán, ví dụ:

operator+ (định nghĩa chồng phép +)
operator- (định nghĩa chồng phép -)

7.3.2. Các đối của hàm toán tử:

a. Với các phép toán có 2 toán hạng, thì hàm toán tử cần có 2 đối. Đối thứ nhất ứng với toán hạng thứ nhất, đối thứ hai ứng với toán hạng thứ hai. Do vậy, với các phép toán không giao hoán (như phép-) thì thứ tự đối là rất quan trọng.

Ví dụ các hàm toán tử cộng, trừ phân số được khai báo như sau:

```
struct PS
{
    int a; // Tử số
    int b; // Mẫu số
};
PS operator+(PS p1, PS p2); // p1 + p2
PS operator-(PS p1, PS p2); // p1 - p2
PS operator*(PS p1, PS p2); // p1 * p2
PS operator/(PS p1, PS p2); // p1 / p2
```

b. Với các phép toán có một toán hạng, thì hàm toán tử có một đối. Ví dụ hàm toán tử đổi dấu ma trận (đổi dấu tất cả các phần tử của ma trận) được khai báo như sau:


```
struct MT
{
    double a[20][20] ; // Mảng chứa các phần tử ma trận
    int m ; // Số hàng ma trận
    int n ; // Số cột ma trận
};
MT operator-(MT x) ;
```

7.3.3. Thân của hàm toán tử: Viết như thân của hàm thông thường. Ví dụ hàm đổi dấu ma trận có thể được định nghĩa như sau:

```
struct MT
{
    double a[20][20] ; // Mảng chứa các phần tử ma trận
    int m ; // Số hàng ma trận
    int n ; // Số cột ma trận
};
MT operator-(MT x)
{
    MT y;
    for (int i=1; i<= m ;++i)
        for (int j=1; j<= n ;++j)
            y[i][j] = - x[i][j] ;
    return y;
}
```

7.4. Cách dùng hàm toán tử

Có 2 cách dùng:

Cách 1: Dùng như một hàm thông thường bằng cách viết lời gọi.

Ví dụ:

```
PS p, q, u, v ;
u = operator+(p, q) ; // u = p + q
v = operator-(p, q) ; // v = p - q
```

Cách 2: Dùng như phép toán của C++ .

Ví dụ:

```
PS p, q, u, v ;
u = p + q ; // u = p + q
v = p - q ; // v = p - q
```

Chú ý: Khi dùng các hàm toán tử như phép toán của C++ ta có thể kết hợp nhiều phép toán để viết các công thức phức tạp. Cũng cho phép dùng dấu ngoặc tròn để quy định thứ tự thực hiện các phép tính. Thứ tự ưu tiên của các phép tính vẫn tuân theo các quy tắc ban đầu của C++ . Chẳng hạn các phép * và / có thứ ưu tiên cao hơn so với các phép + và -

Ví dụ:

```
PS p, q, u, v, s1, s2 ;  
s1 = p*q - u/v ; // s1 = (p*q)  
s2 = (p - q)/(u + v) ; // s2 = (p - q)/(u + v)
```

§ 8. Các ví dụ về định nghĩa chồng toán tử

Ví dụ 1: Trong ví dụ này ngoài việc sử dụng các hàm toán tử để thực hiện 4 phép tính trên phân số, còn định nghĩa chồng các phép toán << và >> để xuất và nhập phân số (xem chi tiết trong chương 7).

Hàm `operator<<` có 2 đối kiểu `ostream&` và `PS` (Phân số). Hàm trả về giá trị kiểu `ostream&`. Hàm được khai báo như sau:

```
ostream& operator<< (ostream& os, PS p);
```

Tương tự hàm `operator>>` được khai báo như sau:

```
istream& operator>> (istream& is, PS &p);
```

Dưới đây sẽ chỉ ra cách xây dựng và sử dụng các hàm toán tử. Chúng ta cũng sẽ thấy việc sử dụng các hàm toán tử rất tự nhiên, ngắn gọn và tiện lợi.

Chương trình dưới đây có nội dung như chương trình trong §6.2, nhưng thay các hàm bằng các hàm toán tử.

```
#include <conio.h>  
#include <iostream.h>  
#include <math.h>  
typedef struct  
{  
    int a,b;  
} PS;  
ostream& operator<< (ostream& os, PS p);  
istream& operator>> (istream& is, PS &p);  
int uscln(int x, int y);  
PS rutgon(PS p);  
PS operator+(PS p1, PS p2);  
PS operator-(PS p1, PS p2);  
PS operator*(PS p1, PS p2);  
PS operator/(PS p1, PS p2);  
ostream& operator<< (ostream& os, PS p)  
{  
    os << p.a << '/' << p.b ;  
    return os;  
}  
istream& operator>> (istream& is, PS &p)  
{  
    cout << "Nhập tu và mau: " ;  
    is >> p.a >> p.b ;  
    return is;
```

```
    }
int uscln(int x, int y)
{
    x=abs(x); y=abs(y);
    if (x*y==0) return 1;
    while (x!=y)
        if (x>y) x-=y;
        else y-=x;
    return x;
}
PS rutgon(PS p)
{
    PS q;
    int x;
    x=uscln(p.a,p.b);
    q.a = p.a / x ;
    q.b = p.b / x ;
    return q;
}
PS operator+(PS p1, PS p2)
{
    PS q;
    q.a = p1.a*p2.b + p2.a*p1.b;
    q.b = p1.b * p2.b ;
    return rutgon(q);
}
PS operator-(PS p1, PS p2)
{
    PS q;
    q.a = p1.a*p2.b - p2.a*p1.b;
    q.b = p1.b * p2.b ;
    return rutgon(q);
}
PS operator*(PS p1, PS p2)
{
    PS q;
    q.a = p1.a * p2.a ;
    q.b = p1.b * p2.b ;
    return rutgon(q);
}
PS operator/(PS p1, PS p2)
{
    PS q;
```

```
    q.a = p1.a * p2.b ;
    q.b = p1.b * p2.a ;
    return rutgon(q);
}

void main()
{
    PS p, q, z, u, v ;
    PS s;
    cout << "\nNhap cac PS p, q, z, u, v:\n " ;
    cin >> p >> q >> z >> u >> v ;
    s = (p - q*z) / (u + v) ;
    cout << "\n Phan so s = " << s;
    getch();
}
```

Ví dụ 2: Chương trình đưa vào các hàm toán tử:

- operator- có một đối dùng để đảo dấu một đa thức
- operator+ có 2 đối dùng để cộng 2 đa thức
- operator- có 2 đối dùng để trừ 2 đa thức
- operator* có 2 đối dùng để nhân 2 đa thức
- operator^ có 2 đối dùng để tính giá đa thức tại x
- operator<< có 2 đối dùng để in đa thức
- operator>> có 2 đối dùng để nhập đa thức

Chương trình sẽ nhập 4 đa thức: p, q, r, s. Sau đó tính đa thức:

$$f = -(p+q)*(r-s)$$

Cuối cùng tính giá trị $f(x)$, với x là một số thực nhập từ bàn phím.

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
struct DT
{
    double a[20]; // Mang chua cac he so da thuc a0, a1,...
    int n ; // Bac da thuc
};

ostream& operator<< (ostream& os, DT d);
istream& operator>> (istream& is,DT &d);
DT operator-(const DT& d);
DT operator+(DT d1, DT d2);
DT operator-(DT d1, DT d2);
DT operator*(DT d1, DT d2);
double operator^(DT d, double x); // Tinh gia tri da thuc
```

ostream& operator<< (ostream& os, DT d)

```
{
    os << " - Cac he so (tu ao): " ;
    for (int i=0 ; i<= d.n ; ++i)
        os << d.a[i] <<" " ;
    return os;
}
```

istream& operator>> (istream& is, DT &d)

```
{
    cout << " - Bac da thuc: " ;
    cin >> d.n;
    cout << "Nhap cac he so da thuc:\n" ;
    for (int i=0 ; i<= d.n ; ++i)
    {
        cout << "He so bac " << i << " = " ;
        is >> d.a[i] ;
    }
    return is;
}
```

DT operator-(const DT& d)

```
{
    DT p;
    p.n = d.n;
    for (int i=0 ; i<=d.n ; ++i)
        p.a[i] = -d.a[i];
    return p;
}
```

DT operator+(DT d1, DT d2)

```
{
    DT d;
    int k,i;
    k = d1.n > d2.n ? d1.n : d2.n ;
    for (i=0; i<=k ; ++i)
        if (i<=d1.n && i<=d2.n)
            d.a[i] = d1.a[i] + d2.a[i];
        else if (i<=d1.n)
            d.a[i] = d1.a[i];
        else
            d.a[i] = d2.a[i];
    i=k;
    while (i>0 && d.a[i]==0.0) --i;
    d.n = i;
    return d ;
}
```

```
    }
DT operator-(DT d1, DT d2)
{
    return (d1 + (-d2));
}

DT operator*(DT d1, DT d2)
{
    DT d;
    int k, i, j;
    k = d.n = d1.n + d2.n ;
    for (i=0; i<=k; ++i) d.a[i] = 0;
    for (i=0 ; i<= d1.n ; ++i)
        for (j=0 ; j<= d2.n ; ++j)
            d.a[i+j] += d1.a[i]*d2.a[j] ;
    return d;
}

double operator^(DT d, double x)
{
    double s=0.0 , t=1.0;
    for (int i=0 ; i<= d.n ; ++i)
    {
        s += d.a[i]*t;
        t *= x;
    }
    return s;
}

void main()
{
    DT p,q,r,s,f;
    double x,g;
    clrscr();
    cout <<"\nNhap da thuc P " ; cin >> p;
    cout <<"\nNhap da thuc Q " ; cin >> q;
    cout <<"\nNhap da thuc R " ; cin >> r;
    cout <<"\nNhap da thuc S " ; cin >> s;
    cout << "\nNhap so thuc x: " ; cin >> x;
    f = -(p+q)*(r-s);
    g = f^x;
    cout << "\nDa thuc f " << f ;
```

```
cout << "\n x = " << x;
cout << "\nf(x) = " << g;
getch();
}
```

§ 9. Các bài toán về ma trận và véc tơ

Trong mục này sẽ xét các ma trận thực vuông cấp n và các véc tơ thực cấp n . Chúng được biểu diễn thông qua các kiểu cấu trúc MT và VT:

```
struct MT
{
    double a[20][20]; // Mang a chứa các phần tử ma trận
    int n;             // Cấp ma trận
};
struct VT
{
    double b[20]; // Mang chứa các phần tử của véc tơ
    int n; // Cấp véc tơ
};
```

Để xử lý ma trận và véc tơ, chúng ta xây dựng 9 hàm toán tử:

```
ostream& operator<< (ostream& os, const MT& x); // In ma trận
ostream& operator<< (ostream& os, const VT& v); // In véc tơ
istream& operator>> (istream& is, MT& x);      // Nhập ma trận
istream& operator>> (istream& is, VT &v);      // Nhập véc tơ
MT operator+(const MT& x1, const MT& x2); // Cộng 2 ma trận
MT operator-(const MT& x1, const MT& x2); // Trừ 2 ma trận
MT operator*(const MT& x1, const MT& x2); // Nhân 2 ma trận
VT operator*(const MT& x, const VT& v); // Nhân ma trận véc tơ
MT operator!(MT x);                       // Nghịch đảo ma trận
```

Thuật toán cho 8 hàm toán tử đầu tương đối quen thuộc không có gì phải bàn. Để nghịch đảo ma trận có nhiều cách, ở đây chúng ta dùng phương pháp Jordance như sau. Giả sử cần nghịch đảo ma trận x cấp n . Ta dùng thêm ma trận đơn vị y . Sau đó thực hiện đồng thời các phép tính trên cả x và y sao cho x trở thành đơn vị. Kết quả y chính là nghịch đảo của x . Thuật toán được tiến hành trên n bước. Nội dung của bước k ($k = 1, \dots, n$) như sau:

Tìm chỉ số r ($k \leq r \leq n$) sao cho

$$\text{abs}(x[r,k]) = \max \{ \text{abs}(x[i,k]) \text{ với } i = k, \dots, n \}$$

Nếu $\text{abs}(x[r,k]) = 0$ thì ma trận không có nghịch đảo và thuật toán kết thúc giữa chừng.

Hoán vị hàng k với hàng r trong cả 2 ma trận x và y .

Chia hàng k của cả x và y cho $\text{tg} = x[k,k]$ (mục đích làm cho $x[k,k] = 1$).

Biến đổi để cột k của x trở thành véc tơ đơn vị bằng cách làm cho các phần tử $x[i,k] = 0$ (với i khác k). Muốn vậy ta thực hiện các phép tính sau trên cả x và y :

$$(\text{hàng } i) = (\text{hàng } i) - x[i,k](\text{hàng } k), \text{ với mọi } i \text{ khác } k$$

Nội dung chương trình là nhập 4 ma trận X, Y, R, S và véc tơ u. Sau đó tính véc tơ v theo công thức:

$$v = ((X + Y) * (R - S))^{-1} u$$

Như sẽ thấy trong hàm main() dưới đây, nhờ các hàm toán tử mà câu lệnh tính v được viết gần giống như công thức toán học nêu trên.

```
/* Chương trình */
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
struct MT
{
    double a[20][20]; // Mang chua cac phan tu ma tran
    int n; // Cap ma tran
};
struct VT
{
    double b[20]; // Mang chua cac phan tu cua vec to
    int n; // Cap vec to
};
ostream& operator<< (ostream& os, const MT& x);
ostream& operator<< (ostream& os, const VT& v);
istream& operator>> (istream& is, MT& x);
istream& operator>> (istream& is, VT& v);
MT operator+(const MT& x1, const MT& x2);
MT operator-(const MT& x1, const MT& x2);
MT operator*(const MT& x1, const MT& x2);
VT operator*(const MT& x, const VT& v);
MT operator!(MT x); // Tinh ma tran nghich dao
ostream& operator<< (ostream& os, const MT& x)
{
    os << setprecision(2) << setiosflags(ios::showpoint);
    for (int i=1; i<= x.n; ++i)
    {
        os << "\n";
        for (int j=1; j<=x.n; ++j)
            os << setw(6) << x.a[i][j];
    }
    os << "\n";
    return os;
}
```



```
}

ostream& operator<< (ostream& os, const VT& v)
{
    os << setprecision(2) << setiosflags(ios::showpoint);
    for (int i=1 ; i<= v.n ; ++i)
        os << setw(6) << v.b[i] ;
    os << "\n" ;
    return os;
}

istream& operator>> (istream& is, MT& x)
{
    cout << " - Cap ma tran: " ;
    is >> x.n;
    cout << "Nhap cac phan tu :\n" ;
    for (int i=1 ; i<= x.n ; ++i)
        for (int j=1; j<=x.n; ++j)
        {
            cout << "PT hang " << i << " cot " << j << " = " ;
            is >> x.a[i][j] ;
        }
    return is;
}

istream& operator>> (istream& is, VT& v)
{
    cout << " - Cap vec to: " ;
    is >> v.n;
    cout << "Nhap cac phan tu :\n" ;
    for (int i=1 ; i<= v.n ; ++i)
    {
        cout << "Phan tu thu " << i << " = " ;
        is >> v.b[i] ;
    }
    return is;
}

MT operator+(const MT& x1, const MT& x2)
{
    if (x1.n!=x2.n)
    {
        cout << "\nKhong thuc hien duoc phep cong vi 2 MT khong cung cap";
        getch();
        return x1;
    }
    else
```

```
{
    MT x;
    int i, j, n;
    n = x.n = x1.n ;
    for (i=1; i<=n; ++i)
        for (j=1; j<=n ;++j)
            x.a[i][j] = x1.a[i][j] + x2.a[i][j] ;
    return x;
}
}
```

MT operator-(const MT& x1, const MT& x2)

```
{
    if (x1.n!=x2.n)
    {
        cout << "\nKhong thuc hien duoc phep tru vi 2 MT khong cung cap";
        getch();
        return x1;
    }
    else
    {
        MT x;
        int i, j, n;
        n = x.n = x1.n;
        for (i=1; i<=n; ++i)
            for (j=1; j<=n ;++j)
                x.a[i][j] = x1.a[i][j] - x2.a[i][j] ;
        return x;
    }
}
```

MT operator*(const MT& x1, const MT& x2)

```
{
    if (x1.n!=x2.n)
    {
        cout << "\nKhong thuc hien duoc phep nhan vi 2 MT khong cung cap";
        getch();
        return x1;
    }
    else
    {
        MT x;
        int n, i, j,k;
```

```
n = x.n = x1.n;
for (i=1; i<=n; ++i)
    for (j=1; j<=n ;++j)
    {
        x.a[i][j] = 0.0 ;
        for (k=1 ; k<=n; ++k)
            x.a[i][j] += x1.a[i][k]*x2.a[k][j] ;
    }
return x;
}
}

VT operator*(const MT& x, const VT& v)
{
    if (x.n != v.n)
    {
        cout << "\n Cap ma tran khac cap vec to, phép nhân vô nghĩa";
        getch();
        return v;
    }
    else
    {
        VT u; int n;
        n = u.n = v.n ;
        for (int i=1; i <=n ; ++i)
        {
            u.b[i] = 0;
            for (int j=1; j<=n; ++j)
                u.b[i] += x.a[i][j]*v.b[j];
        }
        return u;
    }
}

MT operator!(MT x)
{
    MT y;
    int i,j,k,r,n;
    double tg;
    n = y.n = x.n ;
    for (i=1 ; i<=n ; ++i)
```

```
for (j=1 ; j<=n ; ++j)
    if (i==j) y.a[i][j] = 1;
    else y.a[i][j] = 0;
for (k=1; k<=n; ++k)
{
    r=k;
    for (i=k+1; i<=n; ++i)
        if (abs(x.a[i][k]) > abs(x.a[r][k]) ) r = i;
    if (abs(x.a[r][k]) < 1.0E-8)
    {
        cout << "\n Ma tran suy bien, khong co nghich dao" ;
        getch();
        return x;
    }
}
/* Hoan vi hang r va hang k */
for (j=1 ; j<=n ; ++j)
{
    tg = x.a[k][j];
    x.a[k][j] = x.a[r][j];
    x.a[r][j] = tg;
    tg = y.a[k][j];
    y.a[k][j] = y.a[r][j];
    y.a[r][j] = tg;
}
/* Chia hang k cho a[k,k] */
tg = x.a[k][k] ;
for (j=1 ; j<=n ; ++j)
{
    x.a[k][j] /= tg;
    y.a[k][j] /= tg;
}
/* Khu cot k : lam cho a[i,k] = 0 voi i != k */
for (int i=1; i<= n ; ++i)
    if (i != k)
    {
        tg = x.a[i][k] ;
        for (j=1 ; j<=n ; ++j)
        {
            x.a[i][j] -= tg*x.a[k][j] ;
```

```
        y.a[i][j] -= tg*y.a[k][j] ;
    }
}
}
return y;
}
void main()
{
    MT x,y,r,s;
    VT u,v;
    clrscr();
    cout <<"\nNhap ma tran X " ; cin >> x;
    cout <<"\nNhap ma tran Y " ; cin >> y;
    cout <<"\nNhap ma tran R " ; cin >> r;
    cout <<"\nNhap ma tran S " ; cin >> s;
    cout <<"\nNhap vec to u " ; cin >> u;
    v = !((x+y)*(r-s))*u ;
    cout << "\nVec to v = xu " << v ;
    getch();
}
```

Chương 3

Khái niệm về lớp

Như đã nói ở trên, lớp là khái niệm trung tâm của lập trình hướng đối tượng, nó là sự mở rộng của các khái niệm cấu trúc (struct) của C và bản ghi (record) của PASCAL. Ngoài các thành phần dữ liệu (như cấu trúc), lớp còn chứa các thành phần hàm, còn gọi là phương thức (method) hay hàm thành viên (member function). Cũng giống như cấu trúc, lớp có thể xem như một kiểu dữ liệu. Vì vậy lớp còn gọi là kiểu đối tượng và lớp được dùng để khai báo các biến, mảng đối tượng (như thể dùng kiểu int để khai báo các biến mảng nguyên). Như vậy từ một lớp có thể tạo ra (bằng cách khai báo) nhiều đối tượng (biến, mảng) khác nhau. Mỗi đối tượng có vùng nhớ riêng của mình. Vì vậy cũng có thể quan niệm lớp là tập hợp các đối tượng cùng kiểu.

Chương này sẽ trình bày cách định nghĩa lớp, cách xây dựng phương thức, giải thích về phạm vi truy nhập, sử dụng các thành phần của lớp, cách khai báo biến, mảng cấu trúc, lời gọi tới các phương thức.

Bài 1. Định nghĩa lớp

1. Lớp được định nghĩa theo mẫu:

```
class tên_lớp
{
    // Khai báo các thành phần dữ liệu (thuộc tính)
    // Khai báo các phương thức
};
// Định nghĩa (xây dựng) các phương thức
```

Chú ý:

Thuộc tính của lớp có thể là các biến, mảng, con trỏ có kiểu chuẩn (int, float, char, char*, long,...) hoặc kiểu ngoài chuẩn đã định nghĩa trước (cấu trúc, hợp, lớp, ...). Thuộc tính của lớp không thể có kiểu của chính lớp đó, nhưng có thể là kiểu con trỏ lớp này, ví dụ:

```
class A
{
    A x; // Không cho phép, vì x có kiểu lớp A
    A *p; // Cho phép, vì p là con trỏ kiểu lớp A
    ...
};
```

2. Khi báo các thành phần của lớp (thuộc tính và phương thức) có thể dùng các từ khoá private và public để quy định phạm vi sử dụng của các thành phần. Nếu không quy định cụ thể (không dùng các từ khoá private và public) thì C++ hiểu đó là private.

Các thành phần private (riêng) chỉ được sử dụng bên trong lớp (trong thân của các phương thức của lớp). Các hàm không phải là phương thức của lớp không được phép sử dụng các thành phần này.

Các thành phần public (công cộng) được phép sử dụng ở cả bên trong và bên ngoài lớp.

3. Các thành phần dữ liệu thường (nhưng không bắt buộc) khai báo là private để bảo đảm tính giấu kín, bảo vệ an toàn dữ liệu của lớp, không cho phép các hàm bên ngoài xâm nhập vào dữ liệu của lớp.

4. Các phương thức thường khai báo là public để chúng có thể được gọi tới (sử dụng) từ các hàm khác trong chương trình.

5. Các phương thức có thể được xây dựng bên ngoài hoặc bên trong định nghĩa lớp. Thông thường, các phương thức ngắn được viết bên trong định nghĩa lớp, còn các phương thức dài thì viết bên ngoài định nghĩa lớp.

6. Trong thân phương thức của một lớp (giả sử lớp A) có thể sử dụng:

- + Các thuộc tính của lớp A
- + Các phương thức của lớp A
- + Các hàm tự lập trong chương trình. Vì phạm vi sử dụng của hàm là toàn chương trình.

7. Giá trị trả về của phương thức có thể có kiểu bất kỳ (chuẩn và ngoài chuẩn)

Ví dụ sau sẽ minh họa các điều nói trên. Chúng ta sẽ định nghĩa lớp để mô tả và xử lý các điểm trên màn hình đồ họa. Lớp được đặt tên là DIEM.

+ Các thuộc tính của lớp gồm:

```
int x ; // hoành độ (cột)
int y ; // tung độ (hàng)
int m ; // màu
```

+ Các phương thức:

```
Nhập dữ liệu một điểm
Hiển thị một điểm
ấn một điểm
```

Lớp điểm được xây dựng như sau:

```
class DIEM
{
    private:
        int x, y, m ;
    public:
        void nhapsl() ;
        void hien() ;
        void an()
        {
            putpixel(x, y, getbkcolor());
        }
};

void DIEM::nhap()
{
    cout << "\nNhập hoành độ (cột) và tung độ (hàng) của điểm: "
    cin >> x >> y ;
    cout << "\nNhập mã màu của điểm: "
    cin >> m ;
}

void DIEM::hien()
{
    int mau_ht ;
```

```
    mau_ht = getcolor();  
    putpixel(x, y, m);  
    setcolor(mau_ht);  
}
```

Qua ví dụ trên có thể rút ra một số điều cần nhớ sau:

- + Trong cả 3 phương thức (dù viết trong hay viết ngoài định nghĩa lớp) đều được phép truy nhập đến các thuộc tính x, y và m của lớp.
- + Các phương thức viết bên trong định nghĩa lớp (như phương thức an()) được viết như một hàm thông thường.
- + Khi xây dựng các phương thức bên ngoài lớp, cần dùng thêm tên lớp và toán tử phạm vi :: đặt ngay trước tên phương thức để quy định rõ đây là phương thức của lớp nào.

Bài 2. Biến, mảng đối tượng

Như đã nói ở trên, một lớp (sau khi định nghĩa) có thể xem như một kiểu đối tượng và có thể dùng để khai báo các biến, mảng đối tượng. Cách khai báo biến, mảng đối tượng cũng giống như khai báo biến, mảng các kiểu khác (như int, float, cấu trúc, hợp, ...), theo mẫu sau:

Tên_lớp danh sách đối ;
Tên_lớp danh sách mảng ;

Ví dụ sử dụng lớp DIEM ở §1, có thể khai báo các biến, mảng DIEM như sau:

DIEM d1, d2, d3 ; // Khai báo 3 biến đối tượng d1, d2, d3
DIEM d[20] ; // Khai báo mảng đối tượng d gồm 20 phần tử

Mỗi đối tượng sau khi khai báo sẽ được cấp phát một vùng nhớ riêng để chứa các thuộc tính của chúng. Chú ý rằng sẽ không có vùng nhớ riêng để chứa các phương thức cho mỗi đối tượng. Các phương thức sẽ được sử dụng chung cho tất cả các đối tượng cùng lớp. Như vậy về bộ nhớ được cấp phát thì đối tượng giống cấu trúc. Trong trường hợp này:

sizeof(d1) = sizeof(d2) = sizeof(d3) = 3*sizeof(int) = 6
sizeof(d) = 20*6 = 120

Thuộc tính của đối tượng:

Trong ví dụ trên, mỗi đối tượng d1, d2, d3 và mỗi phần tử d[i] đều có 3 thuộc tính là x, y, m. Chú ý là mỗi thuộc đều thuộc về một đối tượng, vì vậy không thể viết tên thuộc một cách riêng rẽ mà bao giờ cũng phải có tên đối tượng đi kèm, giống như cách viết trong cấu trúc của C hay bản ghi của PASCAL. Nói cách khác, cách viết thuộc tính của đối tượng như sau:

tên_đối_tượng.Tên_thuộc_tính

Với các đối tượng d1, d2, d3 và mảng d, có thể viết như sau:

d1.x // Thuộc tính x của đối tượng d1
d2.x // Thuộc tính x của đối tượng d2
d3.y // Thuộc tính y của đối tượng d3
d[2].m // Thuộc tính m của phần tử d[2]
d1.x = 100 ; // Gán 100 cho d1.x
d2.y = d1.x; // Gán d1.x cho d2.y

Sử dụng các phương thức

Cũng giống như hàm, một phương thức được sử dụng thông qua lời gọi. Tuy nhiên trong lời gọi phương thức bao giờ cũng phải có tên đối tượng để chỉ rõ phương thức thực hiện trên các thuộc tính của đối tượng nào. Ví dụ lời gọi:

```
d1.nhapsl();
```

sẽ thực hiện nhập số liệu vào các thành phần d1.x, d1.y và d1.m

Câu lệnh

```
d[3].nhapsl() ;
```

sẽ thực hiện nhập số liệu vào các thành phần d[3].x, d[3].y và d[3].m

Chúng ta sẽ minh họa các điều nói trên bằng một chương trình đơn giản sử dụng lớp DIEM để nhập 3 điểm, hiện rồi ẩn các điểm vừa nhập. Trong chương trình đưa vào hàm kd_do_hoa() dùng để khởi động hệ đồ họa.

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

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

```
#include <graphics.h>
```

```
class DIEM
```

```
{
```

```
private:
```

```
int x, y, m ;
```

```
public:
```

```
void nhapsl();
```

```
void an()
```

```
{
```

```
putpixel(x,y,getbkcolor());
```

```
}
```

```
void hien();
```

```
};
```

```
void DIEM::nhapsl()
```

```
{
```

```
cout << "\nNhap hoành đo (cot) va tung đo (hang) cua diem: " ;
```

```
cin >> x >> y ;
```

```
cout << " \nNhap ma mau cua diem: " ;
```

```
cin >> m ;
```

```
}
```

```
void DIEM::hien()
```

```
{
```

```
int mau_ht;
```

```
mau_ht = getcolor() ;
```

```
putpixel(x,y,m);
```

```
setcolor(mau_ht);
```

```
}
```

```
void kd_do_hoa()
```

```
{
    int mh, mode ;
    mh=mode=0;
    initgraph(&mh, &mode, "");
}

void main()
{
    DIEM d1, d2, d3 ;
    d1.nhapsl();
    d2.nhapsl();
    d3.nhapsl();
    kd_do_hoa();
    setbkcolor(BLACK);
    d1.hien();
    d2.hien();
    d3.hien();
    getch();
    d1.an();
    d2.an();
    d3.an();
    getch();
    closegraph();
}
```

Bài 3. Con trỏ đối tượng

Con trỏ đối tượng dùng để chứa địa chỉ của biến, mảng đối tượng. Nó được khai báo như sau:

Tên_lớp *con trỏ ;

Ví dụ dùng lớp DIEM có thể khai báo:

DIEM *p1 , *p2, *p3 ; // khai báo 3 con trỏ p1, p2, p3

DIEM d1, d2 ; // Khai báo 2 đối tượng d1, d2

DIEM d[20] ; // Khai báo mảng đối tượng

và có thể thực hiện các câu lệnh:

p1 = &d2 ; // p1 chứa địa chỉ của d2 , hay p1 trỏ tới d2

p2 = d ; // p2 trỏ tới đầu mảng d

p3 = new DIEM // Tạo một đối tượng và chứa địa chỉ của nó
// vào p3

Để sử dụng thuộc tính của đối tượng thông qua con trỏ, ta viết như sau:

Tên_con_trỏ->Tên_thuộc_tính

Chú ý: Nếu con trỏ chứa địa chỉ đầu của mảng, có thể dùng con trỏ như tên mảng.

Như vậy sau khi thực hiện các câu lệnh trên thì:

p1->x và d2.x là như nhau

p2[i].y và d[i].y là như nhau

Tóm lại ta có quy tắc sau

Quy tắc sử dụng thuộc tính: Để sử dụng một thuộc tính của đối tượng ta phải dùng phép . hoặc phép -> . Trong chương trình, không cho phép viết tên thuộc tính một cách đơn độc mà phải đi kèm tên đối tượng hoặc tên con trỏ theo các mẫu sau:

Tên_đối_tượng.Tên_thuộc_tính

Tên_con_trỏ->Tên_thuộc_tính

Tên_mảng_đối_tượng[chỉ_số].Tên_thuộc_tính

Tên_con_trỏ[chỉ_số].Tên_thuộc_tính

Chương trình dưới đây cũng sử dụng lớp DIEM (trong §1) để nhập một dãy điểm, hiển thị và ấn các điểm vừa nhập. Chương trình dùng một con trỏ kiểu DIEM và dùng toán tử new để tạo ra một dãy đối tượng.

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

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

```
#include <graphics.h>
```

```
class DIEM
```

```
{
```

```
private:
```

```
int x, y, m ;
```

```
public:
```

```
void nhapsl();
```

```
void an()
```

```
{
```

```
putpixel(x,y,getbkcolor());
```

```
}
```

```
void hien();
```

```
};
```

```
void DIEM::nhapsl()
```

```
{
```

```
cout << "\nNhập hoành độ (cot) và tung độ (hang) của điểm: " ;
```

```
cin >> x >> y ;
```

```
cout << " \nNhập màu của điểm: " ;
```

```
cin >> m ;
```

```
}
```

```
void DIEM::hien()
```

```
{
```

```
int mau_ht;
```

```
mau_ht = getcolor() ;
```

```
putpixel(x,y,m);
```

```
    setcolor(mau_ht);
}
void kd_do_hoa()
{
    int mh, mode ;
    mh=mode=0;
    initgraph(&mh, &mode, "");
}
void main()
{
    DIEM *p;
    int i, n;
    cout << "So diem: " ;
    cin >> n;
    p = new DIEM[n+1];
    for (i=1; i<=n; ++i)
        p[i].nhapsl();
    kd_do_hoa();
    for (i=1; i<=n; ++i)
        p[i].hien();
    getch();
    for (i=1; i<=n; ++i)
        p[i].an();
    getch();
    closegraph();
}
```

Bài 4. Đối của phương thức, con trỏ this

4.1. Con trỏ this là đối thứ nhất của phương thức

Chúng ta hãy xem lại phương thức nhapsl của lớp DIEM

```
void DIEM::nhapsl()
{
    cout << "\nNhap hoành do (cot) va tung do (hang) cua diem:" ;
    cin >> x >> y ;
    cout << " \nNhap ma mau cua diem: " ;
    cin >> m ;
}
```

Rõ ràng trong phương thức này chúng ta sử dụng tên các thuộc tính x, y và m một cách đơn độc. Điều này có vẻ như mâu thuẫn với quy tắc sử dụng thuộc tính nêu trong mục trước. Song sự thể như sau:

C++ sử dụng con trỏ đặc biệt **this** trong các phương thức. Các thuộc tính viết trong phương thức được hiểu là thuộc một đối tượng do con trỏ **this** trỏ tới. Như vậy phương thức `nhapsl()` có thể viết một cách tường minh như sau:

```
void DIEM::nhapsl()
{
    cout << "\nNhap hoành do (cot) va tung do (hang) cua diem: ";
    cin >> this->x >> this->y;
    cout << "\nNhap ma mau cua diem: ";
    cin >> this->m;
}
```

Từ góc độ hàm số có thể kết luận rằng: Phương thức bao giờ cũng có ít nhất một đối là con trỏ **this** và nó luôn luôn là đối đầu tiên của phương thức.

4.2. Tham số ứng với đối con trỏ **this**

Xét một lời gọi tới phương thức `nhapsl()` :

```
DIEM d1;
d1.nhapsl();
```

Trong trường hợp này tham số truyền cho con trỏ **this** chính là địa chỉ của `d1`:

```
this = &d1
```

Do đó:

```
this->x chính là d1.x
this->y chính là d1.y
this->m chính là d1.m
```

Như vậy câu lệnh

```
d1.nhapsl();
```

sẽ nhập dữ liệu cho các thuộc tính của đối tượng `d1`. Từ đó có thể rút ra kết luận sau:

Tham số truyền cho đối con trỏ **this** chính là địa chỉ của đối tượng đi kèm với phương thức trong lời gọi phương thức.

4.3. Các đối khác của phương thức

Ngoài đối đặc biệt **this** (đối này không xuất hiện một cách tường minh), phương thức còn có các đối khác được khai báo như trong các hàm. Đối của phương thức có thể có kiểu bất kỳ (chuẩn và ngoài chuẩn).

Ví dụ để xây dựng phương thức vẽ đường thẳng qua 2 điểm ta cần đưa vào 3 đối: Hai đối là 2 biến kiểu `DIEM`, đối thứ ba kiểu nguyên xác định mã màu. Vì đã có đối ngầm định **this** là đối thứ nhất, nên chỉ cần khai báo thêm 2 đối. Phương thức có thể viết như sau:

```
void DIEM::doan_thang(DIEM d2, int mau)
{
    int mau_ht;
    mau_ht = getcolor();
    setcolor(mau);
    line(this->x, this->y, d2.x, d2.y);
}
```

```
    setcolor(mau_ht);  
}
```

Chương trình sau minh họa các phương thức có nhiều đối. Ta vẫn dùng lớp DIEM nhưng có một số thay đổi:

- + Bỏ thuộc tính m (màu)
- + Bỏ các phương thức hien và an
- +Đưa vào 4 phương thức mới:
 - ve_doan_thang (Vẽ đoạn thẳng qua 2 điểm)
 - ve_tam_giac (Vẽ tam giác qua 3 điểm)
 - do_dai (Tính độ dài của đoạn thẳng qua 2 điểm)
 - chu_vi (Tính chu vi tam giác qua 3 điểm)

Chương trình còn minh họa:

- + Việc phương thức này sử dụng phương thức khác (phương thức ve_tam_giac sử dụng phương thức ve_doan_thang, phương thức chu_vi sử dụng phương thức do_dai)
- + Sử dụng con trỏ this trong thân các phương thức ve_tam_giac và chu_vi

Nội dung chương trình là nhập 3 điểm, vẽ tam giác có đỉnh là 3 điểm vừa nhập sau đó tính chu vi tam giác.

```
#include <conio.h>  
#include <iostream.h>  
#include <graphics.h>  
#include <math.h>  
#include <stdio.h>  
class DIEM  
{  
private:  
    int x, y;  
public:  
    void nhapsl();  
    void ve_doan_thang(DIEM d2, int mau) ;  
    void ve_tam_giac(DIEM d2, DIEM d3,int mau) ;  
    double do_dai(DIEM d2)  
    {  
        DIEM d1 = *this ;  
        return sqrt( pow(d1.x - d2.x,2) + pow(d1.y - d2.y,2) ) ;  
    }  
    double chu_vi(DIEM d2, DIEM d3);  
};  
void DIEM::nhapsl()  
{  
    cout <<"\nNhap hoành do (cot) va tung do (hang) cua diem:" ;  
    cin >> x >> y ;  
}  
void kd_do_hoa()
```

```
{
    int mh, mode ;
    mh=mode=0;
    initgraph(&mh, &mode, "");
}

void DIEM::ve_doan_thang(DIEM d2, int mau)
{
    setcolor(mau);
    line(this->x,this->y,d2.x,d2.y);
}

void DIEM::ve_tam_giac(DIEM d2, DIEM d3,int mau)
{
    (*this).ve_doan_thang(d2,mau);
    d2.ve_doan_thang(d3,mau);
    d3.ve_doan_thang(*this,mau);
}

double DIEM::chu_vi(DIEM d2, DIEM d3)
{
    double s;
    s= (*this).do_dai(d2) + d2.do_dai(d3) + d3.do_dai(*this) ;
    return s;
}

void main()
{
    DIEM d1, d2, d3;
    char tb_cv[20] ;
    d1.nhapsl();
    d2.nhapsl();
    d3.nhapsl();
    kd_do_hoa();
    d1.ve_tam_giac(d2,d3,15);
    double s = d1.chu_vi(d2,d3);
    sprintf(tb_cv,"Chu vi = %0.2f", s);
    outtextxy(10,10,tb_cv);
    getch();
    closegraph();
}
```

Một số nhận xét về đối của phương thức và lời gọi phương thức

+ Quan sát nguyên mẫu phương thức:

```
void ve_doan_thang(DIEM d2, int mau) ;
```

sẽ thấy phương thức có 3 đối:

Đối thứ nhất là một đối tượng DIEM do this trỏ tới

Đối thứ hai là đối tượng DIEM d2

Đối thứ ba là biến nguyên mau

Nội dung phương thức là vẽ một đoạn thẳng đi qua các điểm *this và d2 theo mã màu mau. Xem thân của phương sẽ thấy được nội dung này:

```
void DIEM::ve_doan_thang(DIEM d2, int mau)
{
    setcolor(mau);
    line(this->x,this->y,d2.x,d2.y);
}
```

Tuy nhiên trong trường hợp này, vai trò của this không cao lắm, vì nó được đưa vào chỉ cốt làm rõ đối thứ nhất. Trong thân phương thức có thể bỏ từ khoá this vẫn được.

+ Vai trò của this trở nên quan trọng trong phương thức ve_tam_giac:

```
void ve_tam_giac(DIEM d2, DIEM d3,int mau) ;
```

Phương thức này có 4 đối là:

this trỏ tới một đối tượng kiểu DIEM

d2 một đối tượng kiểu DIEM

d3 một đối tượng kiểu DIEM

mau một biến nguyên

Nội dung phương thức là vẽ 3 cạnh:

cạnh 1 đi qua *this và d2

cạnh 2 đi qua d2 và d3

cạnh 3 đi qua d3 và *this

Các cạnh trên được vẽ nhờ sử dụng phương thức ve_doan_thang:

Vẽ cạnh 1 dùng lệnh: (*this).ve_doan_thang(d2,mau) ;

Vẽ cạnh 2 dùng lệnh: d2.ve_doan_thang(d3,mau);

Vẽ cạnh 3 dùng lệnh: d3.ve_doan_thang(*this,mau);

Trong trường này rõ ràng vai trò của this rất quan trọng. Nếu không dùng nó thì công việc trở nên khó khăn, dài dòng và khó hiểu hơn. Chúng ta hãy so sánh 2 phương án:

Phương án dùng this trong phương thức ve_tam_giac:

```
void DIEM::ve_tam_giac(DIEM d2, DIEM d3,int mau)
{
    (*this).ve_doan_thang(d2,mau);
    d2.ve_doan_thang(d3,mau);
    d3.ve_doan_thang(*this,mau);
}
```

Phương án không dùng this trong phương thức ve_tam_giac:

```
void DIEM::ve_tam_giac(DIEM d2, DIEM d3,int mau)
{
    DIEM d1;
```



```
d1.x = x;
d1.y = y;
d1.ve_doan_thang(d2, mau);
d2.ve_doan_thang(d3, mau);
d3.ve_doan_thang(d1, mau);
}
```

Bài 5. Nói thêm về kiểu phương thức và kiểu đối của phương thức

5.1. Kiểu phương thức

Phương thức có thể không có giá trị trả về (kiểu void) hoặc có thể trả về một giá trị có kiểu bất kỳ, kể cả giá trị kiểu đối tượng, con trỏ đối tượng, tham chiếu đối tượng.

5.2. Đối của phương thức

Đối của phương thức (cũng giống như đối của hàm) có thể có kiểu bất kỳ:

- + Kiểu dữ liệu chuẩn như int, float, char,... . Con trỏ hoặc tham chiếu đến kiểu dữ liệu chuẩn như int*, float*, char*, int&, float&, char&,...
- + Các kiểu ngoài chuẩn đã định nghĩa trước như đối tượng, cấu trúc, hợp, enum,... . Con trỏ hoặc tham chiếu đến các kiểu ngoài chuẩn này.
- + Kiểu đối tượng của chính phương thức, con trỏ hoặc tham chiếu đến kiểu đối tượng này.

5.3. Các ví dụ

Ví dụ 1 minh họa:

- + Thuộc tính (thành phần dữ liệu) của lớp có thể là đối tượng của lớp khác đã định nghĩa bên trên.
- + Phương thức có giá trị trả về kiểu đối tượng và con trỏ đối tượng.

Nội dung chương trình là nhập một dãy hình chữ nhật, sau đó tìm hình chữ nhật có max diện tích và hình chữ nhật có max chu vi.

Chương trình được tổ chức thành 2 lớp:

+ Lớp HINH_CN gồm:

- Các thuộc tính: d và r (chiều dài và chiều rộng)
- Các phương thức

```
void nhapsl() ; // Nhập chiều dài, rộng
```

```
int dien_tich(); // Tính diện tích
```

```
int chu_vi() ; // Tính chu vi
```

+ Lớp DAY_HINH_CN gồm

- Các thuộc tính:

```
int n ; //số hình chữ nhật của dãy
```

```
HINH_CN *h; //Con trỏ tới dãy đối tượng của lớp HINH_CN
```

- Các phương thức

```
void nhapsl(); // Nhập một dãy hình chữ nhật
```

```
HINH_CN hinh_dt_max() ; //Trả về hình chữ nhật có
```

```
// diện tích max
```

```
HINH_CN *hinh_cv_max() ; // Trả về con trỏ tới HCN có
                           // chu vi max

#include <conio.h>
#include <iostream.h>
class HINH_CN
{
private:
    int d, r; // chieu dai va chieu rong
public:
    void nhapsl()
    {
        cout << " \nNhap chieu dai va chieu rong: ";
        cin >> d >> r ;
    }
    void in()
    {
        cout << "\nchieu dai = " << d ;
        cout << " chieu rong= " << r;
    }
    int dien_tich()
    {
        return d*r;
    }
    int chu_vi()
    {
        return 2*(d+r);
    }
};

class DAY_HINH_CN
{
private:
    int n; // So hình ch nhật
    HINH_CN *h;
public:
    void nhapsl();
    HINH_CN hinh_dt_max() ;
    HINH_CN *hinh_cv_max() ;
};

void DAY_HINH_CN::nhapsl()
{
    cout << "So hình CN = " ;
```

```
cin >> n;
h = new HINH_CN[n+1];
for (int i=1; i<=n; ++i)
    h[i].nhapsl();
}

HINH_CN DAY_HINH_CN::hinh_dt_max()
{
    HINH_CN hdtmax;
    hdtmax = h[1];
    for (int i=2; i<=n; ++i)
        if (h[i].dien_tich() > hdtmax.dien_tich() )
            hdtmax = h[i];
    return hdtmax;
}

HINH_CN *DAY_HINH_CN::hinh_cv_max()
{
    int imax = 1;
    for (int i=2; i<=n; ++i)
        if (h[i].chu_vi() > h[imax].chu_vi() )
            imax = i ;
    return (h+imax);
}

void main()
{
    DAY_HINH_CN d;
    HINH_CN hdtmax;
    d.nhapsl();
    hdtmax = d.hinh_dt_max();
    hdtmax.in() ;
    HINH_CN *hcvmax=d.hinh_cv_max();
    hcvmax->in() ;
    getch();
}
```

Ví dụ 2 minh hoạ:

- + Thuộc tính (thành phần dữ liệu) của lớp có thể là đối tượng của lớp khác đã định nghĩa bên trên.
- + Phương thức có giá trị trả về kiểu đối tượng
- + Vai trò của con trỏ this (xem phương thức maxdt của lớp TAM_GIAC)
- + Phương thức tĩnh (xem phương thức tao_tg của lớp TAM_GIAC)

Nội dung chương trình là nhập một dãy các điểm, sau đó tìm tam giác lớn nhất (về diện tích) có đỉnh là các điểm vừa nhập.

Chương trình được tổ chức thành 2 lớp:

+ Lớp DIEM gồm:

- Các thuộc tính: x và y (toạ độ của điểm)

- Các phương thức

```
void nhapsl() ; // Nhập x, y
```

```
void in() ; // In toạ độ
```

```
double do_dai(DIEM d2) ; // Tính độ dài đoạn thẳng qua  
// 2 điểm (điểm ẩn xác định bởi this và điểm d2)
```

+ Lớp TAM_GIAC gồm:

- Các thuộc tính:

```
DIEM d1,d2,d3; // 3 đỉnh của tam giác
```

- Các phương thức:

```
void nhapsl(); // Nhập toạ độ 3 đỉnh
```

```
void in(); // In toạ độ 3 đỉnh
```

```
// Tạo một đối tượng TAM_GIAC từ 3 đối tượng DIEM
```

```
static TAM_GIAC tao_tg(DIEM e1, DIEM e2, DIEM e3)
```

```
double dien_tich() ; // Tính diện tích
```

```
// Tìm tam giác có diện tích max trong 2 tam giác *this và t2
```

```
TAM_GIAC maxdt(TAM_GIAC t2);
```

+ Các vấn đề đáng chú ý trong chương trình là:

- Phương thức tĩnh tao_tg (sẽ giải thích bên dưới)

- Phương thức maxdt

+ Thuật toán là:

- Duyệt qua các tổ hợp 3 điểm.

- Dùng phương thức tao_tg để lập tam giác từ 3 điểm

- Dùng phương thức maxdt để chọn tam giác có diện tích lớn hơn trong 2 tam giác: tam giác vừa tạo và tam giác có diện tích max (trong số các tam giác đã tạo)

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

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

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

```
class DIEM
```

```
{
```

```
private:
```

```
double x,y; // Toa do cua diem
```

```
public:
```

```
void nhapsl()
```

```
{
```

```
cout << " Toa do x, y: " ;
```

```
        cin >> x >> y ;
    }
    void in()
    {
        cout << "x = " << x << "y = " << y;
    }
    double do_dai(DIEM d2)
    {
        return sqrt(pow(x-d2.x,2) + pow(y-d2.y,2) );
    }
};

class TAM_GIAC
{
private:
    DIEM d1,d2,d3; // 3 đỉnh tam giác
public:
    void nhapsl();
    void in();
    static TAM_GIAC tao_tg(DIEM e1, DIEM e2, DIEM e3)
    {
        TAM_GIAC t;
        t.d1=e1; t.d2 = e2; t.d3=e3;
        return t;
    }
    double dien_tich() ;
    TAM_GIAC maxdt(TAM_GIAC t2);
};

void TAM_GIAC::nhapsl()
{
    cout << "\nĐỉnh 1 - ";
    d1.nhapsl();
    cout << "\nĐỉnh 2 - ";
    d2.nhapsl();
    cout << "\nĐỉnh 3 - ";
    d3.nhapsl();
}

void TAM_GIAC::in()
{
    cout << "\nĐỉnh 1: "; d1.in();
```

```
cout << "\nDinh 2: "; d2.in();
cout << "\nDinh 3: "; d3.in();
}
double TAM_GIAC::dien_tich()
{
    double a,b,c,p,s;
    a=d1.do_dai(d2);
    b=d2.do_dai(d3);
    c=d3.do_dai(d1);
    p=(a+b+c)/2;
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
TAM_GIAC TAM_GIAC::maxdt(TAM_GIAC t2)
{
    if (this->dien_tich() > t2.dien_tich())
        return *this ;
    else
        return t2;
}
void main()
{
    DIEM d[50];
    int n, i ;
    clrscr();
    cout << "\n So diem= ";
    cin >> n;
    for (i=1; i<=n; ++i)
    {
        cout << "\nNhap diem " << i << " - ";
        d[i].nhapsl();
    }
    int j, k ;
    TAM_GIAC tmax, t;
    tmax = TAM_GIAC::tao_tg(d[1],d[2],d[3]);
    for (i=1;i<=n-2;++i)
        for (j=i+1;j<=n-1;++j)
            for (k=j+1;k<=n;++k)
            {
                t=TAM_GIAC::tao_tg(d[i],d[j],d[k]);
                tmax = tmax.maxdt(t);
            }
}
```

```
cout << "\n\nTam giac co dien tich lon nhat: " ;
tmax.in();
cout << "\nDien tich = " << tmax.dien_tich();
getch();
}
```

Chú ý 1: Để tạo một đối tượng TAM_GIAC từ 3 đối tượng DIEM ta đã dùng phương thức tĩnh:
static TAM_GIAC tao_tg(DIEM e1, DIEM e2, DIEM e3)

```
{
    TAM_GIAC t;
    t.d1=e1; t.d2 = e2; t.d3=e3;
    return t;
}
```

Phương thức tĩnh (sẽ nói thêm trong các mục bên dưới) có các đặc điểm sau:

+ Nó giống phương thức thông thường ở chỗ: Trong thân của nó có thể truy nhập tới các thành phần của lớp (cụ thể là lớp TAM_GIAC).

+ Nó khác phương thức thông thường ở chỗ:

- Không có đối ngầm định xác định bởi con trỏ this (như phương thức thông thường). Như vậy phương thức tao_tg có đúng 3 đối.

- Nó không gắn với một đối tượng cụ thể nào của lớp, nên trong lời gọi tới phương thức ảo có thể dùng tên lớp, ví dụ (xem hàm main):

```
t=TAM_GIAC::tao_tg(d[i],d[j],d[k]);
```

Chú ý 2: Không thể thay phương thức tĩnh tao_tg bằng hàm, vì trong thân hàm không được truy xuất đến các thuộc tính của lớp TAM_GIAC. Tuy nhiên có một giải pháp khác là dùng khái niệm hàm bạn (friend). Hàm bạn của một lớp có quyền truy nhập đến các thuộc tính của lớp. Trong ví dụ 3 dưới đây ta sẽ xây dựng hàm tao_tg như một hàm bạn của lớp TAM_GIAC.

Chú ý 3: còn một giải pháp nữa là dùng hàm tạo (constructor) sẽ trình bày trong các chương sau:

Chương trình dưới đây có nội dung giống như ví dụ 2, nhưng thay phương thức tĩnh tao_tg bằng hàm bạn tao_tg.

Ví dụ 3: Minh họa cách dùng hàm bạn. Nội dung chương trình giống như trong ví dụ 2.

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DIEM
{
    private:
        double x,y; // Toa do cua diem
    public:
        void nhapsl()
        {
            cout << " Toa do x, y: " ;
            cin >> x >> y ;
        }
        void in()
```

```
{
    cout << "x = " << x << "y = " << y;
}

double do_dai(DIEM d2)
{
    return sqrt(pow(x-d2.x,2) + pow(y-d2.y,2) );
}

};

class TAM_GIAC
{
private:
    DIEM d1,d2,d3; // 3 đỉnh tam giác
public:
    void nhapsl();
    void in();
    friend TAM_GIAC tao_tg(DIEM e1, DIEM e2, DIEM e3)
    {
        TAM_GIAC t;
        t.d1=e1; t.d2 = e2; t.d3=e3;
        return t;
    }
    double dien_tich() ;
    TAM_GIAC maxdt(TAM_GIAC t2);
};

void TAM_GIAC::nhapsl()
{
    cout << "\nĐỉnh 1 - ";
    d1.nhapsl();
    cout << "\nĐỉnh 2 - ";
    d2.nhapsl();
    cout << "\nĐỉnh 3 - ";
    d3.nhapsl();
}

void TAM_GIAC::in()
{
    cout << "\nĐỉnh 1: "; d1.in();
    cout << "\nĐỉnh 2: "; d2.in();
    cout << "\nĐỉnh 3: "; d3.in();
}
```



```
double TAM_GIAC::dien_tich()
{
    double a,b,c,p,s;
    a=d1.do_dai(d2);
    b=d2.do_dai(d3);
    c=d3.do_dai(d1);
    p=(a+b+c)/2;
    return sqrt(p*(p-a)*(p-b)*(p-c));
}

TAM_GIAC TAM_GIAC::maxdt(TAM_GIAC t2)
{
    if (this->dien_tich() > t2.dien_tich())
        return *this ;
    else
        return t2;
}

void main()
{
    DIEM d[50];
    int n, i ;
    clrscr();
    cout << "\n So diem = ";
    cin >> n;
    for (i=1; i<=n; ++i)
    {
        cout << "\nNhap diem " << i << " - ";
        d[i].nhapsl();
    }
    int j, k ;
    TAM_GIAC tmax, t;
    tmax = tao_tg(d[1],d[2],d[3]);
    for (i=1;i<=n-2;++i)
        for (j=i+1;j<=n-1;++j)
            for (k=j+1;k<=n;++k)
            {
                t=tao_tg(d[i],d[j],d[k]);
                tmax = tmax.maxdt(t);
            }
    cout << "\n\nTam giac co dien tich lon nhat: ";
    tmax.in();
}
```

```
cout << "\nDien tich = " << tmax.dien_tich();
getch();
}
```

Chú ý: Hàm bạn có thể xây dựng bên trong định nghĩa lớp (như chương trình trên) hoặc có thể khai báo bên trong và xây dựng bên ngoài định nghĩa lớp như sau:

```
class TAM_GIAC
{
private:
    DIEM d1,d2,d3; // 3 dinh tam giac
public:
    void nhapsl();
    void in();
    friend TAM_GIAC tao_tg(DIEM e1,DIEM e2,DIEM e3);
    double dien_tich() ;
    TAM_GIAC maxdt(TAM_GIAC t2);
};

TAM_GIAC tao_tg(DIEM e1, DIEM e2, DIEM e3)
{
    TAM_GIAC t;
    t.d1=e1; t.d2 = e2; t.d3=e3;
    return t;
}
```

Nhận xét: Không cho phép dùng từ khoá *friend* khi xây dựng hàm (bên ngoài lớp)

Bài 6. Hàm, hàm bạn

6.1. Hàm có các tính chất sau:

+ Phạm vi của hàm là toàn bộ chương trình, vì vậy hàm có thể được gọi tới từ bất kỳ chỗ nào. Như vậy trong các phương thức có thể sử dụng hàm.

+ Đối của hàm có thể là các đối tượng, tuy nhiên có một hạn chế là trong thân hàm không cho phép truy nhập tới thuộc tính của các đối này. Ví dụ giả sử đã định nghĩa lớp:

```
class DIEM
{
private:
    double x,y; // Toa do cua diem
public:
    void nhapsl()
    {
        cout << "Toa do x, y: ";
        cin >> x >> y ;
    }
}
```

```
    }  
    void in()  
    {  
        cout << "x = " << x << "y = " << y;  
    }  
};
```

Dùng lớp DIEM, ta xây dựng hàm tính độ dài của đoạn thẳng đi qua 2 điểm như sau:

```
double do_dai(DIEM d1, DIEM d2)  
{  
    return sqrt(pow(d1.x-d2.x,2) + pow(d1.y-d2.y,2));  
}
```

Hàm này sẽ bị báo lỗi khi dịch, vì trong thân hàm không cho phép sử dụng các thuộc tính d1.x, d1.y, d2.x, d2.y của các đối tượng d1 và d2 thuộc lớp DIEM.

+ Phạm vi sử dụng của các phương thức (public) là toàn chương trình, vì vậy trong thân hàm có thể gọi tới các phương thức. Ví dụ giả sử đã định nghĩa lớp:

```
class DIEM  
{  
    private:  
        double x,y; // Toa do cua diem  
    public:  
        void nhapsl()  
        {  
            cout << "Toa do x, y: ";  
            cin >> x >> y;  
        }  
        void in()  
        {  
            cout << "x = " << x << "y = " << y;  
        }  
        double do_dai(DIEM d2)  
        {  
            return sqrt(pow(x-d2.x,2) + pow(y-d2.y,2));  
        }  
};
```

Khi đó bằng cách dùng phương thức do_dai, ta có thể viết hàm tính diện tích tam giác có đỉnh là các đối tượng d1, d2, d3 của lớp DIEM như sau:

```
double dt_tg(DIEM d1, DIEM d2, DIEM d3)  
{  
    double a,b,c,p,s;
```

```
a=d1.do_dai(d2);
b=d2.do_dai(d3);
c=d3.do_dai(d1);
p=(a+b+c)/2;
return sqrt(p*(p-a)*(p-b)*(p-c));
}
```

Bằng cách dùng hàm `dt_tg`, có thể tổ chức lại chương trình tìm tam giác có diện tích lớn nhất (ở mục trên) một cách đơn giản hơn(bỏ đi lớp `TAM_GIAC`) như ví dụ sau.

Ví dụ 1:

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DIEM
{
private:
    double x,y; // Toa do cua diem
public:
    void nhapsl()
    {
        cout << " Toa do x, y: ";
        cin >> x >> y ;
    }
    void in()
    {
        cout << " x = " << x << " y = " << y;
    }
    double do_dai(DIEM d2)
    {
        return sqrt(pow(x-d2.x,2) + pow(y-d2.y,2) );
    }
};
double dt_tg(DIEM d1, DIEM d2, DIEM d3)
{
    double a,b,c,p,s;
    a=d1.do_dai(d2);
    b=d2.do_dai(d3);
    c=d3.do_dai(d1);
    p=(a+b+c)/2;
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
void main()
```

```
{
    DIEM d[50];
    int n, i, j, k, imax, jmax, kmax ;
    clrscr();
    cout << "\n So diem = ";
    cin >> n;
    for (i=1; i<=n; ++i)
    {
        cout << "\nNhap diem " << i << " - ";
        d[i].nhapsl();
    }
    imax=1; jmax=2; kmax=3;
    for (i=1; i<=n-2; ++i)
        for (j=i+1; j<=n-1; ++j)
            for (k=j+1; k<=n; ++k)
                if (dt_tg(d[i], d[j], d[k]) > dt_tg(d[imax], d[jmax], d[kmax]))
                {
                    imax = i ;
                    jmax = j;
                    kmax = k;
                }
    cout << "\n\nTam giac co dien tich lon nhat: ";
    cout << "\nDinh 1 - "; d[imax].in();
    cout << "\nDinh 2 - "; d[jmax].in();
    cout << "\nDinh 3 - "; d[kmax].in();
    cout << "\nDien tich = " << dt_tg(d[imax], d[jmax], d[kmax]) ;
    getch();
}
```

Nhận xét: Chương trình trên làm việc trên mảng d kiểu DIEM. Bây giờ nếu ta dùng mảng ngoài thì từ số thứ tự sẽ suy ra phần tử của mảng. Như vậy hàm

```
double dt_tg(DIEM d1, DIEM d2, DIEM d3);
```

có 3 đối kiểu DIEM có thể thay bằng hàm có 3 đối nguyên:

```
double dt_tg(int i, int j, int k);
```

để tính diện tích tam giác có đỉnh là d[i], d[j] và d[k] . ý tưởng này được thể hiện trong ví dụ sau.

Ví dụ 2: Chương trình dùng mảng đối tượng ngoài.

Chú ý: Khai báo mảng đối tượng phải đặt sau định nghĩa kiểu đối tượng (định nghĩa lớp).

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

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

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

```
double dt_tg(int i, int j, int k); // Khai báo hàm dt_tg
```

```
class DIEM
```

```
{
```

```
    private:
```

```
double x,y; // Toa do cua diem
public:
    void nhapsl();
    void in();
    double do_dai(DIEM d2);
};
// Chú ý: Khai báo mảng kiểu DIEM phải đặt sau định nghĩa
// lớp DIEM
DIEM d[50];
void DIEM::nhapsl()
{
    cout << " Toa do x, y: ";
    cin >> x >> y;
}
void DIEM::in()
{
    cout << " x = " << x << " y = " << y;
}
double DIEM::do_dai(DIEM d2)
{
    return sqrt(pow(x-d2.x,2) + pow(y-d2.y,2) );
}
double dt_tg(int i, int j, int k)
{
    double a,b,c,p,s;
    a=d[i].do_dai(d[j]);
    b=d[j].do_dai(d[k]);
    c=d[k].do_dai(d[i]);
    p=(a+b+c)/2;
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
void main()
{
    int n, i,j,k,imax,jmax,kmax ;
    clrscr();
    cout << "\n So diem = ";
    cin >> n;
    for (i=1; i<=n; ++i)
    {
        cout << "\nNhap diem " << i << " - ";
        d[i].nhapsl();
    }
}
```

```
    }
    imax=1; jmax=2; kmax=3;
    for (i=1; i<=n-2; ++i)
        for (j=i+1; j<=n-1; ++j)
            for (k=j+1; k<=n; ++k)
                if (dt_tg(i,j,k) > dt_tg(imax,jmax,kmax))
                    {
                        imax = i ;
                        jmax = j;
                        kmax = k;
                    }
    cout << "\n\nTam giác có diện tích lớn nhất: " ;
    cout << "\nĐỉnh 1 - "; d[imax].in();
    cout << "\nĐỉnh 2 - "; d[jmax].in();
    cout << "\nĐỉnh 3 - "; d[kmax].in();
    cout << "\nDiện tích = " << dt_tg(imax,jmax,kmax);
    getch();
}
```

6.2. Hàm bạn (friend function)

6.2.1. Để một hàm trở thành bạn của một lớp, có 2 cách viết:

Cách 1: Dùng từ khoá friend để khai báo hàm trong lớp và xây dựng hàm bên ngoài như các hàm thông thường (không dùng từ khoá friend). Mẫu viết như sau:

```
class A
{
    private:
        // Khai báo các thuộc tính
    public:
        ...
        // Khai báo các hàm bạn của lớp A
        friend void f1(...);
        friend double f2(...);
        friend A f3(...);
        ...
};

// Xây dựng các hàm f1, f2, f3
void f1(...)
{
    ...
}
```

```
double f2(...)
{
    ...
}
A f3(...)
{
    ...
}
```

Cách 2: Dùng từ khoá friend để xây dựng hàm trong định nghĩa lớp. Mẫu viết như sau:

```
class A
{
    private:
        // Khai báo các thuộc tính
    public:
        ...
        // Xây dựng các hàm bạn của lớp A
        void f1(...)
        {
            ...
        }
        double f2(...)
        {
            ...
        }
        A f3(...)
        {
            ...
        }
        ...
};
```

6.2.2. Tính chất của hàm bạn

Trong thân hàm bạn của một lớp có thể truy nhập tới các thuộc tính của các đối tượng thuộc lớp này. Đây là sự khác nhau duy nhất giữa hàm bạn và hàm thông thường. Chú ý rằng hàm bạn không phải là phương thức của lớp. Phương thức có một đối ẩn (ứng với con trỏ this) và lời gọi của phương thức phải gắn với một đối tượng nào đó (địa chỉ đối tượng này được truyền cho con trỏ this). Lời gọi của hàm bạn giống như lời gọi của hàm thông thường.

Ví dụ sau sẽ so sánh phương thức, hàm bạn và hàm tự do (hàm thông thường). Xét lớp SP (số phức). Hãy so sánh 3 phương án để thực hiện việc cộng 2 số phức:

Phương án 1: Dùng phương thức

```
class SP
```



```
{
    private:
        double a; // Phần thực
        double b; // Phần ảo
    public:
        SP cong(SP u2)
        {
            SP u;
            u.a = this->a + u2.a ;
            u.b = this->b + u2.b ;
            return u;
        }
};
```

Cách dùng

```
SP u, u1, u2;
u = u1.cong(u2);
```

Phương án 2: Dùng hàm bạn

```
class SP
{
    private:
        double a; // Phần thực
        double b; // Phần ảo
    public:
        friend SP cong(SP u1, SP u2)
        {
            SP u;
            u.a = u1.a + u2.a ;
            u.b = u1.b + u2.b ;
            return u;
        }
};
```

Cách dùng

```
SP u, u1, u2;
u = cong(u1, u2);
```

Phương án 3: Dùng hàm tự do

```
class SP
{
    private:
```

```
double a; // Phần thực
double b; // Phần ảo
public:
    ...
};

SP cong(SP u1, SP u2)
{
    SP u;
    u.a = u1.a + u2.a;
    u.b = u1.b + u2.b;
    return u;
}
```

Phương án này không được chấp nhận, Trình biên dịch sẽ báo lỗi vì trong thân hàm không được quyền truy xuất đến các thuộc tính riêng (private) a, b của các đối tượng u, u1 và u2 thuộc lớp SP.

6.2.3. Một hàm có thể là bạn của nhiều lớp được không? Câu trả lời là được. Khi một hàm là bạn của nhiều lớp, thì nó có quyền truy nhập tới tất cả các thuộc tính của các đối tượng trong các lớp này. Để làm cho hàm f trở thành bạn của các lớp A, B và C ta sử dụng mẫu viết sau:

```
class B; // Khai báo trước lớp A
class B; // Khai báo trước lớp B
class C; // Khai báo trước lớp C
// Định nghĩa lớp A
class A
{
    // Khai báo f là bạn của A
    friend void f(...);
};

// Định nghĩa lớp B
class B
{
    // Khai báo f là bạn của B
    friend void f(...);
};

// Định nghĩa lớp C
class C
{
    // Khai báo f là bạn của C
    friend void f(...);
};

// Xây dựng hàm f
```

```
void f(...)\n{\n    ... \n}
```

Chương trình sau đây minh họa cách dùng hàm bạn (bạn của một lớp và bạn của nhiều lớp). Chương trình đưa vào 2 lớp VT (véc tơ), MT (ma trận) và 3 hàm bạn để thực hiện các thao tác trên 2 lớp này:

```
// Hàm bạn với lớp VT dùng để in một véc tơ\nfriend void in(const VT &x);\n// Hàm bạn với lớp MT dùng để in một ma trận\nfriend void in(const MT &a);\n// Hàm bạn với cả 2 lớp MT và VT dùng để nhân ma trận với véc tơ\nfriend VT tich(const MT &a,const VT &x);
```

Nội dung chương trình là nhập một ma trận vuông cấp n và một véc tơ cấp n, sau đó thực hiện phép nhân ma trận với véc tơ vừa nhập.

```
// Chương trình CT3_09.CPP\n#include <conio.h>\n#include <iostream.h>\n#include <math.h>\n\nclass VT;\nclass MT ;\nclass VT\n{\n    private:\n        int n;\n        double x[20]; // Toa do cua diem\n    public:\n        void nhapsl();\n        friend void in(const VT &x);\n        friend VT tich(const MT &a,const VT &x) ;\n};\nclass MT\n{\n    private:\n        int n;\n        double a[20][20];\n    public:\n        friend VT tich(const MT &a,const VT &x);\n        friend void in(const MT &a);\n        void nhapsl();\n};
```

```
void VT::nhapsl()
```

```
{
    cout << "\n Cap vec to = ";
    cin >> n;
    for (int i=1; i<=n; ++i)
    {
        cout << "\nPhan tu thu " << i << " = ";
        cin >> x[i];
    }
}
```

```
void MT::nhapsl()
```

```
{
    cout << "\n Cap ma tran = ";
    cin >> n;
    for (int i=1; i<=n; ++i)
        for (int j=1; j<=n; ++j)
        {
            cout << "\nPhan tu thu hang " << i << " cot " << j << " = ";
            cin >> a[i][j];
        }
}
```

```
VT tich(const MT &a,const VT &x)
```

```
{
    VT y;
    int n=a.n;
    if (n!=x.n)
        return x;
    y.n = n;
    for (int i=1; i<=n; ++i)
    {
        y.x[i]=0;
        for (int j=1; j<=n; ++j)
            y.x[i] += a.a[i][j]*x.x[j];
    }
    return y;
}
```

```
void in(const VT &x)
```

```
{
    cout << "\n";
}
```

```
    for (int i=1; i<=x.n; ++i)
        cout << x.x[i] << " ";
    }
void in(const MT &a)
{
    for (int i=1; i<=a.n; ++i)
    {
        cout << "\n";
        for (int j=1; j<=a.n; ++j)
            cout << a.a[i][j] << " ";
        }
    }
void main()
{
    MT a; VT x,y;
    clrscr();
    a.nhapsl();
    x.nhapsl();
    y=tich(a,x);
    clrscr();
    cout << "\nMa tran A:";
    in(a);
    cout << "\n\nVec to x: ";
    in(x);
    cout << "\n\nVec y = Ax: ";
    in(y);
    getch();
}
```

Bài 7. Phạm vi truy xuất

7.1. Các từ khoá private và public

Các thành phần (thuộc tính và phương thức) của lớp có thể khai báo là private hoặc public theo mẫu:

private:

// Khai báo các thành phần riêng của lớp

public:

// Khai báo các thành phần chung (công cộng)

Chú ý: Các thành phần khai báo mặc định (không dùng các từ khoá private và public) được xem là các thành phần private.

7.2. Các thành phần riêng của lớp chỉ được sử dụng trong phạm vi của lớp (trong thân các phương thức của lớp). Chúng không thể đem ra sử dụng bên ngoài lớp.

+ Một thuộc tính private: Thuộc tính này (của một đối tượng nào đó) chỉ có thể được sử dụng trong thân của các phương thức cùng lớp.

+ Một phương thức private: Chỉ được sử dụng trong thân của các phương thức cùng lớp.

Ví dụ sau minh hoạ cách dùng phương thức private. Xét lớp PS (phân số) với 2 thuộc tính nguyên là t (tử) và m (mẫu). Giả sử cần xây dựng các phương thức để thực hiện các phép toán cộng trừ, nhân, chia phân số. Do các phép toán này cần dùng trong toàn bộ chương trình, nên các phương thức thực hiện các phép toán cần khai báo là public. Để thực hiện các phép tính trên phân số cần dùng đến phép rút gọn phân số. Ta có thể dùng một phương thức private để làm điều này vì việc rút gọn chỉ dùng trong nội bộ lớp.

7.3. Các thành phần công cộng của lớp có phạm vi sử dụng trong toàn chương trình. Như vậy nếu một thuộc tính được khai báo là public, thì nó có thể được truy nhập trong thân của bất kỳ hàm nào trong chương trình.

Ví dụ trong §6 đã chỉ ra phương án dùng một hàm (tự do) để thực hiện phép cộng 2 số phức như sau là sai:

Phương án 3: Dùng hàm tự do

```
class SP
{
    private:
        double a; // Phần thực
        double b; // Phần ảo
    public:
        ...
};
```

```
SP cong(SP u1, SP u2)
{
    SP u;
    u.a = u1.a + u2.a;
    u.b = u1.b + u2.b;
    return u;
}
```

Tuy nhiên nếu sửa chữa bằng cách khai báo các thuộc tính a và b là public thì lại được.

Nhận xét: Các thuộc tính thường khai báo là private để đảm bảo tính dấu kín, an toàn dữ liệu của lớp.

Bài 8. Các phương thức toán tử

8.1. Cách đặt tên

Các phương thức toán tử được xây dựng như các phương thức thông thường, chỉ có khác cách đặt tên. Tên các phương thức toán tử (cũng giống như hàm toán tử) được tạo bằng cách ghép từ khoá operator với một phép toán, ví dụ:

```
operator+  
operator<<  
operator>>
```

8.2. Con trỏ this

Cũng giống như phương thức thông thường, phương thức toán tử có đối đầu tiên (đối không tường minh) là con trỏ this.

8.3. Toán tử một toán hạng

Các phương thức toán tử một toán hạng: Dùng ngay con trỏ this để biểu thị toán hạng duy nhất này, nên trong phương thức sẽ không có đối tường minh. Ví dụ phương thức toán tử - (đổi dấu) một đối tượng kiểu SP (số phức) có thể viết như sau:

```
class SP  
{  
    private:  
        double a; // Phần thực  
        double b; // Phần ảo  
    public:  
        SP operator-();  
};
```

```
SP SP:: operator-()  
{  
    SP u ;  
    u.a = - this->a ;  
    u.b = - this->b ;  
    return u;  
}
```

Cách dùng:

```
SP    u, v;  
u = -v;
```

8.4. Toán tử hai toán hạng

Các phương thức toán tử hai toán hạng: Con trỏ this ứng với toán hạng thứ nhất, nên trong phương thức chỉ cần dùng một đối tượng minh để biểu thị toán hạng thứ hai. Ví dụ phương thức toán tử + (cộng) hai đối tượng kiểu SP (số phức) có thể viết như sau:

```
class SP  
{  
    private:  
        double a; // Phần thực  
        double b; // Phần ảo
```

```
public:
    SP operator+(SP u2);
};

SP SP::operator+(SP u2)
{
    SP u;
    u.a = this->a + u2.a;
    u.b = this->b + u2.b;
    return u;
}
```

Cách dùng:

```
SP p, p, r;
r = p + q;
```

8.5. Lớp DT (Đa thức)

Chương trình sau sẽ định nghĩa lớp DT và đưa vào các phương thức, hàm:

+ Các thuộc tính:

```
int n; // bậc đa thức
double *a; // trỏ tới vùng nhớ chứa các hệ số đa thức
```

+ Các phương thức operator+, operator- dùng để đổi dấu các hệ số đa thức

```
operator+    dùng để cộng 2 đa thức
operator-    dùng để trừ 2 đa thức
operator*    dùng để nhân 2 đa thức
operator^    dùng để tính giá trị đa thức
operator[]    dùng để cho biết bậc và hệ số của đa thức
```

+ Các hàm bạn:

```
operator<<    dùng để in các hệ số đa thức
operator>>    dùng để nhập các hệ số đa thức
```

+ Hàm (tự do)

double F(DT p, double x) dùng để tính p(x)-giá trị đa thức tại x

+ Nói thêm về phương thức chỉ số và hàm tự do F

- Nếu p là đối tượng của lớp DT, thì hàm chỉ số cho biết:

```
p[-1] = double(n)
p[i] = a[i], i=0, 1, ..., n
```

- Hàm tự do F sẽ dùng phương thức chỉ số để xác định n, các hệ số đa thức và dùng chúng để tính giá trị đa thức.

+ Trong chương trình sử dụng hàm new để cấp phát vùng nhớ chứa hệ số đa thức.

+ Nội dung chương trình gồm:

- Nhập, in các đa thức p, q, r, s
- Tính đa thức: $f = -(p + q) * (r - s)$

- Nhập các số thực x1 và x2
- Tính $f(x1)$ (bằng cách dùng phương thức operator[^])
- Tính $f(x2)$ (bằng cách dùng hàm F)

// Chương trình CT3_10.CPP

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

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

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

```
class DT
```

```
{
```

```
    private:
```

```
        int n; // Bac da thuc
```

```
        double *a; // Tro toi vung nho chua cac he so da thuc  
                // a0, a1,...
```

```
    public:
```

```
        friend ostream& operator<< (ostream& os,const DT &d);
```

```
        friend istream& operator>> (istream& is,DT &d);
```

```
        DT operator-();
```

```
        DT operator+(const DT &d2);
```

```
        DT operator-(DT d2);
```

```
        DT operator*(const DT &d2);
```

```
        double operator^(const double &x); // Tinh gia tri da thuc
```

```
        double operator[](int i)
```

```
        {
```

```
            if(i<0)
```

```
                return double(n);
```

```
            else
```

```
                return a[i];
```

```
        }
```

```
    };
```

```
// Ham tinh gia tri da thuc
```

```
double F(DT d,double x)
```

```
{
```

```
    double s=0.0 , t=1.0;
```

```
    int n;
```

```
    n = int(d[-1]);
```

```
    for (int i=0; i<=n; ++i)
```

```
    {
```

```
        s += d[i]*t;
```

```
        t *= x;
```

```
    }
```

```
    return s;
```

```
}
```

```
ostream& operator<< (ostream& os,const DT &d)
```

```
{
    os << " - Cac he so (tu ao): ";
    for (int i=0 ; i<= d.n ; ++i)
        os << d.a[i] << " ";
    return os;
}

istream& operator>> (istream& is,DT &d)
{
    cout << " - Bac da thuc: ";
    cin >> d.n;
    d.a = new double[d.n+1];
    cout << "Nhap cac he so da thuc:\n";
    for (int i=0 ; i<= d.n ; ++i)
    {
        cout << "He so bac " << i << " = ";
        is >> d.a[i] ;
    }
    return is;
}

DT DT::operator-()
{
    DT p;
    p.n = n;
    p.a = new double[n+1];
    for (int i=0 ; i<=n ; ++i)
        p.a[i] = -a[i];
    return p;
}

DT DT::operator+(const DT &d2)
{
    DT d;
    int k,i;
    k = n > d2.n ? n : d2.n ;
    d.a = new double[k+1];
    for (i=0; i<=k ; ++i)
        if (i<=n && i<=d2.n)
            d.a[i] = a[i] + d2.a[i];
        else if (i<=n)
            d.a[i] = a[i];
        else
            d.a[i] = d2.a[i];
}
```

```
i=k;
while(i>0 && d.a[i]==0.0) --i;
d.n = i;
return d ;
}

DT DT::operator-(DT d2)
{
    return (*this + (-d2));
}

DT DT::operator*(const DT &d2)
{
    DT d;
    int k, i, j;
    k = d.n = n + d2.n ;
    d.a = new double[k+1];
    for (i=0; i<=k; ++i) d.a[i] = 0;
    for (i=0 ; i<= n ; ++i)
        for (j=0 ; j<= d2.n ; ++j)
            d.a[i+j] += a[i]*d2.a[j] ;
    return d;
}

double DT::operator^(const double &x)
{
    double s=0.0 , t=1.0;
    for (int i=0 ; i<= n ; ++i)
    {
        s += a[i]*t;
        t *= x;
    }
    return s;
}

void main()
{
    DT p,q,r,s,f;
    double x1,x2,g1,g2;
    clrscr();
    cout <<"\nNhap da thuc P " ; cin >> p;
    cout << "\nDa thuc p " << p ;
    cout <<"\nNhap da thuc Q " ; cin >> q;
```

```
cout << "\nDa thuc q " << q ;
cout << "\nNhap da thuc R " ; cin >> r;
cout << "\nDa thuc r " << r ;
cout << "\nNhap da thuc S " ; cin >> s;
cout << "\nDa thuc s " << s ;
f = -(p+q)*(r-s);
cout << "\nNhap so thuc x1: " ; cin >> x1;
cout << "\nNhap so thuc x2: " ; cin >> x2;
g1 = f^x1;
g2 = F(f,x2);
cout << "\nDa thuc f " << f ;
cout << "\nf(" << x1 << ") = " << g1;
cout << "\nf(" << x2 << ") = " << g2;
getch();
}
```

Chương 4

Hàm tạo, hàm huỷ và các vấn đề liên quan

Chương này trình bày một số vấn đề có tính chuyên sâu hơn về lớp như:

- + Hàm tạo (constructor)
- + Hàm huỷ (destructor)
- + Toán tử gán và hàm tạo sao chép
- + Mối liên quan giữa hàm tạo và đối tượng thành phần
- + Các thành phần tĩnh
- + Lớp bạn, hàm bạn
- + Đối tượng hằng
- + Phương thức inline

Bài 1. Hàm tạo (constructor)

1.1. Công dụng

Hàm tạo cũng là một phương thức của lớp (nhưng khá đặc biệt) dùng để tạo dựng một đối tượng mới. Chương trình dịch sẽ cấp phát bộ nhớ cho đối tượng sau đó sẽ gọi đến hàm tạo. Hàm tạo sẽ khởi gán giá trị cho các thuộc tính của đối tượng và có thể thực hiện một số công việc khác nhằm chuẩn bị cho đối tượng mới.

1.2. Cách viết hàm tạo

1.2.1. Điểm khác của hàm tạo và các phương thức thông thường

Khi viết hàm tạo cần để ý 3 sự khác biệt của hàm tạo so với các phương thức khác như sau:

- + Tên của hàm tạo: Tên của hàm tạo bắt buộc phải trùng với tên của lớp.
- + Không khai báo kiểu cho hàm tạo.
- + Hàm tạo không có kết quả trả về.

1.2.2. Sự giống nhau của hàm tạo và các phương thức thông thường

Ngoài 3 điểm khác biệt trên, hàm tạo được viết như các phương thức khác:

- + Hàm tạo có thể được xây dựng bên trong hoặc bên ngoài định nghĩa lớp.
- + Hàm tạo có thể có đối hoặc không có đối.
- + Trong một lớp có thể có nhiều hàm tạo (cùng tên nhưng khác bộ đối).

Ví dụ sau định nghĩa lớp DIEM_DH (Điểm đồ hoạ) có 3 thuộc tính:

int x; // hoành độ (cột) của điểm

int y; // tung độ (hàng) của điểm

int m; // màu của điểm

và đưa vào 2 hàm tạo để khởi gán cho các thuộc tính của lớp:

// Hàm tạo không đối: Dùng các giá trị cố định để khởi gán cho

// x, y, m

DIEM_DH();

// Hàm tạo có đối: Dùng các đối x1, y1, m1 để khởi gán cho

// x, y, m

```
// Đối m1 có giá trị mặc định 15 (màu trắng)
DIEM_DH(int x1, int y1, int m1=15);
class DIEM_DH
{
private:
    int x, y, m;
public:
    //Hàm tạo không đối: khởi gán cho x=0, y=0, m=1
    // Hàm này viết bên trong định nghĩa lớp
    DIEM_DH()
    {
        x=y=0;
        m=1;
    }
    // Hàm tạo này xây dựng bên ngoài định nghĩa lớp
    DIEM_DH(int x1, int y1, int m1=15);
    // Các phương thức khác
};
// Xây dựng hàm tạo bên ngoài định nghĩa lớp
DIEM_DH::DIEM_DH(int x1, int y1, int m1)
{
    x=x1; y=y1; m=m1;
}
```

1.3. Dùng hàm tạo trong khai báo

+ Khi đã xây dựng các hàm tạo, ta có thể dùng chúng trong khai báo để tạo ra một đối tượng đồng thời khởi gán cho các thuộc tính của đối tượng được tạo. Dựa vào các tham số trong khai báo mà Trình biên dịch sẽ biết cần gọi đến hàm tạo nào.

+ Khi khai báo một biến đối tượng có thể sử dụng các tham số để khởi gán cho các thuộc tính của biến đối tượng.

+ Khi khai báo mảng đối tượng không cho phép dùng các tham số để khởi gán.

+ Câu lệnh khai báo một biến đối tượng sẽ gọi tới hàm tạo 1 lần

+ Câu lệnh khai báo một mảng n đối tượng sẽ gọi tới hàm tạo n lần.

Ví dụ:

```
DIEM_DH d; // Gọi tới hàm tạo không đối.
           // Kết quả d.x=0, d.y=0, d.m=1
DIEM_DH u(200,100,4); // Gọi tới hàm tạo có đối.
           // Kết quả u.x=200, u.y=100, d.m=4
DIEM_DH v(300,250); // Gọi tới hàm tạo có đối.
           // Kết quả v.x=300, v.y=250, d.m=15
DIEM_DH p[10]; // Gọi tới hàm tạo không đối 10 lần
```

Chú ý: Với các hàm có đối kiểu lớp, thì đối chỉ xem là các tham số hình thức, vì vậy khai báo đối (trong dòng đầu của hàm) sẽ không tạo ra đối tượng mới và do đó không gọi tới các hàm tạo.

1.4. Dùng hàm tạo trong cấp phát bộ nhớ

+ Khi cấp phát bộ nhớ cho một đối tượng có thể dùng các tham số để khởi gán cho các thuộc tính của đối tượng, ví dụ:

```
DIEM_DH *q = new DIEM_DH(50,40,6); // Gọi tới hàm tạo có đối
// Kết quả q->x=50, q->y=40, q->m=6
DIEM_DH *r = new DIEM_DH; // Gọi tới hàm tạo không đối
// Kết quả r->x=0, r->y=0, r->m=1
```

+ Khi cấp phát bộ nhớ cho một dãy đối tượng không cho phép dùng tham số để khởi gán, ví dụ:

```
int n=20;
DIEM_DH *s = new DIEM_DH[n]; // Gọi tới hàm tạo không
// đối 20 lần.
```

1.5. Dùng hàm tạo để biểu diễn các đối tượng hằng

+ Như đã biết, sau khi định nghĩa lớp DIEM_DH thì có thể xem lớp này như một kiểu dữ liệu như int, double, char, ...

Với kiểu int chúng ta có các hằng int, như 356.

Với kiểu double chúng ta có các hằng double, như 98.75

Khái niệm hằng kiểu int, hằng kiểu double có thể mở rộng cho hằng kiểu DIEM_DH

+ Để biểu diễn một hằng đối tượng (hay còn gọi: Đối tượng hằng) chúng ta phải dùng tới hàm tạo. Mẫu viết như sau:

Tên_lớp(danh sách tham số);

Ví dụ đối với lớp DIEM_DH nói trên, có thể viết như sau:

```
DIEM_DH(345,123,8) // Biểu thị một đối tượng kiểu DIEM_DH
// có các thuộc tính x=345, y=123, m=8
```

Chú ý: Có thể sử dụng một hằng đối tượng như một đối tượng. Nói cách khác, có thể dùng hằng đối tượng để thực hiện một phương thức, ví dụ nếu viết:

```
DIEM_DH(345,123,8).in();
```

thì có nghĩa là thực hiện phương thức in() đối với hằng đối tượng.

1.6. Ví dụ minh họa

Chương trình sau đây minh họa cách xây dựng hàm tạo và cách sử dụng hàm tạo trong khai báo, trong cấp phát bộ nhớ và trong việc biểu diễn các hằng đối tượng.

```
//CT4_02.CPP
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
class DIEM_DH
{
private:
    int x,y,m;
```

```
public:
    // Hàm bạn dùng để in đối tượng DIEM_DH
    friend void in(DIEM_DH d)
    {
        cout << "\n " << d.x << " " << d.y << " " << d.m ;
    }
    // Phương thức dùng để in đối tượng DIEM_DH
    void in()
    {
        cout << "\n " << x << " " << y << " " << m ;
    }
    //Hàm tạo không đối
    DIEM_DH()
    {
        x=y=0;
        m=1;
    }
    //Hàm tạo có đối, đối m1 có giá trị mặc định là 15 (màu trắng)
    DIEM_DH(int x1,int y1,int m1=15);
};

//Xây dựng hàm tạo
DIEM_DH::DIEM_DH(int x1,int y1,int m1)
{
    x=x1; y=y1; m=m1;
}

void main()
{
    DIEM_DH d1; // Gọi tới hàm tạo không đối
    DIEM_DH d2(200,200,10); // Gọi tới hàm tạo có đối
    DIEM_DH *d;
    d= new DIEM_DH(300,300); // Gọi tới hàm tạo có đối
    clrscr();
    in(d1); //Gọi hàm bạn in()
    d2.in();//Gọi phương thức in()
    in(*d); //Gọi hàm bạn in()
    DIEM_DH(2,2,2).in();//Gọi phương thức in()
    DIEM_DH t[3]; // 3 lần gọi hàm tạo không đối
    DIEM_DH *q; // Gọi hàm tạo không đối
    int n;
    cout << "\nN= ";
    cin >> n;
```



```
q=new DIEM_DH[n+1]; // (n+1) lần gọi hàm tạo không đối  
for (int i=0;i<=n;++i)  
    q[i]=DIEM_DH(300+i,200+i,8); //(n+1) lần gọi hàm tạo có đối  
for (i=0;i<=n;++i)  
    q[i].in(); // Gọi phương thức in()  
for (i=0;i<=n;++i)  
    DIEM_DH(300+i,200+i,8).in(); // Gọi phương thức in()  
getch();  
}
```

Bài 2. Lớp không có hàm tạo và hàm tạo mặc định

Các chương trình nêu trong chương 3 đều không có hàm tạo. Vậy khi đó các đối tượng được hình thành như thế nào ?

2.1. Nếu lớp không có hàm tạo, Chương trình dịch sẽ cung cấp một hàm tạo mặc định không đối (default). Hàm này thực chất không làm gì cả. Như vậy một đối tượng tạo ra chỉ được cấp phát bộ nhớ, còn các thuộc tính của nó chưa được xác định. Chúng ta có thể kiểm chứng điều này, bằng cách chạy chương trình sau:

```
//CT4_03.CPP  
// Hàm tạo mặc định  
#include <conio.h>  
#include <iostream.h>  
class DIEM_DH  
{  
    private:  
        int x,y,m;  
    public:  
        // Phương thức  
        void in()  
        {  
            cout << "n " << x << " " << y << " " << m ;  
        }  
};  
void main()  
{  
    DIEM_DH d;  
    d.in();  
    DIEM_DH *p;  
    p= new DIEM_DH[10];  
    clrscr();  
    d.in();  
    for (int i=0;i<10;++i)
```

```
(p+i)->in();  
getch();  
}
```

2.2. Nếu trong lớp đã có ít nhất một hàm tạo, thì hàm tạo mặc định sẽ không được phát sinh nữa. Khi đó mọi câu lệnh xây dựng đối tượng mới đều sẽ gọi đến một hàm tạo của lớp. Nếu không tìm thấy hàm tạo cần gọi thì Chương trình dịch sẽ báo lỗi. Điều này thường xảy ra khi chúng ta không xây dựng hàm tạo không đối, nhưng lại sử dụng các khai báo không tham số như ví dụ sau:

```
#include <conio.h>  
#include <iostream.h>  
class DIEM_DH  
{  
    private:  
        int x,y,m;  
    public:  
        // Phương thức dùng để in đối tượng DIEM_DH  
        void in()  
        {  
            cout << "\n " << x << " " << y << " " << m ;  
        }  
        //Hàm tạo có đối  
        DIEM_DH::DIEM_DH(int x1,int y1,int m1)  
        {  
            x=x1; y=y1; m=m1;  
        }  
};  
void main()  
{  
    DIEM_DH d1(200,200,10); // Gọi tới hàm tạo có đối  
    DIEM_DH d2; // Gọi tới hàm tạo không đối  
    d2= DIEM_DH(300,300,8); // Gọi tới hàm tạo có đối  
    d1.in();  
    d2.in();  
    getch();  
}
```

Trong các câu lệnh trên, chỉ có câu lệnh thứ 2 trong hàm main() là bị báo lỗi. Câu lệnh này sẽ gọi tới hàm tạo không đối, mà hàm này chưa được xây dựng.

Giải pháp: Có thể chọn một trong 2 giải pháp sau:

- Xây dựng thêm hàm tạo không đối.
- Gán giá trị mặc định cho tất cả các đối x1, y1 và m1 của hàm tạo đã xây dựng ở trên.

Theo phương án 2, chương trình có thể sửa như sau:

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

```
#include <iostream.h>
class DIEM_DH
{
    private:
        int x,y,m;
    public:
        // Phương thức dùng để in đối tượng DIEM_DH
        void in()
        {
            cout << "\n " << x << " " << y << " " << m ;
        }
        //Hàm tạo có đối , tất cả các đối đều có giá trị mặc định
        DIEM_DH::DIEM_DH(int x1=0,int y1=0,int m1=15)
        {
            x=x1; y=y1; m=m1;
        }
};

void main()
{
    DIEM_DH d1(200,200,10); // Gọi tới hàm tạo, không dùng
                           // tham số mặc định
    DIEM_DH d2; // Gọi tới hàm tạo , dùng 3 tham số mặc định
    d2= DIEM_DH(300,300); // Gọi tới hàm tạo, dùng 1 tham số
                           // mặc định

    d1.in();
    d2.in();
    getch();
}
```

Bài 3. Lớp đa thức

Chương trình dưới đây là sự cải tiến chương trình trong mục 8.5 của chương 3 bằng cách đưa vào 2 hàm tạo:

```
//Hàm tạo không đối
DT()
{
    this->n=0; this->a=NULL;
}

//Hàm tạo có đối
DT(int n1)
{
    this->n=n1 ;
    this->a = new double[n1+1];
}
```

Hàm tạo có đối sẽ tạo một đối tượng mới (kiểu DT) gồm 2 thuộc tính là biến nguyên n và con trỏ a. Ngoài ra còn cấp phát bộ vùng nhớ (cho a) để chứa các hệ số của đa thức.

Nếu không xây dựng hàm tạo, mà sử dụng hàm tạo mặc định thì các đối tượng (kiểu DT) tạo ra bởi các lệnh khai báo sẽ chưa có bộ nhớ để chứa đa thức. Như vậy đối tượng tạo ra chưa hoàn chỉnh và chưa dùng được. Để có một đối tượng hoàn chỉnh phải qua 2 bước:

+ Dùng khai báo để tạo các đối tượng, ví dụ:

DT d;

+ Cấp phát vùng nhớ (cho đối tượng) để chứa đa thức, ví dụ:

d.n = m;

d.a = new double[m+1] ;

Quy trình này được áp dụng trong các phương thức toán tử của chương trình trong mục 8.5 chương 3. Rõ ràng quy trình này vừa dài vừa không tiện lợi, lại hay mắc lỗi, vì người lập trình hay quên không cấp phát bộ nhớ.

Việc dùng các hàm tạo để sản sinh ra các đối tượng hoàn chỉnh tỏ ra tiện lợi hơn, vì tránh được các thao tác phụ (như cấp phát bộ nhớ) nằm bên ngoài khai báo. Phương án dùng hàm tạo sẽ được sử dụng trong các phương thức toán tử của chương trình dưới đây:

+ Nội dung chương trình gồm:

- Nhập, in các đa thức p, q, r, s
- Tính đa thức: $f = -(p + q) * (r - s)$
- Nhập các số thực x1 và x2
- Tính f(x1) (bằng cách dùng phương thức operator^)
- Tính f(x2) (bằng cách dùng hàm F)

// CT4_05.CPP

#include <conio.h>

#include <iostream.h>

#include <math.h>

class DT

{

private:

int n; // Bac da thuc

double *a; // Tro toi vung nho chua cac he so da thuc

// a0, a1,...

public:

DT()

{

this->n=0; this->a=NULL;

}

DT(int n1)

{

this->n=n1 ;

this->a = new double[n1+1];

}

friend ostream& operator<< (ostream& os,const DT &d);

```
friend istream& operator>> (istream& is,DT &d);
DT operator-();
DT operator+(const DT &d2);
DT operator-(DT d2);
DT operator*(const DT &d2);
double operator^(const double &x); // Tính giá trị đã thực
double operator[](int i)
{
    if (i<0)
        return double(n);
    else
        return a[i];
}
};

// Hàm tính giá trị đã thực
double F(DT d,double x)
{
    double s=0.0 , t=1.0;
    int n;
    n = int(d[-1]);
    for (int i=0; i<=n; ++i)
    {
        s += d[i]*t;
        t *= x;
    }
    return s;
}

ostream& operator<< (ostream& os,const DT &d)
{
    os << " - Các hệ số (tu ao): ";
    for (int i=0 ; i<= d.n ; ++i)
        os << d.a[i] <<" ";
    return os;
}

istream& operator>> (istream& is,DT &d)
{
    if (d.a!=NULL) delete d.a;
    cout << " - Nhập giá trị: ";
    cin >> d.n;
    d.a = new double[d.n+1];
```

```
cout << "Nhap cac he so da thuc:\n" ;
for (int i=0 ; i<= d.n ; ++i)
{
    cout << "He so bac " << i << " = " ;
    is >> d.a[i] ;
}
return is;
}
```

DT DT::operator-()

```
{
    DT p(this->n);
    for (int i=0 ; i<=n ; ++i)
        p.a[i] = -a[i];
    return p;
}
```

DT DT::operator+(const DT &d2)

```
{
    int k,i;
    k = n > d2.n ? n : d2.n ;
    DT d(k);
    for (i=0; i<=k ; ++i)
        if (i<=n && i<=d2.n)
            d.a[i] = a[i] + d2.a[i];
        else if (i<=n)
            d.a[i] = a[i];
        else
            d.a[i] = d2.a[i];
    i=k;
    while(i>0 && d.a[i]==0.0) --i;
    d.n = i;
    return d ;
}
```

DT DT::operator-(DT d2)

```
{
    return (*this + (-d2));
}
```

DT DT::operator(const DT &d2)*

```
{
    int k, i, j;
```

```

    k = n + d2.n ;
    DT d(k);
    for (i=0; i<=k; ++i) d.a[i] = 0;
    for (i=0 ; i<= n ; ++i)
        for (j=0 ; j<= d2.n ; ++j)
            d.a[i+j] += a[i]*d2.a[j] ;
    return d;
}

double DT::operator^(const double &x)
{
    double s=0.0 , t=1.0;
    for (int i=0 ; i<= n ; ++i)
    {
        s += a[i]*t;
        t *= x;
    }
    return s;
}

void main()
{
    DT p,q,r,s,f;
    double x1,x2,g1,g2;
    clrscr();
    cout << "\nNhap da thuc P " ; cin >> p;
    cout << "\nDa thuc p " << p ;
    cout << "\nNhap da thuc Q " ; cin >> q;
    cout << "\nDa thuc q " << q ;
    cout << "\nNhap da thuc R " ; cin >> r;
    cout << "\nDa thuc r " << r ;
    cout << "\nNhap da thuc S " ; cin >> s;
    cout << "\nDa thuc s " << s ;
    f = -(p+q)*(r-s);
    cout << "\nNhap so thuc x1: " ; cin >> x1;
    cout << "\nNhap so thuc x2: " ; cin >> x2;
    g1 = f^x1;
    g2 = F(f,x2);
    cout << "\nDa thuc f " << f;
    cout << "\nf("<<x1<<") = " << g1;
    cout << "\nf("<<x2<<") = " << g2;
    getch();
}

```

}

Bài 4. Hàm tạo sao chép (copy constructor)

4.1. Hàm tạo sao chép mặc định

Giả sử đã định nghĩa một lớp nào đó, ví dụ lớp PS (phân số). Khi đó:

+ Ta có thể dùng câu lệnh khai báo hoặc cấp phát bộ nhớ để tạo các đối tượng mới, ví dụ:

PS p1, p2 ;

PS *p = new PS ;

+ Ta cũng có thể dùng lệnh khai báo để tạo một đối tượng mới từ một đối tượng đã tồn tại, ví dụ:

PS u;

PS v(u) ; // Tạo v theo u

ý nghĩa của câu lệnh này như sau:

- Nếu trong lớp PS chưa xây dựng hàm tạo sao chép, thì câu lệnh này sẽ gọi tới một hàm tạo sao chép mặc định (của C++). Hàm này sẽ sao chép nội dung từng bit của u vào các bit tương ứng của v. Như vậy các vùng nhớ của u và v sẽ có nội dung như nhau. Rõ ràng trong đa số các trường hợp, nếu lớp không có các thuộc tính kiểu con trỏ hay tham chiếu, thì việc dùng các hàm tạo sao chép mặc định (để tạo ra một đối tượng mới có nội dung như một đối tượng cho trước) là đủ và không cần xây dựng một hàm tạo sao chép mới.

- Nếu trong lớp PS đã có hàm tạo sao chép (cách viết sẽ nói sau) thì câu lệnh:

PS v(u) ;

sẽ tạo ra đối tượng mới v, sau đó gọi tới hàm tạo sao chép để khởi gán v theo u.

Ví dụ sau minh họa cách dùng hàm tạo sao chép mặc định:

Trong chương trình đưa vào lớp PS (phân số):

+ Các thuộc tính gồm: t (tử số) và m (mẫu).

+ Trong lớp không có phương thức nào cả mà chỉ có 2 hàm bạn là các hàm toán tử nhập (>>) và xuất (<<).

+ Nội dung chương trình là: Dùng lệnh khai báo để tạo một đối tượng u (kiểu PS) có nội dung như đối tượng đã có d.

//CT4_06.CPP

// Hàm tạo sao chép mặc định

#include <conio.h>

#include <iostream.h>

class PS

{

private:

int t,m ;

public:

friend ostream& operator<< (ostream& os,const PS &p)

{

os << " = " << p.t << "/" << p.m;

return os;

}


```
friend istream& operator>> (istream& is, PS &p)
{
    cout << " - Nhập tu và mau: ";
    is >> p.t >> p.m ;
    return is;
}

};

void main()
{
    PS d;
    cout << "\n Nhập PS d"; cin >> d;
    cout << "\n PS d " << d;
    PS u(d);
    cout << "\n PS u " << u;
    getch();
}
```

4.2. Cách xây dựng hàm tạo sao chép

+ Hàm tạo sao chép sử dụng một đối **kiểu tham chiếu đối tượng** để khởi gán cho đối tượng mới. Hàm tạo sao chép được viết theo mẫu:

```
Tên_lớp (const Tên_lớp &dt)
{
    // Các câu lệnh dùng các thuộc tính của đối tượng dt
    // để khởi gán cho các thuộc tính của đối tượng mới
}
```

+ **Ví dụ** có thể xây dựng hàm tạo sao chép cho lớp PS như sau:

```
class PS
{
    private:
        int t,m ;
    public:
        PS (const PS &p)
        {
            this->t = p.t ;
            this->m = p.m ;
        }
    ...
};
```

4.3. Khi nào cần xây dựng hàm tạo sao chép

+ **Nhận xét:** Hàm tạo sao chép trong ví dụ trên không khác gì hàm tạo sao chép mặc định.

+ Khi lớp không có các thuộc tính kiểu con trỏ hoặc tham chiếu, thì dùng hàm tạo sao chép mặc định là đủ.

+ Khi lớp có các thuộc tính con trỏ hoặc tham chiếu, thì hàm tạo sao chép mặc định chưa đáp ứng được yêu cầu. Ví dụ lớp DT (đa thức) trong §3:

```
class DT
{
    private:
        int n; // Bac da thuc
        double *a; // Tro toi vung nho chua cac he so da thuc
                    // a0, a1,...

    public:
        DT()
        {
            this->n=0; this->a=NULL;
        }
        DT(int n1)
        {
            this->n=n1 ;
            this->a = new double[n1+1];
        }
        friend ostream& operator<< (ostream& os,const DT &d);
        friend istream& operator>> (istream& is,DT &d);
        ....
};
```

Bây giờ chúng ta hãy theo dõi xem việc dùng hàm tạo mặc định trong đoạn chương trình sau sẽ dẫn đến sai lầm như thế nào:

```
DT d ;
// Tạo đối tượng d kiểu DT
cin >> d ;
/* Nhập đối tượng d , gồm: nhập một số nguyên dương và
   gán cho d.n, cấp phát vùng nhớ cho d.a, nhập các hệ số
   của đa thức và chứa vào vùng nhớ được cấp phát
*/

DT u(d) ;
/* Dùng hàm tạo mặc định để xây dựng đối tượng u theo d
   Kết quả: u.n = d.n và u.a = d.a. Như vậy 2 con trỏ u.a và
   d.a cùng trỏ đến một vùng nhớ.
*/
```

Nhận xét: Mục đích của ta là tạo ra một đối tượng u giống như d, nhưng độc lập với d. Nghĩa là khi d thay đổi thì u không bị ảnh hưởng gì. Thế nhưng mục tiêu này không đạt được, vì u và d có

chung một vùng nhớ chứa hệ số của đa thức, nên khi sửa đổi các hệ số của đa thức trong d thì các hệ số của đa thức trong u cũng thay đổi theo. Còn một trường hợp nữa cũng dẫn đến lỗi là khi một trong 2 đối tượng u và d bị giải phóng (thu hồi vùng nhớ chứa đa thức) thì đối tượng còn lại cũng sẽ không còn vùng nhớ nữa.

Ví dụ sau sẽ minh họa nhận xét trên: Khi d thay đổi thì u cũng thay đổi và ngược lại khi u thay đổi thì d cũng thay đổi theo.

```
//CT4_07.CPP
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DT
{
private:
    int n; // Bac da thuc
    double *a; // Tro toi vung nho chua cac he so da thuc
                // a0, a1,...
public:
    DT()
    {
        this->n=0; this->a=NULL;
    }
    DT(int n1)
    {
        this->n=n1 ;
        this->a = new double[n1+1];
    }
    friend ostream& operator<< (ostream& os,const DT &d);
    friend istream& operator>> (istream& is,DT &d);
};
ostream& operator<< (ostream& os,const DT &d)
{
    os << " - Cac he so (tu ao): ";
    for (int i=0 ; i<= d.n ; ++i)
        os << d.a[i] << " ";
    return os;
}
istream& operator>> (istream& is,DT &d)
{
    if (d.a!=NULL) delete d.a;
    cout << " - Bac da thuc: ";
    cin >> d.n;
    d.a = new double[d.n+1];
    cout << "Nhap cac he so da thuc:\n";
    for (int i=0 ; i<= d.n ; ++i)
```

```
{
    cout << "He so bac " << i << " = " ;
    is >> d.a[i] ;
}
return is;
}

void main()
{
    DT d;
    clrscr();
    cout << "\nNhap da thuc d "; cin >> d;
    DT u(d);
    cout << "\nDa thuc d " << d ;
    cout << "\nDa thuc u " << u ;
    cout << "\nNhap da thuc d "; cin >> d;
    cout << "\nDa thuc d " << d ;
    cout << "\nDa thuc u " << u ;
    cout << "\nNhap da thuc u "; cin >> u;
    cout << "\nDa thuc d " << d ;
    cout << "\nDa thuc u " << u ;
    getch();
}
```

4.4. Ví dụ về hàm tạo sao chép

Trong chương trình trên đã chỉ rõ: Hàm tạo sao chép mặc định là chưa thoả mãn đối với lớp DT. Vì vậy cần viết hàm tạo sao chép để xây dựng đối tượng mới (ví dụ u) từ một đối tượng đang tồn tại (ví dụ d) theo các yêu cầu sau:

- + Gán d.n cho u.n
- + Cấp phát một vùng nhớ cho u.a để có thể chứa được (d.n + 1) hệ số.
- + Gán các hệ số chứa trong vùng nhớ của d.a sang vùng nhớ của u.a

Như vậy chúng ta sẽ tạo được đối tượng u có nội dung ban đầu giống như d, nhưng độc lập với d.

Để đáp ứng các yêu cầu nêu trên, hàm tạo sao chép cần được xây dựng như sau:

DT::DT(const DT &d)

```
{
    this->n = d.n;
    this->a = new double[d.n+1];
    for (int i=0;i<=d.n;++i)
        this->a[i] = d.a[i];
}
```

Chương trình sau sẽ minh hoạ điều này: Sự thay đổi của d không làm ảnh hưởng đến u và ngược lại sự thay đổi của u không làm ảnh hưởng đến d.

```
//CT4_08.CPP
// Viết hàm tạo sao chép cho lớp DT
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DT
{
private:
    int n; // Bac da thuc
    double *a; // Tro toi vung nho chua cac he so da thuc
                // a0, a1,...
public:
    DT()
    {
        this->n=0; this->a=NULL;
    }
    DT(int n1)
    {
        this->n=n1 ;
        this->a = new double[n1+1];
    }
    DT(const DT &d);
    friend ostream& operator<< (ostream& os,const DT &d);
    friend istream& operator>> (istream& is,DT &d);
};
DT::DT(const DT &d)
{
    this->n = d.n;
    this->a = new double[d.n+1];
    for (int i=0;i<=d.n;++i)
        this->a[i] = d.a[i];
}
ostream& operator<< (ostream& os,const DT &d)
{
    os << " - Cac he so (tu ao): ";
    for (int i=0 ; i<= d.n ; ++i)
        os << d.a[i] <<" ";
    return os;
}
istream& operator>> (istream& is,DT &d)
```

```
{
    if (d.a!=NULL) delete d.a;
    cout << " - Bac da thuc: " ;
    cin >> d.n;
    d.a = new double[d.n+1];
    cout << "Nhap cac he so da thuc:\n" ;
    for (int i=0 ; i<= d.n ; ++i)
    {
        cout << "He so bac " << i << " = " ;
        is >> d.a[i] ;
    }
    return is;
}

void main()
{
    DT d;
    clrscr();
    cout << "\nNhap da thuc d " ; cin >> d;
    DT u(d);
    cout << "\nDa thuc d " << d ;
    cout << "\nDa thuc u " << u ;
    cout << "\nNhap da thuc d " ; cin >> d;
    cout << "\nDa thuc d " << d ;
    cout << "\nDa thuc u " << u ;
    cout << "\nNhap da thuc u " ; cin >> u;
    cout << "\nDa thuc d " << d ;
    cout << "\nDa thuc u " << u ;
    getch();
}
```

Bài 5. Hàm huỷ (Destructor)

5.1. Công dụng của hàm huỷ

Hàm huỷ là một hàm thành viên của lớp (phương thức) có chức năng ngược với hàm tạo. Hàm huỷ được gọi trước khi giải phóng (xoá bỏ) một đối tượng để thực hiện một số công việc có tính “dọn dẹp” trước khi đối tượng được huỷ bỏ, ví dụ như giải phóng một vùng nhớ mà đối tượng đang quản lý, xoá đối tượng khỏi màn hình nếu như nó đang hiển thị, ...

Việc huỷ bỏ một đối tượng thường xảy ra trong 2 trường hợp sau:

- + Trong các toán tử và các hàm giải phóng bộ nhớ, như delete, free,...
- + Giải phóng các biến, mảng cục bộ khi thoát khỏi hàm, phương thức.

5.2. Hàm huỷ mặc định

Nếu trong lớp không định nghĩa hàm huỷ, thì một hàm huỷ mặc định không làm gì cả được phát sinh. Đối với nhiều lớp thì hàm huỷ mặc định là đủ, và không cần đưa vào một hàm huỷ mới.

5.3. Quy tắc viết hàm huỷ

Mỗi lớp chỉ có một hàm huỷ viết theo các quy tắc sau:

- + Kiểu của hàm: Hàm huỷ cũng giống như hàm tạo là hàm không có kiểu, không có giá trị trả về.
- + Tên hàm: Tên của hàm huỷ gồm một dấu ngã (đứng trước) và tên lớp:
 ~Tên_lớp
- + Đối: Hàm huỷ không có đối

Ví dụ có thể xây dựng hàm huỷ cho lớp DT (đa thức) ở §3 như sau:

```
class DT
{
private:
    int n; // Bac da thuc
    double *a; // Tro toi vung nho chua cac he so da thuc
                // a0, a1,...

public:
    ~DT()
    {
        this->n=0;
        delete this->a;
    }
    ...
};
```

5.4. Vai trò của hàm huỷ trong lớp DT

5.4.1. Khiếm khuyết của chương trình trong bài 3

Chương trình trong §3 định nghĩa lớp DT (đa thức) khá đầy đủ gồm:

- + Các hàm tạo
- + Các hàm toán tử nhập >>, xuất <<
- + Các hàm toán tử thực hiện các phép tính + - *

Tuy nhiên vẫn còn thiếu hàm huỷ để giải phóng vùng nhớ mà đối tượng kiểu DT (cần huỷ) đang quản lý.

Chúng ta hãy phân tích các khiếm khuyết của chương trình này:

+ Khi chương trình gọi tới một phương thức toán tử để thực hiện các phép tính cộng, trừ, nhân đa thức, thì một đối tượng trung gian được tạo ra. Một vùng nhớ được cấp phát và giao cho nó (đối tượng trung gian) quản lý.

+ Khi thực hiện xong phép tính sẽ ra khỏi phương thức. Đối tượng trung gian bị xoá, tuy nhiên chỉ vùng nhớ của các thuộc tính của đối tượng này được giải phóng. Còn vùng nhớ (chứa các hệ số của đa thức) mà đối tượng trung gian đang quản lý thì không hề bị giải phóng. Như vậy số vùng nhớ bị chiếm dụng vô ích sẽ tăng lên.

5.4.2. Cách khắc phục

Nhược điểm trên dễ dàng khắc phục bằng cách đưa vào lớp DT hàm huỷ viết trong 5.3 (mục trên).

5.5. Lớp hình tròn đồ hoạ

Chương trình dưới đây gồm:

Lớp HT (hình tròn) với các thuộc tính:

```
int r;    // Bán kính
int m;    // Màu hình tròn
int xhien,yhien; // Vị trí hiển thị hình tròn trên màn hình
char *pht;    // Con trỏ trỏ tới vùng nhớ chứa ảnh hình tròn
int hienmh;    // Trạng thái hiện (hienmh=1), ẩn (hienmh=0)
```

Các phương thức:

- + Hàm tạo không đối
HT();
thực hiện việc gán giá trị bằng 0 cho các thuộc tính của lớp.
- + Hàm tạo có đối
HT(int r1,int m1=15);
thực hiện các việc:
 - Gán r1 cho r, m1 cho m
 - Cấp phát bộ nhớ cho pht
 - Vẽ hình tròn và lưu ảnh hình tròn vào vùng nhớ của pht
- + Hàm huỷ
~HT();
thực hiện các việc:
 - Xoá hình tròn khỏi màn hình (nếu đang hiển thị)
 - Giải phóng bộ nhớ đã cấp cho pht
- + Phương thức
void hien(int x, int y);
có nhiệm vụ hiển thị hình tròn tại (x,y)
- + Phương thức
void an();
có nhiệm vụ làm ẩn hình tròn

Các hàm độc lập:

```
void ktdh(); //Khởi tạo đồ hoạ
void ve_bau_troi(); // Vẽ bầu trời đầy sao
void ht_di_dong_xuong(); // Vẽ một cặp 2 hình tròn di
                        // chuyển xuống
void ht_di_dong_len();// Vẽ một cặp 2 hình tròn di
                        // chuyển lên trên
```

Nội dung chương trình là tạo ra các chuyển động xuống và lên của các hình tròn.


```
//CT4_09.CPP
// Lop do hoa
// Ham huy
// Trong ham huy co the goi PT khac
#include <conio.h>
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <dos.h>
void ktdh();
void ve_bau_troi();
void ht_di_dong_xuong();
void ht_di_dong_len();
int xmax,ymax;
class HT
{
private:
    int r,m ;
    int xhien,yhien;
    char *pht;
    int hienmh;
public:
    HT();
    HT(int r1,int m1=15);
    ~HT();
    void hien(int x, int y);
    void an();
};
HT:: HT()
{
    r=m=hienmh=0;
    xhien=yhien=0;
    pht=NULL;
}

HT::HT(int r1,int m1)
{
    r=r1; m=m1; hienmh=0;
    xhien=yhien=0;
```

```
if (r<0) r=0;
if (r==0)
{
    pht=NULL;
}
else
{
    int size; char *pmh;
    size = imagesize(0,0,r+r,r+r);
    pmh = new char[size];
    getimage(0,0,r+r,r+r,pmh);
    setcolor(m);
    circle(r,r,r);
    setfillstyle(1,m);
    floodfill(r,r,m);
    pht = new char[size];
    getimage(0,0,r+r,r+r,pht);
    putimage(0,0,pmh,COPY_PUT);
    delete pmh;
    pmh=NULL;
}
}

void HT::hien(int x, int y)
{
    if (pht!=NULL && !hienmh) // chua hien
    {
        hienmh=1;
        xhien=x; yhien=y;
        putimage(x,y,pht,XOR_PUT);
    }
}

void HT::an()
{
    if (hienmh) // dang hien
    {
        hienmh=0;
        putimage(xhien,yhien,pht,XOR_PUT);
    }
}

HT::~~HT()
{

```

```
an();
if (pht!=NULL)
{
    delete pht;
    pht=NULL;
}
}

void ktdh()
{
    int mh=0,mode=0;
    initgraph(&mh,&mode,"");
    xmax = getmaxx();
    ymax = getmaxy();
}

void ve_bau_troi()
{
    for (int i=0;i<2000;++i)
        putpixel(random(xmax), random(ymax), 1+random(15));
}

void ht_di_dong_xuong()
{
    HT h(50,4);
    HT u(60,15);
    h.hien(0,0);
    u.hien(40,0);
    for (int x=0;x<=340;x+=10)
    {
        h.an();
        u.an();
        h.hien(x,x);
        delay(200);
        u.hien(x+40,x);
        delay(200);
    }
}

void ht_di_dong_len()
{
    HT h(50,4);
    HT u(60,15);
```

```
h.hien(340,340);
u.hien(380,340);
for (int x=340;x>=0;x-=10)
{
    h.an();
    u.an();
    h.hien(x,x);
    delay(200);
    u.hien(x+40,x);
    delay(200);
}
}
```

```
void main()
{
    ktdh();
    ve_bau_troi();
    ht_di_dong_xuong();
    ht_di_dong_len();
    getch();
    closegraph();
}
```

Các nhận xét:

1. Trong thân hàm huỷ gọi tới phương thức an().
2. Điều gì xảy ra khi bỏ đi hàm huỷ:

+ Khi gọi hàm ht_di_dong_xuong() thì có 2 đối tượng kiểu HT được tạo ra. Trong thân hàm sử dụng các đối tượng này để vẽ các hình tròn di chuyển xuống. Khi thoát khỏi hàm thì 2 đối tượng (tạo ra ở trên) được giải phóng. Vùng nhớ của các thuộc tính của chúng bị thu hồi, nhưng vùng nhớ cấp phát cho thuộc tính pht chưa được giải phóng và ảnh của 2 hình tròn (ở phía dưới màn hình) vẫn không được cất đi.

+ Điều tương tự xảy ra sau khi ra khỏi hàm ht_di_dong_len() : vùng nhớ cấp phát cho thuộc tính pht chưa được giải phóng và ảnh của 2 hình tròn (ở phía trên màn hình) vẫn không được thu dọn.

Bài 6. Toán tử gán

6.1. Toán tử gán mặc định

Toán tử gán (cho lớp) là một trường hợp đặc biệt so với các toán tử khác. Nếu trong lớp chưa định nghĩa một phương thức toán tử gán thì Trình biên dịch sẽ phát sinh một toán tử gán mặc định để thực hiện câu lệnh gán 2 đối tượng của lớp, ví dụ:

```
HT h1, h2(100,6);
h1 = h2 ; // Gán h2 cho h1
```

Toán tử gán mặc định sẽ sao chép đối tượng nguồn (h2) vào đối tượng đích (h1) theo từng bit một.

Trong đa số các trường hợp khi lớp không có các thành phần con trở hay tham chiếu thì toán tử gán mặc định là đủ dùng và không cần định nghĩa một phương thức toán tử gán cho lớp. Nhưng đối với các lớp có thuộc tính con trở như lớp DT (đa thức), lớp HT (hình tròn) thì toán tử gán mặc định không thích hợp và việc xây dựng toán tử gán là cần thiết.

6.2. Cách viết toán tử gán

Cũng giống như các phương thức khác, phương thức toán tử gán dùng đối con trở this để biểu thị đối tượng đích và dùng một đối tượng minh để biểu thị đối tượng nguồn. Vì trong thân của toán tử gán không nên làm việc với bản sao của đối tượng nguồn, mà phải làm việc trực tiếp với đối tượng nguồn, nên kiểu đối tượng minh nhất thiết phải là kiểu tham chiếu đối tượng.

Phương thức toán tử gán có thể có hoặc không có giá trị trả về. Nếu không có giá trị trả về (kiểu void), thì khi viết chương trình không được phép viết câu lệnh gán liên tiếp nhiều đối tượng, như:

$u = v = k = h ;$

Nếu phương thức toán tử gán trả về tham chiếu của đối tượng nguồn, thì có thể dùng toán tử gán để thực hiện các phép gán liên tiếp nhiều đối tượng.

Ví dụ đối với lớp HT (trong mục trước), có thể xây dựng toán tử gán như sau:

```
void HT::operator=(const HT &h)
{
    r = h.r ; m = h.m ;
    xhien = yhien = 0;
    hienmh = 0 ;
    if (h.pht==NULL)
        pht = NULL;
    else
    {
        int size;
        size = imagesize(0,0,r+r,r+r);
        pht = new char[size];
        memcpy(pht,h.pht,size);
    }
}
```

Với toán tử gán này, chỉ cho phép gán đối tượng nguồn cho một đối tượng đích.

Như vậy câu lệnh sau là sai:

```
HT u, v, h ;
u = v = h ;
```

Bây giờ ta sửa lại toán gán để nó trả về tham chiếu đối tượng nguồn như sau:

```
const HT & HT::operator=(const HT &h)
{
    r = h.r ; m = h.m ;
    xhien = yhien = 0;
    hienmh = 0 ;
    if (h.pht==NULL)
```

```
pht = NULL;
else
{
    int size;
    size = imagesize(0,0,r+r,r+r);
    pht = new char[size];
    memcpy(pht,h.pht,size);
}
return h ;
}
```

Với toán tử gán mới này, ta có thể viết câu lệnh để gán đối tượng nguồn cho nhiều đối tượng đích. Như vậy các câu lệnh sau là được:

```
HT u, v, h ;
u = v = h ;
```

6.3. Toán tử gán và hàm tạo sao chép

- + Toán tử gán không tạo ra đối tượng mới, chỉ thực hiện phép gán giữa 2 đối tượng đã tồn tại.
- + Hàm tạo sao chép được dùng để tạo một đối tượng mới và gán nội dung của một đối tượng đã tồn tại cho đối tượng mới vừa tạo.
- + Nếu đã xây dựng toán tử gán mà lại dùng hàm tạo sao chép mặc định thì chưa đủ, vì việc khởi gán trong câu lệnh khai báo sẽ không gọi tới toán tử gán mà lại gọi tới hàm tạo sao chép.
- + Như vậy đối với lớp có thuộc tính con trỏ, thì ngoài hàm tạo, cần xây dựng thêm:
 - Hàm hủy
 - Hàm tạo sao chép
 - Phương thức toán tử gán

Chú ý: Không phải mọi câu lệnh chứa có dấu = đều gọi đến toán tử gán. Cần phân biệt 3 trường hợp:

1. Câu lệnh new (chứa dấu =) sẽ gọi đến hàm tạo, ví dụ:
HT *h= new HT(50,6); // gọi đến hàm tạo có đối
2. Câu lệnh khai báo và khởi gán (dùng dấu =) sẽ gọi đến hàm tạo sao chép, ví dụ:
HT k=*h; // gọi đến hàm tạo sao chép
3. Câu lệnh gán sẽ gọi đến toán tử gán, ví dụ:
HT u;
u=*h; // gọi đến phương thức toán tử gán

6.4. Ví dụ minh họa

Chương trình dưới đây định nghĩa lớp HT (hình tròn) và minh họa:

- + Hàm tạo và hàm hủy
- + Phương thức toán tử gán có kiểu tham chiếu
- + Hàm tạo sao chép
- + Cách dùng con trỏ this trong hàm tạo sao chép
- + Cách dùng con trỏ `_new_handler` để kiểm tra việc cấp phát bộ nhớ.

```
//CT4_10.CPP
// Lop do hoa
// Ham huy
// toan tu gan - tra ve tham chieu
// Ham tao sao chep
// Trong ham huy co the goi PT khac
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <graphics.h>
#include <new.h>
#include <mem.h>
static void kiem_tra_bo_nho() ;
void ktdh();
int xmax,ymax;
void kiem_tra_bo_nho()
{
    outtextxy(1,1,"LOI BO NHO");
    getch();
    closegraph();
    exit(1);
}
class HT
{
    private:
        int r,m ;
        int xhien,yhien;
        char *pht;
        int hienmh;
    public:
        HT();
        HT(int r1,int m1=15);
        HT(const HT &h);
        ~HT();
        void hien(int x, int y);
        void an();
        const HT &operator=(const HT &h);
};
const HT & HT::operator=(const HT &h)
{
```

```
// outtextxy(1,1,"Gan"); getch();
r = h.r ; m = h.m ;
xhien = yhien = 0;
hienmh = 0 ;
if (h.pht==NULL)
    pht = NULL;
else
{
    int size;
    size = imagesize(0,0,r+r,r+r);
    pht = new char[size];
    memcpy(pht,h.pht,size);
}
return h;
}

HT::HT(const HT &h)
{
    //outtextxy(300,1,"constructor sao chep"); getch();
    *this = h;
}

HT:: HT()
{
    r=m=hienmh=0;
    xhien=yhien=0;
    pht=NULL;
}

HT::HT(int r1,int m1)
{
    r=r1; m=m1; hienmh=0;
    xhien=yhien=0;
    if (r<0) r=0;
    if (r==0)
    {
        pht=NULL;
    }
    else
    {
        int size; char *pmh;
        size = imagesize(0,0,r+r,r+r);
        pmh = new char[size];
```



```
    getimage(0,0,r+r,r+r,pmh);
    setcolor(m);
    circle(r,r,r);
    setfillstyle(1,m);
    floodfill(r,r,m);
    pht = new char[size];
    getimage(0,0,r+r,r+r,pht);
    putimage(0,0,pmh,COPY_PUT);
    delete pmh;
    pmh=NULL;
}
}

void HT::hien(int x, int y)
{
    if (pht!=NULL && !hienmh) // chua hien
    {
        hienmh=1;
        xhien=x; yhien=y;
        putimage(x,y,pht,XOR_PUT);
    }
}

void HT::an()
{
    if (hienmh) // dang hien
    {
        hienmh=0;
        putimage(xhien,yhien,pht,XOR_PUT);
    }
}

HT::~HT()
{
    an();
    if (pht!=NULL)
    {
        delete pht;
        pht=NULL;
    }
}

void ktdh()
{

```

```
int mh=0,mode=0;
initgraph(&mh,&mode,"");
xmax = getmaxx();
ymax = getmaxy();
}

void main()
{
    _new_handler = kiem_tra_bo_nho ;
    ktdh();
    HT *h= new HT(50,6); // gọi hàm tạo có đối
    h->hien(100,200);
    HT k=*h;    // gọi hàm tạo sao chép
    k.hien(200,200);
    HT t,v,u;
    t = v = u = *h; // gọi toán tử gán
    u.hien(300,200);
    v.hien(400,200);
    t.hien(500,200);
    getch();
    closegraph();
}
```

6.5. Vai trò của phương thức toán tử gán

Chương trình trên sẽ vẽ 5 hình tròn trên màn hình. Điều gì sẽ xảy ra nếu bỏ đi phương thức toán tử gán và hàm tạo sao chép?

- + Nếu bỏ cả hai, thì chỉ xuất hiện một hình tròn tại vị trí (100,200).
- + Nếu bỏ toán tử gán (giữ hàm tạo sao chép) thì chỉ xuất hiện 2 hình tròn tại các vị trí (100,200) và (200,200).
- + Nếu bỏ hàm tạo sao chép (giữ toán tử gán) thì xuất hiện 4 hình tròn.

Bài 7. Phân loại phương thức, phương thức inline

7.1. Phân loại các phương thức

Có thể chia phương thức thành các nhóm:

1. Các phương thức thông thường
2. Các phương thức dùng để xây dựng và huỷ bỏ đối tượng gồm:
 - + Hàm tạo không đối,
 - + Hàm tạo có đối
 - + Hàm tạo sao chép
 - + Hàm huỷ
3. Các phương thức toán tử

7.2. Con trỏ this

Mọi phương thức đều dùng con trỏ this như đối thứ nhất (đối ẩn). Ngoài ra trong phương thức có thể đưa vào các đối tượng mình được khai báo như đối của hàm.

- + Với các phương thức thông thường, thì đối ẩn biểu thị đối tượng chủ thể trong lời gọi phương thức.
- + Với các hàm tạo, thì đối ẩn biểu thị đối tượng mới được hình thành.
- + Với các hàm huỷ, thì đối ẩn biểu thị đối tượng sắp bị huỷ bỏ.
- + Với các phương thức toán tử, thì đối ẩn biểu thị toán hạng đối tượng thứ nhất.

7.3. Phương thức inline.

Có 2 cách để biên dịch phương thức theo kiểu inline:

Cách 1: Xây dựng phương thức bên trong định nghĩa lớp.

Cách 2: Thêm từ khoá inline vào định nghĩa phương thức (bên ngoài định nghĩa lớp).

Chú ý là chỉ các phương thức ngắn không chứa các câu lệnh phức tạp (như chu trình, goto, switch, đệ quy) mới có thể trở thành inline. Nếu có ý định biên dịch theo kiểu inline các phương thức chứa các câu lệnh phức tạp nói trên, thì Trình biên dịch sẽ báo lỗi.

Trong chương trình dưới đây, tất cả các phương thức của lớp PS (phân số) đều là phương thức inline

```
//CT4_11.CPP
// Lop PS
// Inline
#include <conio.h>
#include <iostream.h>
class PS
{
private:
    int t,m ;
public:
    PS()
    {
        t=0;m=1;
    }
    PS(int t1, int m1);
    void nhap();
    void in();
    PS operator*=(PS p2)
    {
        t*=p2.t;
        m*=p2.m;
        return *this;
    }
};

inline PS::PS(int t1, int m1)
```

```
{
    t=t1;
    m=m1;
}

inline void PS::nhap()
{
    cout << "\nNhap tu va mau: ";
    cin >> t >> m;
}

inline void PS::in()
{
    cout << "\nPS = " << t << "/" << m ;
}

void main()
{
    PS q,p,s(3,5);
    cout << "\n Nhap PS p";
    p.nhap();
    s.in();
    p.in();
    q = p*s;
    p.in();
    q.in();
    getch();
}
```

Bài 8. Hàm tạo và đối tượng thành phần

8.1. Lớp bao, lớp thành phần

Một lớp có thuộc tính là đối tượng của lớp khác gọi là lớp bao, ví dụ:

```
class A
{
    private:
        int a, b;
        ...
};

class B
{
    private:
```

```
double x, y, z;
...
};
class C
{
private:
int m, n;
A u;
B p, q;
...
};
```

Trong ví dụ trên thì:

C là lớp bao

A, B là các lớp thành phần (của C)

8.2. Hàm tạo của lớp bao

+ Chú ý là trong các phương thức của lớp bao không cho phép truy nhập trực tiếp đến các thuộc tính của các đối tượng của các lớp thành phần.

+ Vì vậy, khi xây dựng hàm tạo của lớp bao, phải sử dụng các hàm tạo của lớp thành phần để khởi gán cho các đối tượng thành phần của lớp bao.

Ví dụ khi xây dựng hàm tạo của lớp C, cần dùng các hàm tạo của lớp A để khởi gán cho đối tượng thành phần u và dùng các hàm tạo của lớp B để khởi gán cho các đối tượng thành phần p, q.

8.3. Cách dùng hàm tạo của lớp thành phần để xây dựng hàm tạo của lớp bao

+ Để dùng hàm tạo (của lớp thành phần) khởi gán cho đối tượng thành phần của lớp bao, ta sử dụng mẫu:

tên_đối_tượng(danh sách giá trị)

+ Các mẫu trên cần viết bên ngoài thân hàm tạo, ngay sau dòng đầu tiên. Nói một cách cụ thể hơn, hàm tạo sẽ có dạng:

tên_lớp(danh sách đối) : tên_đối_tượng(danh sách giá trị),

...

tên_đối_tượng(danh sách giá trị)

{

// Các câu lệnh trong thân hàm tạo

}

Chú ý là các dấu ngoặc sau tên đối tượng luôn luôn phải có, ngay cả khi danh sách giá trị bên trong là rỗng.

+ Danh sách giá trị lấy từ danh sách đối. Dựa vào danh sách giá trị, Trình biên dịch sẽ biết cần dùng hàm tạo nào để khởi gán cho đối tượng. Nếu danh sách giá trị là rỗng thì hàm tạo không đối sẽ được sử dụng.

+ Các đối tượng muốn khởi gán bằng hàm tạo không đối có thể bỏ qua, không cần phải liệt kê trong hàm tạo. Nói cách khác: Các đối tượng không được liệt kê trên dòng đầu hàm tạo của lớp bao, đều được khởi gán bằng hàm tạo không đối của lớp thành phần.

Ví dụ:

class A

```
{
    private:
        int a, b;
    public:
        A()
        {
            a=b=0;
        }
        A(int a1, int b1)
        {
            a = a1; b = b1;
        }
        ...
};
```

class B

```
{
    private:
        double x, y, z;
    public:
        B()
        {
            x = y = z = 0.0 ;
        }
        B(double x1, double y1)
        {
            x = x1; y = y1; z = 0.0 ;
        }
        B(double x1, double y1, double z1)
        {
            x = x1; y = y1; z = z1 ;
        }
        ...
};
```

class C

```
{
    private:
        int m, n;
        A u, v;
        B p, q, r;
    public:
        C(int m1, int n1, int a1, int b1, double x1, double y1, double x2, double y2, double z2) :
            u(), v(a1, b1), q(x1, y1), r(x2, y2, z2)
        {
            m = m1; n = n1;
        }
};
```

Trong hàm tạo nói trên của lớp C, thì các đối tượng thành phần được khởi gán như sau:

u được khởi gán bằng hàm tạo không đối của lớp A

v được khởi gán bằng hàm tạo 2 đối của lớp A

q được khởi gán bằng hàm tạo 2 đối của lớp B

r được khởi gán bằng hàm tạo 3 đối của lớp B

p (không có mặt) được khởi gán bằng hàm tạo không đối của lớp B

8.4. Sử dụng các phương thức của lớp thành phần

Mặc dù lớp bao có các thành phần đối tượng, nhưng trong lớp bao lại không được phép truy nhập đến các thuộc tính của các đối tượng này. Vì vậy giải pháp thông thường là:

- + Trong các lớp thành phần, xây dựng sẵn các phương thức để có thể lấy ra các thuộc tính của lớp.

- + Trong lớp bao dùng các phương thức của lớp thành phần để nhận các thuộc tính của các đối tượng thành viên cần dùng đến.

8.5. Các ví dụ

Hai chương trình dưới đây minh họa các điều đã nói trong các mục trên.

Ví dụ 1:

Trong ví dụ này xét 2 lớp:

DIEM (Điểm) và DT (Đoạn thẳng)

Lớp DIEM là lớp thành phần của lớp DT

```
//CT4_12.CPP
```

```
// Thuoc tinh doi tuong
```

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

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

```
class DIEM
```

```
{
```

```
    private:
```

```
        int x,y ;
```

```
    public:
```

```
DIEM()
{
    x=y=0;
}
DIEM(int x1, int y1)
{
    x= x1; y=y1;
}
void in()
{
    cout << "(" << x << ", " << y << ")";
}
};

class DT
{
private:
    DIEM d1, d2;
    int m;
public:
    DT() : d1(), d2()
    {
        m=0;
    }
    DT(int m1,int x1, int y1, int x2, int y2) : d1(x1,y1), d2(x2,y2)
    {
        m=m1;
    }
    DT(int m1,DIEM t1, DIEM t2)
    {
        m=m1;
        d1 = t1;
        d2 = t2;
    }
    void in()
    {
        cout << "\n Diem dau : "; d1.in();
        cout << "\n Diem cuoi: "; d2.in();
        cout << "\n Mau : " << m;
    }
};
```



```
void main()
{
    DT u, v(1,100,100,200,200), s(2,DIEM(300,300),
        DIEM(400,400)) ;
    clrscr();
    u.in();
    v.in();
    s.in();
    getch();
}
```

Ví dụ 2:

Trong ví dụ này xét 3 lớp:

Diem (Điểm)

DTron (Đường tròn)

HTron (Hình tròn)

Lớp DTron có một lớp thành phần là lớp Diem.

Lớp HTron có 2 lớp thành phần là lớp DTron và lớp Diem.

Trong lớp DTron đưa vào phương thức vẽ đường tròn.

Trong lớp HTron đưa vào phương thức vẽ và tô màu hình tròn.

Khi xây dựng phương thức của lớp bao cần sử dụng các phương thức của lớp thành phần.

```
//CT4_13.CPP
// Thuoc tinh doi tuong
#include <conio.h>
#include <iostream.h>
#include <graphics.h>
class Diem
{
    private:
        int x,y ;
    public:
        Diem()
        {
            x=y=0;
        }
        Diem(int x1, int y1)
        {
            x= x1; y=y1;
        }
        int getx()
        {
            return x;
        }
    };
};
```

```
    }
    int gety()
    {
        return y;
    }
};

class DTron // Duong tron
{
private:
    Diem t ; // tam
    int r ;
    int m;
public:
    DTron()
    {
        r=m=0;
    }
    DTron(int x1,int y1,int r1,int m1): t(x1,y1)
    {
        m=m1; r=r1;
    }
    int mau()
    {
        return m;
    }
    void ve()
    {
        setcolor(m);
        circle(t.getx(),t.gety(),r);
    }
};

class HTron
{
private:
    DTron dt;
    Diem d;
    int m;
public:
    HTron()
    {
        m=0;
    }
};
```

```
    }
    HTron(int x1, int y1, int r1, int m1,
          int x, int y, int mt): dt(x1,y1,r1,m1), d(x,y)
    {
        m = mt;
    }
    void ve()
    {
        dt.ve();
        setfillstyle(1,m);
        floodfill(d.getx(),d.gety(),dt.mau());
    }
};

void main()
{
    int mh=0, mode=0;
    initgraph(&mh,&mode,"");
    setbkcolor(1);
    DTron dt(100,100,80,6);
    HTron ht(300,300,150,15,300,300,4);
    dt.ve();
    ht.ve();
    getch();
    closegraph();
}
```

Bài 9. Các thành phần tĩnh

9.1. Thành phần dữ liệu tĩnh

+ Thành phần dữ liệu được khai báo bằng từ khoá static gọi là tĩnh, ví dụ:

```
class A
{
    private:
        static int ts ; // Thành phần tĩnh
        int x;
        ....
};
```

+ Thành phần tĩnh được cấp phát một vùng nhớ cố định. Nó tồn tại ngay cả khi lớp chưa có một đối tượng nào cả.

+ Thành phần tĩnh là chung cho cả lớp, nó không phải là riêng của mỗi đối tượng. Ví dụ xét 2 đối tượng:

A u,v ; // Khai báo 2 đối tượng

thì giữa các thành phần x và ts có sự khác nhau như sau:

u.x và v.x có 2 vùng nhớ khác nhau

u.ts và v.ts chỉ là một, chúng cùng biểu thị một vùng nhớ thành phần ts tồn tại ngay khi u và v chưa khai báo

+ Để biểu thị thành phần tĩnh, ta có thể dùng tên lớp, ví dụ: Đối với ts thì 3 cách viết sau là tương đương:

A::ts u.ts v.ts

+ Khai báo và khởi gán giá trị cho thành phần tĩnh

Thành phần tĩnh sẽ được cấp phát bộ nhớ và khởi gán giá trị ban đầu bằng một câu lệnh khai báo đặt sau định nghĩa lớp (bên ngoài các hàm, kể cả hàm main), theo các mẫu:

int A::ts ; // Khởi gán cho ts giá trị 0

int A::ts = 1234; // Khởi gán cho ts giá trị 1234

Chú ý: Khi chưa khai báo thì thành phần tĩnh chưa tồn tại. Ví dụ xét chương trình sau:

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

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

```
class HDBH // Hoá đơn bán hàng
```

```
{
```

```
private:
```

```
    char *tenhang ; // Tên hàng
```

```
    double tienban ; // Tiền bán
```

```
    static int tshd ; // Tổng số hoá đơn
```

```
    static double tstienban ; // Tổng số tiền bán
```

```
public:
```

```
    static void in()
```

```
    {
```

```
        cout << "\n" << tshd;
```

```
        cout << "\n" << tstienban;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    HDBH::in();
```

```
    getch();
```

```
}
```

Các thành phần tĩnh tshd và tstienban chưa khai báo, nên chưa tồn tại. Vì vậy các câu lệnh in giá trị các thành phần này trong phương thức in là không logic. Khi dịch chương trình, sẽ nhận được các thông báo lỗi (tại phương thức in) như sau:

Undefined symbol HDBH::tshd in module ...

Undefined symbol HDBH::tstienban in module ...

Có thể sửa chương trình trên bằng cách đưa vào các lệnh khai báo các thành phần tĩnh tshd và tstienban như sau:

```
//CT4_14.CPP
```

```
// thanh phan tinh
// Lop HDBH (hoa don ban hang)
#include <conio.h>
#include <iostream.h>
class HDBH
{
    private:
        int shd ;
        char *tenhang ;
        double tienban ;
        static int tshd ;
        static double tstienban ;
    public:
        static void in()
        {
            cout << "\n" << tshd;
            cout << "\n" << tstienban;
        }
};
int HDBH::tshd=5;
double HDBH::tstienban=20000.0;
void main()
{
    HDBH::in();
    getch();
}
```

9.2. Khi nào cần sử dụng các thành phần dữ liệu tĩnh

Xét một ví dụ về quản lý các hoá đơn bán hàng. Mỗi hoá đơn có: Tên hàng, số tiền bán. Rõ ràng các thuộc tính nói trên là riêng của mỗi hoá đơn. Mặt khác nếu chúng ta quan tâm đến tổng số hoá đơn đã bán, tổng số tiền đã bán, thì các thông tin này là chung. Vì vậy khi thiết kế lớp HDBH (hoá đơn bán hàng) , thì ta có thể đưa vào 4 thành phần dữ liệu là:

tenhang (tên hàng)
tienban (tiền bán)
tshd (tổng số hoá đơn)
tstienban (tổng số tiền bán)

Các thuộc tính tenhang và tienban là riêng của mỗi hoá đơn, nên chúng được chọn là các thuộc tính thông thường. Còn các thuộc tính tshd và tstienban là chung cho cả lớp nên chúng được chọn là các thuộc tính tĩnh.

9.3. Phương thức tĩnh

+ Có 2 cách viết phương thức tĩnh:

Cách 1: Dùng từ khoá static đặt trước định nghĩa phương thức viết bên trong định nghĩa lớp (như phương thức in() ví dụ cuối của mục 9.1).

Cách 2: Nếu phương thức xây dựng bên ngoài định nghĩa lớp, thì dùng từ khoá static đặt trước khai báo phương thức bên trong định nghĩa lớp. Chú ý không cho phép dùng từ khoá static đặt trước định nghĩa phương thức viết bên ngoài định nghĩa lớp.

+ Phương thức tĩnh là chung cho cả lớp, nó không lệ thuộc vào một đối tượng cụ thể, nó tồn tại ngay khi lớp chưa có đối tượng nào (xem ví dụ trong mục 9.1).

+ Lời gọi phương thức tĩnh:

Có thể xuất phát từ một đối tượng nào đó (như vẫn dùng khi gọi các phương thức khác)

Có thể dùng tên lớp

Ví dụ xét lớp HDBH trong mục 9.1 và xét các câu lệnh:

HDBH u, v;

Khi đó để gọi phương thức tĩnh in() có thể dùng một trong các lệnh sau:

u.in();

v.in();

HDBH::in();

+ Vì phương thức tĩnh là độc lập với các đối tượng, nên không thể dùng phương thức tĩnh để xử lý dữ liệu của các đối tượng chủ thể trong lời gọi phương thức tĩnh. Nói cách khác không cho phép truy nhập tới các thuộc tính (trừ thuộc tính tĩnh) trong thân phương thức tĩnh. Điều đó cũng đồng nghĩa với việc không cho phép dùng con trỏ this trong phương thức tĩnh.

Ví dụ nếu lập phương thức tĩnh in() để in các thuộc tính của lớp HDBH như sau:

```
class HDBH
{
    private:
        int shd ;
        char *tenhang ;
        double tienban ;
        static int tshd ;
        static double tstienban ;
    public:
        static void in()
        {
            cout << "\n" << tshd;
            cout << "\n" << tstienban;
            cout << "\n" << tenhang;
            cout << "\n" << tienban;
        }
};
```

thì sẽ bị lỗi, vì trong thân phương thức tĩnh không cho phép truy nhập đến các thuộc tính tenhang và tienban.

9.4. Ví dụ minh hoạ việc dùng phương thức tĩnh

Xét bài toán quản lý hoá đơn bán hàng. Mỗi hoá đơn có 2 dữ liệu là tên hàng và tiền bán. Sử dụng hàm tạo để tạo ra các hoá đơn, dùng hàm huỷ để bỏ đi (loại đi) các hoá đơn không cần lưu trữ, dùng một phương thức để sửa chữa nội dung hoá đơn (nhập lại tiền bán). Vấn đề đặt ra là sau một số thao tác: Tạo, sửa và huỷ hoá đơn thì tổng số hoá đơn còn lại là bao nhiêu và tổng số tiền trên các hoá đơn còn lại là bao nhiêu?

Chương trình dưới đây nhằm đáp ứng yêu cầu đặt ra.

```
//CT4_14.CPP
// thanh phan tinh
// Lop HDBH (hoa don ban hang)
#include <conio.h>
#include <iostream.h>
class HDBH
{
private:
    char *tenhang ;
    double tienban ;
    static int tshd ;
    static double tstienban ;
public:
    HDBH(char *tenhang1=NULL,double tienban1=0.0 )
    {
        tienban=tienban1;
        tenhang=tenhang1;
        ++tshd;
        tstienban += tienban;
    }
    ~HDBH()
    {
        --tshd;
        tstienban -= tienban;
    }
    void sua();
    static void in();
};
int HDBH::tshd=0;
double HDBH::tstienban=0;
void HDBH::in()
{
    cout << "\n\nTong so hoa don: " << tshd;
    cout << "\nTong so tien: " << tstienban;
```

```
}  
  
void HDBH::sua()  
{  
    cout << "\n\nTen hang: " << tenhang;  
    cout << "\nTien ban : " << tienban;  
    tstienban -= tienban;  
    cout << "\nSua tien ban thanh : ";  
    cin >> tienban;  
    tstienban += tienban;  
}  
  
void main()  
{  
    HDBH *h1 = new HDBH("Xi mang",2000);  
    HDBH *h2 = new HDBH("Sat thep",3000);  
    HDBH *h3 = new HDBH("Ti vi",4000);  
    clrscr();  
    HDBH::in();  
    getch();  
    delete h1;  
    HDBH::in();  
    getch();  
    h2->sua();  
    HDBH::in();  
    getch();  
    delete h3;  
    HDBH::in();  
    getch();  
}
```

Bài 10. Mảng đối tượng

10.1. Khai báo

Có thể dùng tên lớp để khai báo mảng đối tượng (giống như khai báo mảng int, float, char, ...) theo mẫu:

Tên_lớp tên_mảng[kích_cỡ];

Ví dụ giả sử đã định nghĩa lớp DIEM (Điểm), khi đó có thể khai báo các mảng đối tượng DIEM như sau:

DIEM a[10], b[20];

ý nghĩa: a là mảng kiểu DIEM gồm 10 phần tử

b là mảng kiểu DIEM gồm 20 phần tử

Câu lệnh khai báo mảng sẽ gọi tới hàm tạo không đối để tạo các phần tử mảng. Trong ví dụ trên, hàm tạo được gọi 30 lần để tạo 30 phần tử mảng đối tượng.

10.2. Khai báo và khởi gán

Để khai báo mảng và khởi gán giá trị cho các phần tử mảng đối tượng, cần dùng các hàm tạo có đối theo mẫu sau:

Tên_lớp tên_mảng[kích_có] = { Tên_lớp(các tham số), ...,
Tên_lớp(các tham số) } ;

Ví dụ giả sử lớp DIEM đã định nghĩa:

```
class DIEM
{
    private:
        int x, y ;
    public:
        DIEM()
        {
            x=y=0;
        }
        DIEM(int x1, int y1)
        {
            x=x1; y=y1;
        }
        void nhapsl();
        void ve_doan_thang(DIEM d2, int mau) ;
};
```

Khi đó các câu lệnh khai báo dưới đây đều đúng:

DIEM d[5] = {DIEM(1,1),DIEM(200,200)};

DIEM u[] = {DIEM(1,1),DIEM(200,200)};

ý nghĩa của các lệnh này như sau:

Câu lệnh đầu gọi tới hàm tạo 2 lần để khởi gán cho d[1], d[2] và gọi tới hàm tạo không đối 3 lần để tạo các phần tử d[3], d[4] và d[5].

Câu lệnh sau gọi tới hàm tạo 2 lần để khởi gán cho u[1], u[2]. Mảng u sẽ gồm 2 phần tử.

10.3. Biểu thị thành phần của phần tử mảng

Để biểu thị thuộc tính của phần tử mảng đối tượng, ta viết như sau:

Tên_mảng[chỉ số] . Tên_thuộc_tính

Để thực hiện phương thức đối với phần tử mảng ta viết như sau:

Tên_mảng[chỉ số] . Tên_phương_thức(danh sách tham số) ;

Ví dụ để vẽ đoạn thẳng nối điểm d[1] với d[2] theo màu đỏ, ta có thể dùng phương thức ve_doan_thang như sau:

d[1].ve_doan_thang(d[2], 4); // Thực hiện phương thức đối với d[1]

10.4. Ví dụ

Chương trình dưới đây đưa vào lớp TS (thí sinh) và xét bài toán: Nhập một danh sách thí sinh, sắp xếp danh sách theo thứ tự giảm của tổng điểm. Chương trình minh họa:

- + Cách dùng mảng đối tượng.
- + Vai trò con trỏ this (trong phương thức hv(hoán vị)) .
- + Các hàm tạo, hàm huỷ.
- + Vai trò của toán tử gán (nếu sử dụng phép gán mặc định chương trình sẽ cho kết quả sai).

```
//CT4_15.CPP
// mang doi tuong
// Lop TS (thi sinh)
// Chu y vai tro cua toan tu gan
#include <conio.h>
#include <iostream.h>
#include <string.h>
class TS
{
private:
    char *ht;
    double td;
public:
    TS()
    {
        ht = new char[20];
        td = 0;
    }
    ~TS()
    {
        delete ht;
    }
    const TS &operator=(const TS &ts2)
    {
        this->td = ts2.td;
        strcpy(this->ht,ts2.ht);
        return ts2;
    }
    void nhap(int i);
    void in();
    double gettd()
    {
        return td;
    }
}
```

```
    }
    void hv(TS &ts2)
    {
        TS tg;
        tg = *this ;
        *this = ts2 ;
        ts2 = tg;
    }
};

void TS::in()
{
    cout << "\nHo ten: " << ht << "   Tong diem: " << td;
}

void TS::nhap(int i)
{
    cout << "\nNhap thi sinh " << i ;
    cout << "\nHo ten: " ; cin >> ht;
    cout << "Tong diem: " ; cin >> td;
}

void main()
{
    TS ts[100];
    int n, i, j;
    clrscr();
    cout << "\n So thi sinh: " ;
    cin >> n;
    for (i=1; i<= n; ++i)
        ts[i].nhap(i);
    cout << "\n Danh sach nhap vao:";
    for (i=1; i<= n; ++i)
        ts[i].in();
    for (i=1; i<n ; ++i)
        for (j=i+1 ; j<=n; ++j)
            if (ts[i].gettd() < ts[j].gettd())
                ts[i].hv(ts[j]);
    cout << "\n\n Danh sach sau khi sap xep:";
    for (i=1; i<= n; ++i)
        ts[i].in();
    getch();
}
```

Bài 11. cấp phát bộ nhớ cho đối tượng

11.1. Cách cấp phát bộ nhớ cho đối tượng

Có thể dùng new và tên lớp để cấp phát một vùng nhớ cho một hoặc một dãy các đối tượng. Bộ nhớ cấp phát được quản lý bởi một con trỏ kiểu đối tượng. Ví dụ sau khi đã định nghĩa lớp DIEM như trong mục trên, ta có thể thực hiện các lệnh cấp phát bộ nhớ như sau:

```
int n = 10;
DIEM *p, *q, *r ;
p = new DIEM ; // Cấp phát bộ nhớ cho một đối tượng
q = new DIEM[n] ; //Cấp phát bộ nhớ cho n đối tượng
r = new DIEM(200,100); // Cấp phát bộ nhớ và khởi gán cho
                        // một đối tượng
```

11.2. Làm việc với đối tượng thông qua con trỏ

+ Giả sử con trỏ p trỏ tới vùng nhớ của một đối tượng nào đó. Khi đó:

- Để biểu thị một thành phần (thuộc tính hoặc phương thức) của đối tượng, ta dùng mẫu viết sau:

p -> tên_thành_phần

- Để biểu thị đối tượng, ta dùng mẫu viết sau:

*p

+ Giả sử con trỏ q trỏ tới địa chỉ đầu vùng nhớ của một dãy đối tượng. Khi đó:

- Để biểu thị một thành phần (thuộc tính hoặc phương thức) của đối tượng thứ i, ta dùng một trong các mẫu viết sau:

q[i].tên_thành_phần

(q+i)-> tên_thành_phần

- Để biểu thị đối tượng thứ i, ta dùng một trong các mẫu viết sau:

q[i]

*(q+i)

11.3. Bài toán sắp xếp thí sinh

Trong mục 10.4. đã sử dụng mảng đối tượng để giải quyết bài toán: Nhập một danh sách thí sinh, sắp xếp danh sách theo thứ tự giảm của tổng điểm. Dưới đây sẽ đưa ra phương án mới bằng cách dùng con trỏ và cấp phát bộ nhớ cho các đối tượng. Chương trình chỉ thay đổi hàm main() như sau:

```
void main()
{
    TS *ts;
    int n, i, j;
    clrscr();
    cout << "\n So thi sinh: " ;
    cin >> n;
    ts = new TS[n+1];
    for (i=1; i<= n; ++i)
        ts[i].nhap(i);
    cout << "\n Danh sach nhap vao:";
    for (i=1; i<= n; ++i)
```

```
    ts[i].in();
    for (i=1; i<n ; ++i)
        for (j=i+1 ; j<=n; ++j)
            if (ts[i].gettd() < ts[j].gettd())
                ts[i].hv(ts[j]);
    cout << "\n\n Danh sach sau khi sap xep:";
    for (i=1; i<= n; ++i)
        ts[i].in();
    getch();
}
```

Nhận xét: Sự khác biệt giữa hàm main mới và hàm main trong 10.4 là rất ít.

11.4. Danh sách móc nối

Chương trình dưới đây định nghĩa lớp tự trở TS (lớp có thuộc tính kiểu *TS). Lớp này được dùng để tổ chức danh sách móc nối. Chương trình nhập một danh sách thí sinh và chứa trong một danh sách móc nối. Sau đó duyệt trên danh sách này để in các thí sinh trúng tuyển. So với lớp TS nêu trong mục 10.4, lớp TS ở đây có một số điểm khác như sau:

+ Thêm thuộc tính:

TS *dc; // Dùng để chứa địa chỉ của một đối tượng kiểu TS

+ Thêm các phương thức:

void setdc(TS *dc1) ; // Gán dc1 cho thuộc tính dc

TS *getdc() ; // Nhận giá trị của dc

+ Phương thức nhập trong chương trình trước có kiểu void nay sửa là:

int nhap(int i);

Phương thức trả về 1 nếu họ tên nhập vào khác trống, trả về 0 nếu trái lại.

+ Bỏ đi các phương thức không dùng đến như: Toán tử gán, hoán vị.

//CT4_16.CPP

// Danh sách móc nối

// Lop TS (thi sinh)

#include <conio.h>

#include <iostream.h>

#include <string.h>

#include <ctype.h>

#include <stdio.h>

class TS

{

private:

char *ht;

double td;

TS *dc;

```
public:
    TS()
    {
        ht = new char[20];
        td = 0;
        dc=NULL;
    }
    ~TS()
    {
        delete ht; dc=NULL ;
    }
    int nhap(int i);
    void in();
    double gettd()
    {
        return td;
    }
    void setdc(TS *dc1)
    {
        dc=dc1;
    }
    TS *getdc()
    {
        return dc;
    }
};

void TS::in()
{
    cout << "\nHo ten: " << ht << "   Tong diem: " << td;
}

int TS::nhap(int i)
{
    cout << "\nNhap thi sinh " << i ;
    cout << "\nHo ten (Bấm Enter để kết thúc nhập): " ;
    fflush(stdin);
    gets(ht);
    if (ht[0]==0) return 0;
    cout << "Tong diem: " ; cin >> td;
    dc=NULL;
    return 1;
}

void main()
```

```
{
    int i=0;
    TS *pdau,*p,*q;
    pdau=NULL;
    clrscr();
    while(1)
    {
        q=new TS;
        ++i;
        if (q->nhap(i)==0)
        {
            delete q; break;
        }
        if (pdau==NULL)
            pdau = p = q;
        else
        {
            p->setdc(q) ;
            p = q;
        }
    }
    /* In */
    double diemchuan;
    cout << "\nDiem chuan: " ;
    cin >> diemchuan;
    cout << "\nDanh sach trung tuyen:" ;
    p=pdau;
    while (p!=NULL)
    {
        if (p->gettd()>=diemchuan)
            p->in();
        p = p->getdc();
    }
    getch();
}
```

Bài 12. Đối tượng hằng, phương thức hằng

+ Cũng giống như các phần tử dữ liệu khác, một đối tượng có thể được khai báo là hằng bằng cách dùng từ khoá const. Ví dụ:

```
class DIEM
{
    private:
        int x, y;
```

```
int m;
public:
    DIEM()
    {
        x = y = m = 0;
    }
    DIEM(int x1, int y1, int m1=15)
    {
        x = x1; y = y1; m = m1;
    }
    ...
};
```

const DIEM d = DIEM(200,100); // Khai báo đối tượng hằng

+ Khi khai báo cần sử dụng các hàm tạo để khởi gán giá trị cho đối tượng hằng. Giá trị khởi tạo có thể là các hằng, các biến, các biểu thức và các hàm, ví dụ:

```
int x0=100, y0=50; m0=4;
```

```
const DIEM d5 = DIEM(x0 + getmaxx()/2, y0 + getmaxy()/2, m0);
```

+ Các phương thức có thể sử dụng cho các đối tượng hằng là hàm tạo và hàm huỷ. Về lý thuyết các đối tượng hằng không thể bị thay đổi, mà chỉ được tạo ra hoặc huỷ bỏ đi.

Khi dùng một phương thức cho đối tượng hằng, thì CTBD (Chương trình biên dịch) sẽ cảnh báo (warning):

Non-const function called for const object

Tuy nhiên, chương trình EXE vẫn được tạo và khi thực hiện chương trình, thì nội dung các đối tượng hằng vẫn bị thay đổi. Chương trình dưới đây sẽ minh hoạ điều này. Chương trình đưa vào lớp PS (phân số). Phương thức toán tử ++ vẫn có thể làm thay đổi đối tượng hằng (mặc dù khi biên dịch có 3 cảnh báo).

```
//CT4_19.CPP
// doi tuong const
// Lop PS (phan so)
#include <conio.h>
#include <iostream.h>
#include <string.h>
#include <math.h>
class PS
{
private:
    int t,m;
public:
    PS()
    {
        t = m = 0;
    }
}
```



```
PS(int t1, int m1)
{
    t = t1; m = m1;
}
PS operator++()
{
    t += m ;
    return *this ;
}
void in()
{
    cout << "\nPS= " << t << "/" << m;
}
void nhap()
{
    cout << "\n Nhap tu va mau: " ;
    cin >> t >> m;
}
};

void main()
{
    int t1=-3, m1=5;
    const PS p = PS(abs(t1)+2,m1+2); // Khai báo đối tượng hằng
    clrscr();
    p.in();
    ++p;
    p.in();
    getch();
}
```

+ Phương thức const

Để biến một phương thức thành const ta chỉ việc viết thêm từ khoá const vào sau dòng đầu của phương thức.

Chú ý: Nếu phương thức được khai báo bên trong và định nghĩa bên ngoài lớp, thì từ khoá const cần được bổ sung cả trong khai báo và định nghĩa phương thức.

Trong thân phương thức const không cho phép làm thay đổi các thuộc tính của lớp. Vì vậy việc dùng phương thức const cho các đối tượng hằng sẽ đảm bảo giữ nguyên nội dung của các đối tượng hằng.

Đương nhiên các phương thức const vẫn dùng được cho các đối tượng khác.

Ví dụ sau về lớp PS (phân số) minh hoạ việc dùng phương thức const.

// Đối tượng const

// Phương thức const

```
// Lop PS (phan so)
#include <conio.h>
#include <iostream.h>
#include <string.h>
#include <math.h>
class PS
{
    private:
        int t,m;
    public:
        PS()
        {
            t = m = 0;
        }
        PS(int t1, int m1)
        {
            t = t1; m = m1;
        }
        PS operator++()
        {
            t += m ;
            return *this ;
        }
        void in() const ;
        void nhap()
        {
            cout << "\n Nhap tu va mau: " ;
            cin >> t >> m;
        }
};

void PS::in() const
{
    cout << "\nPS= " << t << "/" << m;
}

void main()
{
    int t1=-3, m1=5;
    const PS p = PS(abs(t1)+2,m1+2);
    PS q;
```

```
clrscr();
q.nhap();
p.in();
q.in();
getch();
}
```

Bài 13. Hàm bạn, lớp bạn

13.1. Hàm bạn (xem mục bài 6, chương 3) của một lớp, tuy không phải là phương thức của lớp, nhưng có thể truy nhập đến các thành phần riêng (private) của lớp. Một hàm có thể là bạn của nhiều lớp.

13.2. Nếu lớp A được khai báo là bạn của lớp B thì tất cả các phương thức của A đều có thể truy nhập đến các thành phần riêng của lớp B. Một lớp có thể là bạn của nhiều lớp khác. Cũng có thể khai báo A là bạn của B và B là bạn của A.

13.3. Cách khai báo lớp bạn

Giả sử có 3 lớp A, B và C. Để khai báo lớp này là bạn của lớp kia, ta viết theo mẫu sau:

```
// Khai báo trước các lớp
class A;
class B ;
class C;
// Định nghĩa các lớp
class A
{
    ...
    friend class B ; // Lớp B là bạn của A
    friend class C ; // Lớp C là bạn của A
    ...
};
class B
{
    ...
    friend class A ; // Lớp A là bạn của B
    friend class C ; // Lớp C là bạn của B
    ...
};
class C
{
    ...
    friend class B ; // Lớp B là bạn của C
    ...
}
```

```
};
```

13.4. Ví dụ

Chương trình dưới đây có 2 lớp:

MT (ma trận vuông)

VT (véc tơ)

Lớp MT là bạn của VT và lớp VT là bạn của MT. Trong chương trình sử dụng các phương thức trùng tên:

2 phương thức nhập():

nhập ma trận

nhập véc tơ

2 phương thức in():

in ma trận

in véc tơ

4 phương thức tích():

tích ma trận với ma trận, kết quả là ma trận

tích ma trận với véc tơ, kết quả là véc tơ

tích véc tơ với ma trận, kết quả là véc tơ

tích véc tơ với véc tơ, kết quả là số thực

Nội dung chương trình là:

+ Nhập các ma trận A, B, C

+ Nhập các véc tơ

+ Tính tích $D = AB$

+ Tính tích $u = Dy$

+ Tính tích $v = xC$

+ Tính tích $s = vu$

```
//CT4_17.CPP
```

```
// Lop ban
```

```
// Lop MT , lop VT
```

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

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

```
class MT;
```

```
class VT;
```

```
class MT
```

```
{
```

```
private:
```

```
double a[10][10];
```

```
int n;
```

```
public:
```

```
friend class VT;
```

```
MT()
```

```
{
```

```
        n=0;
    }
    void nhap();
    void in();
    VT tich(const VT &y);
    MT tich(const MT &b) ;
};

class VT
{
private:
    double x[10];
    int n;
public:
    friend class MT;
    VT()
    {
        n=0;
    }
    void nhap();
    void in();
    VT tich(const MT &b);
    double tich(const VT &y) ;
};

void MT::nhap()
{
    cout << "\n Cap ma tran: ";
    cin >> n;
    for (int i=1; i<=n; ++i)
        for (int j=1; j<=n; ++j)
        {
            cout << "\nPhan tu hang " << i << " cot " << j << " = ";
            cin >> a[i][j];
        }
}

void MT::in()
{
    for (int i=1; i<=n; ++i)
    {
        cout << "\n";
        for (int j=1; j<=n; ++j)
            cout << a[i][j] << " ";
    }
}
```

```
    }  
}  
void VT::nhap()  
{  
    cout << "\n Cap vec to: ";  
    cin >> n;  
    for (int i=1; i<=n; ++i)  
    {  
        cout << "\nPhan tu thu " << i << " = ";  
        cin >> x[i];  
    }  
}  
void VT::in()  
{  
    for (int i=1; i<=n; ++i)  
        cout << x[i] << " ";  
}  
VT MT::tich(const VT &y)  
{  
    VT z;  
    int i,j;  
    for (i=1; i<=n; ++i)  
    {  
        z.x[i] = 0.0 ;  
        for (j=1; j<=n; ++j)  
            z.x[i] += a[i][j]*y.x[j];  
    }  
    z.n = n;  
    return z;  
}  
MT MT::tich(const MT &b)  
{  
    MT c;  
    int i,j,k;  
    for (i=1; i<=n; ++i)  
        for (j=1; j<=n; ++j)  
        {  
            c.a[i][j] = 0.0 ;  
            for (k=1; k<=n; ++k)  
                c.a[i][j] += a[i][k]*b.a[k][j];  
        }  
    c.n = n;  
    return c;  
}
```

```
VT VT::tich(const MT &b)
{
    VT z;
    int i,j;
    for (j=1; j<=n; ++j)
    {
        z.x[j] = 0.0 ;
        for (i=1; i<=n; ++i)
            z.x[j] += b.a[i][j]*x[i];
    }
    z.n = n;
    return z;
}
double VT::tich(const VT &y)
{
    double tg=0.0;
    for (int i=1; i<=n; ++i)
        tg += x[i]*y.x[i];
    return tg;
}
void main()
{
    MT a,b,c;
    VT x,y;
    clrscr();
    cout << "\nMa tran A";
    a.nhap();
    cout << "\nMa tran B";
    b.nhap();
    cout << "\nMa tran C";
    c.nhap();
    cout << "\nvec to X";
    x.nhap();
    cout << "\nvec to Y";
    y.nhap();
    MT d= a.tich(b);
    VT u = d.tich(y);
    VT v = x.tich(c);
    double s = v.tich(u);
    cout << "\n\nVec to v\n";
}
```

```
v.in();  
cout << "\n\nMa tran D";  
d.in();  
cout << "\n\nVec to y\n";  
y.in();  
cout << "\n\nS= vDy = " << s;  
getch();  
}
```


chương 5

Dẫn xuất và thừa kế

Có 2 khái niệm rất quan trọng đã làm nên toàn bộ thế mạnh của phương pháp lập trình hướng đối tượng đó là tính kế thừa (inheritance) và tính tương ứng bội (polymorphism). Tính kế thừa cho phép các lớp được xây dựng trên các lớp đã có. Trong chương này sẽ nói về sự thừa kế của các lớp.

Bài 1. Sự dẫn xuất và tính thừa kế

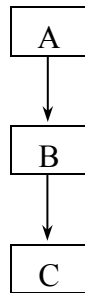
1.1. Lớp cơ sở và lớp dẫn xuất

Một lớp được xây dựng thừa kế một lớp khác gọi là lớp dẫn xuất. Lớp dùng để xây dựng lớp dẫn xuất gọi là lớp cơ sở.

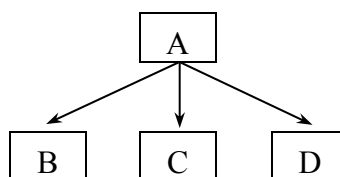
Lớp nào cũng có thể là một lớp cơ sở. Hơn thế nữa, một lớp có thể là cơ sở cho nhiều lớp dẫn xuất khác nhau. Đến lượt mình, lớp dẫn xuất lại có thể dùng làm cơ sở để xây dựng các lớp dẫn xuất khác. Ngoài ra một lớp có thể dẫn xuất từ nhiều lớp cơ sở.

Dưới đây là một số sơ đồ về quan hệ dẫn xuất của các lớp:

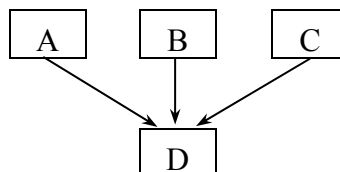
Sơ đồ 1: Lớp B dẫn xuất từ lớp A, lớp C dẫn xuất từ lớp B



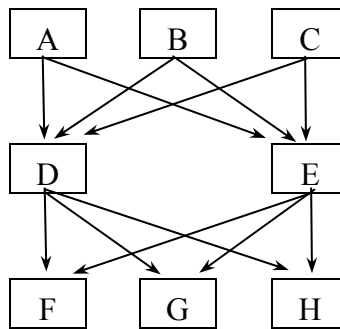
Sơ đồ 2: Lớp A là cơ sở của các lớp B, C và D



Sơ đồ 3: Lớp D dẫn xuất từ 3 lớp A, B, C



Sơ đồ 4: Lược đồ dẫn xuất tổng quát



Tính thừa kế: Một lớp dẫn xuất ngoài các thành phần của riêng nó, nó còn được thừa kế tất cả các thành phần của các lớp cơ sở có liên quan. Ví dụ trong sơ đồ 1 thì lớp C được thừa kế các thành phần của các lớp B và A. Trong sơ đồ 3 thì lớp D được thừa kế các thành phần của các lớp A, B và C. Trong sơ đồ 4 thì lớp G được thừa kế các thành phần của các lớp D, E, A, B và C.

1.2. Cách xây dựng lớp dẫn xuất

Giả sử đã định nghĩa các lớp A và B. Để xây dựng lớp C dẫn xuất từ A và B, ta viết như sau:

```
class C : public A, public B
{
    private:
        // Khai báo các thuộc tính
    public:
        // Các phương thức
};
```

1.3. Thừa kế private và public

Trong ví dụ trên, lớp C thừa kế public các lớp A và B. Nếu thay từ khoá public bằng private, thì sự thừa kế là private.

Chú ý: Nếu bỏ qua không dùng từ khoá thì hiểu là private, ví dụ nếu định nghĩa:

```
class C : public A, B
{
    private:
        // Khai báo các thuộc tính
    public:
        // Các phương thức
};
```

thì A là lớp cơ sở public của C, còn B là lớp cơ sở private của C.

Theo kiểu thừa kế public thì tất cả các thành phần public của lớp cơ sở cũng là các thành phần public của lớp dẫn xuất.

Theo kiểu thừa kế private thì tất cả các thành phần public của lớp cơ sở sẽ trở thành các thành phần private của lớp dẫn xuất.

1.4. Thừa kế các thành phần dữ liệu (thuộc tính)

Các thuộc tính của lớp cơ sở được thừa kế trong lớp dẫn xuất. Như vậy tập thuộc tính của lớp dẫn xuất sẽ gồm: các thuộc tính mới khai báo trong định nghĩa lớp dẫn xuất và các thuộc tính của lớp cơ sở.

Tuy vậy trong lớp dẫn xuất không cho phép truy nhập đến các thuộc tính private của lớp cơ sở.

Chú ý: Cho phép đặt trùng tên thuộc tính trong các lớp cơ sở và lớp dẫn xuất.

Ví dụ:

```
class A
{
    private:
        int a, b, c;
    public:
        ...
};

class B
{
    private:
        double a, b, x;
    public:
        ...
};

class C : public A, B
{
    private:
        char *a, *x;
        int b;
    public:
        ...
};
```

Khi đó lớp C sẽ có các thuộc tính:

A::a, A::b, A::c (kiểu int) - thừa kế từ A

B::a, B::b, B::x (kiểu double) - thừa kế từ B

a, x (kiểu char*) và b (kiểu int) - khai báo trong C

Trong các phương thức của C chỉ cho phép truy nhập trực tiếp tới các thuộc tính khai báo trong C.

1.5. Thừa kế phương thức

Trừ:

- + Hàm tạo
- + Hàm huỷ
- + Toán tử gán

các phương thức (public) khác của lớp cơ sở được thừa kế trong lớp dẫn xuất.

Ví dụ: Trong chương trình dưới đây:

+ Đầu tiên định nghĩa lớp DIEM có:

Các thuộc tính x, y

Hai hàm tạo

Phương thức in()

+ Sau đó xây dựng lớp HINH_TRON dẫn xuất từ lớp DIEM, đưa thêm:

Thuộc tính r

Hai hàm tạo

Phương thức getR

Chú ý cách dùng hàm tạo của lớp cơ sở (lớp DIEM) để xây dựng hàm tạo của lớp dẫn xuất.

+ Trong hàm main:

Khai báo đối tượng h kiểu HINH_TRON

Sử dụng phương thức in() đối với h (sự thừa kế)

Sử dụng phương thức getR đối với h

//CT5-01

// Lop co so

#include <conio.h>

#include <iostream.h>

class DIEM

{

private:

double x, y;

public:

DIEM()

{

x = y = 0.0;

}

DIEM(double x1, double y1)

{

x = x1; y = y1;

}

void in()

{

cout << "\nx= " << x << " y= " << y;

}

};

class HINH_TRON : public DIEM

{

private:

double r;

public:

```
HINH_TRON()
{
    r = 0.0;
}
HINH_TRON(double x1, double y1,
           double r1): DIEM(x1,y1)
{
    r = r1;
}
double getR()
{
    return r;
}
};

void main()
{
    HINH_TRON h(2.5,3.5,8);
    clrscr();
    cout << "\nHinh tron co tam: ";
    h.in();
    cout << "\nCo ban kinh= " << h.getR();
    getch();
}
```

1.6. Lớp cơ sở và đối tượng thành phần

Lớp cơ sở thường được xử lý giống như một thành phần kiểu đối tượng của lớp dẫn xuất. Ví dụ chương trình trong 1.5 có thể thay bằng một chương trình khác trong đó thay việc dùng lớp cơ sở DIEM bằng một thành phần kiểu DIEM trong lớp HINH_TRON. Chương trình mới có thể viết như sau:

```
//CT5-02
// Lop co doi tuong thanh phan
#include <conio.h>
#include <iostream.h>
class DIEM
{
private:
    double x, y;
public:
    DIEM()
    {
        x = y = 0.0;
    }
};
```

```
    }
    DIEM (double x1, double y1)
    {
        x = x1; y = y1;
    }
    void in()
    {
        cout << "\nx= " << x << " y= " << y;
    }
};

class HINH_TRON
{
private:
    DIEM d;
    double r;
public:
    HINH_TRON() : d()
    {
        r = 0.0;
    }
    HINH_TRON(double x1, double y1, double r1): d(x1,y1)
    {
        r = r1;
    }
    void in()
    {
        d.in();
    }
    double getR()
    {
        return r;
    }
};

void main()
{
    HINH_TRON h(2.5,3.5,8);
    clrscr();
    cout << "\nHinh tron co tam: ";
    h.in();
    cout << "\nCo ban kinh= " << h.getR();
    getch();
}
```

Bài 2. Hàm tạo, hàm huỷ đối với tính thừa kế

2.1. Lớp dẫn xuất không thừa kế các hàm tạo, hàm huỷ, toán tử gán của các lớp cơ sở

2.2. Cách xây dựng hàm tạo của lớp dẫn xuất

- + Hàm tạo cần có các đối để khởi gán cho các thuộc tính (thành phần dữ liệu) của lớp.
- + Có thể phân thuộc tính làm 3 loại ứng với 3 cách khởi gán khác nhau:

1. Các thuộc tính mới khai báo trong lớp dẫn xuất. Trong các phương thức của lớp dẫn xuất có thể truy xuất đến các thuộc tính này. Vì vậy chúng thường được khởi gán bằng các câu lệnh gán viết trong thân hàm tạo.

2. Các thành phần kiểu đối tượng. Trong lớp dẫn xuất không cho phép truy nhập đến các thuộc tính của các đối tượng này. Vì vậy để khởi gán cho các đối tượng thành phần cần dùng hàm tạo của lớp tương ứng. Điều này đã trình bày trong mục 8 chương 4.

3. Các thuộc tính thừa kế từ các lớp cơ sở. Trong lớp dẫn xuất không được phép truy nhập đến các thuộc tính này. Vì vậy để khởi gán cho các thuộc tính nói trên, cần sử dụng hàm tạo của lớp cơ sở. Cách thức cũng giống như khởi gán cho các đối tượng thành phần, chỉ khác nhau ở chỗ: Để khởi gán cho các đối tượng thành phần ta dùng tên đối tượng thành phần, còn để khởi gán cho các thuộc tính thừa kế từ các lớp cơ sở ta dùng tên lớp cơ sở:

Tên_đối_tượng_thành_phần(danh sách giá trị)

Tên_lớp_cơ_sở(danh sách giá trị)

Danh sách giá trị lấy từ các đối của hàm tạo của lớp dẫn xuất đang xây dựng

(xem ví dụ mục 2.4 và bài 6, ví dụ 1)

2.3. Hàm huỷ

Khi một đối tượng của lớp dẫn xuất được giải phóng (bị huỷ), thì các đối tượng thành phần và các đối tượng thừa kế từ các lớp cơ sở cũng bị giải phóng theo. Do đó các hàm huỷ tương ứng sẽ được gọi đến.

Như vậy khi xây dựng hàm huỷ của lớp dẫn xuất, chúng ta chỉ cần quan tâm đến các thuộc tính (không phải là đối tượng) khai báo thêm trong lớp dẫn xuất mà thôi. Ta không cần để ý đến các đối tượng thành phần và các thuộc tính thừa kế từ các lớp cơ sở. (xem ví dụ mục 2.4 và bài 6, ví dụ 2)

2.4. Ví dụ xét các lớp

+ Lớp NGUOI gồm:

- Các thuộc tính

char *ht ; // Họ tên

int ns ;

- Hai hàm tạo, phương thức in() và hàm huỷ

+ Lớp MON_HOC gồm:

- Các thuộc tính

char *monhoc ; // Tên môn học

int st ; // Số tiết

- Hai hàm tạo, phương thức in() và hàm huỷ

+ Lớp GIAO_VIEN :

- Kế thừa từ lớp NGUOI

- Đưa thêm các thuộc tính

```
char *bomon ;    // Bộ môn công tác
```

```
MON_HOC mh ; // Môn học đang dạy
```

- Hai hàm tạo , phương thức in() và hàm huỷ

Hãy đề ý cách xây dựng các hàm tạo, hàm huỷ của lớp dẫn xuất GIAO_VIEN. Trong lớp GIAO_VIEN có thể gọi tới 2 phương thức in():

```
GIAO_VIEN::in() // Được xây dựng trong lớp GIAO_VIEN
```

```
NGUOI::in()      // Thừa kế từ lớp NGUOI
```

Hãy chú ý cách gọi tới 2 phương thức in() trong chương trình dưới đây.

//CT5-03

// Ham tao cua lop dan suat

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

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

```
#include <string.h>
```

```
class MON_HOC
```

```
{
```

```
private:
```

```
    char *monhoc;
```

```
    int st;
```

```
public:
```

```
    MON_HOC()
```

```
{
```

```
    monhoc=NULL;
```

```
    st=0;
```

```
}
```

```
    MON_HOC(char *monhoc1, int st1)
```

```
{
```

```
    int n = strlen(monhoc1);
```

```
    monhoc = new char[n+1];
```

```
    strcpy(monhoc,monhoc1);
```

```
    st=st1;
```

```
}
```

```
    ~MON_HOC()
```

```
{
```

```
    if (monhoc!=NULL)
```

```
    {
```

```
        delete monhoc;
```

```
        st=0;
```

```
    }
```

```
}
```

```
void in()
```

```
{
```



```
        cout << "\nTen mon: " << monhoc << " so tiet: " << st;
    }
};
class NGUOI
{
private:
    char *ht;
    int ns;
public:
    NGUOI()
    {
        ht=NULL;
        ns=0;
    }
    NGUOI(char *ht1, int ns1)
    {
        int n = strlen(ht1);
        ht = new char[n+1];
        strcpy(ht,ht1);
        ns=ns1;
    }
    ~NGUOI()
    {
        if (ht!=NULL)
        {
            delete ht;
            ns=0;
        }
    }
    void in()
    {
        cout << "\nHo ten : " << ht << " nam sinh: " << ns;
    }
};
class GIAO_VIEN : public NGUOI
{
private:
    char *bomon;
    MON_HOC mh;
public:
    GIAO_VIEN():mh(),NGUOI()//Su dung ham tao khong doi
    {
```

```
        bomon=NULL;
    }
    GIAO_VIEN(char *ht1, int ns1, char *monhoc1,int st1, char *bomon1 ):
    NGUOI(ht1,ns1),mh(monhoc1, st1)
    {
        int n = strlen(bomon1);
        bomon = new char[n+1];
        strcpy(bomon,bomon1);
    }
    ~GIAO_VIEN()
    {
        if (bomon!=NULL)
            delete bomon;
    }
    void in()
    {
        // Su dung phuong thuc in
        NGUOI::in();
        cout << "\n Cong tac tai bo mon: " << bomon;
        mh.in();
    }
};

void main()
{
    clrscr();
    GIAO_VIEN g1; // Goi toi cac ham tao khong doi
    GIAO_VIEN *g2;
    //Goi toi cac ham tao co doi
    g2 = new GIAO_VIEN("PHAM VAN AT", 1945, "CNPM",
                      60, "TIN HOC");

    g2->in();
    /*
    co the viet
    g2->GIAO_VIEN::in();
    */
    g2->NGUOI::in();
    getch();
    delete g2; // Goi toi cac ham huy
    getch();
}
```

Bài 3. Phạm vi truy nhập đến các thành phần của lớp cơ sở

3.1. Các từ khoá quy định phạm vi truy nhập của lớp cơ sở

+ Mặc dù lớp dẫn xuất được thừa kế tất cả các thành phần của lớp cơ sở, nhưng trong lớp dẫn xuất không thể truy nhập tới tất cả các thành phần này. Giải pháp thường dùng là sử dụng các phương thức của lớp cơ sở để truy nhập đến các thuộc tính của chính lớp cơ sở đó. Cũng có thể sử dụng các giải pháp khác dưới đây.

+ Các thành phần private của lớp cơ sở không cho phép truy nhập trong lớp dẫn xuất.

+ Các thành phần public của lớp cơ sở có thể truy nhập bất kỳ chỗ nào trong chương trình. Như vậy trong các lớp dẫn xuất có thể truy nhập được tới các thành phần này.

+ Các thành phần khai báo là protected có phạm vi truy nhập rộng hơn so với các thành phần private, nhưng hẹp hơn so với các thành phần public. Các thành phần protected của một lớp chỉ được mở rộng phạm vi truy nhập cho các lớp dẫn xuất trực tiếp từ lớp này.

3.2. Hai kiểu dẫn xuất

Có 2 kiểu dẫn xuất là private và public, chúng cho các phạm vi truy nhập khác nhau tới lớp cơ sở. Cụ thể như sau:

+ Các thành phần public và protected của lớp cơ sở sẽ trở thành các thành phần public và protected của lớp dẫn xuất theo kiểu public.

+ Các thành phần public và protected của lớp cơ sở sẽ trở thành các thành phần private của lớp dẫn xuất theo kiểu private.

Ví dụ :

Giả sử lớp A có:

thuộc tính public a1

thuộc tính protected a2

và lớp B dẫn xuất public từ A, thì A::a1 trở thành public trong B, A::a2 trở thành protected trong B.

Do đó nếu dùng B làm lớp cơ để xây dựng lớp C. Thì trong C có thể truy nhập tới A::a1 và A::a2.

Thế nhưng nếu sửa đổi để B dẫn xuất private từ A, thì cả A::a1 và A::a2 trở thành private trong B, và khi đó trong C không được phép truy nhập tới các thuộc tính A::a1 và A::a2.

Để biết tường tận hơn, chúng ta hãy biên dịch chương trình:

```
//CT5-04
```

```
// Phạm vi truy nhập
```

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

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

```
#include <string.h>
```

```
class A
```

```
{
```

```
protected:
```

```
int a1;
```

```
public:
```

```
int a2;
```

```
A()
```

```
{
```

```
a1=a2=0;
```

```
}
```

```
A(int t1, int t2)
{
    a1=t1; a2= t2;
}
void in()
{
    cout << a1 <<" " << a2 ;
}
};

class B: private A
{
    protected:
        int b1;
    public:
        int b2;
        B()
        {
            b1=b2=0;
        }
        B(int t1, int t2, int u1, int u2)
        {
            a1=t1; a2=t2; b1=u1;b2=u2;
        }
        void in()
        {
            cout << a1 <<" " << a2 << " " << b1 << " " << b2;
        }
};

class C : public B
{
    public:
        C()
        {
            b1=b2=0;
        }
        C(int t1, int t2, int u1,int u2)
        {
            a1=t1; a2=t2; b1=u1;b2=u2;
        }
}
```

```
void in()
{
    cout << a1;
    cout << " " << a2 << " " << b1 << " " << b2;
}

};

void main()
{
    C c(1,2,3,4);
    c.in();
    getch();
}
```

Chúng ta sẽ nhận được 4 thông báo lỗi sau trong lớp C (tại hàm tạo có đối và phương thức in):

A::a1 is not accessible

A::a2 is not accessible

A::a1 is not accessible

A::a2 is not accessible

Bây giờ nếu sửa đổi để lớp B dẫn xuất public từ A thì chương trình sẽ không có lỗi và làm việc tốt.

Bài 4. Thừa kế nhiều mức và sự trùng tên

4.1. Sơ đồ xây dựng các lớp dẫn xuất theo nhiều mức

Như đã biết:

+ Khi đã định nghĩa một lớp (ví dụ lớp A), ta có thể dùng nó làm cơ sở để xây dựng lớp dẫn xuất (ví dụ B).

+ Đến lượt mình, B có thể dùng làm cơ sở để xây dựng lớp dẫn xuất mới (ví dụ C).

+ Tiếp đó lại có thể dùng C làm cơ sở để xây dựng lớp dẫn xuất mới.

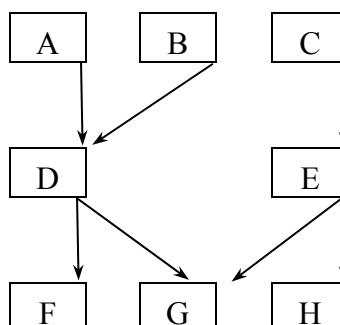
+ Sự tiếp tục theo cách trên là không hạn chế.

Sơ đồ trên chính là sự thừa kế nhiều mức. Ngoài ra chúng ta cũng đã biết:

+ Một lớp có thể được dẫn xuất từ nhiều lớp cơ sở.

+ Một lớp có thể dùng làm cơ sở cho nhiều lớp.

Hình vẽ dưới đây là một ví dụ về sơ đồ thừa kế khá tổng quát, thể hiện được các điều nói trên:



Diễn giải:

Lớp D dẫn xuất từ A và B

Lớp E dẫn xuất từ C

Lớp F dẫn xuất từ D

Lớp G dẫn xuất từ D và E

Lớp H dẫn xuất từ E

4.2. Sự thừa kế nhiều mức.

+ Như đã biết: Lớp dẫn xuất thừa kế tất cả các thành phần (thuộc tính và phương thức) của lớp cơ sở, kể cả các thành phần mà lớp cơ sở được thừa kế.

+ Hãy áp dụng nguyên lý trên để xét lớp G:

- Lớp G thừa kế các thành phần của các lớp D và E
- Lớp D thừa kế các thành phần của lớp A và B
- Lớp E thừa kế các thành phần của lớp C

Như vậy các thành phần có thể sử dụng trong lớp G gồm:

- Các thành phần khai báo trong G (của riêng G)
- Các thành phần khai báo trong các lớp D, E, A, B, C (được thừa kế).

4.3. Sự trùng tên

Như đã nói trong 4.2: Trong lớp G có thể sử dụng (trực tiếp hay gián tiếp) các thành phần của riêng G và các thành phần mà nó được thừa kế từ các lớp D, E, A, B, C. Yêu cầu về cách đặt tên ở đây là:

- + Tên các lớp không được trùng lặp
- + Tên các thành phần trong một lớp cũng không được trùng lặp
- + Tên các thành phần trong các lớp khác nhau có quyền được trùng lặp.

Để phân biệt các thành phần trùng tên trong lớp dẫn xuất, cần sử dụng thêm tên lớp (xem ví dụ trong 4.4).

4.4. Sử dụng các thành phần trong lớp dẫn xuất

Như đã nói ở trên: Thành phần của lớp dẫn xuất gồm:

- + Các thành phần khai báo trong lớp dẫn xuất
- + Các thành phần mà lớp dẫn xuất thừa kế từ các lớp cơ sở

Quy tắc sử dụng các thành phần trong lớp dẫn xuất:

Cách 1: Dùng tên lớp và tên thành phần. Khi đó Chương trình dịch C++ dễ dàng phân biệt thành phần thuộc lớp nào. Ví dụ:

D h; // h là đối tượng của lớp D dẫn xuất từ A và B

h.D::n là thuộc tính n khai báo trong D

h.A::n là thuộc tính n thừa kế từ A (khai báo trong A)

h.D::nhap() là phương thức nhập() định nghĩa trong D

h.A::nhap() là phương thức nhập() định nghĩa trong A

Cách 2: Không dùng tên lớp, chỉ dùng tên thành phần. Khi đó Chương trình dịch C++ phải tự phán đoán để biết thành phần đó thuộc lớp nào. Cách phán đoán như sau: Trước tiên xem thành phần đang xét có trùng tên với một thành phần nào của lớp dẫn xuất không? Nếu trùng thì đó là thành phần của lớp dẫn xuất. Nếu không trùng thì tiếp tục xét các lớp cơ sở theo thứ tự: Các lớp có quan hệ gần với lớp dẫn xuất xét trước, các lớp quan hệ xa xét sau. Hãy chú ý trường hợp sau:

Thành phần đang xét có mặt đồng thời trong 2 lớp cơ sở có cùng một đẳng cấp quan hệ với lớp dẫn xuất. Gặp trường hợp này Chương trình dịch C++ không thể quyết định được thành phần này thừa kế từ lớp nào và buộc phải đưa ra một thông báo lỗi (xem ví dụ dưới đây). Cách khắc phục: Trường hợp này phải sử dụng thêm tên lớp như trình bày trong cách 1.

Ví dụ xét lớp dẫn xuất D. Lớp D có 2 cơ sở là các lớp A và B. Giả sử các lớp A, B và D được định nghĩa:

```
class A
{
    private:
        int n;
        float a[20];
    public:
        void nhap();
        void xuat();
};

class B
{
    private:
        int m,n;
        float a[20][20];
    public:
        void nhap();
        void xuat();
};

class D : public A, public B
{
    private:
        int k;
    public:
        void nhap();
        void xuat();
};
```

Hãy chú ý các điểm sau:

1. Dùng các phương thức của các lớp A, B để xây dựng các phương thức của D

// Xây dựng phương thức nhap()

void D::nhap()

```
{
    cout << "\n Nhập k : “ ;
    cin >> k ; // k là thuộc tính của D
    A::nhap(); // Nhập các thuộc tính mà D thừa kế từ A
```

```
B::nhap(); // Nhập các thuộc tính mà D thừa kế từ B
}
// Xây dựng phương thức xuất()
void D::xuat()
{
    cout << "\n k = " << k ;
    A::xuat(); // Xuất các thuộc tính mà D thừa kế từ A
    B::xuat(); // Xuất các thuộc tính mà D thừa kế từ B
}
```

2. Làm việc với các đối tượng của lớp dẫn xuất

```
D h ; // Khai báo h là đối tượng của lớp D
h.nhap() ; // tương đương với h.D::nhap();
h.A::xuat(); // In giá trị các thuộc tính h.A::n và h.A::a
h.B::xuat(); // In giá trị các thuộc tính h.B::m, h.B::n và h.B::a
h.D::xuat() ; // In giá trị tất cả các thuộc tính của h
h.xuat() ; // tương đương với h.D::xuat() ;
```

Bài 5. Các lớp cơ sở ảo

5.1. Một lớp cơ sở xuất hiện nhiều lần trong lớp dẫn xuất

Một điều hiển nhiên là không thể khai báo 2 lần cùng một lớp trong danh sách của các lớp cơ sở cho một lớp dẫn xuất. Chẳng hạn ví dụ sau là không cho phép:

```
class B : public A, public A
{
};
```

Tuy nhiên vẫn có thể có trường hợp cùng một lớp cơ sở được đề cập nhiều hơn một lần trong các lớp cơ sở trung gian của một lớp dẫn xuất. Ví dụ:

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

```
class A
```

```
{
    public:
        int a;
};
```

```
class B : public A
```

```
{
    public:
        int b;
};
```

```
class C : public A
```

```
{
    public:
        int c;
```



```
};  
class D : public B , public C  
{  
    public:  
        int d;  
};  
void main()  
{  
    D h ;  
    h.d = 4 ; // tốt  
    h.c = 3 ; // tốt  
    h.b = 2 ; // tốt  
    h.a = 1 ; // lỗi  
}
```

Trong ví dụ này A là cơ sở cho cả 2 lớp cơ sở trực tiếp của D là B và C. Nói cách khác có 2 lớp cơ sở A cho lớp D. Vì vậy trong câu lệnh:

```
h.a = 1 ;
```

thì Chương trình dịch C++ không thể nhận biết được thuộc tính a thừa kế thông qua B hay thông qua C và nó sẽ đưa ra thông báo lỗi sau:

Member is ambiguous: 'A::a' and 'A::a'

5.2. Các lớp cơ sở ảo

Giải pháp cho vấn đề nói trên là khai báo A như một lớp cơ sở kiểu virtual cho cả B và C. Khi đó B và C được định nghĩa như sau:

```
class B : virtual public A  
{  
    public:  
        int b;  
};  
class C : virtual public A  
{  
    public:  
        int c;  
};
```

Các lớp cơ sở ảo (virtual) sẽ được kết hợp để tạo một lớp cơ sở duy nhất cho bất kỳ lớp nào dẫn xuất từ chúng. Trong ví dụ trên, hai lớp cơ sở A (A là cơ sở của B và A là cơ sở của C) sẽ kết hợp lại để trở thành một lớp cơ sở A duy nhất cho bất kỳ lớp dẫn xuất nào từ B và C. Như vậy bây giờ D sẽ chỉ có một lớp cơ sở A duy nhất, do đó phép gán:

```
h.a = 1 ;
```

sẽ gán 1 cho thuộc tính a của lớp cơ sở A duy nhất mà D kế thừa.

Bài 6. Một số ví dụ về hàm tạo, hàm hủy trong thừa kế nhiều mức

Ví dụ 1. Ví dụ này minh họa cách xây dựng hàm tạo trong các lớp dẫn xuất. Ngoài ra còn minh họa cách dùng các phương thức của các lớp cơ sở trong lớp dẫn xuất và cách xử lý các đối tượng thành phần.

Xét 4 lớp A, B, C và D. Lớp C dẫn xuất từ B, lớp D dẫn xuất từ C và có thành phần là đối tượng kiểu A.

```
//CT5-06
// Thua ke nhieu muc
// Ham tao
#include <conio.h>
#include <iostream.h>
#include <string.h>
class A
{
private:
    int a;
    char *str ;
public:
    A()
    {
        a=0; str=NULL;
    }
    A(int a1,char *str1)
    {
        a=a1; str=strdup(str1);
    }
    void xuat()
    {
        cout << "\n" << "So nguyen lop A= " << a
            << " Chuoi lop A: " << str ;
    }
};
class B
{
private:
    int b;
    char *str ;
public:
    B()
    {
        b=0; str=NULL;
    }
    B(int b1,char *str1)
    {
        b=b1; str=strdup(str1);
    }
};
```

```
    }
    void xuat()
    {
        cout << "\n" << "So nguyen lop B = " << b
            << " Chuoi lop B: " << str ;
    }
};

class C : public B
{
    private:
        int c;
        char *str ;
    public:
        C():B()
        {
            c=0; str=NULL;
        }
        C(int b1,char *strb,int c1, char *strc) : B(b1,strb)
        {
            c=c1; str=strdup(strc);
        }
        void xuat()
        {
            B::xuat();
            cout << "\n" << "So nguyen lop C = " << c
                << " Chuoi lop C: " << str ;
        }
};

class D : public C
{
    private:
        int d;
        char *str ;
        A u;
    public:
        D():C(),u()
        {
            d=0; str=NULL;
        }
        D(int a1, char *stra,int b1,char *strb,int c1, char *strc,
```

```

        int d1, char *strd) : u(a1,stra), C(b1,strb,c1,src)
    {
        d=d1; str=strdup(strd);
    }
void xuất()
{
    u.xuat();
    C::xuat();
    cout << "\n" << "So nguyen lop D = " << d
        << " Chuoi lop D: " << str ;
}
};
void main()
{
    D h(1,"AA",2,"BB",3,"CC",4,"DD");
    clrscr();
    cout << "\n\n Cac thuoc tinh cua h thua ke B: " ;
    h.B::xuat();
    cout << "\n\n Cac thuoc tinh cua h thua ke B va C: " ;
    h.C::xuat();
    cout << "\n\n Cac thuoc tinh cua h thua ke B,C va khai bao trong D:" ;
    h.xuat();
    getch();
}

```

Ví dụ 2. Ví dụ này minh hoạ cách xây dựng hàm huỷ trong lớp dẫn xuất. Chương trình trong ví dụ này lấy từ chương trình của ví dụ 1, sau đó đưa thêm vào các hàm huỷ.

```

//CT5-07
// Thua ke nhieu muc
// Ham tao
// Ham huy
#include <conio.h>
#include <iostream.h>
#include <string.h>
class A
{
private:
    int a;
    char *str ;
public:
    A()
    {

```

```

        a=0; str=NULL;
    }
    A(int a1,char *str1)
    {
        a=a1; str=strdup(str1);
    }
    ~A()
    {
        cout << "\n Huy A"; getch();
        a=0;
        if (str!=NULL) delete str;
    }
    void xuat()
    {
        cout << "\n" << "So nguyen lop A= " << a
            << " Chuoi lop A: " << str ;
    }
};
class B
{
    private:
        int b;
        char *str ;
    public:
        B()
        {
            b=0; str=NULL;
        }
        B(int b1,char *str1)
        {
            b=b1; str=strdup(str1);
        }
        ~B()
        {
            cout << "\n Huy B"; getch();
            b=0;
            if (str!=NULL) delete str;
        }
        void xuat()
        {
            cout << "\n" << "So nguyen lop B = " << b
                << " Chuoi lop B: " << str ;
        }
};

```

```
class C : public B
{
private:
    int c;
    char *str ;
public:
    C():B()
    {
        c=0; str=NULL;
    }
    C(int b1,char *strb,int c1, char *strc) : B(b1,strb)
    {
        c=c1; str=strdup(strc);
    }
    ~C()
    {
        cout << "\n Huy C"; getch();
        c=0;
        if (str!=NULL) delete str;
    }
    void xuat()
    {
        B::xuat();
        cout << "\n" << "So nguyen lop C = " << c
        << " Chuoi lop C: " << str ;
    }
};

class D : public C
{
private:
    int d;
    char *str ;
    A u;
public:
    D():C(),u()
    {
        d=0; str=NULL;
    }
    D(int a1, char *stra,int b1,char *strb,int c1, char *strc,
        int d1, char *strd) : u(a1,stra), C(b1,strb,c1,strc)
    {
```

```
        d=d1; str=strdup(strd);
    }
    ~D()
    {
        cout << "\n Huy D"; getch();
        d=0;
        if (str!=NULL) delete str;
    }
    void xuat()
    {
        u.xuat();
        C::xuat();
        cout << "\n" << "So nguyen lop D = " << d
              << " Chuoi lop D: " << str ;
    }
};

void main()
{
    D *h;
    h = new D(1,"AA",2,"BB",3,"CC",4,"DD");
    clrscr();
    cout << "\n\n Cac thuoc tinh cua h thua ke B: ";
    h->B::xuat();
    cout << "\n\n Cac thuoc tinh cua h thua ke B va C: ";
    h->C::xuat();
    cout << "\n\n Cac thuoc tinh cua h thua ke B,C va khai bao trong D:";
    h->xuat();
    delete h; // Lan luot gọi tới các hàm hủy của các lớp D, A, C, B
    getch();
}
```

Bài 7. Toán tử gán của lớp dẫn xuất

7.1. Khi nào cần xây dựng toán tử gán: Khi lớp dẫn xuất có các thuộc tính (kể cả thuộc tính thừa kế từ các lớp cơ sở) là con trỏ, thì nhất thiết không được dùng toán tử gán mặc định, mà phải xây dựng cho lớp dẫn xuất một toán tử gán.

7.2. Cách xây dựng toán tử gán cho lớp dẫn xuất

- + Trước hết cần xây dựng toán tử gán cho các lớp cơ sở
- + Vấn đề mấu chốt là: Khi xây dựng toán tử gán cho lớp dẫn xuất thì làm thế nào để sử dụng được các toán tử gán của lớp cơ sở. Cách giải quyết như sau:

- Xây dựng các phương thức (trong các lớp cơ sở) để nhận được địa chỉ của đối tượng ẩn của lớp. Phương thức này được viết đơn giản theo mẫu:

```
Tên_lớp * get_DT ()
{
    return this ;
}
```

- Trong thân của toán tử gán (cho lớp dẫn xuất), sẽ dùng phương thức trên để nhận địa chỉ đối tượng của lớp cơ sở mà lớp dẫn xuất thừa kế. Sau đó thực hiện phép gán trên các đối tượng này.

Phương pháp nêu trên có thể minh họa một cách hình thức như sau: Giả sử lớp B dẫn xuất từ A. Để xây dựng toán tử gán cho B, thì:

1. Trong lớp A cần xây dựng toán tử gán và phương thức cho địa chỉ của đối tượng ẩn. Cụ thể A cần được định nghĩa như sau:

```
class A
{
    ...
    A & operator=(A& h)
    {
        //các câu lệnh để thực hiện gán trong A
    }
    // Phương thức nhận địa chỉ đối tượng ẩn của A
    A* get_A()
    {
        return this;
    }
    ...
};
```

2. Toán tử gán trong lớp B cần như sau:

```
class B : public A
{
    ...
    B & operator=(B& h)
    {
        A *u1, *u2;
        u1 = this->get_A();
        u2 = h.get_A();
        *u1 = *u2 ; // Sử dụng phép gán trong A để gán các
                    // thuộc tính mà B kế thừa từ A
        //Các câu lệnh thực hiện gán các thuộc tính riêng của B
    }
    ...
};
```


7.3. Ví dụ

Chương trình dưới đây minh họa cách xây dựng toán tử gán cho lớp D có 2 lớp cơ sở là C và B (C là lớp cơ sở trực tiếp, còn B là cơ sở của C) . Ngoài ra D còn có một thuộc tính là đối tượng của lớp A.

```
//CT5-08
// Thua ke nhieu muc
// gan
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <string.h>
class A
{
private:
    int a;
    char *str ;
public:
    A()
    {
        a=0; str=NULL;
    }
    A& operator=(A& h)
    {
        this->a = h.a;
        if (this->str!=NULL) delete this->str;
        this->str = strdup(h.str);
        return h;
    }
    void nhap()
    {
        cout << "\nNhap so nguyen lop A: "; cin >> a ;
        if (str!=NULL) delete str;
        cout << "\nNhap chuoi lop A: ";
        char tg[30];
        fflush(stdin); gets(tg);
        str = strdup(tg);
    }
    void xuat()
    {
        cout << "\n" << "So nguyen lop A= " << a
```

```
        << " Chuoi lop A: " << str ;
    }
};

class B
{
private:
    int b;
    char *str ;
public:
    B()
    {
        b=0; str=NULL;
    }
    B* getB()
    {
        return this;
    }
    B& operator=(B& h)
    {
        this->b = h.b;
        if (this->str!=NULL) delete this->str;
        this->str = strdup(h.str);
        return h;
    }
    void nhap()
    {
        cout << "\nNhap so nguyen lop B: "; cin >> b ;
        if (str!=NULL) delete str;
        cout << "\nNhap chuoi lop B: ";
        char tg[30];
        fflush(stdin); gets(tg);
        str = strdup(tg);
    }
    void xuat()
    {
        cout << "\n" << "So nguyen lop B = " << b
            << " Chuoi lop B: " << str ;
    }
};

class C : public B
```

```
{
private:
    int c;
    char *str ;
public:
    C():B()
    {
        c=0; str=NULL;
    }
    C* getC()
    {
        return this;
    }
    C& operator=(C& h)
    {
        B *b1, *b2;
        b1 = this->getB();
        b2 = h.getB();
        *b1 = *b2;
        this->c = h.c;
        if (this->str!=NULL) delete this->str;
        this->str = strdup(h.str);
        return h;
    }
    void nhap()
    {
        B::nhap();
        cout << "\nNhap so nguyen lop C: "; cin >> c ;
        if (str!=NULL) delete str;
        cout << "\nNhap chuoi lop C: ";
        char tg[30];
        fflush(stdin); gets(tg);
        str = strdup(tg);
    }

    void xuat()
    {
        B::xuat();
        cout << "\n" << "So nguyen lop C = " << c
            << " Chuoi lop C: " << str ;
    }
}
```

```

    }
};
class D : public C
{
private:
    int d;
    char *str ;
    A u;
public:
    D():C(),u()
    {
        d=0; str=NULL;
    }
    D& operator=(D& h)
    {
        this->u = h.u;
        C *c1,*c2;
        c1 = this->getC();
        c2 = h.getC();
        *c1 = *c2;
        this->d = h.d;
        if (this->str!=NULL) delete this->str;
        this->str = strdup(h.str);
        return h;
    }
    void nhap()
    {
        u.nhap();
        C::nhap();
        cout << "\nNhap so nguyen lop D: " ; cin >> d ;
        if (str!=NULL) delete str;
        cout << "\nNhap chuoi lop D: " ;
        char tg[30];
        fflush(stdin); gets(tg);
        str = strdup(tg);
    }
    void xuat()
    {
        u.xuat();
        C::xuat();
        cout << "\n" << "So nguyen lop D = " << d

```

```
        << " Chuoi lop D: " << str ;
    }
};

void main()
{
    D h1,h2,h3;
    clrscr();
    h1.nhap();
    h3=h2=h1;
    cout<<"\n\nH2:";
    h2.xuat();
    cout<<"\n\nH3:";
    h3.xuat();
    h1.nhap();
    cout<<"\n\nH2:";
    h2.xuat();
    cout<<"\n\nH3:";
    h3.xuat();
    cout<<"\n\nH1:";
    h1.xuat();
    getch();
}
```

Bài 8. Hàm tạo sao chép của lớp dẫn xuất

8.1. Khi nào cần xây dựng hàm tạo sao chép: Khi lớp dẫn xuất có các thuộc tính (kể cả thuộc tính thừa kế từ các lớp cơ sở) là con trỏ, thì nhất thiết không được dùng hàm tạo sao chép mặc định, mà phải xây dựng cho lớp dẫn xuất một hàm tạo sao chép.

8.2. Cách xây dựng hàm tạo sao chép cho lớp dẫn xuất

- + Trước hết cần xây dựng toán tử gán cho lớp dẫn xuất (xem §7).
- + Sau đó xây dựng hàm tạo sao chép cho lớp dẫn xuất theo mẫu:

Tên_lớp_dẫn_xuất (Tên_lớp_dẫn_xuất &h)

```
{
    *this = h ;
}
```

8.3. Ví dụ

Chương trình dưới đây minh họa cách xây dựng hàm tạo sao chép cho lớp D có 2 lớp cơ sở là C và B (C là lớp cơ sở trực tiếp, còn B là cơ sở của C) . Ngoài ra D còn có một thuộc tính là đối tượng của lớp A. Chương trình này dựa trên chương trình trong mục 7.3 với 2 thay đổi:

- + Xây dựng thêm hàm tạo sao chép cho lớp D.

+ Thay đổi một số câu lệnh trong hàm main để sử dụng hàm tạo sao chép.

Để thấy rõ vai trò của hàm tạo sao chép chúng ta hãy so sánh kết quả nhận được trong 2 trường hợp: Có hàm tạo sao chép và bỏ đi hàm này.

//CT5-09

// Thua ke nhieu muc

// Ham tao sao chep

#include <conio.h>

#include <stdio.h>

#include <iostream.h>

#include <string.h>

class A

{

private:

int a;

*char *str ;*

public:

A()

{

a=0; str=NULL;

}

A& operator=(A& h)

{

this->a = h.a;

if (this->str!=NULL) delete this->str;

this->str = strdup(h.str);

return h;

}

void nhap()

{

cout << "\nNhap so nguyen lop A: "; cin >> a ;

if (str!=NULL) delete str;

cout << "\nNhap chuoi lop A: ";

char tg[30];

fflush(stdin); gets(tg);

str = strdup(tg);

}

void xuat()

{

cout << "\n" << "So nguyen lop A= " << a

<< " Chuoi lop A: " << str ;

}

```
};  
class B  
{  
    private:  
        int b;  
        char *str ;  
    public:  
        B()  
        {  
            b=0; str=NULL;  
        }  
        B* getB()  
        {  
            return this;  
        }  
        B& operator=(B& h)  
        {  
            this->b = h.b;  
            if (this->str!=NULL) delete this->str;  
            this->str = strdup(h.str);  
            return h;  
        }  
        void nhap()  
        {  
            cout << "\nNhap so nguyen lop B: "; cin >> b ;  
            if (str!=NULL) delete str;  
            cout << "\nNhap chuoi lop B: ";  
            char tg[30];  
            fflush(stdin); gets(tg);  
            str = strdup(tg);  
        }  
        void xuat()  
        {  
            cout << "\n" << "So nguyen lop B = " << b  
                << " Chuoi lop B: " << str ;  
        }  
};  
class C : public B  
{  
    private:
```

```
int c;
char *str ;
public:
    C():B()
    {
        c=0; str=NULL;
    }
    C* getC()
    {
        return this;
    }
    C& operator=(C& h)
    {
        B *b1, *b2;
        b1= this->getB();
        b2= h.getB();
        *b1 = *b2;
        this->c = h.c;
        if (this->str!=NULL) delete this->str;
        this->str = strdup(h.str);
        return h;
    }
    void nhap()
    {
        B::nhap();
        cout << "\nNhap so nguyen lop C: "; cin >> c ;
        if (str!=NULL) delete str;
        cout << "\nNhap chuoi lop C: ";
        char tg[30];
        fflush(stdin); gets(tg);
        str = strdup(tg);
    }

    void xuat()
    {
        B::xuat();
        cout << "\n" << "So nguyen lop C = " << c
            << " Chuoi lop C: " << str ;
    }
};
```



```
class D : public C
{
    private:
        int d;
        char *str ;
        A u;
    public:
        D():C(),u()
        {
            d=0; str=NULL;
        }
        D(D& h) // Ham tao sao chep
        {
            *this=h;
        }
        D& operator=(D& h)
        {
            this->u = h.u;
            C *c1,*c2;
            c1 = this->getC();
            c2 = h.getC();
            *c1 = *c2;
            this->d = h.d;
            if (this->str!=NULL) delete this->str;
            this->str = strdup(h.str);
            return h;
        }
        void nhap()
        {
            u.nhap();
            C::nhap();
            cout << "\nNhap so nguyen lop D: "; cin >> d ;
            if (str!=NULL) delete str;
            cout << "\nNhap chuoi lop D: ";
            char tg[30];
            fflush(stdin); gets(tg);
            str = strdup(tg);
        }
        void xuat()
        {
```

```
        u.xuat();
        C::xuat();
        cout << "\n" << "So nguyen lop D = " << d
                << " Chuoi lop D: " << str ;
    }
};

void main()
{
    D h1;
    clrscr();
    h1.nhap();
    D h2(h1);
    cout<<"\n\nH2:";
    h2.xuat();
    h1.nhap();
    cout<<"\n\nH2:";
    h2.xuat();
    cout<<"\n\nH1:";
    h1.xuat();
    getch();
}
```

Bài 9. Phát triển, hoàn thiện chương trình

Có thể dùng tính thừa kế để phát triển khả năng của chương trình.

9.1. ý tưởng của việc phát triển chương trình như sau: Sau khi đã xây dựng được một lớp, ta sẽ phát triển lớp này bằng cách xây một lớp dân xuất trong đó đưa thêm các thuộc tính và phương thức mới. Quá trình trên lại tiếp tục với lớp vừa nhận được. Ta cũng có thể xây dựng các lớp mới có thuộc tính là đối tượng của các lớp cũ. Bằng cách này, sẽ nhận được một dãy các lớp càng ngày càng hoàn thiện và có nhiều khả năng hơn.

9.2. Ví dụ về việc phát triển chương trình

Giả sử cần xây dựng chương trình vẽ một số hình phẳng. Chúng ta có thể phát triển chương trình này như sau:

Đầu tiên định nghĩa lớp DIEM (Điểm) gồm 2 thuộc tính x, y. Từ lớp DIEM xây dựng lớp DUONG_TRON (Đường tròn) bằng cách bổ sung 2 biến nguyên là r (bán kính) và md (mẫu đường). Từ lớp DUONG_TRON xây dựng lớp HINH_TRON bằng cách thêm vào biến nguyên mt (mẫu tô). Đi theo một nhánh khác: Xây dựng lớp DOAN_THANG (Đoạn thẳng) gồm 2 đối tượng kiểu DIEM, và lớp TAM_GIAC gồm 3 đối tượng DIEM.

Chương trình dưới đây cho phép vẽ các đường tròn, hình tròn, đoạn thẳng và hình tam giác.

Chương trình còn minh họa cách dùng con trỏ **this** trong lớp dẫn xuất để thực hiện các phương thức của lớp cơ sở. Ngoài ra còn minh họa cách dùng toán tử chỉ số [] để nhận các tọa độ x, y từ các đối tượng của lớp DIEM.

//CT5-10

```
// Phát triển chương trình
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <graphics.h>
class DIEM
{
private:
    int x,y;
public:
    DIEM()
    {
        x=y=0;
    }
    DIEM(int x1, int y1)
    {
        x=x1; y=y1;
    }
    DIEM(DIEM &d)
    {
        this->x= d.x;
        this->y= d.y;
    }
    int operator[](int i)
    {
        if (i==1) return x;
        else return y;
    }
};
class DUONG_TRON : public DIEM
{
private:
    int r,md;
public:
    DUONG_TRON() : DIEM()
    {
        r=md=0;
    }
    DUONG_TRON(DIEM d, int r1, int md1) : DIEM(d)
    {
```

```
        r=r1; md=md1;
    }
    void ve()
    {
        setcolor(md);
        circle ( (*this)[1],(*this)[2],r);
    }
    int getmd()
    {
        return md;
    }
};

class HINH_TRON : public DUONG_TRON
{
private:
    int mt;
public:
    HINH_TRON() : DUONG_TRON()
    {
        mt=0;
    }
    HINH_TRON(DIEM d, int r1, int md1, int mt1) :
        DUONG_TRON(d,r1,md1)
    {
        mt=mt1;
    }
    void ve()
    {
        DUONG_TRON::ve();
        setfillstyle(1,mt);
        floodfill((*this)[1],(*this)[2],this->getmd());
    }
};

class DOAN_THANG
{
private:
    DIEM d1, d2;
    int md;
public:
    DOAN_THANG() : d1(), d2()
    {
        md=0;
    }
};
```

```
    }
    DOAN_THANG(DIEM t1, DIEM t2, int md1)
    {
        d1=t1; d2 = t2; md=md1;
    }
    void ve()
    {
        setcolor(md);
        line(d1[1],d1[2] ,d2[1],d2[2]);
    }
};

class TAM_GIAC
{
private:
    DIEM d1,d2,d3;
    int md, mt;
public:
    TAM_GIAC(): d1(), d2(), d3()
    {
        md=mt=0;
    }
    TAM_GIAC(DIEM t1,DIEM t2,DIEM t3,int md1,int mt1)
    {
        d1=t1; d2=t2; d3 = t3; md=md1;mt=mt1;
    }
    void ve()
    {
        DOAN_THANG(d1,d2,md).ve();
        DOAN_THANG(d1,d3,md).ve();
        DOAN_THANG(d2,d3,md).ve();
        setfillstyle(1,mt);
        floodfill((d1[1]+d2[1]+d3[1])/3,(d1[2]+d2[2]+d3[2])/3,md);
    }
};

void ktdh()
{
    int mh=0,mode=0;
    initgraph(&mh,&mode,"");
}

void main()
```

```
{
    ktdh();
    DUONG_TRON dt(DIEM(100,100),80,MAGENTA);
    HINH_TRON ht(DIEM(400,100),80,RED,YELLOW);
    DOAN_THANG t(DIEM(100,100),DIEM(400,100),BLUE);
    TAM_GIAC tg(DIEM(250,150), DIEM(100,400),
                DIEM(400,400), CYAN, CYAN);

    dt.ve();
    ht.ve();
    t.ve();
    tg.ve();
    getch();
    closegraph();
}
```

Bài 10. Bổ sung, nâng cấp chương trình

Có thể dùng tính thừa kế để sửa đổi, bổ sung, nâng cấp chương trình.

10.1. ý tưởng của việc nâng cấp chương trình như sau: Giả sử đã có một chương trình hoạt động tốt. Bây giờ cần có một số bổ sung thay đổi không nhiều lắm. Có 2 giải pháp: Hoặc sửa chữa các lớp đang hoạt động ổn định, hoặc xây dựng một lớp dẫn xuất để thực hiện các bổ sung, sửa chữa trên lớp này. Rõ ràng giải pháp thứ 2 tỏ ra hợp lý hơn .

10.2. Ví dụ

Giả sử đã có chương trình quản lý giáo viên gồm 3 lớp MON_HOC (Môn học), GV (Giáo viên) và BM (Bộ môn) :

```
class MON_HOC
{
    private:
        char tenmh[20] ; // Tên môn học
        int  sotiet    ; // Số tiết
    public:
        MON_HOC() ; // Hàm tạo
        const MON_HOC& operator=(const MON_HOC& m) ;
                                   // Gán

        void nhap() ; // Nhập
        void xuat() ; // Xuất
};
class GV
{
```

```
private:
    char ht[25]; // Ho ten
    int ns;    // Nam sinh
    int sm;    // So mon hoc co the day
    MON_HOC *mh ; //Danh sach cac mon hoc
public:
    GV() ; // Hàm tạo
    ~GV() ; //Hàm huỷ
    int getsn() ; // Cho biết số môn (dùng trong BM::sapxep)
    const GV& operator=(const GV& g); // Gán (dùng trong
                                   // BM::sapxep)

    void nhap(); // Nhập
    void xuat(); // Xuất
};

class BM // Bo mon
{
private:
    char tenbm[20]; // Tên bộ môn
    int n; // So giao vien
    GV *gv; // Danh sach giao vien
public:
    BM() // Hàm tạo
    void nhap(); // Nhập
    void xuat(); // Xuất
    void sapxep(); // Sắp xếp
};
```

Chương trình cho phép:

1. Nhập danh sách giáo viên của bộ môn.
2. Sắp xếp danh sách giáo viên theo thứ tự giảm của số môn mà mỗi giáo viên có thể giảng dạy.
3. In danh sách giáo viên.

Nội dung chương trình như sau:

```
//CT5-11
// Nang cap chuong trinh
// CT ban dau
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <string.h>
class MON_HOC
```

```
{
private:
    char tenmh[20];
    int sotiet;
public:
    MON_HOC()
    {
        tenmh[0]=sotiet=0;
    }
    const MON_HOC& operator=(const MON_HOC& m)
    {
        strcpy(this->tenmh,m.tenmh);
        this->sotiet = m.sotiet;
        return m;
    }
    void nhap()
    {
        cout << "\nTen mon hoc:";
        fflush(stdin); gets(tenmh);
        cout << "So tiet: ";
        cin >> sotiet;
    }
    void xuat()
    {
        cout << "\nTen mon hoc:" << tenmh
                << " so tiet: " << sotiet;
    }
};

class GV
{
private:
    char ht[25]; // Ho ten
    int ns;     // Nam sinh
    int sm;     // So mon hoc co the day
    MON_HOC *mh ; //Danh sach cac mon hoc
public:
    GV()
    {
        ht[0]= ns= sm= 0 ;
        mh = NULL;
    }
};
```



```
~GV()
{
    ht[0]= ns= sm= 0 ;
    if (mh) delete mh;
}
int getsm()
{
    return sm;
}
const GV& operator=(const GV& g);
void nhap();
void xuat();
};
const GV& GV::operator=(const GV& g)
{
    strcpy(this->ht,g.ht);
    this->ns=g.ns;
    int n = g.sm;
    this->sm = n;
    if (this->mh) delete this->mh;
    if (n)
    {
        this->mh = new MON_HOC[n+1];
        for (int i=1; i<=n; ++i)
            this->mh[i] = g.mh[i];
    }
    return g;
}
void GV::nhap()
{
    cout << "\nHo ten: ";
    fflush(stdin); gets(ht);
    cout << "Nam sinh: ";
    cin >> ns;
    cout << "So mon co the giang day: ";
    cin >> sm;
    if (this->mh) delete this->mh;
    if(sm)
    {
        this->mh = new MON_HOC[sm+1];
    }
}
```

```
        for (int i=1; i<=sm; ++i)
            this->mh[i].nhap();
    }
}

void GV::xuat()
{
    cout << "\nHo ten: " << ht ;
    cout << "\nNam sinh: " << ns ;
    cout << "\nSo mon co the giang day: " << sm;
    if (sm)
    {
        cout << "\nDo la: ";
        for (int i=1; i<=sm; ++i)
            this->mh[i].xuat();
    }
}

class BM // Bo mon
{
private:
    char tenbm[20];
    int n; // So giao vien
    GV *gv; // Danh sach giao vien
public:
    BM()
    {
        tenbm[0] = n = 0;
        gv = NULL;
    }
    void nhap();
    void xuat();
    void sapxep();
};

void BM::nhap()
{
    cout << "\n\nTen bo mon: ";
    fflush(stdin); gets(tenbm);
    cout << "So giao vien: ";
    cin >> n;
    if (gv) delete gv;
    if (n)
    {
```

```
        gv = new GV[n+1];
        for (int i=1; i<=n; ++i)
            gv[i].nhap();
    }
}

void BM::xuat()
{
    cout << "\nBo mon: " << tenbm;
    cout << "\nSo giao vien: " << n;
    if (n)
    {
        cout << "\n Danh sach giao vien cua bo mon:";
        for (int i=1; i<=n; ++i)
            gv[i].xuat();
    }
}

void BM::sapxep()
{
    GV tg;
    int i,j;
    if (n)
        for (i=1; i<=n; ++i)
            for (j=i+1; j<=n; ++j)
                if (gv[i].getsm() < gv[j].getsm())
                {
                    tg=gv[i]; gv[i]=gv[j]; gv[j]=tg;
                }
}

void main()
{
    BM b;
    b.nhap();
    b.sapxep();
    b.xuat();
    getch();
}
```

Vấn đề đặt ra là: Hiện nay các giáo viên đã bắt đầu hướng dẫn luận văn tốt nghiệp cho sinh viên. Vì vậy cần bổ sung thông tin này vào phần dữ liệu giáo viên. Để nâng cấp chương trình chúng ta sẽ định nghĩa lớp GV2 dẫn xuất từ lớp GV, sau đó trong lớp BM sẽ thay GV bằng GV2. Có 2 chỗ cần bổ sung và một chỗ cần sửa đổi như sau:

1. Bổ sung trong lớp GV phương thức:

GV* getGV()

```
{  
    return this;  
}
```

Phương thức này dùng để xây dựng toán tử gán cho lớp GV2.

2. Trong lớp BM thay GV bằng GV2. Điều này có thể đạt được bằng sửa chữa trực tiếp hoặc bằng một câu lệnh #define :

```
#define GV GV2
```

3. Định nghĩa thêm 2 lớp: LV (Luận văn) và GV2 (Lớp GV2 dẫn xuất từ lớp GV) như sau:

```
class LV // Luan van  
{  
    private:  
        char tenlv[30]; // Ten luan van  
        char tensv[25]; // Ten sinh vien  
        int nambv;      // Nam bao ve luan van  
    public:  
        LV(); // Hàm tạo  
        const LV& operator=(const LV& l); // Gán  
        void nhap(); // Nhập  
        void xuat();  
};  
class GV2 : public GV  
{  
    private:  
        int solv; // Số luận văn đã hướng dẫn  
        LV *lv; // Danh sách luận văn  
    public:  
        GV2(); // Hàm tạo  
        ~GV2(); // Hàm huỷ  
        GV2& operator=(GV2& g); // Gán  
        void nhap(); // Nhập  
        void xuat(); // Xuất  
};
```

Chương trình nâng cấp như sau:

```
//CT5-12B  
// Nang cap chuong trinh  
// CT nang cap  
#include <conio.h>  
#include <stdio.h>  
#include <iostream.h>  
#include <string.h>  
class MON_HOC
```

```
{
    private:
        char tenmh[20];
        int sotiet;
    public:
        MON_HOC()
        {
            tenmh[0]=sotiet=0;
        }
        const MON_HOC& operator=(const MON_HOC& m)
        {
            strcpy(this->tenmh,m.tenmh);
            this->sotiet = m.sotiet;
            return m;
        }
        void nhap()
        {
            cout << "\nTen mon hoc:";
            fflush(stdin); gets(tenmh);
            cout << "So tiet: ";
            cin >> sotiet;
        }
        void xuat()
        {
            cout << "\nTen mon hoc:" << tenmh
                << " so tiet: " << sotiet;
        }
};

// Bo sung phuong thuc getGV cho lop GV
// dung de xay dung toan tu gan cho lop GV2
class GV
{
    private:
        char ht[25]; // Ho ten
        int ns;    // Nam sinh
        int sm;    // So mon hoc co the day
        MON_HOC *mh ; //Danh sach cac mon hoc
    public:
        GV()
        {
            ht[0]= ns= sm= 0 ;
        }
};
```

```

        mh = NULL;
    }
    ~GV()
    {
        ht[0]= ns= sm= 0 ;
        if (mh) delete mh;
    }
    // Bo sung phuong thuc getGV
    GV* getGV()
    {
        return this;
    }
    int getsm()
    {
        return sm;
    }
    const GV& operator=(const GV& g);
    void nhap();
    void xuat();
};

const GV& GV::operator=(const GV& g)
{
    strcpy(this->ht,g.ht);
    this->ns=g.ns;
    int n = g.sm;
    this->sm = n;
    if (this->mh) delete this->mh;
    if (n)
    {
        this->mh = new MON_HOC[n+1];
        for (int i=1; i<=n; ++i)
            this->mh[i] = g.mh[i];
    }
    return g;
}

void GV::nhap()
{
    cout << "\nHo ten: ";
    fflush(stdin); gets(ht);
    cout << "Nam sinh: ";

```

```
cin >> ns;
cout << "So mon co the giang day: " ;
cin >> sm;
if (this->mh) delete this->mh;
if (sm)
{
    this->mh = new MON_HOC[sm+1];
    for (int i=1; i<=sm; ++i)
        this->mh[i].nhap();
}
}
void GV::xuat()
{
    cout << "\nHo ten: " << ht ;
    cout << "\nNam sinh: " << ns ;
    cout << "\nSo mon co the giang day: " << sm;
    if (sm)
    {
        cout << "\nDo la: ";
        for (int i=1; i<=sm; ++i)
            this->mh[i].xuat();
    }
}
```

// Bo sung cac lop LV va GV2

class LV // Luan van

```
{
    private:
        char tenlv[30]; // Ten luan van
        char tensv[25]; // Ten sinh vien
        int nambv;      // Nam bao ve luan van
    public:
        LV()
        {
            tenlv[0]=tensv[0] = nambv = 0;
        }
        const LV& operator=(const LV& l)
        {
            strcpy(this->tenlv,l.tenlv);
            strcpy(this->tensv,l.tensv);
        }
}
```

```
        this->nambv = l.nambv ;
        return l;
    }
    void nhap()
    {
        cout << "\nTen luan van:";
        fflush(stdin); gets(tenlv);
        cout << "Ten sinh vien:";
        fflush(stdin); gets(tensv);
        cout << "Nam bao ve: " ;
        cin >> nambv ;
    }
    void xuat()
    {
        cout << "\nTen lan van:" << tenlv
        << "   Sinh vien: " << tensv
        << "   Nam bao ve: " << nambv;
    }
};

class GV2 : public GV
{
private:
    int solv;
    LV *lv;
public:
    GV2():GV()
    {
        solv = 0 ;
        lv = NULL;
    }
    ~GV2()
    {
        if (solv) delete lv;
    }
    GV2& operator=(GV2& g);
    void nhap();
    void xuat();
};

GV2& GV2::operator=(GV2& g)
{
```



```
GV *g1, *g2;
g1 = this->getGV();
g2 = g.getGV();
*g1 = *g2;
int n = g.solv;
this->solv = n;
if (this->lv) delete this->lv;
if (n)
{
    this->lv = new LV[n+1];
    for (int i=1; i<=n; ++i)
        this->lv[i] = g.lv[i];
}
return g;
}

void GV2::nhap()
{
    GV::nhap();
    cout << "So luan van da huong dan: " ;
    cin >> solv;
    if (this->lv) delete this->lv;
    if (solv)
    {
        this->lv = new LV[solv+1];
        for (int i=1; i<=solv; ++i)
            this->lv[i].nhap();
    }
}

void GV2::xuat()
{
    GV::xuat();
    cout << "\nSo luan van da huong dan: " << solv;
    if (solv)
    {
        cout << "\n Do la: ";
        for (int i=1; i<=solv; ++i)
            this->lv[i].xuat();
    }
}
```

```
// Sua lop BM: thay GV bang GV2
#define GV GV2
class BM // Bo mon
{
    private:
        char tenbm[20];
        int n; // So giao vien
        GV *gv; // Danh sach giao vien
    public:
        BM()
        {
            tenbm[0] = n = 0;
            gv = NULL;
        }
        void nhap();
        void xuat();
        void sapxep();
    };
void BM::nhap()
{
    cout << "\n\nTen bo mon: ";
    fflush(stdin); gets(tenbm);
    cout << "So giao vien: ";
    cin >> n;
    if (gv) delete gv;
    if (n)
    {
        gv = new GV[n+1];
        for (int i=1; i<=n; ++i)
            gv[i].nhap();
    }
}
void BM::xuat()
{
    cout << "\nBo mon: " << tenbm;
    cout << "\nSo giao vien: " << n;
    if (n)
    {
        cout << "\nDanh sach giao vien cua bo mon:";
        for (int i=1; i<=n; ++i)
            gv[i].xuat();
    }
}
```

```
    }
}
void BM::sapxep()
{
    GV tg;
    int i,j;
    if (n)
        for (i=1;i<n;++i)
            for (j=i+1;j<=n;++j)
                if (gv[i].getsm()<gv[j].getsm())
                {
                    tg=gv[i]; gv[i]=gv[j]; gv[j]=tg;
                }
}
#undef GV
void main()
{
    BM b;
    b.nhap();
    b.sapxep();
    b.xuat();
    getch();
}
```

Bài 11. Từ khái quát đến cụ thể

Tính thừa kế cũng thường dùng để thiết kế các bài toán theo hướng từ khái quát đến cụ thể, từ chung đến riêng. Đầu tiên đưa ra các lớp để mô tả những đối tượng chung, sau đó dẫn xuất tới các đối tượng ngày một cụ thể hơn.

Một trường hợp khác cũng thường gặp là: Quản lý nhiều thực thể có những phần dữ liệu chung. Khi đó ta có thể xây dựng một lớp cơ sở gồm các phần dữ liệu chung. Mỗi thực thể sẽ được mô tả bằng một lớp dẫn xuất từ lớp cơ sở này.

Sau đây là một số ví dụ minh họa:

Ví dụ 1 (minh họa tư tưởng đi từ khái quát đến cụ thể) : Giả sử cần quản lý sinh viên của một trường đại học. Khi đó ta có thể bắt đầu từ lớp SINH_VIEN (Sinh viên). Sau đó dùng nó làm cơ sở để dẫn xuất tới các lớp mô tả các đối tượng sinh viên cụ thể hơn, ví dụ: SV Tin, SV Toán, SV Luật, SV Du lịch, ...

Các bài toán kiểu như vậy rất thường gặp trong thực tế.

Ví dụ 2 (minh họa phần chung của nhiều thực thể). Giả sử cần xây dựng phần mềm để thực hiện các phép tính về ma trận vuông và véc tơ cấp n . Ta có nhận xét là n chung cho cả véc tơ và ma trận. Hơn nữa nó còn chung cho tất cả các ma trận và véc tơ cùng xét trong bài toán. Vì vậy có thể định

nghĩa một lớp cơ sở chỉ có một thuộc tính tĩnh (static) n. Các lớp ma trận, véc tơ dẫn xuất từ lớp này và sử dụng chung cùng một giá trị n.

Dưới đây là chương trình thực hiện các phép toán ma trận, véc tơ. Chương trình được tổ chức thành 3 lớp:

Lớp CAP (Cấp ma trận, véc tơ) gồm một thành phần tĩnh n và phương thức nhập n.

Lớp VT (Véc tơ) có một thuộc tính là mảng một chiều (chứa các phần tử của véc tơ) và các phương thức nhập, xuất.

Lớp MT (Ma trận) có một thuộc tính là mảng 2 chiều (chứa các phần tử của ma trận) , các phương thức nhập, xuất và nhân. Lớp MT là bạn của lớp VT.

Chương trình sẽ nhập một ma trận, nhập một véc tơ và tính tích của chúng.

//CT5-13

// ma tran vec to

// Dùng thuộc tính static

#include <conio.h>

#include <stdio.h>

#include <iostream.h>

#include <ctype.h>

class CAP;

class MT;

class VT;

class CAP

{

private:

static int n;

public:

void nhap()

{

int ch;

if (n==0)

{

cout << "\nN= "; cin >> n;

}

else

{

cout << "\n Hien n = " << n;

cout << "\n Co thay doi n? - C/K";

ch=toupper(getch());

if (ch=='C')

{

cout << "\nN= "; cin >> n;

}

}

```
    }
    int getN()
    {
        return n;
    }
};
int CAP::n=0;
class MT : public CAP
{
    private:
        double a[20][20];
    public:
        void nhap();
        void xuat();
        VT operator*(VT x);
};
class VT : public CAP
{
    private:
        double x[20];
    public:
        friend class MT;
        void nhap();
        void xuat();
};
void MT::nhap()
{
    int n,i,j;
    n = this->getN();
    if (n==0)
    {
        this->CAP::nhap();
        n = this->getN();
    }
    for (i=1; i<=n; ++i)
        for (j=1; j<=n; ++j)
        {
            cout << " PT hang " << i << " cot " << j << " = ";
            cin >> a[i][j];
        }
}
```

```
void MT::xuat()
{
    int n,i,j;
    n = this->getN();
    if (n)
        for (int i=1; i<=n; ++i)
        {
            cout << "\n";
            for (int j=1; j<=n; ++j)
                cout << a[i][j] << " ";
        }
}

VT MT::operator*(VT x)
{
    VT y;
    int n,i,j;
    n = this->getN();
    for (i=1; i<=n; ++i)
    {
        y.x[i]=0;
        for (j=1; j<=n; ++j)
            y.x[i] += a[i][j]*x.x[j];
    }
    return y;
}

void VT::nhap()
{
    int n,i;
    n = this->getN();
    if (n==0)
    {
        this->CAP::nhap();
        n = this->getN();
    }
    for (i=1; i<=n; ++i)
    {
        cout << " PT thu " << i << " = ";
        cin >> x[i];
    }
}

void VT::xuat()
```

```
{
    int n,i;
    n = this->getN();
    if (n)
    {
        cout << "n";
        for (int i=1; i<=n; ++i)
        {
            cout << x[i] << " ";
        }
    }
}

void main()
{
    MT a; VT x,y;
    clrscr();
    cout<<"\nNhap ma tran A:";
    a.nhap();
    cout<<"\n\nNhap Vec to X:\n";
    x.nhap();
    y = a*x;
    cout<<"\n\nMa tran A";
    a.xuat();
    cout<<"\n\nVec to X";
    x.xuat();
    cout<<"\n\nVec to Y=AX";
    y.xuat();
    getch();
}
```

Bài 12. Toàn thể và bộ phận

Thông thường khi xem xét, giải quyết một bài toán, ta thường chia nó thành các bài toán nhỏ hơn. Nói cách khác: Một bài toán lớn bao gồm nhiều bài toán bộ phận. Khi đó ta có thể định nghĩa các lớp cho các bài toán bộ phận. Lớp cho bài toán chung được dẫn xuất từ các lớp nói trên.

Xét một thí dụ đơn giản là bài toán quản lý thư viện. Nó gồm 2 bài toán bộ phận là quản lý sách và quản lý độc giả. Chúng ta sẽ xây dựng lớp SACH và lớp DOC_GIA. Sau đó dùng các lớp này làm cơ sở để xây dựng lớp THU_VIEN

Chương 6

Tương ứng bội và phương thức ảo

Tương ứng bội và phương thức ảo là công cụ mạnh của C++ cho phép tổ chức quản lý các đối tượng khác nhau theo cùng một lược đồ. Một khái niệm khác liên quan là: lớp cơ sở trừu tượng. Chương này sẽ trình bày cách sử dụng các công cụ trên để xây dựng chương trình quản lý nhiều đối tượng khác nhau theo một lược đồ thống nhất.

Bài 1. Phương thức tĩnh

1.1. Lời gọi tới phương thức tĩnh

Như đã biết một lớp dẫn xuất được thừa kế các phương thức của các lớp cơ sở tiền bối của nó. Ví dụ lớp A là cơ sở của B, lớp B lại là cơ sở của C, thì C có 2 lớp cơ sở tiền bối là B và A. Lớp C được thừa kế các phương thức của A và B. Các phương thức mà chúng ta vẫn nói là các phương thức tĩnh. Để tìm hiểu thêm về cách gọi tới các phương thức tĩnh, ta xét ví dụ về các lớp A, B và C như sau:

```
class A
{
    public:
        void xuất()
        {
            cout << "\n Lop A ";
        }
};
class B:public A
{
    public:
        void xuất()
        {
            cout << "\n Lop B ";
        }
};
class C:public B
{
    public:
        void xuất()
        {
            cout << "\n Lop C ";
        }
};
```

Lớp C có 2 lớp cơ sở tiền bối là A, B và C kế thừa các phương thức của A và B. Do đó một đối tượng của C sẽ có tới 3 phương thức xuất. Hãy theo dõi các câu lệnh sau:

C h; // h là đối tượng kiểu C

h.xuat(); // Gọi tới phương thức h.D::xuat()


```
h.B::xuat() ; // Gọi tới phương thức h.B::xuat()
```

```
h.A::xuat() ; // Gọi tới phương thức h.A::xuat()
```

Các lời gọi phương thức trong ví dụ trên đều xuất phát từ đối tượng h và mọi lời gọi đều xác định rõ phương thức cần gọi.

Bây giờ chúng ta hãy xét các lời gọi không phải từ một biến đối tượng mà từ một con trỏ. Xét các câu lệnh:

```
A *p, *q, *r; // p, q, r là con trỏ kiểu A
```

```
A a; // a là đối tượng kiểu A
```

```
B b; // b là đối tượng kiểu B
```

```
C c; // c là đối tượng kiểu c
```

Chúng ta hãy ghi nhớ mệnh đề sau về con trỏ của các lớp dẫn xuất và cơ sở:

Phép gán con trỏ: Con trỏ của lớp cơ sở có thể dùng để chứa địa chỉ các đối tượng của lớp dẫn xuất.

Như vậy cả 3 phép gán sau đều hợp lệ:

```
p = &a ;
```

```
q = &b ;
```

```
r = &c ;
```

Chúng ta tiếp tục xét các lời gọi phương thức từ các con trỏ p, q, r:

```
p->xuat();
```

```
q->xuat();
```

```
r->xuat();
```

và hãy lý giải xem phương thức nào (trong các phương thức A::xuat, B::xuat và C::xuat) được gọi. Câu trả lời như sau:

Cả 3 câu lệnh trên đều gọi tới phương thức A::xuat() , vì các con trỏ p, q và r đều có kiểu A.

Như vậy có thể tóm lược cách thức gọi các phương thức tĩnh như sau:

Quy tắc gọi phương thức tĩnh: Lời gọi tới phương thức tĩnh bao giờ cũng xác định rõ phương thức nào (trong số các phương thức trùng tên của các lớp có quan hệ thừa kế) được gọi:

1. Nếu lời gọi xuất phát từ một đối tượng của lớp nào, thì phương thức của lớp đó sẽ được gọi.
2. Nếu lời gọi xuất phát từ một con trỏ kiểu lớp nào, thì phương thức của lớp đó sẽ được gọi bất kể con trỏ chứa địa chỉ của đối tượng nào.

1.2. Ví dụ

Xét 4 lớp A, B, C và D. Lớp B và C có chung lớp cơ sở A. Lớp D dẫn xuất từ C. Cả 4 lớp đều có phương thức xuat(). Xét hàm:

```
void hien(A *p)
{
    p->xuat();
}
```

Không cần biết tới địa chỉ của đối tượng nào sẽ truyền cho đối con trỏ p, lời gọi trong hàm luôn luôn gọi tới phương thức A::xuat() vì con trỏ p kiểu A. Như vậy bốn câu lệnh:

```
hien(&a);
```

```
hien(&b);
```

```
hien(&c);
```

```
hien(&d);
```

trong hàm main (của chương trình dưới đây) đều gọi tới A::xuat().

```
//CT6-01
```

```
// Phương thức tính
```

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

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

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

```
#include <ctype.h>
```

```
class A
```

```
{
```

```
private:
```

```
int n;
```

```
public:
```

```
A()
```

```
{
```

```
n=0;
```

```
}
```

```
A(int n1)
```

```
{
```

```
n=n1;
```

```
}
```

```
void xuat()
```

```
{
```

```
cout << "\nLop A: " << n;
```

```
}
```

```
int getN()
```

```
{
```

```
return n;
```

```
}
```

```
};
```

```
class B:public A
```

```
{
```

```
public:
```

```
B():A()
```

```
{
```

```
}
```

```
B(int n1):A(n1)
```

```
{
```

```
}
```

```
void xuat()
```

```
{
```

```
        cout << "\nLop B: " << getN();
    }
};

class C:public A
{
    public:
        C():A()
        {
        }
        C(int n1):A(n1)
        {
        }
        void xuat()
        {
            cout << "\nLop C: " << getN();
        }
};

class D:public C
{
    public:
        D():C()
        {
        }
        D(int n1):C(n1)
        {
        }
        void xuat()
        {
            cout << "\nLop D: " << getN();
        }
};

void hien(A *p)
{
    p->xuat();
}

void main()
{
    A a(1);
    B b(2);
    C c(3);
    D d(4);
```

```
clrscr();
hien(&a);
hien(&b);
hien(&c);
hien(&d);
getch();
}
```

Bài 2. Sự hạn chế của phương thức tĩnh

Ví dụ sau cho thấy sự hạn chế của phương thức tĩnh trong việc sử dụng tính thừa kế để phát triển chương trình.

Giả sử cần xây dựng chương trình quản lý thí sinh. Mỗi thí sinh đưa vào ba thuộc tính: Họ tên, số báo danh và tổng điểm. Chương trình gồm ba chức năng: Nhập dữ liệu thí sinh, in dữ liệu thí sinh ra máy in và xem - in (in họ tên ra màn hình, sau đó lựa chọn hoặc in hoặc không). Chương trình dưới đây sử dụng lớp TS (Thí sinh) đáp ứng được yêu cầu đặt ra.

```
//CT6-02
// Han che phuong thuc tinh
// Lop TS
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>
class TS
{
private:
    char ht[25];
    int sobd;
    float td;
public:
    void nhap()
    {
        cout << "\nHo ten: ";
        fflush(stdin); gets(ht);
        cout << "So bao danh: ";
        cin >> sobd;
        cout << "Tong diem: ";
        cin >> td;
    }
    void in()
    {
        fprintf(stdprn, "\n\nHo ten: %s", ht);
        fprintf(stdprn, "\nSo bao danh: %d", sobd);
        fprintf(stdprn, "\nTong diem: %0.1f", td);
    }
}
```

```
void xem_in()
{
    int ch;
    cout << "\nHo ten: " << ht ;
    cout << "\nCo in khong? - C/K" ;
    ch = toupper(getch());
    if (ch=='C')
        this->in();
}

};

void main()
{
    TS t[100];
    int i, n;
    cout << "\nSo thi sinh: ";
    cin >> n;
    for (i=1; i<=n; ++i)
        t[i].nhap();
    for (i=1; i<=n; ++i)
        t[i].xem_in();
    getch();
}
```

Giả sử Nhà trường muốn quản lý thêm địa chỉ của thí sinh. Vì sự thay đổi ở đây là không nhiều, nên chúng ta không đã động đến lớp TS mà xây dựng lớp mới TS2 dẫn xuất từ lớp TS. Trong lớp TS2 đưa thêm thuộc tính dc (địa chỉ) và các phương thức nhập, in. Cụ thể lớp TS2 được định nghĩa như sau:

```
class TS2:public TS
{
    private:
        char dc[30] ; // Dia chi
    public:
        void nhap()
        {
            TS::nhap();
            cout << "Dia chi: " ;
            fflush(stdin); gets(dc);
        }
        void in()
        {
            TS::in();
            fprintf(stdprn, "\nDia chi: %s", dc);
        }
}
```

```
    }  
};
```

Trong lớp TS2 không xây dựng lại phương thức xem_in, mà sẽ dùng phương thức xem_in của lớp TS. Chương trình mới như sau:

```
//CT6-03  
// Han che phuong thuc tinh  
// Lop TS TS2  
#include <conio.h>  
#include <stdio.h>  
#include <iostream.h>  
#include <ctype.h>  
class TS  
{  
    private:  
        char ht[25];  
        int sobd;  
        float td;  
    public:  
        void nhap()  
        {  
            cout << "\nHo ten: ";  
            fflush(stdin); gets(ht);  
            cout << "So bao danh: ";  
            cin >> sobd;  
            cout << "Tong diem: ";  
            cin >> td;  
        }  
        void in()  
        {  
            fprintf(stdprn, "\n\nHo ten: %s", ht);  
            fprintf(stdprn, "\nSo bao danh: %d", sobd);  
            fprintf(stdprn, "\nTong diem: %0.1f", td);  
        }  
        void xem_in()  
        {  
            int ch;  
            cout << "\nHo ten: " << ht ;  
            cout << "\nCo in khong? - C/K" ;  
            ch = toupper(getch());  
            if (ch=='C')
```

```
        this->in(); //Goi den TS::in() (Vi this la con tro
                        //kieu TS)
    }
};
class TS2:public TS
{
    private:
        char dc[30] ; // Dia chi
    public:
        void nhap()
        {
            TS::nhap();
            cout << "Dia chi: ";
            fflush(stdin); gets(dc);
        }
        void in()
        {
            TS::in();
            fprintf(stdprn, "\nDia chi: %s", dc);
        }
};

void main()
{
    TS2 t[100];
    int i, n;
    cout << "\nSo thi sinh: ";
    cin >> n;
    for (i=1; i<=n; ++i)
        t[i].nhap();
    for (i=1; i<=n; ++i)
        t[i].xem_in();
    getch();
}
```

Khi thực hiện chương trình này, chúng ta nhận thấy: Dữ liệu in ra vẫn không có địa chỉ. Điều này có thể giải thích như sau:

Xét câu lệnh (thứ 2 từ dưới lên trong hàm main):

```
t[i].xem_in() ;
```

Câu lệnh này gọi tới phương thức xem_in của lớp TS2 (vì t[i] là đối tượng của lớp TS2). Nhưng lớp TS2 không định nghĩa phương thức xem_in, nên phương thức TS::xem_in() sẽ được gọi tới. Hãy theo dõi phương thức này:

```
void xem_in()
{
    int ch;
    cout << "\nHo ten: " << ht ;
    cout << "\nCo in khong? - C/K" ;
    ch = toupper(getch());
    if(ch=='C')
        this->in(); //Goi den TS::in() (Vi this la con tro kieu TS)
}
```

Các lệnh đầu của phương thức sẽ in họ tên thí sinh. Nếu chọn có (bấm phím C), thì câu lệnh:

`this->in() ;`

sẽ được thực hiện. Mặc dù địa chỉ của t[i] (là đối tượng của lớp TS2) được truyền cho con trỏ this, thế nhưng câu lệnh này luôn luôn gọi tới phương thức TS::in(), vì con trỏ this ở đây có kiểu TS và vì in() là phương thức tĩnh. Kết quả là không in được địa chỉ của thí sinh.

Như vậy việc sử dụng các phương thức tĩnh in() (trong các lớp TS và TS2) đã không đáp ứng được yêu cầu phát triển chương trình. Có một giải pháp rất đơn giản là: Định nghĩa các phương thức in() trong các lớp TS và TS2 như các phương thức ảo (virtual).

Bài 3. Phương thức ảo và tương ứng bội

3.1. Cách định nghĩa phương thức ảo

Giả sử A là lớp cơ sở, các lớp B, C, D dẫn xuất (trực tiếp hoặc gián tiếp) từ A. Giả sử trong 4 lớp trên đều có các phương thức trùng dòng tiêu đề (trùng kiểu, trùng tên, trùng các đối). Để định nghĩa các phương thức này là các phương thức ảo, ta chỉ cần:

- + Hoặc thêm từ khoá virtual vào dòng tiêu đề của phương thức bên trong định nghĩa lớp cơ sở A.
- + Hoặc thêm từ khoá virtual vào dòng tiêu đề bên trong định nghĩa của tất cả các lớp A, B, C và D.

Ví dụ:

Cách 1:

```
class A
{
    ...
    virtual void hien_thi()
    {
        cout << "\n Đây là lớp A" ;
    };
};

class B : public A
{
    ...
    void hien_thi()
    {
```



```
        cout << "\n Đây là lớp B" ;
    };
};
class C : public B
{
    ...
    void hien_thi()
    {
        cout << "\n Đây là lớp C" ;
    };
};
class D : public A
{
    ...
    void hien_thi()
    {
        cout << "\n Đây là lớp D" ;
    };
};
```

Cách 2:

```
class A
{
    ...
    virtual void hien_thi()
    {
        cout << "\n Đây là lớp A" ;
    };
};
class B : public A
{
    ...
    virtual void hien_thi()
    {
        cout << "\n Đây là lớp B" ;
    };
};
class C : public B
{
    ...
    virtual void hien_thi()
    {
        cout << "\n Đây là lớp C" ;
    };
};
```

```
};  
};  
class D : public A  
{  
...  
virtual void hien_thi()  
{  
    cout << "\n Đây là lớp D" ;  
};  
};
```

Chú ý: Từ khoá virtual không được đặt bên ngoài định nghĩa lớp. Ví dụ nếu viết như sau là sai (CTBD sẽ báo lỗi).

```
class A  
{  
...  
virtual void hien_thi() ;  
};  
virtual void hien_thi() // Sai  
{  
    cout << "\n Đây là lớp A" ;  
};
```

Cần sửa lại như sau:

```
class A  
{  
...  
virtual void hien_thi() ;  
};  
void hien_thi() // Đúng  
{  
    cout << "\n Đây là lớp A" ;  
};
```

3.2. Quy tắc gọi phương thức ảo

Để có sự so sánh với phương thức tĩnh, ta nhắc lại quy tắc gọi phương thức tĩnh nêu trong §1.

3.2.1. Quy tắc gọi phương thức tĩnh

Lời gọi tới phương thức tĩnh bao giờ cũng xác định rõ phương thức nào (trong số các phương thức trùng tên của các lớp có quan hệ thừa kế) được gọi:

1. Nếu lời gọi xuất phát từ một đối tượng của lớp nào, thì phương thức của lớp đó sẽ được gọi.
2. Nếu lời gọi xuất phát từ một con trỏ kiểu lớp nào, thì phương thức của lớp đó sẽ được gọi bất kể con trỏ chứa địa chỉ của đối tượng nào.

3.2.2. Quy tắc gọi phương thức ảo

Phương thức ảo chỉ khác phương thức tĩnh khi được gọi từ một con trỏ (trường hợp 2 nêu trong mục 3.2.1). Lời gọi tới phương thức ảo từ một con trỏ chưa cho biết rõ phương thức nào (trong số các phương thức ảo trùng tên của các lớp có quan hệ thừa kế) sẽ được gọi. Điều này phụ thuộc vào đối tượng cụ thể mà con trỏ đang trỏ tới: Con trỏ đang trỏ tới đối tượng của lớp nào thì phương thức của lớp đó sẽ được gọi.

Ví dụ A, B, C và D là các lớp đã định nghĩa trong 3.1. Ta khai báo một con trỏ kiểu A và 4 đối tượng:

```
A *p ; // p là con trỏ kiểu A
A a ; // a là biến đối tượng kiểu A
B b ; // b là biến đối tượng kiểu B
C c ; // c là biến đối tượng kiểu C
D d ; // d là biến đối tượng kiểu D
```

Xét lời gọi tới các phương thức ảo `hien_thi` sau:

```
p = &a;          // p trỏ tới đối tượng a của lớp A
p->hien_thi() ; // Gọi tới A::hien_thi()
p = &b;          // p trỏ tới đối tượng b của lớp B
p->hien_thi() ; // Gọi tới B::hien_thi()
p = &c;          // p trỏ tới đối tượng c của lớp C
p->hien_thi() ; // Gọi tới C::hien_thi()
p = &d;          // p trỏ tới đối tượng d của lớp D
p->hien_thi() ; // Gọi tới D::hien_thi()
```

3.3. Tương ứng bội

Chúng ta nhận thấy cùng một câu lệnh

```
p->hien_thi();
```

tương ứng với nhiều phương thức khác nhau. Đây chính là tương ứng bội. Khả năng này rõ ràng cho phép xử lý nhiều đối tượng khác nhau, nhiều công việc, thậm chí nhiều thuật toán khác nhau theo cùng một cách thức, cùng một lược đồ. Điều này sẽ được minh họa trong các mục tiếp theo.

3.4. Liên kết động

Có thể so sánh sự khác nhau giữa phương thức tĩnh và phương thức ảo trên khía cạnh liên kết một lời gọi với một phương thức. Trở lại ví dụ trong 3.2:

```
A *p ; // p là con trỏ kiểu A
A a ; // a là biến đối tượng kiểu A
B b ; // b là biến đối tượng kiểu B
C c ; // c là biến đối tượng kiểu C
D d ; // d là biến đối tượng kiểu D
```

Nếu `hien_thi()` là các phương thức tĩnh, thì dù `p` chứa địa chỉ của các đối tượng `a`, `b`, `c` hay `d`, thì lời gọi:

```
p->hien_thi() ;
```

luôn luôn gọi tới phương thức `A::hien_thi()`

Như vậy một lời gọi (xuất phát từ con trỏ) tới phương thức tĩnh luôn luôn liên kết với một phương thức cố định và sự liên kết này xác định trong quá trình biên dịch chương trình.

Cũng với lời gọi:

`p->hien_thi() ;`

như trên, nhưng nếu `hien_thi()` là các phương thức ảo, thì lời gọi này không liên kết cứng với một phương thức cụ thể nào. Phương thức mà nó liên kết (gọi tới) còn chưa xác định trong giai đoạn dịch chương trình. Lời gọi này sẽ:

+ liên kết với `A::hien_thi()` , nếu `p` chứa địa chỉ đối tượng lớp `A`

+ liên kết với `B::hien_thi()` , nếu `p` chứa địa chỉ đối tượng lớp `B`

+ liên kết với `C::hien_thi()` , nếu `p` chứa địa chỉ đối tượng lớp `C`

+ liên kết với `D::hien_thi()` , nếu `p` chứa địa chỉ đối tượng lớp `D`

Như vậy một lời gọi (xuất phát từ con trỏ) tới phương thức ảo không liên kết với một phương thức cố định, mà tùy thuộc vào nội dung con trỏ. Đó là sự liên kết động và phương thức được liên kết (được gọi) thay đổi mỗi khi có sự thay đổi nội dung con trỏ trong quá trình chạy chương trình.

3.5. Quy tắc gán địa chỉ đối tượng cho con trỏ lớp cơ sở

+ Như đã nói trong §1, C++ cho phép gán địa chỉ đối tượng của một lớp dẫn xuất cho con trỏ của lớp cơ sở. Như vậy các phép gán sau (xem 3.2) là đúng:

`A *p ; // p là con trỏ kiểu A`

`A a ; // a là biến đối tượng kiểu A`

`B b ; // b là biến đối tượng kiểu B`

`C c ; // c là biến đối tượng kiểu C`

`D d ; // d là biến đối tượng kiểu D`

`p = &a; // p và a cùng lớp A`

`p = &b; // p là con trỏ lớp cơ sở, b là đối tượng lớp dẫn xuất`

`p = &c; // p là con trỏ lớp cơ sở, c là đối tượng lớp dẫn xuất`

`p = &d; // p là con trỏ lớp cơ sở, d là đối tượng lớp dẫn xuất`

+ **Tuy nhiên cần chú ý là:** Không cho phép gán địa chỉ đối tượng của lớp cơ sở cho con trỏ của lớp dẫn xuất. Như vậy ví dụ sau là sai:

`B *q ;`

`A a ;`

`q = &a;`

Sai vì: Gán địa chỉ đối tượng của lớp cơ sở `A` cho con trỏ của lớp dẫn xuất `B`

3.6. Ví dụ

Ta sửa chương trình trong bài 1 bằng cách định nghĩa các phương thức `xuat()` là ảo. Khi đó bốn câu lệnh:

`hien(&a);`

`hien(&b);`

`hien(&c);`

`hien(&d);`

trong hàm main (của chương trình dưới đây) sẽ lần lượt gọi tới 4 phương thức khác nhau:

`A::xuat()`

`B::xuat()`

```
C::xuat()
D::xuat()
//CT6-01B
// Phương thức ảo và tương ứng bội
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>
class A
{
    private:
        int n;
    public:
        A()
        {
            n=0;
        }
        A(int n1)
        {
            n=n1;
        }
        virtual void xuat()
        {
            cout << "\nLop A: " << n;
        }
        int getN()
        {
            return n;
        }
};
class B:public A
{
    public:
        B():A()
        {
        }
        B(int n1):A(n1)
        {
        }
        void xuat()
        {

```

```
        cout << "\nLop B: "<<getN();
    }
};

class C:public A
{
    public:
        C():A()
        {
        }
        C(int n1):A(n1)
        {
        }
        void xuat()
        {
            cout << "\nLop C: "<<getN();
        }
};

class D:public C
{
    public:
        D():C()
        {
        }
        D(int n1):C(n1)
        {
        }
        void xuat()
        {
            cout << "\nLop D: "<<getN();
        }
};

void hien(A *p)
{
    p->xuat();
}

void main()
{
    A a(1);
    B b(2);
    C c(3);
    D d(4);
    clrscr();
}
```

```
    hien(&a);  
    hien(&b);  
    hien(&c);  
    hien(&d);  
    getch();  
}
```

3.5. Sự thừa kế của các phương thức ảo

Cũng giống như các phương thức thông thường khác, phương thức ảo cũng có tính thừa kế. Chẳng hạn trong chương trình trên (mục 3.4) ta bỏ đi phương thức xuất() của lớp D, thì câu lệnh:

```
    hien(&d) ;
```

(câu lệnh gần cuối trong hàm main) sẽ gọi tới C::xuat() , phương thức này được kế thừa trong lớp D (vì D dẫn xuất từ C).

Bài 4. Sự linh hoạt của phương thức ảo trong phát triển nâng cấp chương trình

Ví dụ về các lớp TS và TS2 trong §2 đã chỉ ra sự hạn chế của phương thức tĩnh trong việc sử dụng tính thừa kế để nâng cấp, phát triển chương trình. Trong §2 cũng đã chỉ ra lớp TS2 chưa đáp ứng được yêu cầu nêu ra là in địa chỉ của thí sinh. Giải pháp cho vấn đề này rất đơn giản: Thay các phương thức tĩnh in() bằng cách dùng chúng như các phương thức ảo. Chương trình khi đó sẽ như sau:

```
//CT6-03B  
// Sự linh hoạt của phương thức ảo  
// Lop TS TS2  
#include <conio.h>  
#include <stdio.h>  
#include <iostream.h>  
#include <ctype.h>  
class TS  
{  
    private:  
        char ht[25];  
        int sobd;  
        float td;  
    public:  
        void nhap()  
        {  
            cout << "\nHo ten: ";  
            fflush(stdin); gets(ht);  
            cout << "So bao danh: ";  
            cin >> sobd;  
            cout << "Tong diem: ";  
            cin >> td;
```

```

    }
    virtual void in()
    {
        fprintf(stdprn, "\n\nHo ten: %s", ht);
        fprintf(stdprn, "\nSo bao danh: %d", sobd);
        fprintf(stdprn, "\nTong diem: %0.1f", td);
    }
    void xem_in()
    {
        int ch;
        cout << "\nHo ten: " << ht ;
        cout << "\nCo in khong? - C/K" ;
        ch = toupper(getch());
        if (ch=='C')
            this->in(); // Vì in() là phương thức ảo nên
            //có thể gọi đến TS::in() hoặc TS2::in()
    }
};

class TS2:public TS
{
private:
    char dc[30] ; // Dia chi
public:
    void nhap()
    {
        TS::nhap();
        cout << "Dia chi: " ;
        fflush(stdin); gets(dc);
    }
    void in()
    {
        TS::in();
        fprintf(stdprn, "\nDia chi: %s", dc);
    }
};

void main()
{
    TS2 t[100];
    int i, n;
    cout << "\nSo thi sinh: ";
    cin >> n;
    for (i=1; i<=n; ++i)

```



```
t[i].nhap();  
for (i=1; i<=n; ++i)  
    t[i].xem_in();  
getch();  
}
```

Khi thực hiện chương trình này, chúng ta nhận thấy: Dữ liệu thí sinh in ra đã có địa chỉ. Điều này có thể giải thích như sau:

Xét câu lệnh (thứ 2 từ dưới lên trong hàm main):

```
t[i].xem_in() ;
```

Câu lệnh này gọi tới phương thức xem_in của lớp TS2 (vì t[i] là đối tượng của lớp TS2). Nhưng lớp TS2 không định nghĩa phương thức xem_in, nên phương thức TS::xem_in() sẽ được gọi tới. Hãy theo dõi phương thức này:

```
void xem_in()  
{  
    int ch;  
    cout << "\nHo ten: " << ht ;  
    cout << "\nCo in khong? - C/K" ;  
    ch = toupper(getch());  
    this->in(); // Vì in() là phương thức ảo nên  
               //có thể gọi đến TS::in() hoặc TS2::in()  
}
```

Các lệnh đầu của phương thức sẽ in họ tên thí sinh. Nếu chọn Có (bấm phím C), thì câu lệnh:

```
this->in() ;
```

sẽ được thực hiện. Địa chỉ của t[i] (là đối tượng của lớp TS2) được truyền cho con trỏ this (của lớp cơ sở TS). Vì in() là phương thức ảo và vì this đang trỏ tới đối tượng t[i] của lớp TS2, nên câu lệnh này gọi tới phương thức TS2::in(). Trong phương thức TS2::in() có in địa chỉ của thí sinh.

Như vậy việc sử dụng các phương thức tĩnh in() (trong các lớp TS và TS2) đã không đáp ứng được yêu cầu phát triển chương trình. Có một giải pháp rất đơn giản là: Định nghĩa các phương thức in() trong các lớp TS và TS2 như các phương thức ảo (virtual).

Bài 5. Lớp cơ sở trừu tượng

5.1. Lớp cơ sở trừu tượng

Một lớp cơ sở trừu tượng là một lớp chỉ được dùng làm cơ sở cho các lớp khác. Không hề có đối tượng nào của một lớp trừu tượng được tạo ra cả, bởi vì nó chỉ được dùng để định nghĩa một số khái niệm tổng quát, chung cho các lớp khác. Một ví dụ về lớp trừu tượng là lớp CON_VAT (con vật), nó sẽ dùng làm cơ sở để xây dựng các lớp con vật cụ thể như lớp CON_CHO (con chó), CON_MEO (con mèo),... (xem ví dụ bên dưới)

Trong C++, thuật ngữ “Lớp trừu tượng” đặc biệt áp dụng cho các lớp có chứa các phương thức ảo thuần túy. Phương thức ảo thuần túy là một phương thức ảo mà nội dung của nó không có gì. Cách thức định nghĩa một phương thức ảo thuần túy như sau:

```
virtual void tên_phương_thức() = 0 ;
```

Ví dụ:

```
class A
```

```
{  
    public:  
        virtual void nhap() = 0 ;  
        virtual void xuat() = 0 ;  
        void chuong();  
};
```

Trong ví dụ trên, thì A là lớp cơ sở trừu tượng. Các phương thức nhap và xuat được khai báo là các lớp ảo thuần túy (bằng cách gán số 0 cho chúng thay cho việc cài đặt các phương thức này). Phương thức chuong() là một phương thức bình thường và sẽ phải có một định nghĩa ở đâu đó cho phương thức này.

Không có đối tượng nào của một lớp trừu tượng lại có thể được phát sinh. Tuy nhiên các con trỏ và các biến tham chiếu đến các đối tượng của lớp trừu tượng thì vẫn hợp lệ. Bất kỳ lớp nào dẫn xuất từ một lớp cơ sở trừu tượng phải định nghĩa lại tất cả các phương thức thuần ảo mà nó thừa hưởng, hoặc bằng các phương thức ảo thuần túy, hoặc bằng những định nghĩa thực sự. Ví dụ:

```
class B : public A  
{  
    public:  
        virtual void nhap() = 0 ;  
        virtual void xuat()  
        {  
            // Các câu lệnh  
        }  
};
```

Theo ý nghĩa về hướng đối tượng, ta vẫn có thể có một lớp trừu tượng mà không nhất thiết phải chứa đựng những phương thức thuần túy ảo.

Một cách tổng quát mà nói thì bất kỳ lớp nào mà nó chỉ được dùng làm cơ sở cho những lớp khác đều có thể được gọi là “lớp trừu tượng”. Một cách dễ dàng để nhận biết một lớp trừu tượng là xem có dùng lớp đó để khai báo các đối tượng hay không? . Nếu không thì đó là lớp cơ sở trừu tượng.

5.2. Ví dụ

Giả sử có 20 ô, mỗi ô có thể nuôi một con chó hoặc một con mèo. Yêu cầu xây dựng chương trình gồm các chức năng:

- + Nhập một con vật mới mua (hoặc chó, hoặc mèo) vào ô rỗng đầu tiên.
- + Xuất (đem bán) một con vật (hoặc chó, hoặc mèo).
- + Thống kê các con vật đang nuôi trong 20 ô.

Chương trình được tổ chức như sau:

- + Trước tiên định nghĩa lớp CON_VAT là lớp cơ sở ảo. Lớp này có một thuộc tính là tên con vật và một phương thức ảo dùng để xưng tên.
- + Hai lớp là CON_MEO và CON_CHO được dẫn xuất từ lớp CON_VAT
- + Cuối cùng là lớp DS_CON_VAT (Danh sách con vật) dùng để quản lý chung cả mèo và chó. Lớp này có 3 thuộc tính là: số con vật cực đại (chính bằng số ô), số con vật đang nuôi và một mảng con trỏ kiểu CON_VAT. Mỗi phần tử mảng sẽ chứa địa chỉ của một đối tượng kiểu CON_MEO hoặc CON_CHO.

Lớp sẽ có 3 phương thức để thực hiện 3 chức năng nêu trên của chương trình. Nội dung chương trình như sau:

```
//CT6-04
// Lop co so truu tuong
// Lop CON_VAT
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>
#include <string.h>
class CON_VAT
{
protected:
    char *ten;
public:
    CON_VAT()
    {
        ten = NULL;
    }
    CON_VAT(char *ten1)
    {
        ten = strdup(ten1);
    }
    virtual void xung_ten()
    {
    }
};
class CON_MEO:public CON_VAT
{
public:
    CON_MEO() : CON_VAT()
    {
    }
    CON_MEO(char *ten1) : CON_VAT(ten1)
    {
    }
    virtual void xung_ten()
    {
        cout << "\nToi la chu meo: " << ten ;
    }
};
```

```
class CON_CHO:public CON_VAT
{
    public:
        CON_CHO() : CON_VAT()
        {
        }
        CON_CHO(char *ten1) : CON_VAT(ten1)
        {
        }
        virtual void xung_ten()
        {
            cout << "\nToi la chu cho: " << ten ;
        }
};

class DS_CON_VAT // Danh sach con vat
{
    private:
        int max_so_con_vat;
        int so_con_vat;
        CON_VAT **h ;
    public:
        DS_CON_VAT(int max);
        ~DS_CON_VAT();
        int nhap(CON_VAT *c);
        CON_VAT* xuat(int n);
        void thong_ke();
};

DS_CON_VAT::DS_CON_VAT(int max)
{
    max_so_con_vat = max;
    so_con_vat = 0;
    h = new CON_VAT*[max];
    for (int i=0; i<max; ++i)
        h[i] = NULL;
}

DS_CON_VAT::~~DS_CON_VAT()
{
    max_so_con_vat = 0;
    so_con_vat = 0;
    delete h;
}
```

```
}
int DS_CON_VAT::nhap(CON_VAT *c)
{
    if (so_con_vat==max_so_con_vat)
        return 0;
    int i=0;
    while (h[i]!=NULL) ++i;
    h[i]=c;
    so_con_vat++;
    return (i+1);
}
CON_VAT* DS_CON_VAT::xuat(int n)
{
    if (n<1 || n > max_so_con_vat)
        return NULL ;
    --n ;
    if (h[n])
    {
        CON_VAT *c = h[n];
        h[n]=NULL;
        so_con_vat--;
        return c;
    }
    else
        return NULL;
}
void DS_CON_VAT::thong_ke()
{
    if (so_con_vat)
    {
        cout << "\n";
        for (int i=0; i<max_so_con_vat; ++i)
            if (h[i])
                h[i]->xung_ten();
    }
}
CON_CHO c1("MUC");
CON_CHO c2("VEN");
CON_CHO c3("LAI");
CON_CHO c4("NHAT");
CON_CHO c5("BONG");
```

```
CON_MEO m1("MUOP");
CON_MEO m2("DEN");
CON_MEO m3("TRANG");
CON_MEO m4("TAM THE");
CON_MEO m5("VANG");
void main()
{
    DS_CON_VAT d(20);
    clrscr();
    d.nhap(&c1);
    int im2 = d.nhap(&m2);
    d.nhap(&c3);
    d.nhap(&m1);
    int ic4 = d.nhap(&c4);
    d.nhap(&c5);
    d.nhap(&m5);
    d.nhap(&c2);
    d.nhap(&m3);
    d.thong_ke();
    d.xuat(im2);
    d.xuat(ic4);
    d.thong_ke();
    getch();
}
```

Chú ý: Theo quan điểm chung về cách thức sử dụng, thì lớp CON_VAT là lớp cơ sở trừu tượng. Tuy nhiên theo quan điểm của C++ thì lớp này chưa phải là lớp cơ sở trừu tượng, vì trong lớp không có các phương thức thuần túy ảo. Phương thức xung_ten:

```
virtual void xung_ten()
{
}
```

là phương thức ảo, được định nghĩa đầy đủ, mặc dù thân của nó là rỗng.

Do vậy khai báo:

```
CON_VAT cv("Con vat chung");
```

vẫn được C++ chấp nhận.

Bây giờ nếu định nghĩa lại phương thức xung_ten như sau:

```
virtual void xung_ten() = 0 ;
```

thì nó trở thành phương thức thuần ảo và C++ sẽ quan niệm lớp CON_VAT là lớp trừu tượng. Khi đó câu lệnh khai báo:

```
CON_VAT cv("Con vat chung");
```

sẽ bị C++ bắt lỗi với thông báo:

```
Cannot create instance of abstract class 'CON_VAT'
```

Bài 6. Sử dụng tương ứng bội và phương thức ảo

6.1. Chiến lược sử dụng tương ứng bội

Tương ứng bội cho phép xét các vấn đề khác nhau, các đối tượng khác nhau, các phương pháp khác nhau, các cách giải quyết khác nhau theo cùng một lược đồ chung.

Các bước áp dụng tương ứng bội có thể tổng kết lại như sau:

1. Xây dựng lớp cơ sở trừu tượng bao gồm những thuộc tính chung nhất của các thực thể cần quản lý. Đưa vào các phương thức ảo hay thuần ảo dùng để xây dựng các nhóm phương thức ảo cho các lớp dẫn xuất sau này. Mỗi nhóm phương thức ảo sẽ thực hiện một chức năng nào đó trên các lớp.

2. Xây dựng các lớp dẫn xuất bắt đầu từ lớp cơ sở ảo. Số mức dẫn xuất là không hạn chế. Các lớp dẫn xuất sẽ mô tả các đối tượng cụ thể cần quản lý.

3. Xây dựng các phương thức ảo trong các dẫn xuất. Các phương thức này tạo thành các nhóm phương thức ảo trong sơ đồ các lớp có quan hệ thừa kế.

4. Xây dựng lớp quản lý các đối tượng. Dữ liệu của lớp này là một dãy con trỏ của lớp cơ sở trừu tượng ban đầu. Các con trỏ này có thể chứa địa chỉ đối tượng của các lớp dẫn xuất. Do vậy có thể dùng các con trỏ này để thực hiện các thao tác trên các đối tượng của bất kỳ lớp dẫn xuất nào.

6.2. Ví dụ

Chương trình quản lý các con vật trong §5 là một ví dụ về cách sử dụng tương ứng bội. Dưới đây là một ví dụ khác. Giả sử có 4 hình vẽ: Đoạn thẳng, hình tròn, hình chữ nhật và hình vuông. Bốn hình cho hiện thẳng hàng trên màn hình tạo thành một bức tranh. Nếu thay đổi thứ tự các hình sẽ nhận được các bức tranh khác nhau. Chương trình dưới đây sẽ cho hiện tất cả các bức tranh khác nhau. Chương trình được tổ chức theo các bước nêu trong 6.1:

+ Lớp cơ sở trừu tượng là lớp HINH (hình) gồm một thuộc tính mau (màu) và một phương thức ảo thuần túy:

```
virtual void draw(int x, int y) = 0 ;
```

+ Các lớp dẫn xuất trực tiếp từ lớp hình là : DTHANG , HTRON và CHUNHAT.

+ Lớp VUONG dẫn xuất từ lớp CHUNHAT.

+ Lớp quản lý chung là lớp picture có thuộc tính là một mảng con trỏ kiểu HINH gồm 4 phần tử dùng để chứa địa chỉ 4 đối tượng: DTHANG, HTRON, CHUNHAT và VUONG. Sử dụng phương thức draw gọi từ 4 phần tử mảng nói trên sẽ nhận được một bức tranh. Bằng cách hoán vị các phần tử này, sẽ nhận được tất cả các bức tranh khác nhau.

```
//CT6-05
```

```
// Lop co so tru tuong
```

```
// Lop hình học
```

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

```
#include <graphics.h>
```

```
class HINH
```

```
{
```

```
private:
```

```
int mau;
```

```
public:
```

```
HINH(int m)
```

```
{
```

```
mau = m;
```

```
    }
    getmau()
    {
        return mau;
    }
    virtual void draw(int x, int y) = 0;
};

class DTHANG : public HINH
{
private:
    int dodai;
public:
    DTHANG(int d, int m):HINH(m)
    {
        dodai = d ;
    }
    virtual void draw(int x, int y)
    {
        setcolor(getmau()) ;
        line(x,y,x+dodai,y);
    }
};

class CHUNHAT: public HINH
{
private:
    int rong, cao;
public:
    CHUNHAT(int r, int c, int m):HINH(m)
    {
        rong = r; cao = c;
    }
    virtual void draw(int x, int y )
    {
        setcolor(getmau()) ;
        rectangle(x,y,x+rong,y+cao);
        setfillstyle(1,getmau());
        floodfill(x+rong/2,y+cao/2, getmau() );
    }
};

class VUONG : public CHUNHAT
{

```



```

    public:
        VUONG(int a, int m): CHUNHAT(a,a,m)
        {
        }
};

class HTRON: public HINH
{
    private:
        int bk; //Ban kinh
    public:
        HTRON(int bk1, int m):HINH(m)
        {
            bk = bk1;
        }
        virtual void draw(int x, int y)
        {
            setcolor(getmau());
            circle(x+bk,y+bk,bk);
            setfillstyle(1,getmau());
            floodfill(x + bk, y + bk,getmau());
        }
};

class picture
{
    private:
        HINH *h[4];
    public:
        picture(HINH *h0,HINH *h1,HINH *h2,HINH *h3)
        {
            h[0]=h0;
            h[1]=h1;
            h[2]=h2;
            h[3]=h3;
        }
        void paint(int *k);
        void listpaint();
};

void picture::paint(int *k)
{
    for (int i=0; i<4; ++i)
        h[k[i]]->draw(10+i*150, 200);
}

```

```
void picture::listpaint()
{
    int k[4],i1,i2,i3,i4;
    for (i1=0;i1<4;++i1)
        for (i2=0;i2<4;++i2)
            if (i2!=i1)
                for (i3=0;i3<4;++i3)
                    if (i3!=i2 && i3!=i1)
                        for (i4=0;i4<4;++i4)
                            if (i4!=i3 && i4!=i2 && i4!=i1)
                                {
                                    k[0]=i1;k[1]=i2;
                                    k[2]=i3;k[3]=i4;
                                    paint(k);
                                    getch();
                                    cleardevice();
                                }
}

DTHANG dt(120,14);
HTRON ht(60,RED);
CHUNHAT cn(120,100,MAGENTA);
VUONG v(120,CYAN);

};

void main()
{
    int mh=0,mode=0;
    initgraph(&mh,&mode,"");
    picture pic(&dt,&ht,&cn,&v);
    pic.listpaint();
    getch();
    closegraph();
}
```

Bài 7. Xử lý các thuật toán khác nhau

Có thể sử dụng tương ứng bội để tổ chức thực hiện các thuật toán khác nhau trên cùng một bài toán như sau:

- + Lớp cơ sở trừu tượng sẽ chứa dữ liệu bài toán và một phương thức ảo.
- + Mỗi lớp dẫn xuất ứng với một thuật toán cụ thể. Phương thức ảo của lớp dẫn xuất sẽ thực hiện một thuật toán cụ thể.

+ Sử dụng một mảng con trỏ của lớp cơ sở và gán cho mỗi phần tử mảng địa chỉ của một đối tượng của lớp dẫn xuất. Sau đó dùng các phần tử mảng con trỏ để gọi tới các phương thức ảo. Bằng cách đó sẽ thực hiện cùng một bài toán theo các thuật toán khác nhau và dễ dàng so sánh hiệu quả của các thuật toán.

Ví dụ sau minh họa việc thực hiện bài toán sắp xếp dãy số nguyên theo thứ tự tăng bằng cách dùng đồng thời 3 thuật toán: Thuật toán lựa chọn (Select_Sort), thuật toán sắp xếp nhanh (Quick_Sort) và thuật toán vun đống (Heap_Sort). Chương trình gồm 4 lớp:

+ Lớp cơ sở trừu tượng:

```
class sort
{
protected:
    int *a;
    void hoan_vi(long i, long j) ;
public:
    virtual void sapxep(int *a1, long n) ;
};
```

Lớp này gồm:

- Một thành phần dữ liệu là con trỏ a trỏ tới một vùng nhớ chứa dãy số nguyên cần sắp xếp.
- Phương thức hoan_vi(i,j) dùng để hoán vị các phần tử a[i] và a[j]. Phương thức này được dùng trong 3 lớp dẫn xuất bên dưới.

- Phương thức ảo sapxep(a1,n) dùng để sắp xếp dãy n số nguyên chứa trong mảng a1.

+ Ba lớp dẫn xuất là: SELECT_SORT, QUICK_SORT và HEAP_SORT. Mỗi lớp đều có phương thức ảo:

```
virtual void sapxep(int *a1, long n) ;
```

để thực hiện việc sắp xếp theo một thuật toán cụ thể.

+ Trong hàm main() sẽ tạo ra một dãy 30000 số nguyên một cách ngẫu nhiên, sau đó lần lượt sử dụng 3 thuật toán sắp xếp để so sánh. Kết quả như sau:

Thời gian sắp xếp theo thuật toán Select sort là: 19.20 giây

Thời gian sắp xếp theo thuật toán Quick sort là: 0.11 giây

Thời gian sắp xếp theo thuật toán Heap sort là: 0.44 giây

Nội dung chương trình như sau:

```
//CT6-06
```

```
// Lop co so truu tuong
```

```
// Lop sort
```

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

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

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

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

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

```
#include <dos.h>
```

```
class sort
```

```
{
```

```
protected:
```

```
int *a;
void hoan_vi(long i, long j)
{
    int tg = a[i];
    a[i] = a[j];
    a[j] = tg;
}
public:
    virtual void sapxep(int *a1, long n)
    {
        a = a1;
    }
};
class select_sort : public sort
{
public:
    virtual void sapxep(int *a1, long n) ;
};
void select_sort::sapxep(int *a1, long n)
{
    long i,j,r;
    sort::sapxep(a1,n);
    for (i=1; i<n; ++i)
    {
        r=i;
        for (j=i+1; j<=n; ++j)
            if(a[j] < a[r]) r = j;
        if(r!=i) hoan_vi(i,r);
    }
}
class quick_sort : public sort
{
private:
    void q_sort(long l, long r);
public:
    virtual void sapxep(int *a1, long n) ;
};
void quick_sort::q_sort(long l, long r)
{
    int x;
    long i,j;
```

```
    if (l < r)
    {
        x = a[l]; i = l; j = r+1;
        do
        {
            ++i; --j;
            while (i<r && a[i] < x) ++i ;
            while (a[j] > x) --j ;
            if (i<j) hoan_vi(i,j);
        } while (i<j);
        hoan_vi(l,j);
        q_sort(l,j-1);
        q_sort(j+1,r);
    }
}

void quick_sort::sapxep(int *a1, long n)
{
    sort::sapxep(a1,n);
    q_sort(1,n);
}

class heap_sort : public sort
{
private:
    void shift(long i, long n);
public:
    virtual void sapxep(int *a1, long n) ;
};

void heap_sort::shift(long i, long n)
{
    long l,r,k;
    l = 2*i; r = l+1;
    if (l>n) return;
    if (l==n)
    {
        if (a[i]<a[l]) hoan_vi(i,l);
        return;
    }
    if (a[l] > a[r])
        k = l;
    else
        k = r;
```

```
    if (a[i]>=a[k])
        return;
    else
    {
        hoan_vi(i,k);
        shift(k,n);
    }
}

void heap_sort::sapxep(int *a1, long n)
{
    long i;
    sort::sapxep(a1,n);
    /* Tao dong */
    for (i=n/2 ; i>=1; --i) shift(i,n);
    /* Lap */
    for (i=n ; i>=2; --i)
    {
        hoan_vi(1,i);
        shift(1,i-1);
    }
}

void main()
{
    long i,n;
    struct time t1,t2;
    int *a, k, tg, sec, hund;
    n=30000;
    a=(int*) malloc((n+1)*sizeof(int));
    if (a==NULL)
    {
        puts("\nLoi BN");
        getch();
        exit(0);
    }
    sort *s[3];
    select_sort ss;
    quick_sort qs;
    heap_sort hs;
    s[0]=&ss; s[1]=&qs; s[2]=&hs;
    clrscr();
```

```
for (k=0; k<3; ++k)
{
    srand(5000);
    for (i=1;i<=n;++i)
        a[i]=rand();
    gettimeofday(&t1);
    s[k]->sapxep(a,n);
    gettimeofday(&t2);
    tg = (t2.ti_sec - t1.ti_sec)*100 + t2.ti_hund - t1.ti_hund ;
    sec = tg / 100;
    hund = tg % 100;
    printf("\n Sap xep %d %d %d %d %d",k+1,
        t2.ti_sec,t2.ti_hund,t1.ti_sec,t1.ti_hund);
    printf("\n Sap xep %d Thoi gian %d sec %d hund",
        k+1,sec,hund);
}
getch();
}
```

Chương 7

Các dòng tập tin (Stream)

C đã cung cấp một thư viện các hàm nhập xuất như printf, scanf, gets, getch(), puts, puch(), fprintf, fscanf, fopen, fwite, fread,... . Các hàm này làm việc khá hiệu quả nhưng không thích ứng với cách tổ chức chương trình hướng đối tượng.

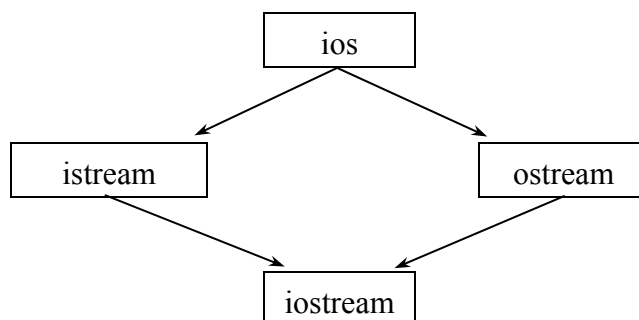
C++ sử dụng khái niệm dòng tin (stream) và đưa ra các lớp dòng tin để tổ chức việc nhập xuất. Dòng tin có thể xem như một dãy các byte. Thao tác nhập là lấy (đọc) các byte từ dòng tin (khi đó gọi là dòng nhập - input) vào bộ nhớ. Thao tác xuất là đưa các byte từ bộ nhớ ra dòng tin (khi đó gọi là dòng xuất - output). Các thao tác này là độc lập thiết bị. Để thực hiện việc nhập, xuất lên một thiết bị cụ thể, chúng ta chỉ cần gắn dòng tin với thiết bị này.

Bài 1. các lớp stream

Có 4 lớp quan trọng cần nhớ là:

- + Lớp cơ sở ios
- + Từ lớp ios dẫn xuất đến 2 lớp istream và ostream
- + Hai lớp istream và ostream lại dẫn xuất tới lớp iostream

Sơ đồ kế thừa giữa các lớp như sau:



Lớp ios

+ Thuộc tính của lớp: Trong lớp ios định nghĩa các thuộc tính được sử dụng làm các cờ định dạng cho việc nhập xuất và các cờ kiểm tra lỗi (xem bên dưới).

+ Các phương thức: Lớp ios cung cấp một số phương thức phục vụ việc định dạng dữ liệu nhập xuất, kiểm tra lỗi (xem bên dưới).

Lớp istream

Lớp này cung cấp toán tử nhập >> và nhiều phương thức nhập khác (xem bên dưới) như các phương thức: get, getline, read, ignore, peek, seekg, tellg,...

Lớp ostream

Lớp này cung cấp toán tử xuất << và nhiều phương thức xuất khác (xem bên dưới) như các phương thức: put, write, flush, seekp, tellp,...

Lớp iostream

Lớp này thừa kế các phương thức nhập xuất của các lớp istream và ostream.

Bài 2. Dòng cin và toán tử nhập

Dòng cin là một đối tượng kiểu istream đã định nghĩa trong C++ . Đó là dòng vào (input) chuẩn gắn với bàn phím (tương tự như stdin của C). Các thao tác nhập trên dòng cin đồng nghĩa với nhập dữ liệu từ bàn phím.

Do cin là một đối tượng của lớp istream nên với cin chúng ta có thể sử dụng toán tử nhập >> và các phương thức nhập của các lớp ios và istream.

Cách dùng toán tử nhập để đọc dữ liệu từ dòng cin như sau:

cin >> Tham_số ;

Trong đó Tham_số có thể là:

- Biến hoặc phần tử mảng nguyên để nhận một số nguyên
- Biến hoặc phần tử mảng thực để nhận một số thực
- Biến hoặc phần tử mảng ký tự để nhận một ký tự
- Con trỏ ký tự để nhận một dãy các ký tự khác trống

Chú ý: Các toán tử nhập có thể viết nối đuôi để nhập nhiều giá trị trên một dòng lệnh như sau:

cin >> Tham_số_1 >> Tham_số_2 >> ... >> Tham_số_k ;

Cách thức nhập như sau: Bỏ qua các ký tự trắng (dấu cách, dấu tab, dấu chuyển dòng) đứng trước nếu có và sau đó đọc vào các ký tự tương ứng với kiểu yêu cầu. Cụ thể đối với từng kiểu như sau:

Khi nhập số nguyên sẽ bỏ qua các ký tự trắng đứng trước nếu có, sau đó bắt đầu nhận các ký tự biểu thị số nguyên. Việc nhập kết thúc khi gặp một ký tự trắng hoặc một ký tự không thể hiểu là thành phần của số nguyên. Ví dụ nếu trên dòng vào (gõ từ bàn phím) chứa các ký tự <space><space>123X2 và Tham_số (bên phải cin) là biến nguyên n thì n sẽ nhận giá trị 123. Con trỏ nhập sẽ dừng tại ký tự X.

Phép nhập một số thực cũng tiến hành tương tự: Bỏ qua các khoảng trắng đứng trước nếu có, sau đó bắt đầu nhận các ký tự biểu thị số Thực. Việc nhập kết thúc khi gặp một ký tự trắng hoặc một ký tự không thể hiểu là thành phần của số thực.

Phép nhập một ký tự cũng vậy: Bỏ qua các khoảng trắng đứng trước nếu có, sau đó nhận một ký tự khác ký tự trắng. Ví dụ nếu gõ <space><space>XY thì ký tự X được nhận và con trỏ nhập dừng tại ký tự Y.

Phép nhập một dãy ký tự: Bỏ qua các khoảng trắng đứng trước nếu có, sau đó bắt đầu nhận từ một ký tự khác ký tự trắng. Việc nhập kết thúc khi gặp một ký tự trắng.

Ví dụ 1: Xét đoạn chương trình:

```
char ten[10], que[12];
char ch;
int n;
float x;
cin >> n >> x >> ch >> ten >> que ;
```

Nếu gõ các ký tự:

```
123<s>3.14<s><s>ZHONG<s>HAI<s>PHONG<Enter>
```

(để cho gọn sẽ ký hiệu <s> là <space>)

thì kết quả nhập như sau:

```
n=123
x=3.14
ch='Z'
ten="HONG"
que = "HAI"
```

Con trỏ nhập sẽ dừng tại ký tự <space> trước từ PHONG. Các ký tự còn lại sẽ được nhận trong các câu lệnh nhập tiếp theo.

Ví dụ 2: Xét đoạn chương trình:

```
int m;  
float y;  
cin >> m >> y;
```

Nếu gõ:

```
<s><s>456<s><s>4.5<Enter>
```

thì kết quả nhập là:

```
m = 456  
y = 4.5
```

Ký tự <Enter> vẫn còn lại trên dòng nhập.

Bài 3. Nhập ký tự và chuỗi ký tự từ bàn phím

Chúng ta nhận thấy toán tử nhập >> chỉ tiện lợi khi dùng để nhập các giá trị số (nguyên, thực). Để nhập ký tự và chuỗi ký tự nên dùng các phương thức sau (định nghĩa trong lớp istream):

cin.get cin.getline cin.ignore

3.1. Phương thức get có 3 dạng (thực chất có 3 phương thức cùng có tên get):

Dạng 1:

```
int cin.get() ;
```

dùng để đọc một ký tự (kể cả khoảng trắng). Cách thức đọc của cin.get có thể minh họa qua ví dụ sau: Xét các câu lệnh

```
char ch;  
ch = cin.get()
```

+ Nếu gõ

```
ABC<Enter>
```

thì biến ch nhận mã ký tự A, các ký tự BC<Enter> còn lại trên dòng vào.

+ Nếu gõ

```
A<Enter>
```

thì biến ch nhận mã ký tự A, ký tự <Enter> còn lại trên dòng vào.

+ Nếu gõ

```
<Enter>
```

thì biến ch nhận mã ký tự <Enter> (bằng 10) và dòng vào rỗng.

Dạng 2:

```
istream& cin.get(char &ch) ;
```

dùng để đọc một ký tự (kể cả khoảng trắng) và đặt vào một biến kiểu char được tham chiếu bởi ch.

Chú ý:

+ Cách thức đọc của cin.get dạng 2 cũng giống như dạng 1

+ Do cin.get() dạng 2 trả về tham chiếu tới cin, nên có thể sử dụng các phương thức get() dạng 2 nối đuôi nhau. Ví dụ 2 nếu khai báo

```
char ch1, ch2;
```

thì 2 câu lệnh:

```
cin.get(ch1);
```

```
cin.get(ch2);
```

có thể viết chung trên một câu lệnh sau:

```
cin.get(ch1).get(ch2);
```

Dạng 3:

```
istream& cin.get(char *str, int n, char delim = '\n');
```

dùng để đọc một dãy ký tự (kể cả khoảng trắng) và đưa vào vùng nhớ do str trỏ tới. Quá trình đọc kết thúc khi xảy ra một trong 2 tình huống sau:

- + Gặp ký tự giới hạn (cho trong delim). Ký tự giới hạn mặc định là '\n' (Enter)
- + Đã nhận đủ (n-1) ký tự

Chú ý:

- + Ký tự kết thúc chuỗi '\0' được bổ sung vào dãy ký tự nhận được
- + ký tự giới hạn vẫn còn lại trên dòng nhập để dành cho các lệnh nhập tiếp theo.

Chú ý:

- + Cũng giống như get() dạng 2, có thể viết các phương thức get() dạng 3 nối đuôi nhau trên một dòng lệnh.
- + Ký tự <Enter> còn lại trên dòng nhập có thể làm trôi phương thức get() dạng 3. Ví dụ xét đoạn chương trình:

```
char ht[25], qq[20], cq[30];
cout << "\nHọ tên: ";
cin.get(ht,25);
cout << "\nQuê quán: ";
cin.get(qq,20);
cout << "\nCơ quan: ";
cin.get(cq,30);
cout << "\n" << ht << " " << qq << " " << cq
```

Đoạn chương trình dùng để nhập họ tên, quê quán và cơ quan. Nếu gõ:

```
Pham Thu Huong<Enter>
```

thì câu lệnh get đầu tiên sẽ nhận được chuỗi "Pham Thu Huong" cất vào mảng ht. Ký tự <Enter> còn lại sẽ làm trôi 2 câu lệnh get tiếp theo. Do đó câu lệnh cuối cùng sẽ chỉ in ra Pham Thu Huong.

Để khắc phục tình trạng trên, có thể dùng một trong các cách sau:

- + Dùng phương thức get() dạng 1 hoặc dạng 2 để lấy ra ký tự <Enter> trên dòng nhập trước khi dùng get (dạng 3).
- + Dùng phương thức ignore để lấy ra một số ký tự không cần thiết trên dòng nhập trước khi dùng get dạng 3. Phương thức này viết như sau:

```
cin.ignore(n); // Lấy ra (loại ra hay bỏ qua) n ký tự trên
               // dòng nhập.
```

Như vậy để có thể nhập được cả quê quán và cơ quan, cần sửa lại đoạn chương trình trên như sau:

```
char ht[25], qq[20], cq[30];
cout << "\nHọ tên: " ;
cin.get(ht,25);
cin.get(); // Nhận <Enter>
cout << "\nQuê quán: " ;
cin.get(qq,20);
ignore(1); // Bỏ qua <Enter>
cout << "\nCơ quan: " ;
cin.get(cq,30);
cout << "\n" << ht << " " << qq << " " << cq
```

3.2. Phương thức getline

Tương tự như get dạng 3, có thể dùng getline để nhập một dãy ký tự từ bàn phím. Phương thức này được mô tả như sau:

```
istream& cin.getline(char *str, int n, char delim = '\n');
```

Phương thức đầu tiên làm việc như get dạng 3, sau đó nó loại <Enter> ra khỏi dòng nhập (ký tự <Enter> không đưa vào dãy ký tự nhận được). Như vậy có thể dùng getline để nhập nhiều chuỗi ký tự (mà không lo ngại các câu lệnh nhập tiếp theo bị trôi).

Ví dụ đoạn chương trình nhập họ tên, quê quán và cơ quan bên trên có thể viết như sau (bằng cách dùng getline):

```
char ht[25], qq[20], cq[30];
cout << "\nHọ tên: " ;
cin.getline(ht,25);
cout << "\nQuê quán: " ;
cin.getline(qq,20);
cout << "\nCơ quan: " ;
cin.get(cq,30);
cout << "\n" << ht << " " << qq << " " << cq
```

Chú ý: Cũng giống như get() dạng 2 và get() dạng 3, có thể viết các phương thức getline() nối đuôi nhau trên một dòng lệnh. Ví dụ đoạn chương trình trên có thể viết lại như sau:

```
char ht[25], qq[20], cq[30];
cout << "\nHọ tên, Quê quán và Cơ quan: " ;
cin.getline(ht,25).getline(qq,20).get(cq,30);
cout << "\n" << ht << " " << qq << " " << cq
```

3.3. Phương thức ignore

Phương thức ignore dùng để bỏ qua (loại bỏ) một số ký tự trên dòng nhập. Trong nhiều trường hợp, đây là việc làm cần thiết để không làm ảnh hưởng đến các phép nhập tiếp theo.

Phương thức ignore được mô tả như sau:

```
istream& cin.ignore(int n=1);
```

Phương thức sẽ bỏ qua (loại bỏ) n ký tự trên dòng nhập.

3.4. Nhập đồng thời giá trị số và ký tự

Như đã nói trong §2, toán tử nhập >> bao giờ cũng để lại ký tự <Enter> trên dòng nhập. Ký tự <Enter> này sẽ làm trôi các lệnh nhập ký tự hoặc chuỗi ký tự bên dưới. Do vậy cần dùng:

hoặc ignore()
hoặc get() dạng 1
hoặc get() dạng 2

để loại bỏ ký tự <Enter> còn sót lại ra khỏi dòng nhập trước khi thực hiện việc nhập ký tự hoặc chuỗi ký tự.

3.5. Ví dụ: Chương trình dưới đây sử dụng lớp TSINH (Thí sinh) với 2 phương thức xuất và nhập.

```
//CT7_04.CPP
// Nhập dữ liệu số và ký tự
#include <iostream.h>
#include <conio.h>
struct TS
{
    int sobd;
    char ht[25];
    float dt,dl,dh,td;
};
class TSINH
{
private:
    TS *ts;
    int sots;
public:
    TSINH()
    {
        ts=NULL;
        sots=0;
    }
    TSINH(int n)
    {
        ts=new TS[n+1];
        sots=n;
    }
    ~TSINH()
    {
        if (sots)
        {
            sots=0;
            ts = NULL;
        }
    }
}
```

```
    }
    void nhap();
    void xuat();
};

void TSINH::nhap()
{
    if (sots)
        for (int i=1; i<=sots; ++i)
        {
            cout << "\nThi sinh "<< i << ": ";
            cout << "\nSo bao danh: ";
            cin >> ts[i].sobd;
            cin.ignore();
            cout << "Ho ten: ";
            cin.get(ts[i].ht,25);
            cout << "Diem toan, ly , hoa: ";
            cin >> ts[i].dt >> ts[i].dl >> ts[i].dh;
            ts[i].td = ts[i].dt + ts[i].dl + ts[i].dh;
        }
}

void TSINH::xuat()
{
    if (sots)
    {
        cout << "\nDanh sach thi sinh: ";
        for (int i=1; i<=sots; ++i)
            cout << "\nHo ten: " << ts[i].ht << " So BD: " << ts[i].sobd
                << " Tong diem: " << ts[i].td;
    }
}

void main()
{
    int n;
    clrscr();
    cout << "\nSo thi sinh: ";
    cin >> n;
    TSINH *t = new TSINH(n);
    t->nhap();
```

```
t->xuat();  
getch();  
delete t;  
}
```

Bài 4. Dòng cout và toán tử xuất

4.1. Dòng cout

Dòng cout là một đối tượng kiểu ostream đã định nghĩa trong C++. Đó là dòng xuất (output) chuẩn gắn với màn hình (tương tự như stdout của C). Các thao tác xuất trên dòng cout đồng nghĩa với xuất dữ liệu ra màn hình.

Do cout là một đối tượng của lớp ostream nên với cout chúng ta có thể sử dụng toán tử xuất << và các phương thức xuất của các lớp ios và ostream.

4.2. Toán tử xuất

C++ định nghĩa chồng toán tử dịch trái << để gửi các ký tự ra dòng xuất.

Cách dùng toán tử xuất để xuất dữ liệu từ bộ nhớ ra dòng cout như sau:

cout << Tham_số ;

Trong đó Tham_số biểu thị một giá trị cần xuất ra màn hình. Giá trị sẽ được biến đổi thành một dãy ký tự trước khi đưa ra dòng xuất. Kiểu của Tham_số có thể như sau:

- Nguyên (xuất giá trị nguyên)
- Thực (xuất giá trị thực)
- ký tự - char (xuất một ký tự)
- con trỏ ký tự - char* (xuất chuỗi ký tự)

Chú ý: Các toán tử xuất có thể viết nối đuôi nhau (để xuất nhiều giá trị) trên một dòng lệnh như sau:

cout << Tham_số_1 << Tham_số_2 << ... << Tham_số_k ;

Chú ý: Toán tử xuất được định nghĩa chồng (trùng tên) với toán tử dịch trái và nó cùng có mức độ ưu tiên như toán tử dịch trái. Xem phụ lục 1 chúng ta thấy toán tử xuất có thứ tự ưu tiên lớn hơn các toán tử trong biểu thức điều kiện. Vì vậy nếu dùng toán tử xuất để in một biểu thức điều kiện như sau:

```
int a=5, b=10;  
cout << "\nMax= " << a>b?a:b ;
```

thì Trình biên dịch sẽ báo lỗi. Để tránh lỗi cần dùng các dấu ngoặc tròn để bao biểu thức điều kiện như sau:

```
int a=5, b=10;  
cout << "\nMax= " << (a>b?a:b) ;
```

Tóm lại: Nên bao các biểu thức trong 2 dấu ngoặc tròn.

4.3. Định dạng (tạo khuôn dạng cho) dữ liệu xuất

Việc định dạng dữ liệu xuất hay tạo khuôn dạng cho dữ liệu xuất là một việc cần thiết. Ví dụ cần in các giá trị thực trên 10 vị trí trong đó có 2 vị trí dành cho phần phân.

Bản thân toán tử xuất chưa có khả năng định dạng, mà cần sử dụng các công cụ sau:

- + Các phương thức định dạng

- + Các cờ định dạng
 - + Các hàm và bộ phận định dạng
- Mục sau sẽ trình bày cách định dạng giá trị xuất.

Bài 5. Các phương thức định dạng

5.1. Nội dung định dạng giá trị xuất

Nội dung định dạng là xác định các thông số:

- Độ rộng quy định
- Độ chính xác
- Ký tự độn
- Và các thông số khác

+ **Độ rộng thực tế của giá trị xuất:** Như đã nói ở trên, C++ sẽ biến đổi giá trị cần xuất thành một chuỗi ký tự rồi đưa chuỗi này ra màn hình. Ta sẽ gọi số ký tự của chuỗi này là độ rộng thực tế của giá trị xuất. Ví dụ với các câu lệnh:

```
int n=4567, m=-23 ;  
float x = -3.1416 ;  
char ht[] = "Tran Van Thong" ;
```

thì:

Độ rộng thực tế của n là 4, của m là 3, của x là 7, của ht là 14.

+ **Độ rộng quy định** là số vị trí tối thiểu trên màn hình dành để in giá trị. Theo mặc định, độ rộng quy định bằng 0. Chúng ta có thể dùng phương thức `cout.width()` để thiết lập rộng này. Ví dụ câu lệnh:

```
cout.width(8);
```

sẽ thiết lập độ rộng quy định là 8.

+ Mỗi quan hệ giữa độ rộng thực tế và độ rộng quy định

- Nếu độ rộng thực tế lớn hơn hoặc bằng độ rộng quy định thì số vị trí trên màn hình chứa giá trị xuất sẽ bằng độ rộng thực tế.

- Nếu độ rộng thực tế nhỏ hơn độ rộng quy định thì số vị trí trên màn hình chứa giá trị xuất sẽ bằng độ rộng quy định. Khi đó sẽ có một số vị trí dư thừa. Các vị trí dư thừa sẽ được độn (lấp đầy) bằng khoảng trống.

+ **Xác định ký tự độn:** Ký tự độn mặc định là dấu cách (khoảng trống). Tuy nhiên có thể dùng phương thức `cout.fill()` để chọn một ký tự độn khác. Ví dụ với các câu lệnh sau:

```
int n=123; // Độ rộng thực tế là 3  
cout.fill('*'); // Ký tự độn là *  
cout.width(5); // Độ rộng quy định là 5  
cout << n ;
```

thì kết quả in ra là:

```
**123
```

+ **Độ chính xác** là số vị trí dành cho phần phân (khi in số thực). Độ chính xác mặc định là 6. Tuy nhiên có thể dùng phương thức `cout.precision()` để chọn độ chính xác. Ví dụ với các câu lệnh:

```
float x = 34.455 ; // Độ rộng thực tế 6  
cout.precision(2) ; // Độ chính xác 2
```



```
cout.width(8);    // Độ rộng quy ước 8
cout.fill('0') ;  // Ký tự độn là số 0
cout << x ;
```

thì kết quả in ra là:

```
0034.46
```

5.2. Các phương thức định dạng

1. Phương thức

int cout.width()

cho biết độ rộng quy định hiện tại.

2. Phương thức

int cout.width(int n)

Thiết lập độ rộng quy định mới là n và trả về độ rộng quy định trước đó.

Chú ý: Độ rộng quy định n chỉ có tác dụng cho một giá trị xuất. Sau đó C++ lại áp dụng độ rộng quy định bằng 0.

Ví dụ với các câu lệnh:

```
int m=1234, n=56;
```

```
cout << "\nAB"
```

```
cout.width(6);
```

```
cout << m ;
```

```
cout << n ;
```

thì kết quả in ra là:

```
AB 123456
```

(giữa B và số 1 có 2 dấu cách).

3. Phương thức

int cout.precision()

Cho biết độ chính xác hiện tại (đang áp dụng để xuất các giá trị thức).

4. Phương thức

int cout.precision(int n)

Thiết lập độ chính xác sẽ áp dụng là n và cho biết độ chính xác trước đó. Độ chính xác được thiết lập sẽ có hiệu lực cho tới khi gặp một câu lệnh thiết lập độ chính xác mới.

5. Phương thức

char cout.fill()

Cho biết ký tự độn hiện tại đang được áp dụng.

6. Phương thức

char cout.fill(char ch)

Quy định ký tự độn mới sẽ được dùng là ch và cho biết ký tự độn đang dùng trước đó. Ký tự độn được thiết lập sẽ có hiệu lực cho tới khi gặp một câu lệnh chọn ký tự độn mới.

Ví dụ xét chương trình:

```
//CT7_06.CPP
```

```
// Cac phuong thuc dinh dang
#include <iostream.h>
#include <conio.h>
void main()
{
    clrscr();
    float x=-3.1551, y=-23.45421;
    cout.precision(2);
    cout.fill('*');
    cout << "\n" ;
    cout.width(8);
    cout << x;
    cout << "\n" ;
    cout.width(8);
    cout << y;
    getch();
}
```

Sau khi thực hiện, chương trình in ra màn hình 2 dòng sau:

***-3.16

** -23.45

Bài 6. Cờ định dạng

6.1. Khái niệm chung về cờ

Mỗi cờ chứa trong một bit. Cờ có 2 trạng thái:

Bật (on) - có giá trị 1

Tắt (off) - có giá trị 0

(Trong 6.3 sẽ trình bày các phương thức dùng để bật, tắt các cờ)

Các cờ có thể chứa trong một biến kiểu long. Trong tệp <iostream.h> đã định nghĩa các cờ sau:

ios::left	ios::right	ios::internal
ios::dec	ios::oct	ios::hex
ios::fixed	ios::scientific	ios::showpos
ios::uppercase	ios::showpoint	ios::showbase

6.2. Công dụng của các cờ

Có thể chia các cờ thành các nhóm:

Nhóm 1 gồm các cờ định vị (căn lề) :

ios::left ios::right ios::internal

Cờ ios::left: Khi bật cờ ios::left thì giá trị in ra nằm bên trái vùng quy định, các ký tự độn nằm sau, ví dụ:

35***

-89**

Cờ ios::right: Khi bật cờ ios:right thì giá trị in ra nằm bên phải vùng quy định, các ký tự đơn nằm trước, ví dụ:

```
***35
```

```
** -89
```

Chú ý: Mặc định cờ ios:right bật.

Cờ ios::internal: Cờ ios:internal có tác dụng giống như cờ ios:right chỉ khác là dấu (nếu có) in đầu tiên, ví dụ:

```
***35
```

```
._**89
```

Chương trình sau minh họa cách dùng các cờ định vị:

```
//CT7_06.CPP
// Các phương thức dinh dang
// Co dinh vi
#include <iostream.h>
#include <conio.h>
void main()
{
    clrscr();
    float x=-87.1551, y=23.45421;
    cout.precision(2);
    cout.fill('*');
    cout.setf(ios::left); // Bật cờ ios::left
    cout << "\n";
    cout.width(8);
    cout << x;
    cout << "\n";
    cout.width(8);
    cout << y;
    cout.setf(ios::right); // Bật cờ ios::right
    cout << "\n";
    cout.width(8);
    cout << x;
    cout << "\n";
    cout.width(8);
    cout << y;
    cout.setf(ios::internal); // Bật cờ ios::internal
    cout << "\n";
    cout.width(8);
    cout << x;
    cout << "\n";
    cout.width(8);
    cout << y;
```

```
    getch();  
}
```

Sau khi thực hiện chương trình in ra 6 dòng như sau:

```
-87.16**  
23.45***  
**-87.16  
***23.45  
-**-87.16  
***23.45
```

Nhóm 2 gồm các cờ định dạng số nguyên:

`ios::dec ios::oct ios::hex`

+ Khi `ios::dec` bật (mặc định): Số nguyên được in dưới dạng cơ số 10

+ Khi `ios::oct` bật : Số nguyên được in dưới dạng cơ số 8

+ Khi `ios::hex` bật : Số nguyên được in dưới dạng cơ số 16

Nhóm 3 gồm các cờ định dạng số thực:

`ios::fixed ios::scientific ios::showpoint`

Mặc định: Cờ `ios::fixed` bật (on) và cờ `ios::showpoint` tắt (off).

+ Khi `ios::fixed` bật và cờ `ios::showpoint` tắt thì số thực in ra dưới dạng thập phân, số chữ số phần phân (sau dấu chấm) được tính bằng độ chính xác `n` nhưng khi in thì bỏ đi các chữ số 0 ở cuối.

Ví dụ nếu độ chính xác `n = 4` thì:

Số thực -87.1500 được in: -87.15

Số thực 23.45425 được in: 23.4543

Số thực 678.0 được in: 678

+ Khi `ios::fixed` bật và cờ `ios::showpoint` bật thì số thực in ra dưới dạng thập phân, số chữ số phần phân (sau dấu chấm) được in ra đúng bằng độ chính xác `n`.

Ví dụ nếu độ chính xác `n = 4` thì:

Số thực -87.1500 được in: -87.1500

Số thực 23.45425 được in: 23.4543

Số thực 678.0 được in: 678.0000

+ Khi `ios::scientific` bật và cờ `ios::showpoint` tắt thì số thực in ra dưới dạng mũ (dạng khoa học). Số chữ số phần phân (sau dấu chấm) của phần định trị được tính bằng độ chính xác `n` nhưng khi in thì bỏ đi các chữ số 0 ở cuối.

Ví dụ nếu độ chính xác `n = 4` thì:

Số thực -87.1500 được in: -8.715e+01

Số thực 23.45425 được in: 2.3454e+01

Số thực 678.0 được in: 6.78e+02

+ Khi `ios::scientific` bật và cờ `ios::showpoint` bật thì số thực in ra dưới dạng mũ. Số chữ số phần phân (sau dấu chấm) của phần định trị được in đúng bằng độ chính xác `n`.

Ví dụ nếu độ chính xác `n = 4` thì:

Số thực -87.1500 được in: -8.7150e+01

Số thực 23.45425 được in: 2.3454e+01

Số thực 678.0 được in: 6.7800e+01

Nhóm 4 gồm các hiển thị:

`ios::showpos ios::showbase ios::uppercase`

Cờ `ios::showpos`

+ Nếu cờ `ios::showpos` tắt (mặc định) thì dấu cộng không được in trước số dương.

+ Nếu cờ `ios::showpos` bật thì dấu cộng được in trước số dương.

Cờ `ios::showbase`

+ Nếu cờ `ios::showbase` bật thì số nguyên hệ 8 được in bắt đầu bằng ký tự 0 và số nguyên hệ 16 được bắt đầu bằng các ký tự 0x. Ví dụ nếu `a = 40` thì:

dạng in hệ 8 là: 050

dạng in hệ 16 là 0x28

+ Nếu cờ `ios::showbase` tắt (mặc định) thì không in 0 trước số nguyên hệ 8 và không in 0x trước số nguyên hệ 16. Ví dụ nếu `a = 40` thì:

dạng in hệ 8 là: 50

dạng in hệ 16 là 28

Cờ `ios::uppercase`

+ Nếu cờ `ios::uppercase` bật thì các chữ số hệ 16 (như A, B, C, ...) được in dưới dạng chữ hoa.

+ Nếu cờ `ios::uppercase` tắt (mặc định) thì các chữ số hệ 16 (như A, B, C, ...) được in dưới dạng chữ thường.

6.3. Các phương thức bật tắt cờ

Các phương thức này định nghĩa trong lớp `ios`.

+ Phương thức

`long cout.setf(long f) ;`

sẽ bật các cờ liệt kê trong `f` và trả về một giá trị `long` biểu thị các cờ đang bật. Thông thường giá trị `f` được xác định bằng cách tổ hợp các cờ trình bày trong mục 6.1.

Ví dụ câu lệnh:

`cout.setf(ios::showpoint | ios::scientific) ;`

sẽ bật các cờ `ios::showpoint` và `ios::scientific`.

+ Phương thức

`long cout.unsetf(long f) ;`

sẽ tắt các cờ liệt kê trong `f` và trả về một giá trị `long` biểu thị các cờ đang bật. Thông thường giá trị `f` được xác định bằng cách tổ hợp các cờ trình bày trong mục 6.1.

Ví dụ câu lệnh:

`cout.unsetf(ios::showpoint | ios::scientific) ;`

sẽ tắt các cờ `ios::showpoint` và `ios::scientific`.

+ Phương thức

`long cout.flags(long f) ;`

có tác dụng giống như `cout.setf(long)`. Ví dụ câu lệnh:

```
cout.flags(ios::showpoint | ios::scientific) ;
```

sẽ bật các cờ ios::showpoint và ios::scientific.

+ Phương thức

```
long cout.flags() ;
```

sẽ trả về một giá trị long biểu thị các cờ đang bật.

Bài 7. Các bộ phận định dạng và các hàm định dạng

7.1. Các bộ phận định dạng (định nghĩa trong <iostream.h>)

Các bộ phận định dạng gồm:

```
dec // như cờ ios::dec
```

```
oct // như cờ ios::oct
```

```
hex // như cờ ios::hex
```

```
endl // xuất ký tự '\n' (chuyển dòng)
```

```
flush // đẩy dữ liệu ra thiết bị xuất
```

Chúng có tác dụng như cờ định dạng nhưng được viết nổi đuôi trong toán tử xuất nên tiện sử dụng hơn.

Ví dụ xét chương trình đơn giản sau:

```
//CT7_08.CPP
```

```
// Bộ phận định dạng
```

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

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

```
void main()
```

```
{
```

```
clrscr();
```

```
cout.setf(ios::showbase)
```

```
cout << "ABC" << endl << hex << 40 << " " << 41;
```

```
getch();
```

```
}
```

Chương trình sẽ in 2 dòng sau ra màn hình:

```
ABC
```

```
0x28 0x29
```

7.2. Các hàm định dạng (định nghĩa trong <iomanip.h>)

Các hàm định dạng gồm:

```
setw(int n) // như cout.width(int n)
```

```
setprecision(int n) // như cout.precision(int n)
```

```
setfill(char ch) // như cout.fill(char ch)
```

```
setiosflags(long l) // như cout.setf(long f)
```

```
resetiosflags(long l) // như cout.unsetf(long f)
```

Các hàm định dạng có tác dụng như các phương thức định dạng nhưng được viết nổi đuôi trong toán tử xuất nên tiện sử dụng hơn.

Chú ý 1: Các hàm định dạng (cũng như các bộ phận định dạng) cần viết trong các toán tử xuất. Một hàm định dạng đứng một mình như một câu lệnh sẽ không có tác dụng định dạng.

Chú ý 2: Muốn sử dụng các hàm định dạng cần bổ sung vào đầu chương trình câu lệnh:

#include <iomanip.h>

Ví dụ có thể thay phương thức

`cout.setf(ios::showbase);` ;

trong chương trình của mục 7.1 bằng hàm

`cout << setiosflags(ios::showbase);`

(chú ý hàm phải viết trong toán tử xuất)

Như vậy chương trình trong 7.1 có thể viết lại theo các phương án sau:

Phương án 1:

`#include <iostream.h>`

`#include <iomanip.h>`

`#include <conio.h>`

`void main()`

```
{
    clrscr();
    cout << setiosflags(ios::showbase);
    cout << "ABC" << endl << hex << 40 << " " << 41;
    getch();
}
```

Phương án 2:

`#include <iostream.h>`

`#include <iomanip.h>`

`#include <conio.h>`

`void main()`

```
{
    clrscr();
    cout << "ABC" << endl << setiosflags(ios::showbase)
    << hex << 40 << " " << 41;
    getch();
}
```

Dưới đây là ví dụ khác về việc dùng các hàm và bộ phận định dạng. Các câu lệnh:

`int i = 23;`

`cout << i << endl << setiosflags(ios::showbase)`

`<< hex << i << dec << setfill('*')`

`<< endl << setw(4) << i << setfill('0')`

`<< endl << setw(5) << i ;`

sẽ in ra màn hình như sau:

23

0x17

**23

00023

7.3. Ví dụ: Chương trình dưới đây minh hoạ cách dùng các hàm định dạng và phương thức định dạng để in danh sách thí sinh dưới dạng bảng với các yêu cầu sau: Số báo danh in 4 ký tự (chèn thêm số 0 vào trước ví dụ 0003), tổng điểm in với đúng một chữ số phần phân.

```
//CT7_08.CPP
// Bo phan dinh dang
// Ham dinh dang
// Co dinh dang
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
struct TS
{
    int sobd;
    char ht[25];
    float dt,dl,dh,td;
};
class TSINH
{
private:
    TS *ts;
    int sots;
public:
    TSINH()
    {
        ts=NULL;
        sots=0;
    }
    TSINH(int n)
    {
        ts=new TS[n+1];
        sots=n;
    }
    ~TSINH()
    {
        if (sots)
        {
            sots=0;
            ts = NULL;
        }
    }
}
```



```
void nhap();
void sapxep();
void xuat();
};
void TSINH::nhap()
{
    if (sots)
        for (int i=1; i<=sots; ++i)
        {
            cout << "\nThi sinh "<< i << ": ";
            cout << "\nSo bao danh: ";
            cin >> ts[i].sobd;
            cin.ignore();
            cout << "Ho ten: ";
            cin.get(ts[i].ht,25);
            cout << "Diem toan, ly , hoa: ";
            cin >> ts[i].dt >> ts[i].dl >> ts[i].dh;
            ts[i].td = ts[i].dt + ts[i].dl + ts[i].dh;
        }
}
void TSINH::sapxep()
{
    int i,j;
    for (i=1; i< sots; ++i)
        for (j=i+1; j<= sots; ++j)
            if (ts[i].td < ts[j].td)
            {
                TS tg;
                tg=ts[i];
                ts[i]=ts[j];
                ts[j]=tg;
            }
}
void TSINH::xuat()
{
    if (sots)
    {
        cout << "\nDanh sach thi sinh:";
        cout.precision(1);
        cout << setiosflags(ios::left);
```

```
cout << "\n" << setw(20) << "Ho ten" << setw(8)
    << "So BD" << setw(10) << "Tong diem";
for (int i=1; i<=sots; ++i)
    cout << "\n" << setw(20) << setiosflags(ios::left) << ts[i].ht
        << setw(4) << setfill('0') << setiosflags(ios::right)
        << ts[i].sobd << "    " << setfill(32)
        << setiosflags(ios::left|ios::showpoint)
        << setw(10) << ts[i].td;
    }
}
void main()
{
    int n;
    clrscr();
    cout << "\nSo thi sinh: ";
    cin >> n;
    TSINH *t = new TSINH(n);
    t->nhap();
    t->sapxep();
    t->xuat();
    getch();
    delete t;
}
```

Bài 8. Các dòng tin chuẩn

Có 4 dòng tin (đối tượng của các lớp Stream) đã định nghĩa trước, được cài đặt khi chương trình khởi động. Hai trong số đó đã nói ở trên là:

cin dòng input chuẩn gắn với bàn phím, giống như stdin của C.

cout dòng output chuẩn gắn với màn hình, giống như stdout của C.

Hai dòng tin chuẩn khác là:

cerr dòng output lỗi chuẩn gắn với màn hình, giống như stderr của C.

clog giống cerr nhưng có thêm bộ đệm.

Chú ý 1: Có thể dùng các dòng cerr và clog để xuất ra màn hình như đã dùng đối với cout.

Chú ý 2: Vì clog có thêm bộ đệm, nên dữ liệu được đưa vào bộ đệm. Khi đầy bộ đệm thì đưa dữ liệu từ bộ đệm ra dòng clog. Vì vậy trước khi kết thúc xuất cần dùng phương thức:

```
clog.flush();
```

để đẩy dữ liệu từ bộ đệm ra clog.

Chương trình sau minh họa cách dùng dòng clog. Chúng ta nhận thấy, nếu bỏ câu lệnh clog.flush() thì sẽ không nhìn thấy kết quả xuất ra màn hình khi chương trình tạm dừng bởi câu lệnh getch().

```
// Dùng clog và flush
#include <iostream.h>
#include <conio.h>
```

```
void main()
{
    clrscr();
    float x=-87.1500, y=23.45425, z=678.0;
    clog.setf(ios::scientific);
    clog.precision(4);
    clog.fill('*');
    clog << "\n";
    clog.width(10);
    clog << x;
    clog << "\n";
    clog.width(10);
    clog << y;
    clog << "\n";
    clog.width(10);
    clog << z;
    clog.flush();
    getch();
}
```

Bài 9. Xuất ra máy in

Trong số 4 dòng tin chuẩn không dòng nào gắn với máy in. Như vậy không thể dùng các dòng này để xuất dữ liệu ra máy in. Để xuất dữ liệu ra máy in (cũng như nhập, xuất trên tệp) cần tạo ra các dòng tin mới và cho nó gắn với thiết bị cụ thể. C++ cung cấp 3 lớp stream để làm điều này, đó là các lớp:

ifstream dùng để tạo dòng nhập

ofstream dùng để tạo dòng xuất

fstream dùng để tạo dòng nhập, dòng xuất hoặc dòng nhập-xuất

Mỗi lớp có 4 hàm tạo dùng để khai báo các dòng tin (đối tượng dòng tin). Trong mục sau sẽ nói thêm về các hàm tạo này.

Để tạo một dòng xuất và gắn nó với máy in ta có thể dùng một trong các hàm tạo sau:

```
ofstream Tên_dòng_tin(int fd) ;
```

```
ofstream Tên_dòng_tin(int fd, char *buf, int n) ;
```

Trong đó:

+ Tên_dòng_tin là tên biến đối tượng kiểu ofstream hay gọi là tên dòng xuất do chúng ta tự đặt.

+ fd (file descriptor) là chỉ số tệp tin. Chỉ số tệp tin định sẵn đối với stdprn (máy in chuẩn) là 4.

+ Các tham số buf và n xác định một vùng nhớ n byte do buf trỏ tới. Vùng nhớ sẽ được dùng làm bộ đệm cho dòng xuất.

Ví dụ 1 câu lệnh:

```
ofstream prn(4) ;
```

sẽ tạo dòng tin xuất prn và gắn nó với máy in chuẩn. Dòng prn sẽ có bộ đệm mặc định. Dữ liệu trước hết chuyển vào bộ đệm, khi đầy bộ đệm thì dữ liệu sẽ được đẩy từ bộ đệm ra dòng prn. Để chủ động yêu cầu đẩy dữ liệu từ bộ đệm ra dòng prn có thể sử dụng phương thức flush hoặc bộ phận định dạng flush. Cách viết như sau:

```
prn.flush(); // Phương thức  
prn << flush ; // Bộ phận định dạng
```

Các câu lệnh sau sẽ xuất dữ liệu ra prn (máy in) và ý nghĩa của chúng như sau:

```
prn << "\nTong = " << (4+9) ; // Đưa một dòng vào bộ đệm  
prn << "\nTich = " << (4*9); // Đưa tiếp dòng thứ 2 vào bộ đệm  
prn.flush(); // Đẩy dữ liệu từ bộ đệm ra máy in (in 2 dòng)
```

Các câu lệnh dưới đây cũng xuất dữ liệu ra máy in nhưng sẽ in từng dòng một:

```
prn << "\nTong = " << (4+9) << flush ; // In một dòng  
prn << "\nTich = " << (4*9) ; << flush // In dòng thứ hai
```

Ví dụ 2: Các câu lệnh

```
char buf[1000] ;  
ofstream prn(4,buf,1000) ;
```

sẽ tạo dòng tin xuất prn và gắn nó với máy in chuẩn. Dòng xuất prn sử dụng 1000 byte của mảng buf làm bộ đệm. Các câu lệnh dưới đây cũng xuất dữ liệu ra máy in:

```
prn << "\nTong = " << (4+9) ; // Đưa dữ liệu vào bộ đệm  
prn << "\nTich = " << (4*9) ; // Đưa dữ liệu vào bộ đệm  
prn.flush() ; // Xuất 2 dòng (ở bộ đệm) ra máy in
```

Chú ý: Trước khi kết thúc chương trình, dữ liệu từ bộ đệm sẽ được tự động đẩy ra máy in.

Chương trình minh họa: Chương trình dưới đây tương tự như chương trình trong mục 7.3 (chỉ sửa đổi phương thức xuất) nhưng thay việc xuất ra màn hình bằng xuất ra máy in.

```
//CT7_08B.CPP  
// Xuat ra may in  
// Bo phan dinh dang  
// Ham dinh dang  
#include <iostream.h>  
#include <iomanip.h>  
#include <conio.h>  
struct TS  
{  
    int sobd;  
    char ht[25];  
    float dt,dl,dh,td;  
};  
class TSINH  
{  
private:  
    TS *ts;
```

```
    int sots;
public:
    TSINH()
    {
        ts=NULL;
        sots=0;
    }
    TSINH(int n)
    {
        ts=new TS[n+1];
        sots=n;
    }
    ~TSINH()
    {
        if (sots)
        {
            sots=0;
            ts = NULL;
        }
    }
    void nhap();
    void sapxep();
    void xuat();
};

void TSINH::nhap()
{
    if (sots)
        for (int i=1; i<=sots; ++i)
        {
            cout << "\nThi sinh "<< i << ": ";
            cout << "\nSo bao danh: ";
            cin >> ts[i].sobd;
            cin.ignore();
            cout << "Ho ten: ";
            cin.get(ts[i].ht,25);
            cout << "Diem toan, ly, hoa: ";
            cin >> ts[i].dt >> ts[i].dl >> ts[i].dh;
            ts[i].td = ts[i].dt + ts[i].dl + ts[i].dh;
        }
}
```

```

    }
void TSINH::sapxep()
{
    int i,j;
    for (i=1; i< sots; ++i)
        for (j=i+1; j<= sots; ++j)
            if (ts[i].td < ts[j].td)
            {
                TS tg;
                tg=ts[i];
                ts[i]=ts[j];
                ts[j]=tg;
            }
}

void TSINH::xuat()
{
    ostream prn(4);
    if (sots)
    {
        prn << "\nDanh sach thi sinh:" ;
        prn.precision(1);
        prn << setiosflags(ios::left);
        prn << "\n" << setw(20) << "Ho ten" << setw(8)
            << "So BD" << setw(10) << "Tong diem";
        for (int i=1; i<=sots; ++i)
            prn << "\n" << setw(20) << setiosflags(ios::left) << ts[i].ht <<
                setw(4) << setfill('0') << setiosflags(ios::right) << ts[i].sobd
                << " " << setfill(32) << setiosflags(ios::left|ios::showpoint)
                << setw(10) << ts[i].td;
    }
}

void main()
{
    int n;
    clrscr();
    cout << "\nSo thi sinh: ";
    cin >> n;
    TSINH *t = new TSINH(n);
    t->nhap() ;
    t->sapxep();
    t->xuat();
}

```

```
    getch();  
    delete t;  
}
```

Bài 10. Làm việc với tệp

10.1. Các lớp dùng để nhập, xuất dữ liệu lên tệp

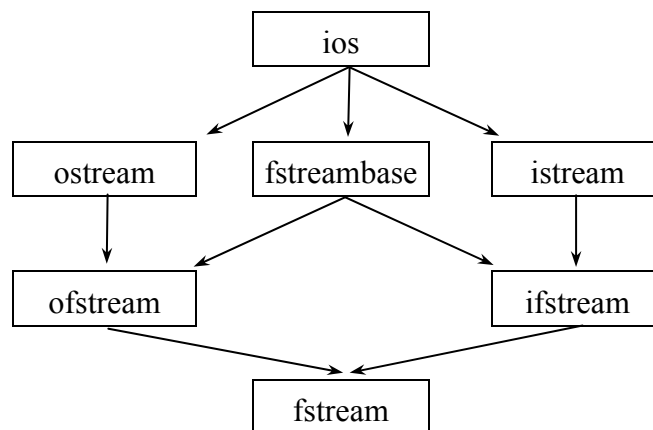
Như đã nói ở trên, C++ cung cấp 4 dòng tin chuẩn để làm việc với bàn phím và màn hình. Muốn nhập xuất lên tệp chúng ta cần tạo các dòng tin mới (khai báo các đối tượng Stream) và gắn chúng với một tệp cụ thể. C++ cung cấp 3 lớp stream để làm điều này, đó là các lớp:

ofstream dùng để tạo các dòng xuất (ghi tệp)

ifstream dùng để tạo các dòng nhập (đọc tệp)

fstream dùng để tạo các dòng nhập, dòng xuất hoặc dòng nhập-xuất

Sơ đồ dẫn xuất các lớp như sau:



10.2. Ghi dữ liệu lên tệp

Thủ tục ghi dữ liệu lên tệp như sau:

1. Dùng lớp ofstream để tạo ra một dòng xuất và gắn nó với một tệp cụ thể. Khi đó việc xuất dữ liệu ra dòng này đồng nghĩa với việc ghi dữ liệu lên tệp.

2. Thực hiện xuất dữ liệu ra dòng xuất vừa tạo như thể xuất dữ liệu ra dòng xuất chuẩn cout.

10.3. Đọc dữ liệu từ tệp

Thủ tục đọc dữ liệu từ tệp như sau:

1. Dùng lớp ifstream để tạo ra một dòng nhập và gắn nó với một tệp cụ thể. Khi đó việc nhập dữ liệu từ dòng này đồng nghĩa với việc đọc dữ liệu từ tệp.

2. Thực hiện nhập dữ liệu từ dòng nhập vừa tạo như thể nhập dữ liệu từ dòng nhập chuẩn cin.

10.4. Đọc - ghi dữ liệu đồng thời trên tệp

Thủ tục đọc-ghi dữ liệu đồng thời trên tệp như sau:

1. Dùng lớp fstream để tạo ra một dòng nhập-xuất và gắn nó với một tệp cụ thể.

2. Thực hiện nhập dữ liệu từ dòng nhập-xuất vừa tạo như thể nhập dữ liệu từ dòng nhập chuẩn cin.

3. Thực hiện xuất dữ liệu ra dòng nhập-xuất vừa tạo như thể xuất dữ liệu ra dòng xuất chuẩn cout.

Nhận xét: Như vậy:

1. Việc xuất dữ liệu ra máy in hoặc lên tệp được thực hiện hoàn toàn giống như xuất dữ liệu ra dòng xuất chuẩn cout (màn hình).
2. Việc đọc dữ liệu từ tệp được thực hiện hoàn toàn giống như nhập dữ liệu từ dòng nhập chuẩn cin (bàn phím).

Bài 11. Ghi dữ liệu lên tệp

11.1. Lớp ofstream

Để ghi dữ liệu lên tệp chúng ta sử dụng lớp ofstream. Lớp ofstream thừa kế các phương thức của các lớp ios và ostream. Nó cũng thừa kế phương thức:

close

của lớp fstreambase. Ngoài ra lớp ofstream có thêm các hàm tạo và các phương thức sau:

1. Hàm tạo:

ofstream() ; // Không đối

dùng để tạo một đối tượng ofstream (dòng xuất), chưa gắn với tệp.

2. Hàm tạo:

ofstream(const char *fn, int mode = ios::out,

int prot = filebuf::openprot); dùng để tạo một đối tượng ofstream, mở tệp có tên fn để ghi và gắn đối tượng vừa tạo với tệp được mở.

+ Tham số fn cho biết tên tệp.

+ Tham số mode có giá trị mặc định là ios::out (mở để ghi). Tham số này có thể là một hợp của các giá trị sau:

ios::binary ghi theo kiểu nhị phân (mặc định theo kiểu văn bản)

ios::out ghi tệp, nếu tệp đã có thì nó bị xóa

ios::app ghi bổ sung vào cuối tệp

ios::ate chuyển con trỏ tệp tới cuối tệp sau khi mở tệp

ios::trunc xóa nội dung của tệp nếu nó tồn tại

ios::nocreate nếu tệp chưa có thì không làm gì (bỏ qua)

ios::noreplace nếu tệp đã có thì không làm gì (bỏ qua)

+ Tham số thứ ba prot quy định cấp bảo vệ của dòng tin, tham số này có thể bỏ qua vì nó đã được gán một giá trị mặc định.

3. Hàm tạo:

ofstream(int fd);

dùng để tạo một đối tượng ofstream và gắn nó với một tệp có chỉ số fd đang mở.

(Để mở và lấy chỉ số (số hiệu) tệp có thể dùng hàm _open, xem cuốn Kỹ thuật Lập trình C của tác giả).

4. Hàm tạo:

ofstream(int fd, char *buf, int n);

dùng để tạo một đối tượng ofstream, gắn nó với một tệp có chỉ số fd đang mở và sử dụng một vùng nhớ n byte do buf trỏ tới làm bộ đệm.

5. Phương thức:

void open(const char *fn, int mode = ios::out,

int prot = filebuf::openprot); dùng để mở tệp có tên fn để ghi và gắn nó với đối tượng ofstream. Các tham số của phương thức có cùng ý nghĩa như trong hàm tạo thứ 2.

11.2. Các cách ghi tệp

Có 2 cách chính sau:

+ *Cách 1:* Dùng hàm tạo 2 để xây dựng một dòng xuất, mở một tệp để ghi và gắn tệp với dòng xuất. Sau đó dùng toán tử xuất << và các phương thức để xuất dữ liệu ra dòng xuất vừa tạo như thể xuất dữ liệu ra cout (xem các mục trên).

+ *Cách 2:* Dùng hàm tạo 1 để xây dựng một dòng xuất. Sau đó dùng phương thức open để mở một tệp cụ thể và cho gắn với dòng xuất vừa xây dựng. Khi không cần làm việc với tệp này nữa, chúng ta có thể dùng phương thức close để chấm dứt mọi ràng buộc giữa dòng xuất và tệp. Sau đó có thể gắn dòng xuất với tệp khác. Theo cách này, có thể dùng một dòng xuất (đối tượng ofstream) để xuất dữ liệu lên nhiều tệp khác nhau.

11.3. Ví dụ

Chương trình 1: Chương trình dưới đây sẽ nhập danh sách n thí sinh. Thông tin thí sinh gồm: Họ tên, tỉnh hoặc thành phố cư trú, số báo danh, các điểm toán lý hoá. Dữ liệu thí sinh được ghi trên 2 tệp: Tệp DS1.DL ghi thí sinh theo thứ tự nhập từ bàn phím, tệp DS2.DL ghi thí sinh theo thứ tự giảm của tổng điểm. Cấu trúc của 2 tệp như sau:

Dòng đầu ghi một số nguyên bằng số thí sinh.

Các dòng tiếp theo ghi dữ liệu của thí sinh. Mỗi thí sinh ghi trên 2 dòng, dòng 1 ghi họ tên trên 24 vị trí và tên tỉnh trên 20 vị trí. Dòng 2 ghi số báo danh (6 vị trí), các điểm toán, lý, hoá và tổng điểm (mỗi điểm ghi trên 6 vị trí trong đó một vị trí chứa phần phân). Chương trình sử dụng lớp TS (Thí sinh) có 3 phương thức: Nhập, sắp xếp và ghi tệp. Cách ghi tệp sử dụng ở đây là cách 1: Dùng hàm tạo dạng 2 của lớp ofstream.

Chương trình 2 ngay bên dưới cũng giải quyết cùng bài toán nêu trên nhưng sử dụng cách ghi tệp thứ 2 (dùng hàm tạo 1 và phương thức open)

Một điều đáng nói ở đây là việc nhập một chuỗi ký tự (như họ tên và tên tỉnh) bằng các phương thức get hoặc getline chưa được thuận tiện, vì 2 lý do sau: thứ nhất là các phương thức này có thể bị ký tự chuyển dòng (còn sót trên cin) làm trôi. Thứ hai là các phương thức này có thể để lại một số ký tự trên dòng cin (nếu số ký tự gõ nhiều hơn so với quy định) và các ký tự này sẽ gây ảnh hưởng đến các phép nhập tiếp theo. Để khắc phục các nhược điểm trên, chúng ta đưa vào 2 chương trình trên hàm getstr để nhập chuỗi ký tự từ bàn phím.

```
//CT7_10.CPP
// Ghi Tệp
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
void getstr(char *str,int n)
{
    char tg[21];
    while(1) // Bỏ qua Enter và nhập tối đa n-1 ký tự
```

```
{
    cin.get(str,n);
    if (str[0])
        break;
    else
        cin.ignore();
}
while(1) // Loại các ký tự còn lại ra khỏi dòng nhập cin
{
    cin.get(tg,20);
    if (tg[0]==0)
    {
        cin.ignore();
        break;
    }
}
}
```

struct TSINH

```
{
    char ht[25];
    char ttinh[21];
    int sobd;
    float dt,dl,dh,td;
};

class TS
{
    private:
        int sots;
        TSINH *ts;
    public:
        TS()
        {
            sots=0;
            ts = NULL;
        }
        void nhap();
        void sapxep();
        void ghitep(char *ttep);
};
```

```
void TS::nhap()
{
    cout << "\n So thi sinh: ";
    cin >> sots;
    int n=sots;
    ts = new TSINH[n+1];
    for (int i=1; i<=n; ++i)
    {
        cout << "\n Nhap thi sinh thu: " << i << endl;
        cout << "Ho ten: ";
        getstr(ts[i].ht,25);
        cout << "Tinh hoac thanh pho: ";
        getstr(ts[i].tinh,21);
        cout << "So bao danh: ";
        cin >> ts[i].sobd;
        cout << "Cac diem toan, ly, hoa: ";
        cin >> ts[i].dt >> ts[i].dl >> ts[i].dh;
        ts[i].td =ts[i].dt + ts[i].dl + ts[i].dh;
    }
}

void TS::sapxep()
{
    int n = sots;
    for (int i=1; i< n; ++i)
        for (int j=i+1; j<= n; ++j)
            if (ts[i].td < ts[j].td)
            {
                TSINH tg = ts[i];
                ts[i] = ts[j];
                ts[j] = tg;
            }
}

void TS::ghitep(char *ttep)
{
    ofstream f(ttep);
    f<< sots;
    f<< setprecision(1) << setiosflags(ios::showpoint);
    for (int i=1; i<=sots; ++i)
    {
        f<< endl << setw(24) << ts[i].ht << setw(20) << ts[i].tinh;
```

```
f << endl << setw(6) << ts[i].sobd
    << setw(6) << ts[i].dt
    << setw(6) << ts[i].dl
    << setw(6) << ts[i].dh
    << setw(6) << ts[i].td ;
}
f.close();
}

void main()
{
    clrscr();
    TS t;
    t.nhap();
    t.ghitep("DS1.DL");
    t.sapxep();
    t.ghitep("DS2.DL");
    cout << "\n Hoan thanh";
    getch();
}
```

Chương trình 2:

```
//CT7_11.CPP
// Ghi Tep
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
void getstr(char *str,int n)
{
    char tg[21];
    while(1)
    {
        cin.get(str,n);
        if (str[0])
            break;
        else
            cin.ignore();
    }
}
```

```
while(1)
{
    cin.get(tg,20);
    if (tg[0]==0)
    {
        cin.ignore();
        break;
    }
}
}

struct TSINH
{
    char ht[25];
    char ttinh[21];
    int sobd;
    float dt,dl,dh,td;
};

class TS
{
private:
    int sots;
    TSINH *ts;
public:
    TS()
    {
        sots=0;
        ts = NULL;
    }
    void nhap();
    void sapxep();
    void ghitep(char *ttep);
};

void TS::nhap()
{
    cout << "\n So thi sinh: ";
    cin >> sots ;
    int n=sots;
    ts = new TSINH[n+1];
    for (int i=1; i<=n; ++i)
    {
```

```
cout << "\n Nhap thi sinh thu: " << i << endl;
cout << "Ho ten: ";
getstr(ts[i].ht,25);
cout << "Tinh hoac thanh pho: ";
getstr(ts[i].tinh,21);
cout << "So bao danh: ";
cin >> ts[i].sobd;
cout << "Cac diem toan, ly, hoa: ";
cin >> ts[i].dt >> ts[i].dl >> ts[i].dh;
ts[i].td = ts[i].dt + ts[i].dl + ts[i].dh;
}
}
void TS::sapxep()
{
    int n = sots;
    for (int i=1; i< n; ++i)
        for (int j=i+1; j<= n; ++j)
            if (ts[i].td < ts[j].td)
            {
                TSINH tg = ts[i];
                ts[i] = ts[j];
                ts[j] = tg;
            }
}
void TS::ghitep(char *ttep)
{
    ofstream f;
    f.open(ttep,ios::out|ios::noreplace);
    if (f.bad())
    {
        cout << "\nTep " << ttep << " da ton tai";
        cout << "\nCo ghi de? - C/K";
        int ch=getch();
        if (toupper(ch)=='C')
        {
            f.close();
            f.open(ttep);
        }
        else
            exit(1);
    }
}
```

```
    }
    f << sots ;
    f << setprecision(1) << setiosflags(ios::showpoint);
    for (int i=1; i<=sots; ++i)
    {
        f << endl << setw(24) << ts[i].ht << setw(20) << ts[i].ttinh ;
        f << endl << setw(6) << ts[i].sobd
        << setw(6) << ts[i].dt
        << setw(6) << ts[i].dl
        << setw(6) << ts[i].dh
        << setw(6) << ts[i].td ;
    }
    f.close();
}

void main()
{
    clrscr();
    TS t;
    t.nhap();
    t.ghitep("DS1.DL");
    t.sapxep();
    t.ghitep("DS2.DL");
    cout << "\n Hoan thanh";
    getch();
}
```

Bài 12. Đọc dữ liệu từ tệp

12.1. Lớp ifstream

Để đọc dữ liệu từ tệp chúng ta sử dụng lớp ifstream. Lớp ifstream thừa kế các phương thức của các lớp ios và istream. Nó cũng thừa kế phương thức:

close

của lớp fstreambase. Ngoài ra lớp ifstream có thêm các hàm tạo và các phương thức sau:

1. Hàm tạo:

ifstream() ; // Không đối

dùng để tạo một đối tượng ifstream (dòng nhập), chưa gắn với tệp.

2. Hàm tạo:

**ifstream(const char *fn, int mode = ios::in,
int prot = filebuf::openprot);**

dùng để tạo một đối tượng ifstream, mở tệp có tên fn để đọc và gắn đối tượng vừa tạo với tệp được mở.

- + Tham số `fn` cho biết tên tệp.
- + Tham số mode có giá trị mặc định là `ios::in` (mở để đọc). Tham số này có thể là một hợp của các giá trị sau:

- `ios::binary` đọc theo kiểu nhị phân (mặc định theo kiểu văn bản)

- `ios::ate` chuyển con trỏ tệp tới cuối tệp sau khi mở tệp

- + Tham số thứ ba `prot` quy định cấp bảo vệ của dòng tin, tham số này có thể bỏ qua vì nó đã được gán một giá trị mặc định.

3. Hàm tạo:

`ifstream(int fd);`

dùng để tạo một đối tượng `ifstream` và gắn nó với một tệp có chỉ số `fd` đang mở.

(Để mở và lấy chỉ số (số hiệu) tệp có thể dùng hàm `_open`, xem cuốn Kỹ thuật Lập trình C của tác giả)

4. Hàm tạo:

`ifstream(int fd, char *buf, int n);`

dùng để tạo một đối tượng `ifstream`, gắn nó với một tệp có chỉ số `fd` đang mở và sử dụng một vùng nhớ `n` byte do `buf` trỏ tới làm bộ đệm.

5. Phương thức:

**`void open(const char *fn, int mode = ios::in,
int prot = filebuf::openprot);`**

dùng để mở tệp có tên `fn` để đọc và gắn nó với đối tượng `ifstream`. Các tham số của phương thức có cùng ý nghĩa như trong hàm tạo thứ 2.

12.2. Các cách đọc tệp

Có 2 cách chính sau:

- + *Cách 1*: Dùng hàm tạo 2 để xây dựng một dòng nhập, mở một tệp để đọc và gắn tệp với dòng nhập. Sau đó dùng toán tử nhập `>>` và các phương thức để nhập dữ liệu từ dòng nhập vừa tạo như thể nhập dữ liệu từ `cin` (xem các mục trên)

- + *Cách 2*: Dùng hàm tạo 1 để xây dựng một dòng nhập. Sau đó dùng phương thức `open` để mở một tệp cụ thể và cho gắn với dòng nhập vừa xây dựng. Khi không cần làm việc với tệp này nữa, chúng ta có thể dùng phương thức `close` để chấm dứt mọi ràng buộc giữa dòng nhập và tệp. Sau đó có thể gắn dòng nhập với tệp khác. Theo cách này, có thể dùng một dòng nhập (đối tượng `ifstream`) để nhập dữ liệu từ nhiều tệp khác nhau.

12.3. Kiểm tra sự tồn tại của tệp, kiểm tra cuối tệp

- + Khi mở một tệp để đọc mà tệp không tồn tại thì sẽ phát sinh lỗi, khi đó phương thức `bad` trả về giá trị khác không. Ví dụ để kiểm tra xem tệp `DSTS` (Danh sách thí sinh) đã tồn tại hay không có thể dùng đoạn chương trình:

```
ifstream fin("DSTS");
if (fin.bad())
{
    cout << "\nTep DSTS không tồn tại";
    exit(1);
}
```

- + Trong quá trình đọc, con trỏ tệp sẽ chuyển dần về cuối tệp. Khi con trỏ tệp đã ở cuối tệp (hết dữ liệu) mà vẫn thực hiện một lệnh đọc thì phương thức `eof` sẽ cho giá trị khác không. Chương trình

dưới đây dùng phương thức eof để xác định độ dài (số byte) của tệp TC.EXE (chú ý cần dùng kiểu đọc nhị phân):

```
//CT7_14.CPP
// Do dai tep
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
    clrscr();
    long dd=0;    char ch;
    ifstream f("TC.EXE",ios::in | ios::binary);
    if (f.bad())
    {
        cout << "\nTep TC.EXE khong ton tai";
        getch();
        exit(1);
    }
    while(f.get(ch),!f.eof()) ++dd;
    cout << "\n Do dai TC.EXE: " << dd;
    getch();
}
```

12.4. Ví dụ

Chương trình dưới đây sẽ:

- + Đọc danh sách thí sinh từ tệp DS1.DL do chương trình trong mục bài 11 tạo ra.
- + In danh sách thí sinh vừa đọc.
- + Sắp xếp dãy thí sinh (vừa nhập từ tệp) theo thứ tự giảm của tổng điểm.
- + Ghi danh sách thí sinh sau khi sắp xếp lên tệp DS3.DL
- + Đọc danh sách thí sinh từ tệp DS3.DL
- + In danh sách thí sinh đọc từ tệp DS3.DL

Chương trình sử dụng lớp TS (Thí sinh) có 4 phương thức:

```
void xuat();
void sapxep();
void ghitep(char *ttep);
void doctep(char *ttep);
//CT7_12.CPP
// Doc tep
#include <iostream.h>
#include <iomanip.h>
```

```
#include <fstream.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
struct TSINH
{
    char ht[25];
    char ttinh[21];
    int sobd;
    float dt,dl,dh,td;
};
class TS
{
private:
    int sots;
    TSINH *ts;
public:
    TS()
    {
        sots=0;
        ts = NULL;
    }
    void xuat();
    void sapxep();
    void ghitep(char *ttep);
    void doctep(char *ttep);
};
void TS::xuat()
{
    cout << "\n\nSo thi sinh: " << sots;
    cout << setprecision(1) << setiosflags(ios::showpoint);
    for (int i=1; i<=sots; ++i)
    {
        cout << "\nThi sinh thu: " << i ;
        cout << "\nHo ten: " << ts[i].ht ;
        cout << "\nTinh - thanh pho: " << ts[i].ttinh ;
        cout << "\nSo bao danh: " << ts[i].sobd ;
        cout << "\nCac diem toan, ly, hoa: "
            << setw(5) << ts[i].dt
            << setw(5) << ts[i].dl
```

```

        << setw(5) << ts[i].dh ;
        cout << "\nTong diem: " << ts[i].td ;
    }
}
void TS::sapxep()
{
    int n = sots;
    for (int i=1; i< n; ++i)
        for (int j=i+1; j<= n; ++j)
            if (ts[i].td < ts[j].td)
            {
                TSINH tg = ts[i];
                ts[i] = ts[j];
                ts[j] = tg;
            }
}
void TS::ghitep(char *ttep)
{
    ofstream f;
    f.open(ttep,ios::out|ios::noreplace);
    if (f.bad())
    {
        cout << "\nTep " << ttep << " da ton tai";
        cout << "\nCo ghi de? - C/K";
        int ch=getch();
        if (toupper(ch)=='C')
        {
            f.close();
            f.open(ttep) ;
        }
        else
            exit(1);
    }
    f<< sots ;
    f<< setprecision(1) << setiosflags(ios::showpoint);
    for (int i=1; i<=sots; ++i)
    {
        f<< endl << setw(24) << ts[i].ht << setw(20) << ts[i].ttinh ;
        f<< endl << setw(6) << ts[i].sobd
        << setw(6) << ts[i].dt
    }
}

```

```
<< setw(6) << ts[i].dl
<< setw(6) << ts[i].dh
<< setw(6) << ts[i].td ;
}
f.close();
}
void TS::doctep(char *ttep)
{
    ifstream f;
    f.open(ttep);
    if (f.bad())
    {
        cout << "\nTep " << ttep << " khong ton tai";
        getch();
        exit(1);
    }
    f>> sots ;
    f.ignore();
    if (ts!=NULL) delete ts;
    ts = new TSINH[sots+1];
    for (int i=1; i<=sots; ++i)
    {
        f.get(ts[i].ht,25).get(ts[i].ttinh,21); ;
        f>> ts[i].sobd >> ts[i].dt >> ts[i].dl
        >> ts[i].dh >> ts[i].td ;
        f.ignore();
    }
    f.close();
}
void main()
{
    clrscr();
    TS t;
    t.doctep("DS1.DL");
    t.xuat();
    t.sapxep();
    t.ghitep("DS3.DL");
    t.doctep("DS3.DL");
    t.xuat();
    cout << "\n Hoan thanh";
    getch();
}
```

}

Bài 13. Đọc ghi đồng thời trên tệp

13.1. Lớp fstream

Để đọc ghi đồng thời trên tệp, chúng ta sử dụng lớp fstream. Lớp fstream thừa kế các phương thức của các lớp ofstream và ifstream. Ngoài ra lớp fstream có các hàm tạo và phương thức sau:

1. Hàm tạo:

fstream() ; // Không đối

dùng để tạo một đối tượng fstream (dòng nhập-xuất), chưa gắn với tệp.

2. Hàm tạo:

**fstream(const char *fn, int mode,
int prot = filebuf::openprot);**

dùng để tạo một đối tượng fstream, mở tệp có tên fn và gắn đối tượng vừa tạo với tệp được mở.

+ Tham số fn cho biết tên tệp.

+ Tham số mode quy định các kiểu truy nhập và có thể là tổ hợp của các giá trị sau:

ios::binary đọc-ghi theo kiểu nhị phân (mặc định theo kiểu văn bản).

ios::out ghi tệp, nếu tệp đã có thì nó bị xóa

ios::in đọc tệp

ios::app ghi bổ sung vào cuối tệp

ios::ate chuyển con trỏ tệp về cuối sau khi mở

ios::trunc xóa nội dung của tệp nếu nó tồn tại

ios::nocreate nếu tệp chưa có thì không làm gì (bỏ qua)

ios::noreplace nếu tệp đã có thì không làm gì (bỏ qua)

Chú ý:

+ Tham số mode không có giá trị mặc định.

+ Tham số thứ ba prot quy định cấp bảo vệ của dòng tin, tham số này có thể bỏ qua vì nó đã được gán một giá trị mặc định.

3. Hàm tạo:

fstream(int fd);

dùng để tạo một đối tượng fstream và gắn nó với một tệp có chỉ số fd đang mở.

(Để mở và lấy chỉ số (số hiệu) tệp có thể dùng hàm _open, xem cuốn Kỹ thuật Lập trình C của tác giả)

4. Hàm tạo:

fstream(int fd, char *buf, int n);

dùng để tạo một đối tượng fstream, gắn nó với một tệp có chỉ số fd đang mở và sử dụng một vùng nhớ n byte do buf trỏ tới làm bộ đệm.

5. Phương thức:

**void open(const char *fn, int mode,
int prot = filebuf::openprot);**

dùng để mở tệp có tên fn và gắn nó với đối tượng fstream. Các tham số của phương thức có cùng ý nghĩa như trong hàm tạo thứ 2.

Chú ý: Tham số mode không có giá trị mặc định.

13.2. Các cách đọc-ghi đồng thời trên tệp

Có 2 cách chính sau:

+ **Cách 1:** Dùng hàm tạo 2 để xây dựng một dòng nhập-xuất, mở một tệp để đọc-ghi và gắn tệp với dòng nhập-xuất. Sau đó dùng toán tử nhập >> , toán tử xuất << và các phương thức nhập, xuất để nhập, xuất dữ liệu ra dòng nhập-xuất vừa tạo (như đối với các dòng chuẩn cin và cout). Ví dụ:

```
fstream f("DU_LIEU", ios::in | ios::out) ;
```

+ **Cách 2:** Dùng hàm tạo 1 để xây dựng một dòng nhập-xuất. Sau đó dùng phương thức open để mở một tệp cụ thể (để đọc và ghi) và cho gắn với dòng nhập-xuất vừa xây dựng. Khi không cần làm việc với tệp này nữa, chúng ta có thể dùng phương thức close để chấm dứt mọi ràng buộc giữa dòng nhập-xuất và tệp. Sau đó có thể gắn dòng nhập-xuất với tệp khác. Theo cách này, có thể dùng một dòng nhập-xuất (đối tượng fstream) để đọc-ghi dữ liệu từ nhiều tệp khác nhau.

Ví dụ:

```
fstream f;  
f.open("DU_LIEU", ios::in | ios::out) ;
```

13.3. Di chuyển con trỏ tệp

13.3.1. Để di chuyển con trỏ tệp trên dòng xuất, chúng ta sử dụng các phương thức sau (của lớp ostream) :

1. Phương thức

ostream& seekp(long n) ;

sẽ chuyển con trỏ tệp tới vị trí (byte) thứ n (số thứ tự các byte tính từ 0).

2. Phương thức

ostream& seekp(long offset, seek_dir dir) ;

sẽ chuyển con trỏ tệp tới vị trí **offset** kể từ vị trí xuất phát **dir**. Giá trị của **offset** có thể âm, còn **dir** có thể nhận một trong các giá trị sau:

ios::beg xuất phát từ đầu tệp

ios::end xuất phát từ cuối tệp

ios::cur xuất phát từ vị trí hiện tại của con trỏ tệp

3. Phương thức

long teelp() ;

cho biết vị trí hiện tại của con trỏ tệp.

13.3.2. Để di chuyển con trỏ tệp trên dòng nhập, chúng ta sử dụng các phương thức sau (của lớp istream):

4. Phương thức

istream& seekg(long n) ;

sẽ chuyển con trỏ tệp tới vị trí (byte) thứ n (số thứ tự các byte tính từ 0)

5. Phương thức

istream& seekg(long offset, seek_dir dir) ;

sẽ chuyển con trỏ tệp tới vị trí **offset** kể từ vị trí xuất phát **dir**. Giá trị của **offset** có thể âm, còn **dir** có thể nhận một trong các giá trị sau:

ios::beg xuất phát từ đầu tệp
ios::end xuất phát từ cuối tệp
ios::cur xuất phát vị trí hiện tại của con trỏ tệp

6. Phương thức

long teelg() ;

cho biết vị trí hiện tại của con trỏ tệp.

13.3.3. Để di chuyển con trỏ tệp trên dòng nhập-xuất, chúng ta có thể sử dụng cả 6 phương thức nêu trên.

13.4. Ví dụ

Ví dụ 1. Trong §12 đã viết chương trình xác định độ dài của tệp TC.EXE. Dưới đây là một phương án khác đơn giản hơn:

```
fstream f("TC.EXE");  
f.seekg(0,ios::end);  
do_dai = f.teelg();
```

Ví dụ 2. Chương trình dưới đây sẽ nhập danh sách n thí sinh từ bàn phím và ghi lên tệp. Sau đó đưa con trỏ tệp về đầu tệp và bắt đầu đọc dữ liệu thí sinh từ tệp để in ra màn hình. Thông tin thí sinh gồm: Họ tên, tỉnh hoặc thành phố cư trú, số báo danh, các điểm toán lý hoá.

```
//CT7_13.CPP  
// ghi - đọc đồng thời  
#include <iostream.h>  
#include <iomanip.h>  
#include <fstream.h>  
#include <conio.h>  
#include <stdlib.h>  
#include <ctype.h>  
#include <stdio.h>  
void main()  
{  
    char ht[25], ttinh[21], ttep[40];  
    int sobd, stt ;  
    float dt, dl, dh, td;  
    fstream f;  
    cout << "\nTen tep: " ;  
    cin >> ttep;  
    f.open(ttep, ios::out|ios::in|ios::noreplace);  
    if (f.bad())  
    {  
        cout << "\nTep " << ttep << " da ton tai";  
        cout << "\nCo ghi de? - C/K";  
        int ch=getch();  
        if (toupper(ch)=='C')
```

```
{
    f.close();
    f.open(ttep,ios::out|ios::in|ios::trunc) ;
}
else
    exit(1);
}
stt=0 ;
f<< setprecision(1) << setiosflags(ios::showpoint);
while(1)
{
    ++stt;
    cout << "\nNhap thi sinh thu: " << stt ;
    cout << "\nHo ten (neu rong thi ket thuc nhap) : ";
    cin.ignore();
    cin.getline(ht,25);
    if (ht[0]==0) break;
    cout << "Tinh - thanh pho: ";
    cin.getline(ttinh,21);
    cout << "SoBD, diem toan, diem ly, diem hoa: " ;
    cin >> sobd >> dt >> dl >> dh ;
    td = dt + dl + dh ;
    if (stt>1) f<< endl;
    f<< setw(24) << ht << setw(20) << ttinh ;
    f<< endl << setw(6) << sobd
        << setw(6) << dt
        << setw(6) << dl
        << setw(6) << dh
        << setw(6) << td ;
}
f.seekg(0);
stt=0;
clrscr();
cout << "\nDanh sach thi sinh\n";
cout << setprecision(1) <<
setiosflags(ios::showpoint);
while(1)
{
    f.getline(ht,25).getline(ttinh,21);
    if (f.eof()) break;
    ++stt;
```



```
f>> sobd >> dt >> dl >> dh >> td;
f.ignore();
cout << "\nThi sinh thu: " << stt ;
cout << "\nHo ten: " << ht;
cout << "\nTinh - thanh pho: " << ttinh;
cout << "\nSo bao danh: " << sobd;
cout << "\nDiem toan, ly, hoa va tong diem: "
    <<setw(6)<< dt << setw(6) <<dl << setw(6) << dh
    << setw(6) << td ;
}
f.close();
cout << "\n Hoan thanh";
getch();
}
```

Bài 14. Xử lý lỗi

Khi làm việc với tệp không phải mọi việc đều trôi chảy mà thường xảy ra nhiều điều trục trặc, chẳng hạn:

1. Mở một tệp để đọc nhưng tệp không tồn tại.
2. Đọc dữ liệu nhưng con trỏ tệp đã ở cuối tệp
3. Ghi dữ liệu nhưng hết không gian đĩa (đĩa đầy).
4. Tạo tệp nhưng đĩa hỏng, hoặc đĩa cấm ghi hoặc đĩa đầy.
5. Dùng tên tệp không hợp lệ
6. Định thực hiện một thao tác nhưng tệp lại không được mở ở mode phù hợp để thực hiện thao tác đó.

Tóm lại khi làm việc với tệp thường gặp nhiều lỗi khác nhau, nếu không biết cách phát hiện xử lý thì chương trình sẽ dẫn đến rối loạn hoặc cho kết quả sai. Trong lớp ios của C++ có nhiều phương thức cho phép phát hiện lỗi khi làm việc với tệp. Đó là:

1. Phương thức

int eof() ;

Nếu con trỏ tệp đã ở cuối tệp mà lại thực hiện một lệnh đọc dữ liệu thì phương thức eof() trả về giá trị khác không, trái lại phương thức có giá trị bằng 0.

2. Phương thức

int fail() ;

Phương thức fail() trả về giá trị khác không nếu lần nhập xuất cuối cùng có lỗi, trái lại phương thức có giá trị bằng 0.

3. Phương thức

int bad() ;

Phương thức `bad()` trả về giá trị khác không khi một phép nhập xuất không hợp lệ hoặc có lỗi mà chưa phát hiện được, trái lại phương thức có giá trị bằng 0.

4. Phương thức

`int good() ;`

Phương thức `good()` trả về giá trị khác không nếu mọi việc đều tốt đẹp (không có lỗi nào xảy ra). Khi có một lỗi nào đó thì phương thức có giá trị bằng 0.

5. Phương thức

`void clear() ;`

dùng để tắt tất cả các bit lỗi.

Ví dụ 1. Khi mở tệp để đọc cần kiểm tra xem tệp có tồn tại không. Để làm điều đó, chúng ta có thể dùng đoạn chương sau:

```
char ten_tep[40] ;
cout << "\n Cho biết tên tệp: " ;
cin >> ten_tep ;
ifstream f(ten_tep);
if (f.bad())
{
    cout << "\n Tệp << ten_tep << "không tồn tại" ;
    exit(1) ;
}
```

Ví dụ 2. Khi tạo tệp mới để ghi cần kiểm tra xem có tạo được tệp hay không. Để làm điều đó, chúng ta có thể dùng đoạn chương sau:

```
char ten_tep[40] ;
cout << "\n Cho biết tên tệp: " ;
cin >> ten_tep ;
ofstream f(ten_tep);
if (f.bad())
{
    cout << "\n Không tạo được tệp << ten_tep ;
    exit(1) ;
}
```

Ví dụ 3. Để xác định độ dài của tệp, có thể dùng phương thức `eof()` và thuật toán sau:

+ Đọc một byte (chú ý phải đọc theo kiểu nhị phân)

+ Nếu việc đọc thành công (`eof()`=0) thì cộng thêm một vào bộ đếm. Nếu việc đọc không thành (`eof()` != 0) thì kết thúc vùng lặp.

Thuật toán trên được thể hiện bằng đoạn chương trình sau:

```
ifstream f(ten_tep, ios::in | ios::binary) ;
long dem = 0 ; char ch;
while (1)
{
    f.get(ch) ;
    if (!eof()) dem++ ;
    else break;
```

}

Bài 15. Nhập xuất nhị phân

15.1. Chọn kiểu nhập xuất nhị phân

Kiểu nhập xuất mặc định là văn bản. Để chọn kiểu nhập xuất nhị phân, thì trong tham số mode (của hàm tạo dạng 2 và phương thức open) cần chứa giá trị:

`ios::binary`

Ví dụ muốn mở tệp “DSTS.DL” để đọc-ghi theo kiểu nhị phân và gắn tệp với dòng nhập-xuất fs, ta dùng câu lệnh sau:

`fstream fs(“DSTS.DL”, ios::in | ios::out | ios::binary);`

15.2. Đọc, ghi ký tự

+ Để ghi một ký tự lên tệp có thể dùng phương thức:

`ostream & put(char);`

+ Để đọc một ký tự từ tệp có thể dùng phương thức:

`istream & get(char &);`

Cần chú ý rằng: Cách đọc ghi ký tự theo kiểu văn bản khác với cách đọc ghi ký tự theo kiểu nhị phân (xem chương 10, cuốn Kỹ thuật lập trình C của tác giả)

Ví dụ để sao tệp có thể dùng thuật toán đơn giản sau:

+ Đọc một ký tự từ tệp nguồn

+ Nếu đọc thành công (phương thức eof() = 0) thì ghi lên tệp đích và lại tiếp tục đọc ký tự tiếp theo.

+ Nếu đọc không thành công (phương thức eof() != 0) thì kết thúc.

Chú ý là phải dùng kiểu nhập xuất nhị phân thì thuật toán mới cho kết quả chính xác. Chương trình sao tệp dưới đây viết theo thuật toán trên.

`//CT7_15.CPP`

`// Sao tệp`

`#include <iostream.h>`

`#include <fstream.h>`

`#include <conio.h>`

`#include <stdlib.h>`

`void main()`

`{`

`clrscr();`

`char tep_nguon[40], tep_dich[40];`

`char ch;`

`fstream fnguồn, fdich;`

`cout << “\nTen tệp nguồn: ” ; cin >> tep_nguon;`

`cout << “\nTen tệp đích: ” ; cin >> tep_dich;`

`fnguồn.open(tep_nguon, ios::in | ios::binary);`

`fdich.open(tep_dich, ios::out | ios::binary);`

`if (fnguồn.bad() || fdich.bad() )`

```
{
    cout << "\n Loi mo tep nguon hoac dich " ;
    getch();
    exit(1);
}
while(fnguồn.get(ch), !fnguồn.eof())
    fdich.put(ch) ;
fnguồn.close();
fdich.close();
cout << "\nHoan thanh" ;
getch();
}
```

15.3. Đọc, ghi một dãy ký tự theo kiểu nhị phân

+ Phương thức:

```
ostream & write(char *buf, int n) ;
```

sẽ xuất n ký tự (byte) chứa trong buf ra dòng xuất.

+ Phương thức:

```
istream & read(char *buf, int n) ;
```

sẽ nhập n ký tự (byte) từ dòng nhập và chứa vào buf.

+ Phương thức

```
int gcount
```

cho biết số ký tự thực sự đọc được trong phương thức read.

Chú ý: Các phương thức write, read chỉ làm việc một cách chính xác trong kiểu nhập-xuất nhị phân.

Dưới đây là chương trình sao tệp sử dụng các phương thức write, read và gcount.

```
//CT7_16.CPP
```

```
// Sao tệp dùng write, read và gcount
```

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

```
#include <fstream.h>
```

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

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

```
void main()
```

```
{
```

```
    clrscr();
```

```
    char tep_nguồn[40], tep_dịch[40] ;
```

```
    char buf[5000];
```

```
    int n;
```

```
    fstream fnguồn, fdich;
```

```
    cout << "\nTen tệp nguồn: " ; cin >> tep_nguồn;
```

```
    cout << "\nTen tệp dịch: " ; cin >> tep_dịch;
```

```
    fnguồn.open(tep_nguồn, ios::in | ios::binary);
```

```
fdich.open(tep_dich,ios::out | ios::binary);
if (fnguồn.bad() || fdich.bad() )
{
    cout << "\n Loi mo tep nguon hoac dich " ;
    getch();
    exit(1);
}
while(fnguồn.read(buf,5000),(n=fnguồn.gcount()))
    fdich.write(buf,n) ;
fnguồn.close();
fdich.close();
cout << "\nHoan thanh" ;
getch();
}
```

Bài 16. Đọc ghi đồng thời theo kiểu nhị phân

Chương trình dưới đây minh họa cách đọc ghi đồng thời trên tệp theo kiểu nhị phân. Chương trình sử dụng các phương thức write, read, các phương thức di chuyển con trỏ tệp và các phương thức kiểm tra lỗi. Chương trình gồm 3 chức năng:

1. Nhập một danh sách thí sinh mới và ghi vào tệp TS.DL
2. Bổ sung thí sinh vào tệp TS.DL
3. Xem sửa thí sinh trên tệp TS.DL

//CT7_18.CPP

// Doc tep

#include <iostream.h>

#include <iomanip.h>

#include <fstream.h>

#include <conio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <stdio.h>

struct TSINH

{
 char ht[25];

int sobd;

float td;

};

class TS

{

private:

TSINH ts;

```
    char ten_tep[40];
    int sots;
    static int size;
public:
    TS(char *ttep);
    void tao_ds();
    void bo_sung();
    void xem_sua();
};
int TS::size = sizeof(TSINH);
TS::TS(char *ttep)
{
    strcpy(ten_tep,ttep);
    fstream f;
    f.open(ten_tep,ios::binary|ios::in|ios::ate);
    if (!f.good())
        sots = 0 ;
    else
    {
        sots=f.tellg()/size ;
    }
}
void TS::tao_ds()
{
    fstream f;
    f.open(ten_tep,ios::binary|ios::out|ios::noreplace);
    if (!f.good())
    {
        cout << "\nDanh sach da ton tai" ;
        cout << "\nCo tao lai khong? - C/K" ;
        char ch=getch();
        if (toupper(ch) != 'C')
            return;
        else
        {
            f.close();
            f.open(ten_tep,ios::binary|ios::out|ios::trunc);
        }
    }
    sots=0;
    while(1)
    {
```

```
cout << "\nThi sinh thu: " << (sots+1) ;
cout << "\nHo ten (Bam Enter de ket thuc): ";
fflush(stdin);
gets(ts.ht);
if (ts.ht[0]==0) break;
cout << "\nSo bao danh: ";
cin >> ts.sobd;
cout << "\nTong diem: ";
cin >> ts.td;
f.write((char*)&ts,size) ;
sots++ ;
}
f.close();
}
void TS::bo_sung()
{
    fstream f;
    f.open(ten_tep,ios::binary|ios::app|ios::nocreate);
    if (!f.good())
    {
        cout << "\nDanh sach chua tao" ;
        cout << "\nCo tao moi khong? - C/K" ;
        char ch=getch();
        if (toupper(ch) != 'C')
            return;
        else
        {
            f.close();
            f.open(ten_tep,ios::binary|ios::out);
        }
    }
    int stt=0;
    while(1)
    {
        cout << "\nBo sung thi sinh thu: " << (stt+1);
        cout << "\nHo ten (Bam Enter de ket thuc): ";
        fflush(stdin);
        gets(ts.ht);
        if (ts.ht[0]==0) break;
        cout << "\nSo bao danh: ";
```

```
    cin >> ts.sobd;
    cout << "\nTong diem: ";
    cin >> ts.td;
    f.write((char*)(&ts),size) ;
    ++stt;
}
sots += stt ;
f.close();
}
void TS::xem_sua()
{
    fstream f; int ch;
    f.open(ten_tep,ios::binary|ios::out|ios::in|ios::nocreate);
    if (!f.good())
    {
        cout << "\nDanh sach chua tao" ;
        getch();
        return ;
    }
    cout << "\nDanh sach gom: " << sots << "thi sinh" ;
    int stt;
    while(1)
    {
        cout << "\nCan xem-sua thi sinh thu (Bam 0 de ket thuc): " ;
        cin >> stt ;
        if (stt<1 || stt > sots) break;
        f.seekg((stt-1)*size,ios::beg);
        f.read((char*)(&ts),size);
        cout << "\nHo ten : " << ts.ht;
        cout << "\nSo ba danh: " << ts.sobd ;
        cout << "\nTong diem: " << ts.td ;
        cout << "\nCo sua khong? - C/K" ;
        ch=getch();
        if (toupper(ch)=='C')
        {
            f.seekg(-size,ios::cur) ;
            cout << "\nHo ten: ";
            fflush(stdin);
            gets(ts.ht);
            cout << "\nSo bao danh: ";
            cin >> ts.sobd;
```



```
        cout << "\nTong diem: ";
        cin >> ts.td;
        f.write((char*)(&ts),size) ;
    }
}
f.close();
}

void main()
{
    int chon;
    clrscr();
    TS t("TS.DL");
    while(1)
    {
        clrscr();
        cout << "\n1. Tao danh sach thi sinh moi" ;
        cout << "\n2. Bo sung danh sach thi sinh" ;
        cout << "\n3. Xem-sua danh sach thi sinh" ;
        cout << "\n4. Ket thuc chuong trinh    " ;
        chon = getch();
        chon = chon - 48;
        clrscr();
        if (chon==1) t.tao_ds();
        else if(chon==2) t.bo_sung();
        else if(chon==3) t.xem_sua();
        else break;
    }
    clrscr();
    cout << "\n Hoan thanh";
    getch();
}
```

Bài 17. Xây dựng toán tử nhập xuất đối tượng trên tệp

Trong các mục trên đã trình bày cách dùng các toán tử nhập >> và xuất << để ghi dữ liệu kiểu chuẩn (nguyên, thực, ký tự, chuỗi ký tự) trên tệp. Mục này trình bày cách xây dựng các toán tử dùng để đọc ghi các đối tượng của một lớp bất kỳ do người dùng định nghĩa.

Giả sử chúng ta muốn sử dụng các toán tử nhập xuất để đọc ghi các đối tượng của lớp TS. Khi đó ta đưa vào các hàm bạn toán tử nhập xuất như sau:

```
class TS
{
private:
    // Khai báo các thuộc tính
```

```
public:
    friend fstream& operator<<(fstream& fs,const TS &t);
    friend fstream& operator>>(fstream& fs,TS &t);
    ...
};
```

Về kiểu ghi: Có thể xây dựng các toán tử để thực hiện các phép đọc ghi theo kiểu văn bản cũng như nhị phân.

Ví dụ 1: Ghi theo kiểu văn bản

Chương trình dưới đây minh họa cách xây dựng và sử dụng các toán tử nhập xuất đối tượng trên màn hình, bàn phím và tệp. Chương trình đưa vào lớp TS (Thí sinh) và các hàm toán tử cho phép nhập xuất các đối tượng TS trên màn hình, bàn phím và tệp. Chương trình gồm các nội dung sau:

- + Tạo tệp TS.DL dùng để đọc và ghi theo kiểu văn bản.
- + Nhập 3 thí sinh từ bàn phím và chứa vào 3 biến đối tượng t1, t2, t3.
- + Ghi nội dung của 3 biến đối tượng t1, t2, t3 lên tệp TS.DL
- + Đọc các đối tượng từ tệp TS.DL và chứa vào 3 biến t4, t5, t6
- + In các biến đối tượng t4, t5, t6 ra màn hình
- + Chuyển con trỏ về đầu tệp, dùng chu trình while để lần lượt đọc các đối tượng từ tệp và in ra màn hình. Dùng phương thức eof để kiểm tra xem đã đọc hết dữ liệu hay chưa.

//CT7_17.CPP

// Cac toan tu doc ghi doi tuong tren Tep

#include <iostream.h>

#include <iomanip.h>

#include <fstream.h>

#include <conio.h>

#include <stdlib.h>

#include <ctype.h>

class TS

{

private:

char ht[25];

float td;

public:

friend ostream& operator<<(ostream& os,const TS &t);

friend istream& operator>>(istream& is,TS &t);

friend fstream& operator<<(fstream& fs,const TS &t);

friend fstream& operator>>(fstream& fs,TS &t);

};

fstream& operator>>(fstream& fs,TS &t)

{

fs.getline(t.ht,25);

fs>>t.td;

fs.ignore();

```
    return fs;
}
ostream& operator<<(ostream& os,const TS &t)
{
    os << "\nHo ten: " << t.ht ;
    os << "\nTong diem: " << t.td;
    return os;
}
fstream& operator<<(fstream& fs,const TS &t)
{
    fs << t.ht << endl;
    fs << t.td << endl;
    return fs;
}
istream& operator>>(istream& is,TS &t)
{
    cout << "\nHo ten: ";
    is.get(t.ht,25);
    cout << "Tong diem: ";
    is >> t.td ;
    is.ignore();
    return is;
}
void main()
{
    clrscr();
    fstream f("TS.DL",ios::out | ios::in | ios::trunc);
    TS t1,t2,t3,t4,t5,t6,t;
    cin >> t1 >> t2 >> t3;
    f<< t1 << t2 <<t3;
    f.seekg(0);
    f>>t4>>t5>>t6;
    cout << t4 << t5 << t6;
    f.seekg(0);
    while (f>>t ,!f.eof())
        cout << t;
    f.close();
    cout << "\n Xong";
    getch();
}
```

}

Ví dụ 2 : Ghi theo kiểu nhị phân

Chương trình dưới đây cũng có các chức năng như chương trình trong ví dụ 1 bên trên, nhưng cách ghi đọc tệp theo kiểu nhị phân.

```
//CT7_19.CPP
// Cac toan tu doc ghi doi tuong tren Tep
// Kieu nhi phan
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
class TS
{
private:
    char ht[25];
    float td;
    static int size;
public:
    friend ostream& operator<<(ostream& os,const TS &t);
    friend istream& operator>>(istream& is,TS &t);
    friend fstream& operator<<(fstream& fs,const TS &t);
    friend fstream& operator>>(fstream& fs,TS &t);
};
int TS::size= sizeof(TS);
fstream& operator>>(fstream& fs,TS &t)
{
    fs.read( (char*)&t , t.size);
    return fs;
}
fstream& operator<<(fstream& fs,const TS &t)
{
    fs.write( (char*)&t , t.size);
    return fs;
}
ostream& operator<<(ostream& os,const TS &t)
{
    os << t.ht << endl;
    os << t.td << endl;
```

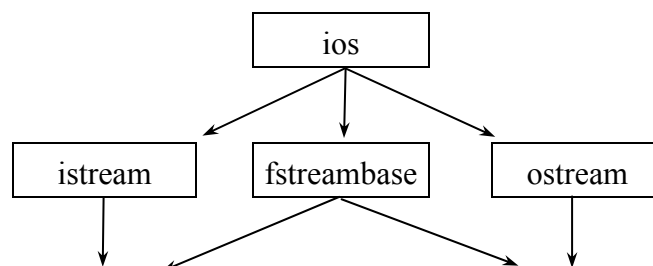
```
    return os;
}
istream& operator>>(istream& is, TS &t)
{
    cout << "\nHo ten: ";
    is.get(t.ht,25);
    cout << "Tong diem: ";
    is >> t.td;
    is.ignore();
    return is;
}

void main()
{
    clrscr();
    fstream f("THU.DL",ios::binary | ios::out|ios::in|ios::trunc);
    TS t1,t2,t3,t4,t5,t6,t;
    cin >> t1 >> t2 >> t3;
    f<< t1 << t2 <<t3;
    f.seekg(0);
    f>>t4>>t5>>t6;
    cout << t4 << t5 << t6;
    f.seekg(0);
    while( f>>t ,!f.eof() )
        cout << t;
    f.close();
    cout << "\n Xong";
    getch();
}
```

Bài 18. Hệ thống các lớp stream

Mục này hệ thống lại các lớp stream mà chúng ta đã sử dụng bên trên để tổ chức xuất nhập trên màn hình, bàn phím, máy in và tệp

18.1. Sơ đồ quan hệ giữa các lớp



ifstream

ofstream

fstream

18.2. Các phương thức của lớp ios

1. int bad()
2. void clear(int=0)
3. int eof()
4. int fail()
5. int fill()
6. int fill(char)
7. long flags()
8. long flags(long)
9. int good()
10. int precision()
11. int precision(int)
12. long setf(long)
13. long setf(long setbits, long field)
14. long unsetf(long)
15. int width()
16. int width(int)

18.3. Các phương thức của lớp istream

1. operator>>
2. int gcount()
3. int get()
4. istream& get(char*, int, char = '\n')
5. istream& get(char&)
6. istream& getline(char*, int, char = '\n')
7. istream& ignore(int n = 1, int delim = EOF)
8. int peek()
9. istream& putback(char)
10. istream& read(char*, int)
11. istream& seekg(long)
12. istream& seekg(long, seek_dir)
13. long tellg()

18.4. Các phương thức của lớp ostream

1. operator<<
2. ostream& flush()

3. ostream& put(char)
4. ostream& seekp(long)
5. ostream& seekp(long, seek_dir)
6. long tellp()
7. ostream& write(char*, int)

18.5. Các phương thức của lớp fstreambase

void close()

18.6. Các phương thức của lớp ifstream

1. ifstream()
2. ifstream(const char*, int = ios::in, int = filebuf::openprot)
3. ifstream(int)
4. ifstream(int , char*, int)
5. void open(const char*, int = ios::in, int = filebuf::openprot)

18.7. Các phương thức của lớp ofstream

1. ofstream()
2. ofstream(const char*, int = ios::out, int = filebuf::openprot)
3. ofstream(int)
4. ofstream(int , char*, int)
5. void open(const char*, int = ios::out, int = filebuf::openprot)

18.8. Các phương thức của lớp fstream

1. fstream()
2. fstream(const char*, int, int = filebuf::openprot)
3. fstream(int)
4. fstream(int , char*, int)
5. void open(const char*, int, int = filebuf::openprot)

Chương 8

Đồ họa

Trong chương này sẽ giới thiệu các hàm để vẽ các đường và hình cơ bản như đường tròn, cung elip, hình quạt, đường gãy khúc, hình đa giác, đường thẳng, đường chữ nhật, hình chữ nhật, hình hộp chữ nhật, ... Ngoài ra còn đề cập tới các vấn đề rất lý thú khác như: xử lý văn bản trên màn hình đồ họa, cửa sổ và kỹ thuật tạo ảnh di động. Các hàm đồ họa được khai báo trong tệp graphics.h.

Bài 1. Khái niệm đồ họa

Để hiểu kỹ thuật lập trình đồ họa, đầu tiên phải hiểu các yếu tố cơ bản của đồ họa. Từ trước đến nay chúng ta chủ yếu làm việc với kiểu văn bản. Nghĩa là màn hình được thiết lập để hiển thị 25 dòng, mỗi dòng có thể chứa 80 ký tự. Trong kiểu văn bản, các ký tự hiển thị trên màn hình đã được phần cứng của máy PC ấn định trước và ta không thể nào thay đổi được kích thước, kiểu chữ.

Ở màn hình đồ họa, ta có thể xử lý đến từng chấm điểm (pixel) trên màn hình và do vậy muốn vẽ bất kỳ thứ gì cũng được. Sự bài trí và số pixel trên màn hình được gọi là độ phân giải (resolution). Do mỗi kiểu màn hình đồ họa có một cách xử lý đồ họa riêng nên TURBO C cung cấp một tệp tin điều khiển riêng cho từng kiểu đồ họa. Bảng 8-1 cho thấy các kiểu đồ họa và các tệp tin điều khiển chúng.

Ngoài các tệp có đuôi BGI chứa chương trình điều khiển đồ họa, TURBO C còn cung cấp các tệp tin đuôi CHR chứa các Font chữ để vẽ các kiểu chữ khác nhau trên màn hình đồ họa. Đó là các tệp:

GOTH.CHR

LITT.CHR

SANS.CHR

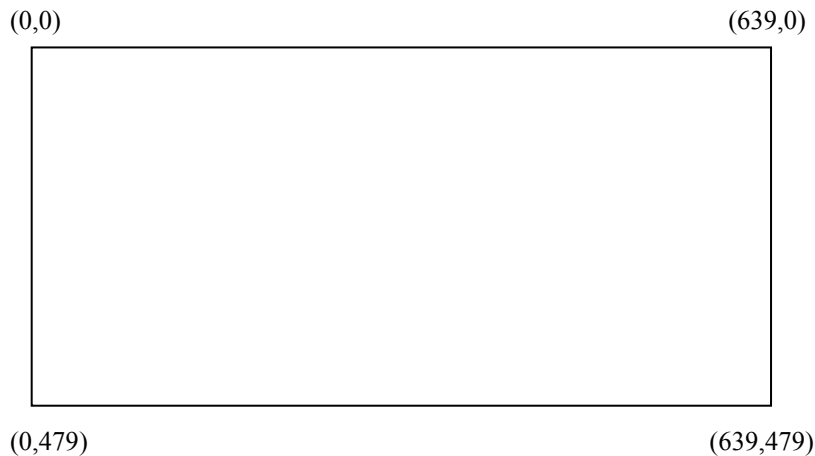
TRIP.CHR

Bảng 8-1. Các tệp tin điều khiển đồ họa của TURBO C++

Tên tệp tin	Kiểu màn hình đồ họa
ATT.BGI	ATT & T6300 (400 dòng)
CGA.BGI	IBMCGA, MCGA và các máy tương thích
EGAVGA.BGI	IBM EGA, VGA và các máy tương thích
HERC.BGI	Hercules monochrome và các máy tương thích
IBM8514.BGI	IBM 8514 và các máy tương thích
PC3270.BGI	IBM 3270 PC

Màn hình đồ họa gồm nhiều điểm ảnh được sắp xếp trên các đường thẳng ngang và dọc. Điều này đúng cho tất cả các kiểu màn hình đồ họa của máy tính. Khác biệt chủ yếu giữa chúng là kích thước và số các điểm ảnh. Trong kiểu CGA (độ phân giải thấp), điểm ảnh có kích thước lớn, chiều ngang có 320 điểm ảnh, còn theo chiều dọc có 200 điểm ảnh. Màn hình VGA có độ phân giải cao hơn: điểm ảnh nhỏ hơn, trên mỗi hàng có 640 điểm ảnh và trên mỗi cột có 480 điểm ảnh. Điểm ảnh càng nhỏ thì số điểm ảnh trên màn hình càng nhiều và chất lượng đồ họa càng cao.

Mỗi kiểu đồ họa dùng một hệ tọa độ riêng. Hệ tọa độ cho màn hình VGA là 640 x 480 như sau :



Hình 8.1. Hệ tọa độ VGA

Nhờ hệ tọa độ này, ta có thể tác động hay tham chiếu đến bất kỳ điểm ảnh nào trên màn hình đồ họa.

Nếu dùng màn hình CGA thì góc dưới phải có tọa độ (319, 199). Độc lập với kiểu đồ họa đang sử dụng, các hàm getmaxx và getmaxy bao giờ cũng cho tọa độ x và y lớn nhất trong kiểu đồ họa đang dùng.

Một chương trình đồ họa thường gồm các phần sau:

- Khởi động hệ thống đồ họa.
- Xác định màu nền (màu màn hình), màu đường vẽ, màu tô và kiểu (mẫu) tô.
- Vẽ, tô màu các hình mà ta mong muốn.
- Các thao tác đồ họa khác như cho hiện các dòng chữ...
- Đóng hệ thống đồ họa để trở về mode văn bản.

Bài 2. Khởi động hệ đồ họa

Mục đích của việc khởi động hệ thống đồ họa là xác định thiết bị đồ họa (màn hình) và một đồ họa sẽ sử dụng trong chương trình. Để làm điều này ta dùng hàm:

```
void initgraph(int *graphdriver, int *graphmode, char *driverpath);
```

trong đó: driverpath là đường dẫn của thư mục chứa các tệp tin điều khiển đồ họa, graphdriver, graphmode cho biết màn hình và một đồ họa sẽ sử dụng trong chương trình. Bảng 8-2 cho thấy các giá trị khả dĩ của graphdriver và graphmode.

Ví dụ 1. Giả sử máy tính của ta có màn hình EGA, các tệp tin đồ họa chứa trong thư mục C:\TC, khi đó ta có thể khởi động hệ thống đồ họa như sau:

```
#include "graphics.h"
main()
{
    int mh=EGA, mode= EGALO;
    initgraph(&mh, &mode, "C:\TC");
    ...
}
```

Bảng 8-2. Các giá trị khả dĩ của graphdriver, graphmode

graphdriver	graphmode	Độ phân giải
Detect (0)		
CGA (1)	CGAC0 (0)	320 x 200
	CGAC1 (1)	320 x 200
	CGAC2 (2)	320 x 200
	CGAC3 (3)	320 x 200
	CGAHi (4)	640 x 200
MCGA (2)	MCGA0 (0)	320 x 200
	MCGA1 (1)	320 x 200
	MCGA2 (2)	320 x 200
	MCGA3 (3)	320 x 200
	MCGAMed (4)	640 x 200
	MCGAHi (5)	640 x 480
EGA (3)	EGALO (0)	640 x 200
	EGAHi (1)	640 x 350
EGA64 (4)	EGA64LO (0)	640 x 200
	EGA64Hi (1)	640 x 350
EGAMONO (5)	EGAMONOHİ (0)	640 x 350
VGA (9)	VGALO (0)	640 x 200
	VGAMED (1)	640 x 350
	VGAHI (2)	640 x 480
HERCMONO (7)	HERCMONOHİ	720 x 348
ATT400 (8)	ATT400C0 (0)	320 x 200
	ATT400C1 (1)	320 x 200
	ATT400C2 (2)	320 x 200
	ATT400C3 (3)	320 x 200
	ATT400MED (4)	640 x 400
	ATT400HI (5)	640 x 400
PC3270 (10)	PC3270HI (0)	720 x 350
IBM8514 (6)	IBM8514LO (0)	640 x 480, 256 màu
	IBM8514HI (1)	1024 x 768, 256 màu

Chú ý 1. Bảng 8-2 cho các tên hằng và giá trị của chúng mà các biến graphdriver, graphmode có thể nhận. Chẳng hạn hằng DETECT có giá trị 0, hằng VGA có giá trị 9, hằng VGALO có giá trị 0... Khi lập trình ta có thể dùng tên hằng hoặc giá trị tương ứng của chúng. Chẳng hạn các phép gán trong ví dụ 1 có thể viết theo một cách khác tương đương như sau:

```
mh=3;
mode=0;
```

Chú ý 2. Bảng 8.2 cho thấy độ phân giải phụ thuộc cả vào màn hình và mode. Ví dụ trong màn hình EGA nếu dùng mode EGALO thì độ phân giải là 640 x 200, hàm getmaxx cho giá trị 639, hàm getmaxy cho giá trị 199 . Nếu cũng màn hình EGA mà dùng mode EGAHI thì độ phân giải là 640x 350, hàm getmaxx cho giá trị 639, hàm getmaxy cho giá trị 349.

Chú ý 3. Nếu không biết chính xác kiểu màn hình đang sử dụng thì ta gán cho biến graphdriver hằng DETECT hay giá trị 0. Khi đó kết quả của hàm initgraph sẽ là:

- Kiểu của màn hình đang sử dụng được phát hiện, giá trị số của nó được gán cho biến graphdriver.

- Mode đồ họa ở độ phân giải cao nhất ứng với màn hình đang sử dụng cũng được phát hiện và giá trị số của nó được gán cho biến `graphmode`.

Như vậy việc dùng hằng số `DETECT` chẳng những có thể khởi động được hệ thống đồ họa của màn hình hiện có theo mode có độ phân giải cao nhất, mà còn giúp ta xác định chính xác kiểu màn hình đang sử dụng.

Ví dụ 2. Chương trình dưới đây xác định kiểu màn hình đang sử dụng:

```
#include "graphics.h"
#include "stdio.h"
main()
{
    int mh=0, mode=0;
    initgraph(&mh, &mode, "");
    printf("\n Giá trị số của màn hình là: %d", mh);
    closegraph();
}
```

Nếu chương trình cho kết quả:

Giá trị số của màn hình là: 3

thì ta có thể khẳng định loại màn hình đang dùng là EGA.

Chú ý 4. Nếu chuỗi dùng để xác định `driverpath` là một chuỗi rỗng (như trong ví dụ 2) thì chương trình dịch sẽ tìm các tệp điều khiển đồ họa trên thư mục chủ.

Bài 3. Lỗi đồ họa

Khi khởi động hệ thống đồ họa nếu máy không tìm thấy các chương trình điều khiển đồ họa thì sẽ phát sinh lỗi đồ họa và việc khởi động coi như không thành. Lỗi đồ họa còn phát sinh khi dùng các hàm đồ họa. Trong mọi trường hợp, hàm `graphresult` cho biết có lỗi hay không lỗi và đó là lỗi gì. Bảng 8-3 cho các mã lỗi mà hàm này phát hiện được. Ta có thể dùng hàm `grapherrormsg` với mã lỗi do hàm `graphresult` trả về để biết được đó là lỗi gì, ví dụ:

```
int maloi;
maloi = graphresult();
printf("\nLỗi đồ họa là: %d", grapherrormsg(maloi));
```

Bảng 8-3. Các mã lỗi của `Graphresult`

Hằng	Trị	Lỗi phát hiện
<code>grOk</code>	0	Không có lỗi
<code>grNoInitGraph</code>	-1	Chưa khởi động hệ đồ họa
<code>grNotDetected</code>	-2	Không có phần cứng đồ họa
<code>grFileNotFound</code>	-3	Không tìm thấy trình điều khiển đồ họa
<code>grInvalidDriver</code>	-4	Trình điều khiển không hợp lệ
<code>grNoLoadMem</code>	-5	Không đủ RAM cho đồ họa
<code>grNoScanMem</code>	-6	Vượt vùng RAM trong Scan fill
<code>grNoFloodMem</code>	-7	Vượt vùng RAM trong flood fill
<code>grFontNotFound</code>	-8	Không tìm thấy tập tin Font
<code>grNoFontMem</code>	-9	Không đủ RAM để nạp Font

grInvalidMode	-10	Kiểu đồ họa không hợp lệ cho trình điều khiển
grError	-11	Lỗi đồ họa tổng quát
grIOerror	-12	Lỗi đồ họa vào ra
grInvalidFont	-13	Tập tin Font không hợp lệ
grInvalidFontNum	-14	Số hiệu Font không hợp lệ

Bài 4. Màu và mẫu

1. Để chọn màu nền ta sử dụng hàm

`void setbkcolor(int color);`

2. Để chọn màu đường vẽ ta dùng hàm

`void setcolor(int color);`

3. Để chọn mẫu (kiểu) tô và màu tô ta dùng hàm

`void setfillstyle(int pattern, int color);`

Trong cả 3 trường hợp color xác định mã của màu. Các giá trị khả dĩ của color cho trong bảng 8-4, pattern xác định mã của mẫu tô (xem bảng 8-5).

Mẫu tô và màu tô sẽ được sử dụng trong các hàm `pieslice`, `fillpoly`, `bar`, `bar3d` và `floodfill` (xem §5 dưới đây).

4. Chọn giải màu

Để thay đổi giải màu đã được định nghĩa trong bảng 8.4 ta dùng hàm

`void setpalette(int colormum, int color);`

Ví dụ câu lệnh

`setpalette(0, Lightcyan);`

biến màu đầu tiên trong bảng màu thành xanh lơ nhạt. Các màu khác không bị ảnh hưởng.

Bảng 8-4. Các giá trị khả dĩ của color

Tên hằng	Giá trị số	Màu hiển thị
BLACK	0	Đen
BLUE	1	Xanh da trời
GREEN	2	Xanh lá cây
CYAN	3	Xanh lơ
RED	4	Đỏ
MAGENTA	5	Tím
BROWN	6	Nâu
LIHGTGRAY	7	Xám nhạt
DARKGRAY	8	Xám sẫm
LIGHTBLUE	9	Xanh da trời nhạt
LIGHTGREEN	10	Xanh lá cây nhạt
LIGHTCYAN	11	Xanh lơ nhạt
LIGHTRED	12	Đỏ nhạt
LIGHTMAGENTA	13	Tím nhạt
YELLOW	14	Vàng

WHITE	15	Trắng

5. Để nhận giải màu hiện hành ta dùng hàm

`void getpalette (struct palettetype *palette);`

ở đây `palettetype` là kiểu đã định nghĩa trước như sau:

```
#define MAXCOLORS 15
```

```
struct palettetype
```

```
{
```

```
    unsigned char size;
```

```
    unsigned char colors[MAXCOLORS+1];
```

```
};
```

ở đây: `size` là số lượng màu trong palette, `colors` là mảng chứa màu với chỉ số mảng chạy từ 0 đến `size - 1`

Bảng 8-5. Các giá trị khả dĩ của pattern

Tên hằng	Giá trị số	Mô tả kiểu tô
EMPTY_FILL	0	Tô bằng màu nền
SOLID_FILL	1	Tô bằng đường nét liền
LINE_FILL	2	Tô bằng - - -
LTSLASH_FILL	3	Tô bằng ///
SLASH_FILL	4	Tô bằng /// in đậm
BKSLASH_FILL	5	Tô bằng \\ in đậm
LTBKSLASH_FILL	6	Tô bằng \\
HATCH_FILL	7	Tô bằng đường gạch bóng nhật
XHATCH_FILL	8	Tô bằng đường gạch bóng chữ thập
INTERLEAVE_FILL	9	Tô bằng đường đứt quãng
WIDE_DOT_FILL	10	Tô bằng dấu chấm thưa
CLOSE_DOT_FILL	11	Tô bằng dấu chấm mau

6. Hàm `getcolor` trả về màu đã xác định trước đó bằng hàm `setcolor`.

7. Hàm `getbkcolor` trả về màu đã xác định trước đó bằng hàm `setbkcolor`.

8. Hàm `getmaxcolor` trả về mã màu cực đại thuộc giải màu hiện đang có hiệu lực. Trên 256 KEGA, hàm `getmaxcolor` luôn cho giá trị 15.

Bài 5. Vẽ và tô màu

Có thể chia các đường và hình thành bốn nhóm chính:

- Đường tròn và ellipse
- Đường gấp khúc và hình đa giác
- Đường thẳng
- Hình chữ nhật

A. Đường tròn và hình tròn

Nhóm này gồm cung tròn, đường tròn, cung ellipse và hình quạt.

1. Cung tròn. Để vẽ một cung tròn ta dùng hàm

```
void arc(int x, int y, int gd, int gc, int r);
```

ở đây:

(x, y) là tọa độ của tâm cung tròn,

r là bán kính

gd là góc đầu

gc là góc cuối

Chú ý: Trong tất cả các hàm dưới đây, góc tính theo độ và có giá trị từ 0 đến 360.

2. Đường tròn. Để vẽ một đường tròn ta dùng hàm

```
void circle(int x, int y, int r);
```

ở đây:

(x, y) là tọa độ của tâm;

r là bán kính đường tròn.

3. Cung ellipse. Để vẽ một cung Ellipse ta dùng hàm

```
void ellipse(int x,int y,int gd,int gc,int xr,int yr);
```

ở đây:

(x, y) là tọa độ của tâm cung Ellipse

gd là góc đầu

gc là góc cuối

xr là bán trục ngang

yr là bán trục đứng.

4. Hình quạt. Để vẽ và tô màu một hình quạt ta dùng hàm

```
void pieslice(int x,int y,int gd,int gc,int r);
```

ở đây:

(x,y) là tọa độ tâm hình quạt

gd là góc đầu

gc là góc cuối

r là bán kính

Ví dụ 1. Chương trình dưới đây sẽ vẽ: một cung tròn ở góc phần tư thứ nhất, một cung ellipse ở góc phần tư thứ ba, một đường tròn và một hình quạt quét từ 90 đến 360 độ.

```
#include <graphics.h>
```

```
main()
```

```
{
```

```
    int mh, mode;
```

```
    // Khởi động đồ họa, màn hình EGA, mode EGALO
```

```
    mh=EGA;
```

```
    mode=EGALO;
```

```
    initgraph(&mh, &mode, "");
```

```
// Màu nền Green, màu đường vẽ
//White, màu tô Red, kiểu tô SlashFill
setbkcolor (GREEN);
setcolor (WHITE);
setfillstyle (SLASH_FILL, RED);
// Vẽ: một cung tròn ở góc phần tư thứ nhất,
// một cung Ellipse ở góc phần tư thứ ba,
// một đường tròn, một quạt tròn
arc(160, 50, 0, 90, 45);
ellipse(480, 50, 180, 270, 150, 45);
circle(160, 150, 45);
pieslice(480, 150, 90, 360, 45);
// Kết thúc chế độ đồ họa
closegraph();
}
```

B. Đường gấp khúc và đa giác

5. Muốn vẽ một đường gấp khúc đi qua n điểm: (x1,y1), ... , (xn,yn) thì trước hết ta phải đưa các tọa độ vào một mảng a nào đó kiểu int. Nói một cách chính xác hơn, cần gán x1 cho a[0], y1 cho a[1], x2 cho a[2], y2 cho a[3],... Sau đó ta viết lời gọi hàm:

```
drawpoly(n, a);
```

Khi điểm cuối (xn, yn) trùng với điểm đầu (x1, y1) ta nhận được một đường gấp khúc khép kín.

6. Giả sử a là mảng đã nói trong điểm 5, khi đó lời gọi hàm

```
fillpoly(n, a);
```

sẽ vẽ và tô màu một đa giác có đỉnh là các điểm

(x1, y1), ... , (xn, yn).

Ví dụ 2. Chương trình dưới đây sẽ vẽ một đường gấp khúc và hai hình tam giác.

```
#include <graphics.h>
// Xây dựng các mảng chứa tọa độ các đỉnh
int poly1[]={5,200,190,5,100,300};
int poly2[]={205,200,390,5,300,300};
int poly3[]={405,200,590,5,500,300,405,200};
main()
{
    int mh=0, mode=0;
    initgraph(&mh, &mode, "");
    // Màu nền CYAN, màu đường vẽ
    // YELLOW, màu tô MAGENTA, mẫu tô SolidFill
    setbkcolor (CYAN); Setcolor (YELLOW);
    setfillstyle (SOLID_FILL, MAGENTA);
```

```
drawpoly (3, poly1); // Đường gấp khúc  
fillpoly (3, poly2); // Hình đa giác  
fillpoly(4, poly3); // Hình đa giác  
closegraph();  
}
```

C. Đường thẳng

7. Hàm

```
void line(int x1,int y1,int x2,int y2);
```

vẽ đường thẳng nối hai điểm (x1, y1) và (x2, y2) nhưng không làm thay đổi vị trí con chạy.

8. Hàm

```
void lineto(int x,int y);
```

vẽ đường thẳng từ điểm hiện tại tới điểm (x, y) và chuyển con chạy đến điểm (x, y).

9. Hàm

```
void linerel(int dx,int dy);
```

vẽ một đường thẳng từ vị trí hiện tại (x, y) của con chạy đến điểm (x + dx, y + dy). Con chạy được di chuyển đến vị trí mới.

10. Hàm

```
void moveto(int x,int y);
```

sẽ di chuyển con chạy tới vị trí (x, y).

Ví dụ 3. Chương trình dưới đây tạo lên một đường gấp khúc bằng các đoạn thẳng. Đường gấp khúc đi qua các đỉnh: (20, 20), (620, 20), (620, 180), (20, 180) và (320, 100).

```
#include <graphics.h>  
main()  
{  
    int mh=0, mode=0;  
    initgraph(&mh, &mode, "");  
    setbkcolor(GREEN);  
    setcolor(YELLOW);  
    moveto(320,100);  
    line(20,20,620,20);  
    linerel(-300,80);  
    lineto(620,180);  
    lineto(620,20);  
    closegraph();  
}
```

D. Hình chữ nhật

11. Hàm

```
void rectangle(int x1,int y1,int x2,int y2);
```


sẽ vẽ một đường chữ nhật có các cạnh song song với các cạnh của màn hình. Tọa độ đỉnh trên bên trái của hình chữ nhật là (x1,y1) và điểm dưới bên phải là (x2,y2).

12. Hàm

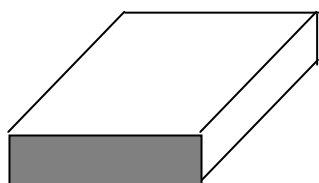
```
void bar(int x1,int y1,int x2,int y2);
```

sẽ vẽ và tô màu một hình chữ nhật. Các giá trị x1,y1,x2 và y2 có ý nghĩa như đã nói trong điểm 11.

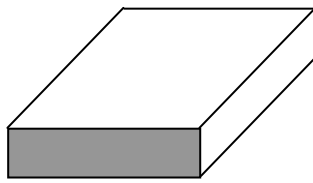
13. Hàm

```
void bar3d(int x1,int y1,int x2,int y2,int depth,int top);
```

sẽ vẽ một khối hộp chữ nhật, mặt ngoài của nó là hình chữ nhật xác định bởi các tọa độ x1,y1,x2,y2 (như đã nói trong điểm 12). Hình chữ nhật này được tô màu. Tham số depth ấn định số điểm ảnh trên bề sâu của khối 3 chiều. Tham số top có thể nhận trị 1 (TOPON) hay 0 (TOPOFF) và khối 3 chiều sẽ có nắp hay không nắp (xem hình vẽ).



TOPON



TOPOFF

Ví dụ 4. Chương trình dưới đây sẽ vẽ một đường chữ nhật, một hình chữ nhật và một khối hộp chữ nhật có nắp.

```
#include <graphics.h>
main()
{
    int mh=0, mode=0;
    initgraph(&mh, &mode, "");
    setbkcolor(GREEN);
    setcolor(RED);
    setfillstyle(CLOSE_DOT_FILL, YELLOW);
    rectangle(5,5,300,160);
    bar(5,175,300,340);
    bar3d(320,100,500,340,100,1);
    closegraph();
}
```

Bài 6. Chọn kiểu đường

1. Hàm

```
void setlinestyle(int linestyle,int pattern,int thickness);
```

tác động đến nét vẽ của các thủ tục line, lineto, rectangle, drawpoly, circle,... Hàm này cho phép ta ấn định 3 yếu tố của đường thẳng là dạng, bề dày và mẫu tự tạo.

+ Dạng đường do tham số linestyle khống chế. Sau đây là các giá trị khả dĩ của linestyle và dạng đường thẳng tương ứng.

```
SOLID_LINE = 0    Nét liền
DOTTED_LINE = 1   Nét chấm
```

CENTER_LINE = 2 Nét chấm gạch

DASHED_LINE = 3 Nét gạch

USERBIT_LINE = 4 Mẫu tự tạo

+ Bề dày do tham số thickness không chế. Giá trị này có thể là:

NORM_WIDTH = 1 Bề dày bình thường

THICK_WIDTH = 3 Bề dày gấp ba

+ Mẫu tự tạo: Nếu tham số thứ nhất là USERBIT_LINE thì ta có thể tạo ra mẫu đường thẳng bằng tham số pattern. Ví dụ xét đoạn chương trình:

```
int pattern= 0x1010;
```

```
setlinestyle(USERBIT_LINE, pattern, NORM_WIDTH);
```

```
line(0,0,100,200);
```

Giá trị của pattern trong hệ 16 là 0x1010 hay trong hệ 2 là

0001 0000 0001 0000

Chỗ nào có bit 1 điểm ảnh sẽ sáng, bit 0 làm tắt điểm ảnh.

2. Để nhận các giá trị hiện hành của 3 yếu tố trên ta dùng hàm:

```
void getlinesettings(struct linesettingstype *lineinfo);
```

với kiểu linesettingstype đã được định nghĩa trước như sau:

```
struct linesettingstype
```

```
{
```

```
  int linestyle;
```

```
  unsigned int upattern;
```

```
  int thickness;
```

```
};
```

Ví dụ 1. Chương trình dưới đây minh họa cách dùng các hàm setlinestyle và getlinesettings để vẽ đường thẳng.

```
// kiểu đường
```

```
#include <graphics.h>
```

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

```
main()
```

```
{
```

```
  struct linesettingstype kieu_cu;
```

```
  int mh=0, mode=0;
```

```
  initgraph(&mh, &mode, "");
```

```
  if (graphresult!= grOk) exit(1);
```

```
  setbkcolor(GREEN); setcolor(RED);
```

```
  line(0,0,100,0);
```

```
  // Lưu lại kiểu hiện tại
```

```
  getlinesettings(kieu_cu);
```

```
  // Thiết lập kiểu mới
```

```
  setlinestyle(DOTTED_LINE,0,THICK_WIDTH);
```

```
  line(0,0,100,10);
```

```
// Phục hồi kiểu cũ
setlinestyle(kieu_cu.linestyle,
kieu_cu.upattern, kieu_cu.thickness);
Line(0,20,100,20);
getch();
closegraph();
}
```

3. Hàm

```
void setwritemode( int writemode);
```

sẽ thiết lập kiểu thể hiện đường thẳng cho các hàm line, drawpoly, linerel, lineto, rectangle. Kiểu thể hiện do tham số writemode không chế:

- Nếu writemode bằng COPY_PUT = 0, thì đường thẳng được viết đè lên dòng đang có trên màn hình.

- Nếu writemode bằng XOR_PUT = 1, thì màu của đường thẳng định vẽ sẽ kết hợp với màu của từng chấm điểm của đường hiện có trên màn hình theo phép toán XOR (chương 3, §3) để tạo lên một đường thẳng mới.

Một ứng dụng của XOR_PUT là: Khi thiết lập kiểu writemode bằng XOR_PUT rồi vẽ lại đường thẳng cùng màu thì sẽ xóa đường thẳng cũ và khôi phục trạng thái của màn hình.

Chương trình dưới đây minh họa cách dùng hàm setwritemode. Khi thực hiện ta sẽ thấy hình chữ nhật thu nhỏ dần vào tâm màn hình.

Ví dụ 2:

```
// Thu hình;
#include <graphics.h>
#include <conio.h>
main()
{
    struct linesettingstype kieu_cu;
    int mh=0, mode=0, x1, y1, x2, y2;
    initgraph(&mh, &mode, "");
    if (graphresult!= grOk) exit(1);
    setbkcolor(GREEN);
    setcolor(RED);
    setfillstyle(CLOSE_DOT_FILL, YELLOW);
    x1=0; y1=0;
    x2=getmaxx(); y2=getmaxy();
    setwritemode(XOR_PUT);
    tt: rectangle(x1,y1,x2,y2); // Vẽ hình chữ nhật
    if ( (x1+1)<(x2-1) && (y1+1)<(y2-1) )
    {
        rectangle(x1,y1,x2,y2); // xóa hình chữ nhật
        x1=x1+1; y1=y1+1; co hình chữ nhật
        x2=x2-1; y2=y2-1;
    }
}
```

```
    goto tt;
}
setwritemode(COPY_PUT); // Trở về overwrite mode
closegraph();
}
```

Bài 7. Cửa sổ (Viewport)

1. Viewport là một vùng chữ nhật trên màn hình đồ họa tựa như window trong textmode. Để thiết lập viewport ta dùng hàm

```
void setviewport(int x1,int y1,int x2,int y2,int clip);
```

trong đó (x1,y1) là tọa độ góc trên bên trái và (x2,y2) là tọa độ góc dưới bên phải. Bốn giá trị này phải thỏa mãn:

$$0 \leq x1 \leq x2$$
$$0 \leq y1 \leq y2$$

Tham số clip có thể nhận một trong hai giá trị:

clip = 1 không cho phép vẽ ra ngoài viewport

clip = 0 Cho phép vẽ ra ngoài viewport.

Ví dụ câu lệnh

```
setviewport(100,50,200,150, 1);
```

sẽ thiết lập viewport. Sau khi lập viewport ta có hệ tọa độ mới mà góc trên bên trái của viewport sẽ có tọa độ (0,0).

2. Để nhận viewport hiện hành ta dùng hàm

```
void getviewsettings(struct viewporttype *vp);
```

ở đây kiểu viewporttype đã được định nghĩa như sau:

```
struct viewporttype
{
    int left, top, right, bottom;
    int clip;
};
```

3. Để xóa viewport ta dùng hàm

```
void clearviewport(void);
```

4. Để xóa màn hình và đưa con chạy về tọa độ (0,0) của màn hình ta dùng hàm

```
void cleardevice(void);
```

Chú ý: Câu lệnh này sẽ xóa mọi thứ trên màn hình.

464

Nhờ sử dụng Viewport có thể viết các chương trình đồ họa theo tọa độ âm dương. Muốn vậy ta thiết lập viewport sao cho tâm tuyệt đối của màn hình là góc trên bên trái của viewport và cho clip = 0 để có thể vẽ ra ngoài giới hạn của viewport. Sau đây là đoạn chương trình thực hiện công việc trên

```
int xc, yc;
xc= getmaxx()/2; yc= getmaxy()/2;
```

```
setviewport(xc, yc, getmaxx(), getmaxy(), 0);
```

Như thế màn hình sẽ được chia làm 4 phần với tọa độ âm dương như sau:

Phần tư trái trên: x âm, y âm

Phần tư trái dưới: x âm, y dương

Phần tư phải trên: x dương, y âm

Phần tư phải dưới: x dương, y dương

Chương trình dưới đây vẽ đồ thị hàm $\sin(x)$ trong hệ trục tọa độ âm dương. Hoành độ x lấy các giá trị từ $-4*PI$ đến $4*PI$. Trong chương trình có dùng hai hàm mới là: `outtextxy` và `putpixel` (xem các mục sau).

Ví dụ 1:

```
// đồ thị hàm sin
```

```
#include <graphics.h>
```

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

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

```
#define SCALEX 20
```

```
#define SCALEY 60
```

```
main()
```

```
{
    int mh=0, mode=0, x, y, i;
    initgraph(&mh, &mode, "");
    if (graphresult!= grOk) exit(1);
    setviewport(getmaxx()/2, getmaxy()/2,
        getmaxx(), getmaxy(), 0);
    // Kê hệ trục tọa độ
    setcolor(BLUE);
    line(-(getmaxx()/2), 0, getmaxx()/2, 0);
    line(0, -(getmaxy()/2), 0, getmaxy()/2);
    setttextjustify(1, 1); setcolor(RED);
    outtextxy(0, 0, "(0,0)");
    for (i=-400; i<=400; ++i)
    {
        x=round(2*M_PI*i*SCALEX/200);
        y=round(sin(2*M_PI*i/200)*SCALEY);
        putpixel(x, y, YELLOW);
    }
    getch();
}
```

Ví dụ 1 tạo lên một đồ thị từ các chấm điểm. Bây giờ ta sửa ví dụ 1 đôi chút: giữ nguyên từ đầu đến `outtextxy`, thay phần cuối bởi đoạn chương trình dưới đây. Ta sẽ được đồ thị từ các đoạn thẳng rất ngắn ghép lại.

Ví dụ 2:

```
// Phần đầu giống ví dụ 1
setcolor(YELLOW);
for (i=-400;i<=400;++i)
{
    x=round(2*M_PI*i*SCALEX/200);
    y=round(sin(2*M_PI*i/200)*SCALEY);
    if(i= -400) moveto(x,y);
    else lineto(x,y);
}
getch();
}
```

Bài 8. Tô điểm, tô miền

1. Hàm

void putpixel(int x, int y, int color);

sẽ tô điểm (x,y) theo màu xác định bởi color.

2. Hàm

unsigned getpixel(int x, int y);

sẽ trả về số hiệu màu của điểm ảnh ở vị trí (x,y). Chú ý: nếu điểm này chưa được tô màu bởi các hàm vẽ hoặc putpixel (mà chỉ mới được tạo màu nền bởi setbkcolor) thì hàm cho giá trị bằng 0. Vì vậy có thể dùng hàm này theo mẫu dưới đây để xác định các nét vẽ trên màn hình đồ họa và vẽ ra giấy.

```
if (getpixel(x,y)!=0)
{
    // Điểm (x,y) được vẽ , đặt một chấm điểm ra giấy
}
```

3. Tô miền

Để tô màu cho một miền nào đó trên màn hình ta dùng hàm

void floodfill(int x, int y, int border);

ở đây:

(x,y) là tọa độ của một điểm nào đó gọi là điểm gieo.

tham số border chứa mã của một màu.

Sự hoạt động của hàm floodfill phụ thuộc vào giá trị của x,y, border và trạng thái màn hình.

a) Khi trên màn hình có một đường (cong hoặc gấp khúc) khép kín mà mã màu của nó bằng giá trị của border thì:

+ Miền giới hạn bởi đường kín sẽ được tô màu nếu điểm gieo (x,y) nằm bên trong miền này.

+ Nếu điểm gieo (x,y) nằm bên ngoài thì phần màn hình bên ngoài miền đóng nói trên được tô màu.

b) Khi trên màn hình không có một đường nào như vậy, thì cả màn hình được tô màu.

Ví dụ 1. Chương trình dưới đây sẽ vẽ một đường tròn đỏ trên màn hình xanh. Tọa độ (x,y) của điểm gieo được nạp vào từ bàn phím. Tùy thuộc vào giá trị cụ thể của x,y, chương trình sẽ tô màu vàng cho hình tròn hoặc phần màn hình bên ngoài hình tròn.

```
#include <graphics.h>
#include <stdio.h>
main()
{
    int mh=0, mode=0, x, y;
    initgraph(&mh, &mode, "");
    if (graphresult!= grOk) exit(1);
    setbkcolor(GREEN);
    setcolor(RED);
    setfillstyle(11,YELLOW);
    circle(320,100,50);
    moveto(1,150);
    outtext(" Tọa độ điểm gieo x,y ");
    scanf("%d%d",&x,&y); floodfill(x,y,RED);
}
```

Ví dụ 2. Minh họa cách dùng hàm Putpixel và hàm getpixel để vẽ các điểm ảnh và sau đó xóa các điểm ảnh. Muốn kết thúc chương trình bấm ESC.

```
#include <conio.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>
int seed = 1962; // Nhân cho bộ tạo số ngẫu nhiên
int numpts = 2000; // Vẽ 2000 chấm điểm
int ESC = 27;
void putpixelplay(void);
main()
{
    int mh=0, mode=0;
    initgraph(&mh, &mode, "");
    if (graphresult()!= grOk)
        exit(1);
    putpixelplay();
    closegraph();
}
void putpixelplay(void)
{
    int i,x,y,color,xmax,ymax,maxcolor,ch;
```

```
struct viewporttype v;
getviewsettings(&v);
xmax=(v.right - v.left -1); ymax=(v.bottom - v.top -1);
maxcolor=getmaxcolor();
while (!kbhit())
{
    //Vẽ các chấm điểm một cách ngẫu nhiên
    srand(seed);
    i=0;
    while(i<=numpts)
    {
        ++i;
        x=random(xmax)+1;y=random(ymax)+1;
        color=random(maxcolor);
        putpixel(x,y,color);
    }
    // Xóa các điểm ảnh
    srand(seed);
    i=0;
    while(i<=numpts)
    {
        ++i;
        x= random(xmax) + 1; y= random(ymax) + 1;
        color=random(maxcolor);
        putpixel(x,y,0);
    }
    if(kbhit())
    {
        ch=getch();
        if (ch==ESC) break;
    }
}
} // Kết thúc hàm putpixelplay
```

Bài 9. Xử lý văn bản trên màn hình đồ hoạ

1. Hiển thị văn bản trên màn hình đồ hoạ

Hàm

```
void outtext (char *s);
```


sẽ hiện chuỗi ký tự (do s trở tới) tại vị trí hiện tại của con trỏ.

Hàm

```
void outtextxy(int x,int y,char *s);
```

sẽ hiện chuỗi ký tự (do s trở tới) tại vị trí (x,y).

Ví dụ 1: Hai cách sau đây sẽ cho cùng kết quả

```
outtextxy (100,100," chao ban ");
```

và

```
moveto (100,100);
```

```
outtext (" chao ban ");
```

Chú ý: Trong một đồ họa vẫn cho phép dùng hàm nhập dữ liệu scanf và các hàm bắt phím getch, kbhit.

2. Fonts

Như đã nói ở trên: Các Fonts nằm trong các tệp tin .CHR trên đĩa. Các Font này cho các kích thước và kiểu chữ khác nhau sẽ hiện thị trên màn hình đồ họa bằng outtext hay outtextxy. Để chọn và nạp Font chúng ta dùng hàm:

```
void settextstyle(int font,int direction,int charsize);
```

(Chú ý: hàm chỉ có tác dụng nếu tồn tại các tệp .CHR)

Với direction là một trong hai hằng số:

```
HORIZ_DIR = 0
```

```
VERT_DIR = 1
```

Nếu direction là HORIZ_DIR, văn bản sẽ hiển thị theo hướng nằm ngang từ trái sang phải. Nếu direction là VERT_DIR, văn bản sẽ hiển thị theo chiều đứng từ dưới lên trên.

Đối charsize là hệ số phóng to ký tự và có giá trị trong khoảng từ 1 đến 10.

- Nếu charsize = 1, ký tự được thể hiện trong hình chữ nhật 8*8 pixel.

- Nếu charsize = 2, ký tự được thể hiện trong hình chữ nhật 16*16 pixel.

...

- Nếu charsize = 10, ký tự được thể hiện trong hình chữ nhật 80*80 pixel.

Cuối cùng là tham số font để chọn kiểu chữ và nhận một trong các hằng sau:

```
DEFAULT_FONT = 0
```

```
TRIPLEX_FONT = 1
```

```
SMALL_FONT = 2
```

```
SANS_SERIF_FONT = 3
```

```
GOTHIC_FONT = 4
```

Các giá trị do settextstyle thiết lập sẽ dữ nguyên cho đến khi gọi một settextstyle mới.

Ví dụ 2:

```
settextstyle (3,VERT_DIR,2);
```

```
outtextxy (50,50," HELLO ");
```

3. Vị trí hiển thị

Hàm settextjustify cho phép ấn định nơi hiển thị văn bản của outtext theo quan hệ với vị trí hiện tại của con chạy hay của outtextxy theo quan hệ với toạ độ (x,y).

Hàm này có dạng

```
void settextjustify(int horiz, int vert);
```

Tham số horiz có thể là một trong các hằng số sau:

LEFT_TEXT = 0 (Văn bản xuất hiện bên phải con chạy)

CENTER_TEXT = 1 (Chỉnh tâm văn bản theo vị trí con chạy)

RIGHT_TEXT = 2 (Văn bản xuất hiện bên trái con chạy)

Tham số Vert có thể là một trong các hằng số sau:

BOTTOM_TEXT = 0 (Văn bản xuất hiện phía trên con chạy)

CENTER_TEXT = 1 (Chỉnh tâm văn bản theo vị trí con chạy)

TOP_TEXT = 2 (Văn bản xuất hiện phía dưới con chạy)

Ví dụ 3:

```
settextjustify(1,1);
```

```
outtextxy(100,100,"ABC");
```

Kết quả là điểm (100,100) sẽ nằm giữa chữ B.

4. Bề rộng và bề cao của văn bản

Hàm

```
void textheight (char *s);
```

trả về chiều cao (theo pixel) của chuỗi do s trở tới. Ví dụ nếu ký tự có kích thước 8*8 thì

```
textheight ("H") = 8
```

Ví dụ 4: Đoạn chương trình dưới đây sẽ cho hiện 5 dòng chữ.

```
#include <graphics.h>
```

```
main()
```

```
{
```

```
int mh=0,mode=0,y,size;
```

```
initgraph(&mh,&mode,"");
```

```
y=10;
```

```
settextjustify(0,0);
```

```
for (size=1; size<=5; ++size)
```

```
{
```

```
settextstyle(0,0,size);
```

```
outtextxy(0,y,"GRAPHICS");
```

```
y += textheight("GRAPHICS") + 10;
```

```
}
```

```
getch();
```

```
closegraph();
```

```
}
```

Hàm

```
void textwidth(char *s);
```

sẽ dựa vào chiều dài của chuỗi, kích thước Font chữ, hệ số khuếch đại chữ để trả về bề rộng (theo pixel) của chuỗi do s trở tới.

Ví dụ 5: Trong chương trình dưới đây sẽ lập các hàm vào ra trên màn hình đồ hoạ.

```
#include <graphics.h>
#include <conio.h>
#define Enter 13
#define Lmargin 10
void text_write(int *x,int *y,char *s);
void text_writeln(int *x,int *y,char *s);
void text_read(int *x,int *y,char *s);
void text_write(int *x,int *y,char *s)
{
    outtextxy(*x,*y,s); *x += textwidth(s);
}
void text_writeln(int *x,int *y,char *s)
{
    outtextxy(*x,*y,s);
    *x=Lmargin;
    *y += textheight(s)+5;
}
void text_read(int *x,int *y,char *s)
{
    int i=0; char ch[2];
    ch[1]=0;
    while(1)
    {
        ch[0]=getch();
        if(ch[0]==Enter) break;
        text_write(x,y,ch);
        s[i]=ch[0]; ++i;
    }
    s[i]=0;
}
main()
{
    int mh=0,mode=0,x,y,xmax,ymax;
    char name[25];
    initgraph(&mh,&mode,"");
    settextstyle(TRIPLEX_FONT,HORIZ_DIR,3);
    x=Lmargin; y=100;
    text_write (&x,&y,"cho ten cua ban: ");
```

```
text_read (&x,&y,name);
text_writeln (&x,&y,"");
text_write(&x,&y,"chao ban ");
text_write(&x,&y,name);
getch();
closegraph();
}
```

Bài 10. Cắt hình, Dán hình và Tạo ảnh chuyển động

1. Hàm

unsigned imagesize(int x1,int y1,int x2,int y2)

trả về số byte cần thiết để lưu trữ ảnh trong phạm vi hình chữ nhật (x1,y1,x2,y2).

2. Hàm

#include <alloc.h>

void *malloc(unsigned n);

trả về con trỏ tới một vùng nhớ n byte mới được cấp phát.

3. Hàm

void getimage(int x1,int y1,int x2,int y2,void *bitmap);

sẽ chép các điểm ảnh của hình chữ nhật (x1,y1,x2,y2) và các thông tin về bề rộng, cao của hình chữ nhật vào vùng nhớ do bitmap trỏ tới. Vùng nhớ và biến bitmap cho bởi hàm malloc. Độ lớn của vùng nhớ được xác định bằng hàm imagesize.

4. Hàm

void putimage(int x,int y,void *bitmap,int copymode);

dùng để sao ảnh lưu trong vùng nhớ bitmap ra màn hình tại vị trí (x,y). Tham số copymode xác định kiểu sao chép ảnh, nó có thể nhận các giá trị sau:

COPY_PUT = 0	Sao chép nguyên xi.
XOR_PUT = 1	Các điểm ảnh trong bitmap kết hợp với các điểm ảnh trên màn hình bằng phép XOR
OR_PUT = 2	Các điểm ảnh trong bitmap kết hợp với các điểm ảnh trên màn hình bằng phép OR
AND_PUT = 3	Các điểm ảnh trong bitmap kết hợp với các điểm ảnh trên màn hình bằng phép AND
NOT_PUT = 4	ảnh xuất hiện trên màn hình theo dạng đảo ngược (phép NOT) với ảnh trong bitmap.

Nhận xét: Nếu dùng mode XOR_PUT để chép hình, rồi lặp lại đúng câu lệnh đó thì hình sẽ bị xoá và màn hình trở lại như cũ. Kỹ thuật này dùng để tạo lên các hình ảnh chuyển động.

Ví dụ 1: Chương trình dưới đây minh hoạ cách dùng imagesize, malloc, getimage và putimage.

```
#include <alloc.h>
#include <graphics.h>
main()
{
    int mh=0,mode=0;
    char *p;
    unsigend size;
    initgraph (&mh,&mode,"");
    bar(0,0,getmaxx(),getmaxy());
    size = imagesize(10,20,30,40);
    p=(char*)malloc(size); // p trỏ tới vùng nhớ size byte
                           // mới được cấp phát
    getimage (10,20,30,40,p);
    getch();
    cleardevice();
    putimage (100,100,p,COPY_PUT);
    getch();
    closegraph();
}
```

5. Tảo ảnh di động

Nguyên tắc tạo ảnh di động giống như phim hoạt hình:

- Vẽ một hình (trong chuỗi hình mô tả chuyển động)
- Delay
- Xoá hình đó
- Vẽ hình kế theo
- Delay
- ...

A) Vẽ hình

Cách 1: Vẽ lại một ảnh nhưng tại các vị trí khác nhau.

Cách 2: Lưu ảnh vào một vùng nhớ rồi đưa ảnh ra màn hình tại các vị trí khác nhau.

B) Xóa ảnh

Cách 1: Dùng hàm cleardevice

Cách 2: Dùng hàm putimage (mode XOR_PUT) để xếp chồng lên ảnh cần xóa.

Cách 3: Lưu trạng thái màn hình vào một chỗ nào đó. Vẽ một hình ảnh. Đưa trạng thái cũ màn hình ra xếp đè lên ảnh vừa vẽ.

Kỹ thuật tạo ảnh di động được minh họa trong các chương trình của §11.

Bài 11. Một số chương trình đồ họa

Chương trình 1: Đầu tiên vẽ bầu trời đầy sao. Sau đó từng chùm pháo hoa được bắn lên bầu trời. Khi bấm phím Enter thì việc bắn pháo hoa kết thúc, ta nhận lại bầu trời đầy sao. Bấm tiếp Enter thì kết thúc chương trình.

// Bắn pháo hoa trên bầu trời đầy sao

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
main()
{
    int x[101],y[101];
    int mh=0,mode=0,i,n;
    char *p[101];
    initgraph(&mh,&mode,"");
    if (graphresult()!=0) exit(1);
    setcolor(RED);
    // Vẽ bầu trời đầy sao
    for (i=1;i<=1000;++i)
    {
        putpixel(random(getmaxx()),
            random(getmaxy()),random(getmaxcolor()));
    }
    // Lưu hiện trạng 100 hình chữ nhật trên màn hình để khôi phục
    for (i=1;i<=100;++i)
    {
        x[i]=random(getmaxx())-10;
        y[i]=random(getmaxy())-10;
        if (x[i]<0) x[i]=0;
        if (y[i]<0) y[i]=0;
        n=imagesize(x[i],y[i],x[i]+10,y[i]+10);
        p[i]=(char*)malloc(n);
        getimage(x[i],y[i],x[i]+10,y[i]+10,p[i]);
    }
    // Chu trình bắn pháo hoa
    do
    {
        // Đưa 100 quả pháo lên màn hình tại các vị trí quy định
        for (i=1;i<=100;++i)
        {
            setfillstyle(SOLID_FILL,i%15+1);
            pieslice(x[i]+5,y[i]+5,0,360,5);
        }
        delay(500);
        //Xoá chùm pháo hoa vừa bắn bằng cách khôi phục màn hình
```

```
for (i=100;i>=1;--i)
    putimage(x[i],y[i],p[i],COPY_PUT);
delay(500);
} while(!kbhit());
getch();
getch();
closegraph();
}
```

Chương trình 2: Vẽ đồng hồ có 3 kim giờ, phút và giây. Đồng hồ chạy đúng theo giờ hệ thống. Muốn kết thúc chương trình bấm Enter.

```
// Đồng hồ
#include <graphics.h>
#include <conio.h>
#include <math.h>
#include <dos.h>
// Hàm kẻ đoạn thẳng từ tâm đồng hồ theo độ, chiều dài,
// độ dày và màu
void ke(int ddo, unsigned dai, unsigned day, unsigned mau);
// Kẻ kim giây khi biết số giây
void ke_giay(unsigned giay);
// Kẻ kim phút khi biết số phút
void ke_phut(unsigned phut);
// Kẻ kim giờ khi biết số giờ
void ke_gio(unsigned gio, unsigned phut);
void chay_kim_giay(void); void chay_kim_phut(void);
void chay_kim_gio(void);
int x0,y0,rgio,rphut,rgiay,mgio,mphut,mgiay;
unsigned phutgioht,gioht,phutht,giayht;
void ke(int ddo, unsigned dai, unsigned day, unsigned mau)
{
    unsigned x,y; float goc;
    while (ddo>=360) ddo=ddo-360;
    goc=(M_PI/180)*ddo;
    x=x0+ (int)(dai*cos(goc)+0.5);
    y=y0- (int)(dai*sin(goc)+0.5);
    setcolor(mau); setlinestyle(0,0,day);
    line(x0,y0,x,y);
}
// Hàm kẻ kim giây
void ke_giay(unsigned giay)
{

```

```
    int ddo;
    ddo = (90 - 6*giay);
    ke(ddo,rgiay,1,mgiay);
}
// Hàm ke kim phut
void ke_phut(unsigned phut)
{
    int ddo;
    ddo= (90-6*phut);
    ke(ddo,rphut,3,mphut);
}
// Hàm ke kim gio
void ke_gio(unsigned gio, unsigned phut)
{
    int ddo;
    ddo = 360 + 90 - 30*(gio%12) - (phut+1)/2;
    ke(ddo,rgio,3,mgio);
}
// Hàm chỉnh giây hiện tại và làm chuyển động kim giây
void chay_kim_giay(void)
{
    unsigned giay; struct time t;
    gettime(&t);
    giay=t.ti_sec;
    if (giay!=giayht)
    {
        ke_giay(giayht);
        giayht=giay;
        ke_giay(giayht);
    }
}
// Hàm chỉnh phút hiện tại và làm chuyển động kim phút
void chay_kim_phut(void)
{
    unsigned phut;
    struct time t;
    gettime(&t);
    phut=t.ti_min;
    if (phut!=phutht)
    {
```



```
        ke_phut(phutht);
        phutht=phut;
        ke_phut(phutht);
    }
}
// Hàm chỉnh giờ phút hiện tại và làm chuyển động kim giờ
void chay_kim_gio(void)
{
    unsigned h,gio,phut,sophut,sophutht;
    struct time t;
    gettime(&t);
    gio=t.ti_hour; phut=t.ti_min;
    sophut = gio*60+phut;
    sophutht = gioht*60+phutgioht;
    if ( sophut<sophutht) sophut=sophut+ 12*60;
    h=sophut-sophutht;
    if (h>=12)
    {
        ke_gio(gioht,phutgioht);
        phutgioht=phut;
        gioht=gio;
        ke_gio(gioht,phutgioht);
    }
}
main()
{
    struct time t;
    char *dso[]={ "", "12", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11"};
    int i,mh=0,mode=0,r,x,y;
    float goc;
    initgraph(&mh,&mode,"");
    x0=(getmaxx()/2)-1; y0=(getmaxy()/2)-1;
    r=y0-2;
    rgiaiy = r-10; rphut=r-50; rgio=r-90;
    mgiay= BROWN; mphut=RED; // mgio:=magenta;
    mgio=YELLOW;
    // Vẽ chu vi đồng hồ
    setcolor(BLUE); setlinestyle(0,0,3); circle(x0,y0,r);
    setfillstyle(1,YELLOW);
    floodfill(0,0,BLUE);
```

```

setfillstyle(1,WHITE); floodfill(x0,y0,BLUE);
setlinestyle(0,0,1);
circle(x0,y0,10);
setfillstyle(1,GREEN); floodfill(x0,y0,BLUE);
settextjustify(1,1); setcolor(MAGENTA);
outtextxy(x0,y0+120,"IBM-JIMIKO");
// Ghi chữ số
settextstyle(3,0,3); settextjustify(1,1); setcolor(BLUE);
for (i=1;i<=12;++i)
{
    goc=(2*M_PI+M_PI/2) - (i-1)*(M_PI/6);
    x = x0+ (int)(rphut*cos(goc)+0.5);
    y = y0- (int)(rphut*sin(goc)+0.5);
    outtextxy(x,y,dso[i]);
}
// Xác định thời điểm đầu
gettime(&t);
gioht=t.ti_hour; phutht=t.ti_min; giayht=t.ti_sec;
phutgioht=phutht;
setwritemode(XOR_PUT);
// Vẽ kim gio,phut,giay
ke_gio(gioht,phutgioht);
ke_phut(phutht);
ke_giay(giayht);
// Làm chuyển động các kim
do
{
    chay_kim_giay(); chay_kim_phut();
    chay_kim_gio();
}
while(!kbhit());
closegraph();
}

```

Chương trình 3: Vẽ một con tàu vũ trụ bay trên bầu trời đầy sao theo quỹ đạo ellipse. Trong khi tàu chuyển động thì các ngôi sao thay nhau nhấp nháy

```

// Tàu vũ trụ chuyển động trên bầu trời đầy sao nhấp nháy
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

```

```
#include <stdlib.h>
#include <math.h>
// Khai báo các hàm trong chương trình
void tau_cd(void); // tàu chuyển động
void nhap_nhay_bt(void); // nhấp nháy bầu trời
void main(void); // hàm main
// Khai báo các biến mảng ngoài
int a,b,x,y,x0,y0,mh=0,mode=0,n,i;
float goc,xt,yt;
char *p;
int xx[1001],yy[1001];
// Hàm main
void main(void)
{
    initgraph(&mh,&mode,"");
    if (graphresult()!=0) exit(1);
    // Vẽ tàu vũ trụ
    setcolor(RED);
    ellipse(100,50,0,360,20,8);
    ellipse (100,46,190,357,20,6);
    line(107,44,110,38);
    circle(110,38,2);
    line(93,44,90,38);
    circle(90,38,2);
    setfillstyle(SOLID_FILL,BLUE);
    floodfill(101,54,RED);
    setfillstyle(SOLID_FILL,MAGENTA);
    floodfill(94,45,RED);
    // Lưu ảnh của tàu vũ trụ vào bộ nhớ
    n=imagesize(79,36,121,59);
    p=(char*)malloc(n);
    getimage(79,36,121,59,p);
    // Vẽ bầu trời đầy sao và lưu vị trí của chúng
    // vào các mảng xx, yy để phục vụ hàm nhap_nhay_bt
    cleardevice();
    for (i=1;i<=1000;++i)
    {
        xx[i]=random(getmaxx()); yy[i]=random(getmaxy());
        putpixel(xx[i],yy[i],random(getmaxcolor()));
    }
    // Xác định giá trị ban đầu cho các biến
```

```
// dùng để điều khiển chuyển động tàu
goc= 2*M_PI + M_PI/2;
x0= (getmaxx() - 42)/2;
y0= (getmaxy() - 25)/2;
a=x0; b=y0;
// chu trình tàu vũ trụ chuyển động và các ngôi sao nhấp nháy
do
{
    tau_cd();
    nhap_nhay_bt();
} while(!kbhit());
getch();
closegraph();
}

void tau_cd(void)
{
    xt=a*cos(goc)+x0;
    yt=-b*sin(goc)+y0;
    x=(int)(xt+0.5); y=(int)(yt+0.5);
    // Đặt tàu vũ trụ lên màn hình
    putimage(x,y,p,XOR_PUT);
    delay(500);
    // Xóa
    putimage(x,y,p,XOR_PUT);
    // Thay đổi góc để làm cho tàu chuyển động
    goc -= M_PI/30;
    if (goc<M_PI/2) goc=2*M_PI+M_PI/2;
}

void nhap_nhay_bt(void)
{
    static i=1; // Lệnh này thực hiện một lần khi dịch
    int j;
    // Cho nhấp nháy bằng cách đổi màu 50 ngôi sao
    for (j=1;j<=50;++j)
    {
        putpixel(xx[i],yy[i],random(getmaxcolor()));
        ++i;
        if (i>1000) i=1;
    }
}
```

Bài 12. In ảnh từ màn hình đồ hoạ

Hàm `in_anh` dưới đây sẽ in ảnh trong miền chữ nhật (`xt, yt, xd, yd`) của màn hình đồ hoạ ra giấy trên các máy in LQ1070, LQ1170 và FX1050.

```
void in_anh(int dd,int xt,int yt,int xd,int yd);
```

Tham số `dd` là độ đậm của nét in. Thực chất `dd` là số lần in lại. Bình thường chọn `dd=1`. Nếu muốn in rõ hơn ta chọn `dd` bằng 2 hay 3.

Trong hàm `in_anh` có dùng hàm `tao_mau`, nó được mô tả như sau:

```
int tao_mau(int k,int x,int y);
```

Hàm này sẽ dò trên `k` chấm điểm theo chiều dọc bắt đầu từ toạ độ (`x,y`) trên màn hình để biết xem chấm điểm nào đã tô màu. Hàm sẽ trả về một giá trị nguyên tạo bởi các bit 1 (ứng với điểm đã tô màu) và 0 (ứng với điểm chưa tô màu).

Hàm `in_anh` sẽ dùng hàm `tao_mau` để duyệt trên miền chữ nhật (`xt,yt,xd,yd`). Mỗi lần duyệt sẽ nhận được một mẫu các chấm điểm (giá trị nguyên) và mẫu này được in ra giấy.

Dưới đây là nội dung của 2 hàm nói trên.

```
// in ảnh
#include "stdio.h"
#include "graphics.h"
int tao_mau(int k,int x,int y);
void in_anh(int dd,int xt,int yt,int xd,int yd);
int tao_mau(int k,int x,int y)
{
    int c=0,i;
    for (i=0;i<k;++i)
        if (getpixel(x,y+i)) c=c|(128>>i);
    return c;
}
void in_anh(int dd,int xt,int yt,int xd,int yd)
{
    //dd - so lan in lai mot dong
    char c,ch1;
    int scot,m,mm,k,dong,cot,i,j,n1,n2;
    dong=(yd-yt+1)/6; mm=(yd-yt+1) % 6;
    cot=xd-xt+1;
    for (i=0;i<=dong;++i)
    {
        if (i<dong) m=6; else m=mm;
        if (m>0)
        {
            scot=0;
            for (j=0;j < cot;++j)
```

```
    if (tao_mau(m,xt+j,yt+i*6)) scot=j+1;
    if (scot)
    {
        n1=scot % 256; n2= scot/256;
        for (k=0;k<dd;++k)
        {
            fprintf(stdprn,"%c%c%c%c%c%c",13,27,'*',
                                0,n1,n2); //LQ
            for (j=0;j < scot;++j)
            {
                if (kbhit())//bat phim
                {
                    if ((ch1=getch())==0) getch();
                    if (ch1==27) goto ket;
                }
                c=tao_mau(m,xt+j,yt+i*6);
                fprintf(stdprn,"%c",c);
            }
        }
        fprintf(stdprn,"%c%c%c",27,'A',m);
        fprintf(stdprn,"\n");
    }
    }
    ket: fprintf(stdprn,"%c%c",27,'@');
}
```

Chương 9

Truy nhập trực tiếp vào bộ nhớ

Trong chương này trình bày các vấn đề:

- + Hai kiểu địa chỉ: Địa chỉ phân đoạn và địa chỉ thực
- + Truy nhập tới địa chỉ phân đoạn
- + Đổi từ địa chỉ phân đoạn sang địa chỉ thực
- + Bộ nhớ màn hình, truy nhập trực tiếp vào bộ nhớ màn hình
- + Dựng con trỏ để lấy dữ liệu từ bộ nhớ phân đoạn
- + Dựng con trỏ hàm để thực hiện các thủ tục của DOS

Bài 1. Các hàm truy nhập theo địa chỉ phân đoạn

1. Hàm pokeb: Gửi một ký tự vào bộ nhớ.

+ Nguyên mẫu trong dos.h như sau:

void pokeb(unsigned seg, unsigned off, char value);

+ Củng dụng: Gửi giá trị ký tự value vào bộ nhớ tại địa chỉ phân đoạn seg:off

2. Hàm peekb: Nhận một ký tự từ bộ nhớ.

+ Nguyên mẫu trong dos.h như sau:

char peekb(unsigned seg, unsigned off);

+ Củng dụng: Nhận một byte tại địa chỉ phân đoạn seg:off

3. Hàm poke: Gửi một số nguyên vào bộ nhớ.

+ Nguyên mẫu trong dos.h như sau:

void poke(unsigned seg, unsigned off, int value);

+ Củng dụng: Gửi giá trị nguyên value vào bộ nhớ tại địa chỉ phân đoạn seg:off

4. Hàm peek: Nhận một số nguyên từ bộ nhớ.

+ Nguyên mẫu trong dos.h như sau:

int peek(unsigned seg, unsigned off);

+ Củng dụng: Nhận một word tại địa chỉ phân đoạn seg:off

5. Hàm movedata: Sao các byte.

+ Nguyên mẫu trong mem.h như sau:

**void movedata(unsigned seg_gui, unsigned off_gui,
 unsigned seg_nhan, unsigned off_nhan, int n);**

+ Củng dụng: Sao n byte từ seg_gui:off_gui đến
 seg_nhan:off_nhan

Bài 2. Bộ nhớ màn hình văn bản

2.1. Cách biểu diễn ký tự trong bộ nhớ màn hình

Bộ nhớ màn hình văn bản bắt đầu từ địa chỉ :

(0xb800:0x0000)

Khi đưa một ký tự vào vùng nhớ màn hình, thì nó sẽ hiện lên màn hình. Mỗi ký tự trên màn hình chiếm 2 byte trong bộ nhớ màn hình: byte đầu chứa mã ASCII, byte thứ hai biểu diễn màu hiển thị gọi là byte thuộc tính. Các bit của byte thuộc tính:

B7B6B5B4B3B2B1B0

được chia làm 3 nhóm:

- + Nhóm 1 gồm bit B7 biểu thị sự nhấp nháy. Nếu B7=0 thì ký tự không nhấp nháy, nếu B7=1 thì ký tự sẽ nhấp nháy.
- + Nhóm 2 gồm các bit B6, B5 và B4. Các bit này chứa được một số nguyên từ 0 đến 7 và biểu thị 8 màu nền của ký tự.
- + Nhóm 3 gồm các bit B3, B2, B1 và B0. Các bit này chứa được một số nguyên từ 0 đến 15 và biểu thị 16 màu của ký tự.

2.2. Trang màn hình

Mỗi trang màn hình gồm 80x25 ký tự, do đó cần $80 \times 25 \times 2 = 4000$ byte bộ nhớ. Thực tế mỗi trang màn hình được phân bổ $4096 = 0x1000$ byte. Như vậy 4 trang màn hình được phân bổ như sau:

- + Trang màn hình thứ 0 bắt đầu từ địa chỉ 0xB800:0x0000
- + Trang màn hình thứ 1 bắt đầu từ địa chỉ 0xB800:0x1000
- + Trang màn hình thứ 2 bắt đầu từ địa chỉ 0xB800:0x2000
- + Trang màn hình thứ 3 bắt đầu từ địa chỉ 0xB800:0x3000

2.3. Chọn trang hiển thị

Tại mỗi thời điểm chỉ có thể hiển thị được một trong 4 trang màn hình. Để hiển thị trang màn hình thứ t ($t=0,1,2,3$) chúng ta sử dụng chức năng 5 của ngắt 0x10 theo mẫu sau:

```
union REGS v,r;  
v.h.ah = 5 ; // Chức năng 5  
v.h.al = t ; // Số hiệu trang màn hình cần hiển thị  
int86(0x10, &v, &r); // Thực hiện ngắt 0x10
```

2.4. Ví dụ minh họa

Ví dụ sau dựng hàm pokeb để đưa các ký tự vào các trang của bộ nhớ màn hình, sau đó dựng chức năng 5 của ngắt 0x10 để chọn trang hiển thị.

```
//CT9_03.CPP  
#include <dos.h>  
#include <conio.h>  
char d1[]={'C',1*16+14,'H',1*16+14,'U',1*16+14,'C',1*16+14};  
char d2[]={'M',2*16+15,'U',2*16+15,'N',2*16+15,'G',2*16+15};  
void main()  
{  
    union REGS v,r;  
    clrscr();  
    //Mặc định hiển thị trang 0  
    for (int i=0;i<8;++i)  
        pokeb(0xb800,i,d1[i]);  
    getch();  
}
```



```
//Hien thi trang 1
v.h.ah = 5 ; v.h.al = 1 ;
int86(0x10,&v,&r);
for (i=0;i<8;++i)
pokeb(0xb800,0x1000+i,d2[i]);
getch();
//Hien thi trang 0
v.h.ah = 5 ; v.h.al = 0 ;
int86(0x10,&v,&r);
getch();
//Hien thi trang 1
v.h.ah = 5 ; v.h.al = 1 ;
int86(0x10,&v,&r);
getch();
}
```

Bài 3. Chuyển đổi địa chỉ

3.1. Để chuyển từ địa chỉ thực sang địa chỉ phân đoạn ta dựng các macro:

unsigned FP_SEG(địa_chỉ_thực)

unsigned FP_OFF(địa_chỉ_thực)

3.2. Để chuyển từ địa chỉ phân đoạn sang địa chỉ thực ta dựng macro:

void far *MK_FP(seg,off)

Ví dụ 1. Sau khi thực hiện các câu lệnh:

char buf[100];

unsigned ds,dx;

ds = FP_SEG(buf); dx = FP_OFF(buf);

thì ds:dx chứa địa chỉ của mảng buf.

Ví dụ 2. Sau khi thực hiện các câu lệnh:

char far *pchar;

pchar = (char far*)MK_FP(0xb800:0);

thì pchar trỏ tới địa chỉ đầu của bộ nhớ màn hình. Khi đó ta có thể sử dụng các lệnh gán để truy cập trực tiếp tới bộ nhớ màn hình.

Bài 4. các ví dụ minh họa

Chương trình 1. Chương trình minh họa cách truy cập trực tiếp vào bộ nhớ màn hình có địa chỉ đầu là 0xB800:0. Chương trình gồm hàm main() và hai hàm sau:

1. Hàm cuaso

void cuaso(int dongt,int cott,int dongd,int cotd,int maucs);

thiết lập một cửa sổ màu có tọa độ góc trên-trái là (dongt, cott) và góc dưới-phải là (dongd,cotd). Màu cho bởi tham số maucs. ở đây sử dụng hàm pokeb và địa chỉ phân đoạn.

2. Hàm duarmh

```
void duarmh(char *day, int dong, int cotd, int cotc,int m_nen, int m_chu);
```

sẽ đưa ra màn hình một dãy ký tự (chứa trong dãy) tại dòng dong, từ cột cotd đến cotc. Màu nền cho bởi m_nen, màu chữ cho bởi m_chữ. ở đây sử dụng toán tử gán trên địa chỉ thực.

Trong hàm main() sẽ sử dụng các hàm cuaso và duarmh để tạo hai cửa sổ và viết hai dòng chữ trên trang màn hình thứ hai (từ dòng 26 đến dòng 50).

```
/*
```

```
chương trình minh họa cách truy nhập trực tiếp vào bộ  
nhớ của màn hình
```

```
*/
```

```
#include "dos.h"
```

```
#include "conio.h"
```

```
void duarmh(char *day, int dong,int cotd, int cotc,int m_nen, int m_chu);
```

```
void cuaso(int dongt,int cott,int dongd,int cotd,int maucs);    495                496
```

```
main()
```

```
{
```

```
    cuaso(26,1,50,80,BLUE);
```

```
    duarmh("Chuc mung nam moi", 28, 30, 50, MAGENTA, WHITE);
```

```
    cuaso(30,20,46,60,RED);
```

```
    duarmh("Chuc mung nam moi", 40, 30, 50, MAGENTA, YELLOW);
```

```
    getch();
```

```
}
```

```
void cuaso(int dongt,int cott,int dongd,int cotd,int maucs)
```

```
/* Dung dia phan doan */
```

```
{
```

```
    int i, j, p, t, dt, dd, mau;
```

```
    union REGS v, r;
```

```
    /* Xac dinh thuoc tinh mau */
```

```
    mau = (maucs << 4)+maucs;
```

```
    /*
```

```
    Xac dinh trang man hinh t
```

```
va cac chi so dong tren dt, dong duoi dd
```

```
trong trang t
```

```
*/
```

```
t=(dongt-1)/25;
```

```
dt=(dongt-1)-t*25; dd=(dongd-1)-t*25;
```

```
/* Chon t la trang hien thi */
```

```
v.h.ah=5;v.h.al=t; int86(0x10,&v,&r);
```

```
/*
```

```
Dua cac khoang trong (ma 32) va thuoc tinh mau
```

```
vao cac vi tri thich hop cua bo nho man hinh
```

```
*/
```

```

for (i=dt;i<=dd;++i)
{
    p=t*4096+i*160+(cott-1)*2;
    for (j=0;j<=cotd-cott;++j)
    {
        pokeb(0xb800,p+2*j,32);
        pokeb(0xb800,p+2*j+1,mau);
    }
}
}

void duarmh(char *day, int dong, int cotd, int cotc, int m_nen,
            int m_chu)
/* Dung dia chi thuc */
{
    int i,p,t,d,kt,mau;
    char far *buf;
    union REGS v,r;
    /* Lay dia chi thuc cua bo nho man hinh */
    buf=(char far*)MK_FP(0xb800,0);
    /* Xac dinh thuoc tinh mau */
    mau = (m_nen << 4)+m_chu;
    /*
    Xac dinh trang man hinh t
    va cac chi so dong d trong trang t
    */
    t=(dong-1)/25; d=dong-1-t*25;
    /* Chon t la trang hien thi */
    v.h.ah=5;v.h.al=t; int86(0x10,&v,&r);
    p=t*4096+d*160+(cotd-1)*2;
    /*
    Dua cac ky tu va thuoc tinh mau
    vao cac vi tri thích hợp của bo nho man hinh
    */
    for (i=0;i<=cotec-cotd;++i)
    {
        if ((kt=day[i])==0) break;
        buf[p+2*i]=kt;
        buf[p+2*i+1]=mau;
    }
}

```

Chương trình 2. Biết địa chỉ của các thủ tục xử lý ngắt được lưu trữ trong bộ nhớ từ địa chỉ 0000:0000 đến 0000:0x0400. Chương trình sẽ cho biết địa chỉ của thủ tục xử lý ngắt n (giá trị n nhập vào từ bàn phím). Số hiệu của ngắt được tính từ 0, nhưng n được đánh số từ 1.

```
/*
Xac dinh dia chi cac thu tuc ngat */
#include "dos.h"
#include "conio.h" #include "stdio.h"
main()
{
    unsigned char far *p; /*p se tro toi bang vecto ngat*/
    int n; /* n - so hieu ngat, n=1,2,... */
    int k; /* vi tri cua ngat n trong bang vecto ngat */
    unsigned seg, off;
    /* p tro toi bang vecto ngat */
    p=(unsigned char far*)MK_FP(0,0);
    clrscr();
    while(1)
    {
        printf("\n So hieu ngat (Bam 0 - Ket thuc): ");
        scanf("%d",&n); if(n==0) break;
        k=(n-1)*4;
        off=p[k]+256*p[k+1]; seg=p[k+2]+256*p[k+3];
        printf("\nDia chi %x:%x",seg,off);
    }
}
```

Chương trình 3. Chương trình minh họa cách dựng con trỏ hàm để thực hiện thủ tục khởi động lại máy của DOS, biết địa chỉ đầu của thủ tục này là 0xFFFF:0000 . Chương trình yêu cầu nhập mật khẩu. Nếu chọn đúng (bấm ABCD và Enter) thì chương trình tiếp tục làm việc, nếu vào sai thì sẽ khởi động lại máy.

```
#include <dos.h>
#include <conio.h>
#include <iostream.h>
#include <ctype.h>
typedef void far (*HAM)(void);
void khoi_dong_may(void)
{
    HAM f;
    f = (HAM)MK_FP(0xFFFF,0);
    f();
}
char mat_khau[] = {'A','B','C','D'};
```

```
int n = sizeof(mat_khau)/sizeof(char);
void main()
{
    char i, ch, sai_mat_khau;
    clrscr();
    i=0;
    sai_mat_khau=0;
    cout << "\nMat khau: ";
    while(1)
    {
        ch=getch();
        if (ch==13) break;
        cout << '*';
        if (i<n)
        {
            if (toupper(ch)!=mat_khau[i])
                sai_mat_khau=1;
        }
        else
            sai_mat_khau=1;
        ++i;
    }
    if (sai_mat_khau)
    {
        cout << "\nSai mat khau, Khoi dong lai may";
        getch();
        khoi_dong_may();
    }
    else
    {
        cout << "\nDung mat khau, tiep tục chương trình";
        getch();
    }
}
```

Chương trình 4. Chương trình minh họa cách dựng biến con trỏ để lấy dữ liệu về thời gian hệ thống chứa trong 4 byte bắt đầu từ địa chỉ 0:0x46C . Chương trình cũng minh họa cách truy nhập trực tiếp bộ nhớ màn hình văn bản (địa chỉ đầu là 0xB800:0) và cách bắt phím tổng quát. Chương trình sẽ in ra màn hình các chữ cái một cách ngẫu nhiên. Khi bấm phím F1 chương trình tạm dừng để thông báo thời gian. Để kết thúc chương trình bấm phím ESC.

```
#include <dos.h>
```

```
#include <conio.h>
#include <stdlib.h>
#define VT 132 // vi tri thong bao
//Dia chi vung nho man hinh
char far *p_mh = (char far*)MK_FP(0xB800,0) ;
//Dia chi 4 byte chua thoi gian
unsigned long far *t_time=(unsigned long far*)MK_FP(0,0x46C);
char buf_time[]={'T',47,'T',47,'M',47,'E',47,':', 47, 32, 47, 32, 47,
                32, 47, 32, 47, 32, 47, 32, 47, 32, 47, 32, 47, 32, 47};
char buf_luu[28];
void thong_bao_thoi_gian()
{
    //Luu trang thai man hinh
    for (int i=0; i<28; ++i)
        buf_luu[i]=p_mh[i];
    // Xac dinh gio, phut, giay
    int gio = (int)(*t_time/65543) ;
    unsigned long du = *t_time%65543 ;
    int phut = (int)(du/1092);
    du = du%1092;
    int giay = (int)(du/18);
    //Doi ra ky tu dua vao mang buf_time
    buf_time[12]=gio/10 + 48;
    buf_time[14]=gio%10 + 48;
    buf_time[18]=phut/10 + 48;
    buf_time[20]=phut%10 + 48;
    buf_time[24]=giay/10 + 48;
    buf_time[26]=giay%10 + 48;
    //Dua thong bao goi ra man hinh
    for (i=0; i<28; ++i)
        p_mh[i] = buf_time[i];
    getch();
    //Khoi phuc man hinh
    for (i=0; i<28; ++i)
        p_mh[i] = buf_luu[i];
}

void main()
{
    int ch1, ch2;
    clrscr();
```

```
while(1)
{
    if (kbhit())
    {
        ch1=getch();
        if (ch1==0) ch2=getch();
        if (ch1==27) //ESC
            break;
        if (ch1==0 && ch2==59) // Bam F1
            thong_bao_thoi_gian();
    }
    //In cac chu cai mot cach ngau nhien
    gotoxy(random(80)+1,random(25)+1);
    putch(random(26)+65);
    delay(400);
}
}
```

Chương 10

Một số chương trình hướng đối tượng trên C++

Chương này trình bày thêm một số chương trình hướng đối tượng trên C++. Đây là các chương trình tương đối phức tạp, hữu ích và sử dụng các công cụ mạnh của C++ như: Cách truy nhập trực tiếp bộ nhớ màn hình, kỹ thuật đồ họa, con trỏ void, tính kế thừa, lớp cơ sở trừu tượng, tương ứng bội, phương thức ảo.

Bài 1. Lớp cửa sổ

Chương trình gồm lớp `cua_so` và lớp `stack`

+ Lớp cửa sổ

Thuộc tính gồm:

```
char *noidung; // Trỏ đến vùng nhớ chứa nội dung
                // soạn thảo trên cửa sổ
int cao,rong ; // Chiều cao và chiều rộng cửa sổ
int mau;      // mau = 16*mau_nen + mau_chu
int ra_mh;    // Cho biết cửa sổ đã được đưa ra màn hình chưa?
int posx,posy; // Vị trí trên trái của cửa sổ trên màn hình
word *pluu;   // Trỏ đến vùng nhớ chứa nội dung
                // phần màn hình bị cửa sổ đè lên
```

Phương thức gồm:

```
cua_so();
cua_so(int c,int r,byte mau_nen, byte mau_chu);
int push(int x,int y); // Đưa cửa sổ ra màn hình tại (x,y)
                        // cho phép soạn thảo trên cửa sổ
                        // Bấm F6 chuyển sang cửa sổ khác
                        // Bấm ESC kết thúc
void pop(); // Tháo gỡ cửa sổ và khôi phục màn hình
int get_ra_mh();
```

+ Lớp stack (dùng để quản lý một dãy cửa sổ)

Thuộc tính gồm:

```
int max; //Số cửa sổ cực đại có thể quản lý
int num; //Số cửa sổ hiện có trong stack
cua_so **pcs; //Con trỏ trỏ đến vùng nhớ chứa
               //địa chỉ của các đối tượng cua_so
```

Phương thức gồm:

```
stack();
stack(int max_cs);
int accept(cua_so *cs,int x,int y); //Đưa một cửa sổ
                                     //vào stack, nó sẽ hiện lên màn hình
void del(); // Loại cửa sổ khỏi stack, nó sẽ bị xóa
```


// khởi màn hình

Nội dung chương trình:

- + Đầu tiên hiện cửa sổ thứ nhất nền GREEN chữ WHITE. Có thể soạn thảo trên đó.
- + Nếu bấm ESC kết thúc chương trình, nếu bấm F6 thì hiện thêm cửa sổ thứ hai nền CYAN chữ MAGENTA. Có thể soạn thảo trên đó.
- + Nếu bấm ESC kết thúc chương trình, nếu bấm F6 thì hiện thêm cửa sổ thứ ba nền RED chữ YELLOW. Có thể soạn thảo trên đó.
- + Đang ở một cửa sổ, nếu bấm ESC thì kết thúc chương trình, nếu bấm F6 thì hiện cửa sổ tiếp theo (theo thứ tự vòng quanh: 1 -> 2 -> 3 -> 1).

Chương trình sử dụng phương pháp truy nhập trực tiếp bộ nhớ màn hình trình bày trong chương 9.

```
// CT10_01.CPP
// lop_cua_so
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <alloc.h>
typedef unsigned int word;
typedef unsigned char byte;
struct kt_word
{
    word kt;
};
struct kt_byte
{
    byte ma, mau;
};
union ky_tu
{
    struct kt_byte h;
    struct kt_word x;
};
typedef union ky_tu far *VP;
VP vptr=(VP)MK_FP(0xb800,0);
// Vi tri x,y tren man hinh
#define VPOS(x,y) (VP)(vptr + ((y)-1)*80+(x)-1)
class cua_so
{
private:
    char *noidung;
    int cao, rong;
    int mau; // mau = 16*mau_nen + mau_chu
```

```
int ra_mh;
int posx,posy;
word *pluu;
public:
    cua_so();
    cua_so(int c,int r,byte mau_nen, byte mau_chu);
    int push(int x,int y);
    void pop();
    int get_ra_mh();
};
cua_so::cua_so()
{
    cao=rong=mau=ra_mh=posx=posy=0;
    noidung=NULL; pluu=NULL;
}
cua_so::cua_so(int c,int r,byte mau_nen, byte mau_chu)
{
    cao=c; rong=r;
    mau= 16*mau_nen+mau_chu;
    ra_mh=posx=posy=0;
    noidung = (char*)malloc(cao*rong);
    for (int i=0;i<cao*rong;++i)
        noidung[i]=32;
    pluu= (word*)malloc(2*cao*rong);
}
int cua_so::push(int x,int y)
{
    word *p= pluu; char *pnd=noidung;
    VP ptr;
    int i,j;
    // Luu man hinh
    if (ra_mh==0)
    {
        ra_mh=1; posx=x;posy=y;
        for (i=posx;i<=posx+rong-1;++i)
            for (j=posy;j<=posy+cao-1;++j)
            {
                ptr=VPOS(i,j); *p=ptr->x.kt; ++p;
            }
    }
    // Hien noi dung dang soan thao tren cua so
```

```
for (i=posx;i<=posx+rong-1;++i)
for (j=posy;j<=posy+cao-1;++j)
{
    ptr=VPOS(i,j);
    ptr->h.mau=mau;
    ptr->h.ma=*pnd; ++pnd;
}
// Soan thao
int xx=posx,yy=posy,ch1,ch2;
while (1)
{
    gotoxy(xx,yy);
    if ((ch1=getch())==0) ch2=getch();
    if (ch1==27)break; // ESC Ket Thuc Soan Thao
    else if (ch1==0&&ch2==64)break; //F6
    else if (ch1==13)
    {
        ++yy; xx=posx; if(yy>=posy+cao) break;
    }
    else if (ch1!=0)
    {
        ptr=VPOS(xx,yy);
        ptr->h.ma=ch1;
        ++xx;
        if (xx>=posx+rong) {++yy; xx=posx;}
        if (yy>=posy+cao) break;
    }
    else if (ch2==72||ch2==80||ch2==75||ch2==77)
    {
        if (ch2==72) yy--;
        else if (ch2==80) ++yy;
        else if (ch2==75) --xx;
        else ++xx;
        if (xx<posx) xx=posx;
        if (xx>=posx+rong) {++yy; xx=posx;}
        if (yy<posy) yy=posy;
        if (yy>=posy+cao) break;
    }
}
// Luu ket qua soan thao
pnd=noidung;
for (i=posx;i<=posx+rong-1;++i)
```

```

    for (j=posy;j<=posy+cao-1;++j)
    {
        ptr=VPOS(i,j);
        *pnd=ptr->h.ma; ++pnd;
    }
    if (ch1==0&&ch2==64) return 0; //F6
    else return 1;
}

void cua_so::pop() // Khoi phuc vung nho bi cua so chiem
{
    if (ra_mh==0) return;
    ra_mh=0;
    word *p=pluu;
    VP ptr;
    int i,j;
    for (i=posx;i<=posx+rong-1;++i)
        for (j=posy;j<=posy+cao-1;++j)
        {
            ptr=VPOS(i,j); ptr->x.kt=*p; ++p;
        }
}

int cua_so::get_ra_mh()
{
    return ra_mh;
}

//class stack
class stack
{
private:
    int max,num;
    cua_so **pcs;
public:
    stack();
    stack(int max_cs);
    int accept(cua_so *cs,int x,int y);
    void del();
};

stack::stack()
{
    max=num=0; pcs=NULL;
}

```

```
stack::stack(int max_cs)
{
    max=max_cs; num=0;
    pcs=(cua_so**)malloc(max*sizeof(cua_so*));
    for (int i=0;i<max;++i) pcs[i]=NULL;
}
int stack::accept(cua_so *cs,int x,int y)
{
    int gt;
    if (num==max)return 0;
    if (!cs->get_ra_mh())
    {
        pcs[num]=cs; ++num;
    }
    gt=cs->push(x,y);
    return gt;
}
void stack::del()
{
    if (num==0) return;
    --num;
    pcs[num]->pop();
    pcs[num]=NULL;
}
main()
{
    int ch;
    cua_so w1(10,40,GREEN,WHITE),
            w2(12,42,CYAN,MAGENTA),
            w3(14,44,RED,YELLOW);
    stack s(4);
    clrscr();
    while(1)
    {
        ch=s.accept(&w1,5,5);
        if(ch==1)break;
        ch=s.accept(&w2,8,8);
        if(ch==1)break;
        ch=s.accept(&w3,11,11);
```

```
    if(ch==1)break;
}
s.del(); s.del(); s.del();
}
```

Bài 2. Lớp menu

Lớp cmenu có 2 phương thức để tạo lập và sử dụng menu:

1. Hàm tạo

```
cmenu(int so_cn_menu,char **nd_menu);
```

dùng để tạo một menu (đối tượng kiểu cmenu). Hàm tạo chứa 2 đối là:

- + Biến so_cn_menu chứa số chức năng của menu
- + Con trỏ nd_menu trỏ tới một vùng nhớ chứa địa chỉ các chuỗi ký tự dùng làm tiêu đề menu và tiêu đề các chức năng menu.

Ví dụ các câu lệnh:

```
char *nd[]={ "Quản lý vật tư", "Nhập số liệu",
            "Tìm kiếm","Kết thúc"};
```

```
cmenu mc(3,nd);
```

sẽ tạo một menu mc gồm 3 chức năng: Nhập số liệu, Tìm kiếm và Kết thúc. Menu có tiêu đề là: Quản lý vật tư

2. Phương thức

```
int menu(int x,int y,int mau_nen,int mau_chon);
```

thực hiện các việc sau:

- + Hiện thị menu tại vị trí (x,y) trên màn hình. Menu có màu nền xác định bởi đối mau_nen và màu chức năng định chọn (hộp sáng) xác định bởi đối mau_chon.
- + Cho phép sử dụng các phím mũi tên lên, xuống để di chuyển hộp sáng và dùng phím Enter để thoát khỏi phương thức.
- + Sau khi thoát khỏi, phương thức trả về giá trị bằng số thứ tự (tính từ 1) của chức năng được chọn.

Chương trình dưới đây xây dựng lớp cmenu và minh họa cách sử dụng lớp này.

```
/*
```

```
CT10_02.CPP
```

```
menu.cpp
```

```
lop cmenu
```

```
*/
```

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

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

```
#include <dos.h>
```

```
#include <alloc.h>
```

```
#include <string.h>
```

```
typedef unsigned int word;
```

```
typedef unsigned char byte;
```

```
struct kt_word
{
    word kt;
};
struct kt_byte
{
    byte ma, mau;
};
union ky_tu
{
    struct kt_byte h;
    struct kt_word x;
};
typedef union ky_tu far *VP;
VP vptr=(VP)MK_FP(0xb800,0);
// Vi tri x,y tren man hinh
#define VPOS(x,y) (VP)(vptr + ((y)-1)*80+(x)-1)
class cmenu
{
private:
    int so_cn,cao,rong,posx,posy;
    int chon;
    char **nd;
private:
    void hiendc(char *dc,int x,int y, int mau);
    void hien_menu(int x,int y,int mau_nen,int mau_chon);
public:
    cmenu(int so_cn_menu,char **nd_menu);
    int menu(int x,int y,int mau_nen,int mau_chon);
};
cmenu::cmenu(int so_cn_menu,char **nd_menu)
{
    cao=so_cn=so_cn_menu; nd=nd_menu;
    rong=0;
    chon=1;
    int d;
    for(int i=0;i<=so_cn;++i)
        if( (d=strlen(nd[i])) > rong) rong=d;
}
void cmenu::hiendc(char *dc,int x,int y, int mau)
```

```
{
    VP ptr; int i;
    byte m=16*mau+15; //chu trang
    for(i=0;i<rong;++i)
    {
        ptr=VPOS(x+i,y);
        ptr->h.mau=m ;
        ptr->h.ma=32;
    }
    for(i=0;i<rong;++i)
    {
        ptr=VPOS(x+i,y);
        if(dc[i]==0)break;
        ptr->h.ma=dc[i];
    }
}

void cmenu::hien_menu(int x,int y,int mau_nen,int mau_chon)
{
    for(int i=0;i<=so_cn;++i)
        hiendc(nd[i],x,y+i,mau_nen);
    hiendc(nd[chon],x,y+chon,mau_chon);
}

int cmenu::menu(int x,int y,int mau_nen,int mau_chon)
{
    int ch1,ch2,chonluu;
    //Trinh bay
    hien_menu(x,y,mau_nen,mau_chon);
    //Bat phim
    while(1)
    {
        if( (ch1=getch())==0 ) ch2=getch();
        if(ch1==13) //chon chuc nang
            return (chon);
        else if( (ch1==0)&&(ch2==80||ch2==72))
        {
            //Di chuyen hop sang
            chonluu=chon;
            if(ch2==80) ++chon;
            else --chon;
            if(chon<1) chon=cao;
            else if(chon>cao) chon=1;
        }
    }
}
```



```
        if(chon!=chonluu)
        {
            hiendc(nd[chonluu],x,y+chonluu,mau_nen);
            hiendc(nd[chon],x,y+chon,mau_chon);
        }
    }
}
}
char *nd[]={"TINH DIEN TICH", "Tam giac", "Hinh tron",
            "Chu nhât", "Hinh vuong", "Ket thuc chuong trinh"};
void main()
{
    cmenu mc(5,nd); int chon;
    clrscr();
    while(1)
    {
        chon=mc.menu(5,5,BLUE,MAGENTA);
        if(chon==1)
        {
            clrscr();
            puts("TAM GIAC");
            getch(); clrscr();
        }
        else if(chon==2)
        {
            clrscr();
            puts("HINH TRON");
            getch();clrscr();
        }
        else if(chon==3)
        {
            clrscr();
            puts("CHU NHAT");
            getch();clrscr();
        }
        else if(chon==4)
        {
            clrscr();
            puts("HINH VUONG");
            getch(); clrscr();
        }
        else break;
    }
}
```

```
}  
}
```

Bài 3. Lớp hình học

Chương trình dưới đây gồm:

- + Lớp “hình” là lớp cơ sở trừu tượng
- + Và 3 lớp dẫn suất từ lớp “hình” là:
 - Lớp “khoihop” biểu thị các khối hộp lập phương
 - Lớp “duong” biểu thị các đoạn thẳng qua 2 điểm
 - Lớp “tron” biểu thị các đường tròn

Chương trình minh họa cách dùng tượng ứng bội và phương thức ảo. Nội dung chương trình như sau:

- + Khi chạy chương trình sẽ thấy xuất hiện một khối hộp lập phương.
- + Có thể di chuyển khối hộp bằng các phím mũi tên.
- + Bấm phím Q sẽ xuất hiện một đoạn thẳng.
- + Có thể di chuyển đoạn thẳng bằng các phím mũi tên.
- + Bấm phím Q sẽ xuất hiện một đường tròn.
- + Có thể di chuyển đường tròn bằng các phím mũi tên.
- + Bấm phím Q sẽ kết thúc chương trình.

/*

CT10_03.CPP

LOP hình học

Minh họa cách dùng:

- + lớp cơ sở trừu tượng
- + Tượng ứng bội và phương thức ảo

*/

```
#include <graphics.h>
```

```
#include <process.h>
```

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

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

```
char getkey(int &dx,int &dy);
```

```
class hình
```

```
{
```

```
protected:
```

```
int mau;
```

```
public:
```

```
hình(void)
```

```
{
```

```
    mau=0;
```

```
}
```

```
    hình(int m)
    {
        mau=m;
    }
    virtual void dchuyen(int b)=0;
};
class khoihop : public hình
{
    private:
        int x,y;
        int a ;
    public:
        khoihop(void):hình()
        {
            x=y=a=0;
        }
        khoihop(int m,int x1,int y1, int a1):hình(m)
        {
            x=x1;
            y=y1;
            a=a1;
        }
        virtual void dchuyen(int b);
        void hien(void)
        {
            setfillstyle(1,mau);
            bar3d(x,y,x+a,y+a,a/2,1);
        }
        void an(void)
        {
            setfillstyle(1,getbkcolor());
            bar(x,y-a/2,x+a+a/2,y+a+a/2);
        }
};
class duong:public hình
{
    private:
        int x1,y1,x2,y2;
    public:
```

```
    duong(void):hinh()
    {
        x1=x2=y1=y2=0;
    }
    duong(int m,int a,int b,int c,int d):hinh(m)
    {
        x1=a;y1=b;x2=c;y2=d;
    }
    virtual void dchuyen(int b);
    void hien(void)
    {
        setcolor(mau);
        line(x1,y1,x2,y2);
    }
    void an(void)
    {
        setcolor(getbkcolor());
        line(x1,y1,x2,y2);
    }
};

class tron:public hinh
{
private:
    int x,y,r;
public:
    tron(void):hinh()
    {
        x=y=r=0;
    }
    tron(int m,int a,int b,int d):hinh(m)
    {
        x=a; y=b; r=d;
    }
    virtual void dchuyen(int b);
    void hien(void)
    {
        setcolor(mau);
        circle(x,y,r);
    }
    void an(void)
    {
```

```
        setcolor(getbkcolor());
        circle(x,y,r);
    }
};

char getkey(int &dx,int &dy)
{
    int ch1,ch2;
    dx=dy=0;
    while (1)
    {
        ch1=getch();
        if (ch1==0)
            ch2=getch();
        if (ch1=='q' || ch1=='Q') return('q');
        if ((ch1==0 && (ch2==80 || ch2==72 || ch2==75 || ch2==77)))
        {
            if (ch2==80) dy=1;
            else if (ch2==72) dy=-1;
            else if (ch2==77) dx=1;
            else dx=-1;
            return(0);
        }
    }
}

void khoihop::dchuyen(int b)
{
    int dx,dy;
    while (1)
    {
        hien();
        if (getkey(dx,dy)=='q') break;
        an();
        x+=b*dx;
        y+=b*dy;
    }
}

void duong::dchuyen(int b)
{
    int dx,dy;
    while (1)
    {
```

```
    hien();
    if (getkey(dx,dy)=='q') break;
    an();
    x1+=b*dx;
    x2+=b*dx;
    y1+=b*dy;
    y2+=b*dy;
}
}
void tron::dchuyen(int b)
{
    int dx,dy;
    while (1)
    {
        hien();
        if (getkey(dx,dy)=='q') break;
        an();
        x+=b*dx;
        y+=b*dy;
    }
}
void main()
{
    int mh=0,mode=0;
    initgraph(&mh,&mode,"");
    if (graphresult())
    {
        printf("\n LOI");
        getch();
        exit(0);
    }
    setbkcolor(0);
    // setwritemode(0);
    hình *h[3];
    khoihop M(4,300,200,15);
    duong D(10,10,10,60,60);
    tron T(14,200,200,50);
    h[0]=&M; h[1]=&D;h[2]=&T;
    for(int i=0;i<3;++i)
        h[i]->dchuyen(10);
    closegraph();
}
```

}

Bài 4. Các lớp ngăn xếp và hàng đợi

Chương trình tổ chức thành 4 lớp chính:

1. Lớp container (thùng chứa) gồm 2 thuộc tính:

```
unsigned long count; //Số phần tử trong thùng chứa
void (*errhandler)(); //Con trỏ tới hàm xử lý lỗi
```

2. Lớp s_list thừa kế từ lớp container, có thêm 2 thuộc tính các con trỏ kiểu cấu trúc listnode:

```
struct listnode
{
    void *dataptr;
    listnode *next;
};
```

```
listnode *head; // Trỏ tới đầu danh sách
```

```
listnode *tail; // Trỏ tới cuối danh sách
```

Các phần tử được chứa trong lớp s_list dưới dạng một danh sách móc nối đơn. Mỗi nút chứa địa chỉ của một phần tử. Do ở đây dùng kiểu con trỏ void nên có thể đưa vào lớp s_list các phần tử có kiểu bất kỳ.

3. Lớp stack thừa kế từ lớp s_list

4. Lớp queue thừa kế từ lớp stack

Các lớp stack và queue không có các thuộc tính riêng. Hai phương thức quan trọng của các lớp này là:

```
virtual int store(void *item) ; // Cất vào một phần tử
virtual void *retrieve () ; // Lấy ra một phần tử
```

Chú ý là: Lớp stack hoạt động theo nguyên tắc LIFO (vào sau ra trước) còn lớp queue hoạt động theo nguyên tắc FIFO (vào trước ra trước) .

Chương trình sau minh hoạ cách dùng liên kết bội, phương thức ảo và con trỏ kiểu void để quản lý các kiểu dữ liệu khác nhau.

Hoạt động của chương trình như sau:

+ Trước tiên lần lượt đưa địa chỉ của biến đối tượng ts1, chuỗi “HA NOI”, biến nguyên a, biến đối tượng ts2 và biến thực x vào ngăn xếp s1 và hàng đợi q1.

+ Thực hiện phép gán các biến đối tượng:

```
s2 = s1 ;
```

```
q2 = q1 ;
```

+ Lấy các phần tử trong ngăn xếp s2 theo trình tự ngược với lúc đưa vào.

+ Lấy các phần tử trong hàng đợi q2 theo trình tự như lúc đưa vào.

/*

CT10_05.CPP

Lop vat chua (container)

Lop danh sach moc noi

Lop ngan xep

Lop hang doi

Chu y:

1. constructor sao chep cua lop dan suat
2. toan tu gan cua lop dan suat
3. co the dung cac phuong thuc khac
de viet constructor va destructor
4. Dung con tro this

```
*/  
#include <stdio.h>  
#include <iostream.h>  
#include <iomanip.h>  
#include <conio.h>  
#include <alloc.h>  
#include <dos.h>  
//Lop container  
class container  
{  
    protected:  
        unsigned long count; //so pt trong thung chua  
        void (*errhandler)();  
    public:  
        container();  
        container(const container &c); // Ham tao sao chep  
        void operator=(const container &c); // Gan  
        unsigned long getcount(); // Cho biet so phan tu  
        // Dinh ham xl loi  
        void seterrorhandler(void (*userhandler)());  
        // 4 phuong thuc thuan ao  
        virtual int store(void *item)=0; //Cat mot phan tu vao thung  
        virtual void *examine()=0; // Xem gia tri mot phan tu  
        virtual void *retrieve ()=0; // Lay mot pt ra  
        virtual void empty()=0; // Lam cho thung tro nen rong  
};  
// Cai dat  
// Ham xl loi mac dinh  
void defaulthandler();  
void defaulthandler()  
{  
    puts("\nContainer error: memory allocation failure");  
}  
container::container ()
```



```
{
    count=0; errhandler= defaulthandler;
}
container::container(const container &c)
{
    count=c.count; errhandler=c.errhandler;
}
// Gan
void container::operator=(const container &c)
{
    count=c.count; errhandler=c.errhandler;
}
// Cho biet so pt
unsigned long container::getcount()
{
    return count;
}
// Dinh ham xl loi
void container::seterrorhandler(void (*userhandler)())
{
    errhandler=userhandler;
}
// Lop danh sach moc noi don
class s_list:public container
{
protected:
    //Cau truc mot nut trong ds
    struct listnode
    {
        void *dataptr;
        listnode *next;
    };
    listnode *head;
    listnode *tail;
private:
    // phuong thuc sao chep
    void copy(const s_list &s1);
public:
    s_list();
    s_list(const s_list &s1);
```

```
~s_list();
void operator=(const s_list &s1);
// 4 phương thức
virtual int store(void *item)=0; // Cat một phần tử vào
                                // thùng
virtual void *examine()=0; // Xem giá trị một phần tử
virtual void *retrieve ()=0; // Lấy một phần tử ra
virtual void empty(); // Làm cho thùng trở nên rỗng
};
//Cài đặt
void s_list::copy(const s_list &s1)
{
    head=NULL; tail=NULL;
    listnode *temp = s1.head;
    while(temp!=NULL)
    {
        if(head==NULL)
        {
            head= new listnode;
            if(head==NULL) errhandler();
            tail=head;
        }
        else
        {
            tail->next = new listnode;
            if(tail->next == NULL) errhandler();
            tail = tail->next;
        }
        tail->dataptr= temp->dataptr;
        tail->next=NULL;
        temp = temp->next;
    }
}
// constructor
s_list::s_list() : container()
{
    head=NULL; tail=NULL;
}
s_list::s_list(const s_list &s1):container(s1)
{
```

```

        copy(s1);
    }
s_list::~s_list()
{
    this->empty();
}
void s_list::operator=(const s_list &s1)
{
    this->empty();
    count=s1.count;
    copy(s1);
}
void s_list::empty()
{
    listnode *q,*p;
    p = head; head=NULL; tail=NULL;
    while (p!=NULL)
    {
        q=p; p=p->next;
        delete q;
    }
}
// Lop stack
class stack:public s_list
{
public:
    stack();
    stack(const stack &st);
    void operator=(const stack &st);
    virtual int store(void *item); // Cat mot phan tu vao thung
    virtual void *examine(); // Xem gia tri mot phan tu
    virtual void *retrieve(); // Lay mot pt ra
};
stack::stack():s_list()
{
}
stack::stack(const stack &st):s_list(st)
{
}
void stack::operator=(const stack &st)
{
    this->s_list::operator=(st); //Dung toan tu gan cua s_list

```

```
}
int stack::store(void *item) // Cat mot phan tu vao thung
{
    //Dua vao dau danh sach
    listnode *p;
    p= new listnode ;
    if(p==NULL) return 1;
    count++;
    p->dataptr=item; p->next=head;
    head=p; return 0;
}
void *stack::examine() // Xem gia tri mot phan tu
{
    if(count==0) return NULL;
    else
        return head->dataptr;
}
void *stack::retrieve() // Lay mot pt ra
{
    if(count==NULL) return NULL;
    else
    {
        listnode *p; void *value;
        value = head->dataptr;
        p=head;
        head = p->next;
        delete p;
        count--;
        return value;
    }
}
// Lop queue
class queue:public stack
{
public:
    queue();
    queue(const queue &q);
    void operator=(const queue &q);
    virtual int store(void *item); // Cat mot phan tu vao thung
};
queue::queue(): stack()
{
```

```
    }
queue::queue(const queue &q):stack(q)
{
}
void queue::operator=(const queue &q)
{
    this->stack::operator=(q); //Dung toan tu gan cua stack
}
int queue::store(void *item)
{
    // Dat vao cuoi
    listnode *q;
    q=new listnode;
    if(q==NULL)return 1;
    // Bo sung
    q->next=NULL; q->dataptr=item;
    if(count==0)
    {
        head=q; tail=q;
    }
    else
    {
        tail->next=q;
        tail=q;
    }
    count++; return 0;
}
class TS
{
private:
    char ht[25];
    int sobd;
    float td;
public:
    void nhap()
    {
        cout << "\nHo ten: ";
        fflush(stdin);
        gets(ht);
        cout << "So bao danh: ";
```

```
        cin >> sobd;
        cout << "Tong diem: " ;
        cin >> td;
    }
    void xuat()
    {
        cout << "\nHo ten: " << ht;
        cout << "\nSo bao danh: " << sobd;
        cout << "\nTong diem: " << setiosflags(ios::showpoint)
            << setprecision(1)<<setw(5)<< td;
    }
};

// Ham main
void main()
{
    stack s1,s2; queue q1,q2;
    TS ts1,ts2,ts;
    int a=123,b;
    float x=3.14,y;
    char *str;
    clrscr();
    ts1.nhap();
    ts2.nhap();
    //Gui vao
    s1.store(&ts1); q1.store(&ts1);
    s1.store("HA NOI"); q1.store("HA NOI");
    s1.store(&a); q1.store(&a);
    s1.store(&ts2); q1.store(&ts2);
    s1.store(&x); q1.store(&x);
    //Lay ra tu ngan xep theo nguyen tac LIFO
    cout << "\n\nLay ra tu ngan xep: " ;
    s2=s1;
    y = *((float*)s2.retrieve());
    cout << "\nSo thuc = " << setiosflags(ios::showpoint)
        << setprecision(2)<< y;
    ts = *((TS*)s2.retrieve());
    ts.xuat();
    b = *((int*)s2.retrieve());
    cout << "\nSo nguyen = " << b;
    str = (char*)s2.retrieve();
```

```
cout << "\nChuoi ky tu: " << str;
ts = *((TS*)s2.retrieve());
ts.xuat();
//Lay ra tu hang doi theo nguyen tac FIFO
cout << "\n\nLay ra tu hang doi:";
q2=q1;
ts = *((TS*)q2.retrieve());
ts.xuat();
str = (char*)q2.retrieve();
cout << "\nChuoi ky tu: " << str;
b = *((int*)q2.retrieve());

cout << "\nSo nguyen = " << b;
ts = *((TS*)q2.retrieve());
ts.xuat();
y = *((float*)q2.retrieve());
cout << "\nSo thuc = " << setiosflags(ios::showpoint)
    << setprecision(2) << y;
getch();
}
```

Bài 5. Các lớp sắp xếp

Trong tệp C_SORT.H dưới đây sẽ chứa 4 lớp sắp xếp: sort, select_sort, quick_sort và heap_sort. tổng quát hơn. So với các lớp sắp xếp trong mục §7 chương 6 thì các lớp ở đây tổng quát hơn ở chỗ:

- + Các lớp trong mục §7 chương 6 chỉ cho phép sắp xếp một dãy số nguyên theo thứ tự tăng dần.
- + Các lớp dưới đây cho phép sắp xếp một dãy phần tử có kiểu bất kỳ (nguyên, thực, cấu trúc, lớp, ...) và theo một tiêu chuẩn sắp xếp bất kỳ.

1. Lớp sort là lớp cơ sở trừu tượng

+ Các thuộc tính:

protected:

```
void *a; // Trỏ tới vùng nhớ chứa dãy
        // phần tử cần sắp xếp
int size; // Độ lớn tính theo byte của phần tử
int (*nho_hon)(void* pt1, void* pt2); // Con trỏ hàm
        // định nghĩa pt1 nhỏ hơn pt2
```

+ Các phương thức:

protected:

```
void hoan_vi(int i, int j); // Hoán vị các phần tử thứ i và j
```

```
void * dia_chi (int m); // Cho địa chỉ của phần tử thứ m
public:
virtual void sapxep(void *a1,int n,int itemsize,
    int (*ss_nho_hon)(void* ,void* )) ; // Sắp xếp dãy
    // n phần tử chứa trong vùng nhớ a1, mỗi phần tử
    // có độ dài itemsize, thứ tự tăng được quy định
    // bởi hàm ss_nho_hon
```

2. Lớp select_sort dẫn xuất từ lớp sort. Lớp này sẽ thực hiện việc sắp xếp theo phương pháp chọn (xem mục §7 chương 6).

+ Các phương thức:

```
public:
virtual void sapxep(void *a1,int n,int itemsize,
    int (*ss_nho_hon)(void* ,void* )) ; // thực hiện
    // sắp xếp theo phương pháp chọn
```

3. Lớp quick_sort dẫn xuất từ lớp sort. Lớp này sẽ thực hiện việc sắp xếp theo phương pháp quick sort (xem mục §7 chương 6)

+ Các phương thức:

```
private:
    void q_sort(int l, int r);
public:
virtual void sapxep(void *a1,int n,int itemsize,
    int (*ss_nho_hon)(void* ,void* )) ; // thực hiện
    // sắp xếp theo phương pháp quick sort
```

4. Lớp heap_sort dẫn xuất từ lớp sort. Lớp này sẽ thực hiện việc sắp xếp theo phương pháp heap sort (xem mục §7 chương 6).

+ Các phương thức:

```
private:
    void shift(int i, int n);
public:
virtual void sapxep(void *a1,int n,int itemsize,
    int (*ss_nho_hon)(void* ,void* )) ; // thực hiện
    // sắp xếp theo phương pháp heap sort
```

Dưới đây là nội dung tệp C_SORT.H

```
//C_SORT.H
```

```
// Lop co so truu tuong
```

```
// Lop sort
```

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



```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
#include <mem.h>
class sort
{
protected:
    void *a;
    int size;
    int (*nho_hon)(void*,void*);
    void* dia_chi(int m)
    {
        return (void*) ((char*)a + size*(m-1));
    }
    void hoan_vi(int i, int j)
    {
        void *tg, *di, *dj;
        di= dia_chi(i);
        dj= dia_chi(j);
        tg = new char[size];
        memcpy(tg,di,size);
        memcpy(di,dj,size);
        memcpy(dj,tg,size);
    }
public:
    virtual void sapxep(void *a1,int n,int itemsize,
        int (*ss_nho_hon)(void*,void*))
    {
        a=a1;
        size=n; // Cho C++ hai long
        size=itemsize;
        nho_hon= ss_nho_hon;
    }
};
class select_sort : public sort
{
public:
    virtual void sapxep(void *a1,int n,int itemsize,
        int (*ss_nho_hon)(void*,void*)) ;
};
void select_sort::sapxep(void *a1,int n,int itemsize,
```

```
int (*ss_nho_hon)(void*,void*)

{
    int i,j,r;
    sort::sapxep(a1,n,itemsz,ss_nho_hon);
    for(i=1; i<n; ++i)
    {
        r=i;
        for(j=i+1; j<=n; ++j)
            if(nho_hon(dia_chi(j),dia_chi(r))) r = j;
        if(r!=i) hoan_vi(i,r);
    }
}

class quick_sort : public sort
{
private:
    void q_sort(int l, int r);
public:
    virtual void sapxep(void *a1,int n,int itemsz,
        int (*ss_nho_hon)(void*,void*)) ;
};

void quick_sort::q_sort(int l, int r)
{
    void *x;
    int i,j;
    x = new char[size];
    if(l < r)
    {
        memcpy(x, dia_chi(l), size);
        i = l; j = r+1;
        do
        {
            ++i; --j;
            while(i<r && nho_hon(dia_chi(i),x)) ++i ;
            while(nho_hon(x,dia_chi(j)) ) --j ;
            if(i<j) hoan_vi(i,j);
        } while (i<j);
        hoan_vi(l,j);
        q_sort(l,j-1);
        q_sort(j+1,r);
    }
}
```

```
}
void quick_sort::sapxep(void *a1,int n,int itemsize,
                        int (*ss_nho_hon)(void*,void*))
{
    sort::sapxep(a1,n,itemsize,ss_nho_hon);
    q_sort(1,n);
}
class heap_sort : public sort
{
private:
    void shift(int i, int n);
public:
    virtual void sapxep(void *a1,int n,int itemsize,
                        int (*ss_nho_hon)(void*,void*));
};
void heap_sort::shift(int i, int n)
{
    int l,r,k;
    l = 2*i; r = l+1;
    if(l>n) return;
    if(l==n)
    {
        if (nho_hon(dia_chi(i), dia_chi(l)))
            hoan_vi(i,l);
        return;
    }
    if(nho_hon(dia_chi(r), dia_chi(l)))
        k = l;
    else
        k = r;
    if (!nho_hon(dia_chi(i), dia_chi(k)))
        return;
    else
    {
        hoan_vi(i,k);
        shift(k,n);
    }
}
void heap_sort::sapxep(void *a1,int n,int itemsize,
                        int (*ss_nho_hon)(void*,void*))
{
    long i;
```

```
sort::sapxep(al,n,itemsz,ss_nho_hon);
// Tao dong
for(i=n/2 ; i>=1; --i) shift(i,n);
// Lap
for(i=n ; i>=2; --i)
{
    hoan_vi(1,i);
    shift(1,i-1);
}
}
```

Bài 6. Ví dụ về Các lớp sắp xếp

Trong mục này trình bày 2 chương trình minh họa cách dùng các lớp nói trên. Chương trình thứ nhất minh họa cách sử dụng các lớp trong tệp C_SORT.H để sắp xếp một dãy thí sinh theo thứ tự giảm và thứ tự tăng của tổng điểm. Chương trình thứ hai minh họa cách dùng các lớp trong C_SORT.H để sắp xếp một dãy số nguyên theo chiều tăng và chiều giảm.

Chương trình 1

```
//CT10-08
// Lop co so truu tuong
// Lop sort
#include "c_sort.h"
class TS
{
private:
    char ht[25];
    int sobd;
    float td;
public:
    float get_td()
    {
        return td;
    }
    void nhap()
    {
        cout << "\nHo ten: ";
        fflush(stdin);
        gets(ht);
        cout << "So bao danh: ";
        cin >> sobd;
        cout << "Tong diem: ";
        cin >> td;
    }
}
```

```
void xuat()
{
    cout << "\nHo ten: " << ht;
    cout << "\nSo bao danh: " << sobd;
    cout << "\nTong diem: " << setiosflags(ios::showpoint)
        << setprecision(1)<<setw(5)<< td;
}

};

int ss_tong_diem_giam(void *ts1, void *ts2)
{
    return ( ((TS*)ts1)->get_td() > ((TS*)ts2)->get_td() );
}

int ss_tong_diem_tang(void *ts1, void *ts2)
{
    return ( ((TS*)ts1)->get_td() < ((TS*)ts2)->get_td() );
}

void main()
{
    TS t[100];
    sort *sa;
    int n,i;
    clrscr();
    cout << "\nSo thi sinh: ";
    cin >> n;
    for(i=1; i<=n; ++i) t[i].nhap();
    for(i=1; i<=n; ++i) t[i].xuat();
    getch();
    cout << "\n\nSap xep giam theo tong diem - PP Select Sort" ;
    sa= new select_sort;
    sa->sapxep( t+1,n,sizeof(TS),ss_tong_diem_giam);
    for(i=1; i<=n; ++i) t[i].xuat();
    delete sa;
    getch();
    cout << "\n\nSap xep tang theo tong diem - PP Select Sort";
    sa= new select_sort;
    sa->sapxep( t+1,n,sizeof(TS),ss_tong_diem_tang);
    for(i=1; i<=n; ++i) t[i].xuat();
    delete sa;
    getch();
    cout << "\n\nSap xep giam theo tong diem - PP Quick Sort" ;
```

```
sa= new quick_sort;
sa->sapxep( t+1,n,sizeof(TS),ss_tong_diem_giam);
for(i=1; i<=n; ++i) t[i].xuat();
delete sa;
getch();
cout << "\n\nSap xep tang theo tong diem - PP Quick Sort" ;
sa= new quick_sort;
sa->sapxep( t+1,n,sizeof(TS),ss_tong_diem_tang);
for(i=1; i<=n; ++i) t[i].xuat();
delete sa;
getch();
cout << "\n\nSap xep giam theo tong diem - PP Heap Sort" ;
sa= new heap_sort;
sa->sapxep( t+1,n,sizeof(TS),ss_tong_diem_giam);
for(i=1; i<=n; ++i) t[i].xuat();
delete sa;
getch();
cout << "\n\nSap xep tang theo tong diem - PP Heap Sort" ;
sa= new heap_sort;
sa->sapxep( t+1,n,sizeof(TS),ss_tong_diem_tang);
for(i=1; i<=n; ++i) t[i].xuat();
delete sa;
getch();
}
```

Chương trình 2

```
//CT10-09
// Lop co so truu tuong
// Lop sort
#include "c_sort.h"
int ss_tang(void *i1,void *i2)
{
    return *((int*)i1) < *((int*)i2);
}
int ss_giam(void *i1,void *i2)
{
    return *((int*)i1) > *((int*)i2);
}
void main()
{
    int i,n;
```

```
struct time t1,t2;
int b[20],a[20], k, tg, sec, hund;
n=10;
sort *s[3];
select_sort ss;
quick_sort qs;
heap_sort hs;
s[0]=&ss; s[1]=&qs; s[2]=&hs;
clrscr();
srand(5000);
for(i=1;i<=n;++i)
b[i]=rand();
cout<<"\nDay ban dau\n ";
for(i=1;i<=n;++i) cout <<b[i]<<" ";
cout<<"\n\nCac day tang sap xep theo ";
cout << "select_sort, quick_sort, heap_sort\n";
for(k=0; k<3; ++k)
{
    for(i=1;i<=n;++i)
        a[i]=b[i];
    s[k]->sapxep (a+1,n,sizeof(int),ss_tang);
    //In
    for(i=1;i<=n;++i) cout <<a[i]<<" ";
    cout<<"\n";
}
cout<<"\n\nCac day giam sap xep theo ";
cout << "select_sort, quick_sort, heap_sort\n";
for(k=0; k<3; ++k)
{
    for(i=1;i<=n;++i)
        a[i]=b[i];
    s[k]->sapxep (a+1,n,sizeof(int),ss_giam);
    //In
    for(i=1;i<=n;++i) cout <<a[i]<<" ";
    cout << "\n";
}
getch();
}
```

Phụ lục 1

Thứ tự ưu tiên của các phép toán

Các phép toán được chia thành 16 nhóm. Các phép toán trong cùng nhóm có mức độ ưu tiên như nhau.

Về trình tự kết hợp thì:

- + Các phép tính của nhóm 2, nhóm 14 và toán tử gán (nhóm 15) kết hợp từ phải sang trái.
- + Các phép toán còn lại kết hợp từ trái qua phải.

1. Nhóm một

- () Gọi hàm (Function call)
- [] Chỉ số mảng (Array subscript)
- > Chọn gián tiếp một thành phần (indirect component selector)
- :: Xác định phạm vi truy nhập (scope access/resolution)
- . Chọn trực tiếp một thành phần (direct component selector)

2. Nhóm hai

- () Gọi hàm (Function call)
- ! Phủ định logic (Logical negation -NOT)
- ~ Lấy phần bù theo bit (Bitwise (1's) complement)
- + Dấu cộng (Unary plus)
- Dấu trừ (Unary minus)
- ++ Phép tăng một (Preincrement or postincrement)
- Phép giảm một (Predecrement or postdecrement)
- & Phép lấy địa chỉ (Address)
- * Truy nhập gián tiếp (Indirection)
- sizeof Cho kích thước của toán hạng (returns size of operand, in bytes)
- new Cấp phát bộ nhớ động (dynamically allocates C++ storage)
- delete Giải phóng bộ nhớ (dynamically deallocates C++ storage)

3. Nhóm ba

- * Nhân (Multiply)
- / Chia (Divide)
- % Lấy phần dư (Remainder - modulus)

4. Nhóm bốn

- .* Gọi gián tiếp tới thành phần từ một biến đối tượng
- >* Gọi gián tiếp tới thành phần từ một con trỏ đối tượng

5. Nhóm năm

- + Cộng (Binary plus)
- Trừ (Binary minus)

6. Nhóm sáu

- << Dịch trái (Shift left)

>> Dịch phải (Shift right)

7. Nhóm bảy

< Nhỏ hơn (Less than)

<= Nhỏ hơn hoặc bằng (Less than or equal to)

> Lớn hơn (Greater than)

>= Lớn hơn hoặc bằng (Greater than or equal to)

8. Nhóm tám

== Bằng (Equal to)

!= Không bằng (Not equal to)

9. Nhóm chín

& Phép và theo bit (Bitwise AND)

10. Nhóm mười

^ Phép hoặc loại trừ theo bit (Bitwise XOR)

11. Nhóm mười một

| Phép hoặc theo bit (Bitwise OR)

12. Nhóm mười hai

&& Phép và logic (Logical AND)

13. Nhóm mười ba

&& Phép hoặc logic (Logical OR)

14. Nhóm mười bốn

?: Toán tử điều kiện (a ? x : y means "if a then x, else y")

15. Nhóm mười năm

= Phép gán đơn giản (Simple assignment)

*= Phép gán sau khi nhân (Assign product)

/= Phép gán sau khi chia (Assign quotient)

%= Phép gán sau khi lấy phần dư (Assign remainder)

+= Phép gán sau khi cộng (Assign sum)

-= Phép gán sau khi trừ (Assign difference)

&= Phép gán sau khi AND theo bit (Assign bitwise AND)

^= Phép gán sau khi XOR theo bit (Assign bitwise XOR)

|= Phép gán sau khi OR theo bit (Assign bitwise OR)

<<= Phép gán sau khi dịch trái (Assign left shift)

>>= Phép gán sau khi dịch phải (Assign right shift)

16. Nhóm mười sáu

, Toán tử phẩy dùng để phân cách các phần tử

Tất cả các toán tử nói trên đều có thể định nghĩa chồng trừ các toán tử sau:

. Chọn trực tiếp một thành phần

- . * Gọi gián tiếp tới thành phần từ một biến đối tượng
- :: Toán tử xác định phạm vi truy nhập
- ?: Toán tử điều kiện

Phụ lục 2

Các từ khóa của c++

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

Phụ lục 3

Bảng mã asskey và giải quyết

1. Bảng mã ascii

Bộ ký tự ASCII gồm 256 ký tự được phân bố như sau:

+ 32 ký tự đầu tiên là các ký tự điều khiển không in được như ký tự Enter (mã 13), ký tự ESC (mã 27).

+ Các mã ASCII 32-47, 58-64, 91-96 và 123-127 là các ký tự đặc biệt như dấu chấm, dấu phẩy, dấu cách, dấu ngoặc, dấu móc, dấu hỏi,...

+ Các mã ASCII 48-57 là 10 chữ số

+ Các mã ASCII 65-90 là các chữ cái hoa từ A đến Z

+ Các mã ASCII 97-122 là các chữ cái thường từ a đến z

Lưu ý: Chữ thường có mã ASCII lớn hơn 32 so với chữ hoa tương ứng. Ví dụ mã ASCII của a là 97 còn mã ASCII của A là 65.

+ Các mã ASCII 128-255 là các ký tự đồ hoạ.

Bảng sau cho mã ASCII của 128 ký tự đầu tiên. Để nhận được các ký tự đồ hoạ (có mã từ 128 đến 255) có thể dùng chương trình sau:

```
// In các ký tự đồ hoạ lên màn hình
```

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

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

```
main()
```

```
{
```

```
    int i;
```

```
    clrscr();
```

```
    for (i=128; i<=255; ++i)
```

```
        printf("%06d%2c",i,i);
```

```
}
```

Bảng mã ASCII

mã (Số TT)	ký tự	mã (Số TT)	ký tự	mã (Số TT)	ký tự
0	NUL	26	SUB	52	4
1	SOH	27	ESC	53	5
2	STX	28	FS	54	6
3	ETX	29	GS	55	7
4	EOT	30	RS	56	8
5	ENQ	31	US	57	9
6	ACK	32	Space	58	:
7	BEL	33	!	59	;
8	BS	34	“	60	<
9	HT	35	#	61	=

10	LF	36	\$	62	>
11	VT	37	%	63	?
12	FF	38	&	64	@
13	CR	39	'	65	A
14	SO	40	(	66	B
15	SI	41	)	67	C
16	DLE	42	*	68	D
17	DC1	43	+	69	E
18	DC2	44	,	70	F
19	DC3	45	-	71	G
20	DC4	46	.	72	H
21	NAK	47	/	73	I
22	SYN	48	0	74	J
23	ETB	49	1	75	K
24	CAN	50	2	76	L
25	EM	51	3	77	M
78	N	95	_	112	p
79	O	96	*	113	q
80	P	97	a	114	r
81	Q	98	b	115	s
82	R	99	c	116	t
83	S	100	d	117	u
84	t	101	e	118	v
85	U	102	f	119	w
86	V	103	g	120	x
87	W	104	h	121	y
88	X	105	i	122	z
89	Y	106	J	123	{
90	Z	107	k	124	
91	[	108	l	125	}
92	\	109	m	126	~
93	]	110	n	127	DEL
94	^	111	o		

2. Bảng mã scan từ bàn phím

Mỗi phím trên bàn phím của IBM PC đều được gán một con số, gọi là mã scan, từ 1 đến 83. IBM PC AT dùng một nhóm mã khác, từ 1 đến 108 các mã này bắt đầu bằng các phím số, các phím chữ, rồi đến các phím chức năng và cuối cùng là các phím cho con trỏ, khi một phím được nhấn thì bộ xử lý của bàn phím gửi cho CPU mã scan tương ứng, khi nó được nhả thì mã scan cộng thêm 80 hex sẽ được gửi tiếp cho CPU.

Hex	Thập phân	Phím của PC	Phím của PC-AT
1	1	ESC	Tilde
2-B	2-11	1-9,0	1-9,0
C	12	trừ, gạch dưới	trừ, gạch dưới
D	13	=, +	=, +
E	14	Backspace	\, thanh đứng
F	15	Tab	Backspace
10	16	Q	Tab
11	17	W	Q
12	18	E	W
13	19	R	E
14	20	T	R
15	21	Y	T
16	22	U	Y
17	23	I	U
18	24	O	I
19	25	P	O
1A	26	[	P
1B	27	]	[
1C	28	Enter	]
1D	29	Ctrl	
1E	30	A	Ctrl
1F	31	S	A
20	32	D	S
21	33	F	D
22	34	G	F
23	35	H	G
24	36	J	H
25	37	K	J
26	38	L	K
27	39	Chấm phẩy, :	L
28	40	Nháy	Chấm phẩy, :
29	41	Tidle	Nháy
2A	42	Shift	trái
2B	43	\, thanh đứng	Enter
2C	44	Z	Shift trái
2D	45	X	
2E	46	C	Z
2F	47	V	X
30	48	B	C

31	49	N	V
32	50	M	B
33	51	Phảy	N
34	52	Chấm	M
35	53	/,?	Phảy
36	54	Shift phải	Chấm
37	55	*, PrtScr	/, ?
38	56	Alt	
39	57	Space bar	Shift phải
3A	58	Caps Lock	Alt
3B	59	F1	
3C	60	F2	
3D	61	F3	Space bar
3E	62	F4	
3F	63	F5	
40	64	F6	Caps Lock
41	65	F7	F2
42	66	F8	F4
43	67	F9	F6
44	68	F10	F8
45	69	Num Lock	F10
46	70	Scroll Lock,Break	F1
47	71	Home	F3
48	72	mũi tên lên	F5
49	73	PgUp	F7
4A	74	Dấu trừ bàn tính	F9
4B	75	Mũi tên trái	
4C	76	5	của bàn tính
4D	77	Mũi tên phải	
4F	79	End	
50	80	Mũi tên xuống	
51	81	PgDn	
52	82	Ins	
53	83	Del	
5A	90		ESC
5B	91		Home
5C	92		Mũi tên trái
5D	93		End
5F	95		Num Lock
60	96		Mũi tên lên

61	97	5 của bàn tính
62	98	Mũi tên xuống
63	99	Ins
64	100	Scroll Lock
65	101	PgUp
66	102	Mũi tên phải
67	103	PgDn
68	104	Del
69	105	Sys
6A	106	*, PrtScr
6B	107	-
6C	108	+

Phụ lục 4

Hàm với đối số bất định trong C

Trong các giáo trình C thường chỉ hướng dẫn cách xây dựng hàm với các đối cố định. Mỗi đối cần có một tham số (cùng kiểu với nó) trong lời gọi hàm. Tuy nhiên một vài hàm chuẩn của C lại không như vậy, mà linh hoạt hơn, chẳng khi dùng hàm printf hay scanf thì số tham số mà ta cung cấp cho hàm là không cố định cả về số lượng lẫn kiểu cách. Ví dụ trong câu lệnh:

```
printf("\n Tổng = %d ", 3+4+5) ;
```

có 2 tham số, nhưng trong câu lệnh:

```
printf("\n Hà Nội" ) ;
```

chỉ có một tham số.

Như vậy cần phân biệt các khái niệm sau:

- Đối số cố định được khai báo trong dòng đầu của hàm, nó có tên và kiểu
- Tham số ứng với đối số cố định gọi là tham số cố định
- Đối số bất định được khai báo bởi ba dấu chấm: bất định cả về số lượng và kiểu
- Tham số bất định (ứng với đối số bất định) là một danh sách giá trị với số lượng và kiểu tùy ý (không xác định)

Trong phụ lục này sẽ trình bày cách xây dựng các hàm với đối số bất định. Công cụ chủ yếu được dùng là con trỏ và danh sách.

1. Biến con trỏ

Biến con trỏ (hay con trỏ) dùng để chứa địa chỉ của biến, mảng, hàm, ... Có nhiều kiểu địa chỉ, vì vậy cũng có nhiều kiểu con trỏ. Biến con trỏ được khai báo theo mẫu:

```
Kiểu *Tên_biến_con_trỏ ;
```

Ví dụ:

```
float px ; // px là con trỏ thực
```

Các phép toán quan trọng trên con trỏ gồm:

+ Gán địa chỉ một vùng nhớ cho con trỏ (dùng toán tử gán, phép lấy địa chỉ, các hàm cấp phát bộ nhớ)

+ Truy nhập vào vùng nhớ thông qua con trỏ, dùng phép toán:

```
*Tên_con_trỏ
```

(Để ý ở đây có 2 vùng nhớ: vùng nhớ của biến con trỏ và vùng nhớ mà địa chỉ đầu của nó chứa trong biến con trỏ)

+ Cộng địa chỉ để con trỏ chứa địa chỉ của phần tử tiếp theo, dùng phép toán:

```
++ Tên_con_trỏ    hoặc    Tên_con_trỏ ++
```

Chú ý rằng các phép toán trên chỉ có thể thực hiện đối với con trỏ có kiểu.

2. Danh sách không cùng kiểu

Dùng con trỏ có kiểu chỉ quản lý được một danh sách giá trị cùng kiểu, ví dụ dãy số thực, dãy số nguyên, dãy các cấu trúc,....

Khi cần quản lý một danh sách các giá trị không cùng kiểu ta phải dùng con trỏ không kiểu (void) khai báo như sau:

```
void * Tên_con_trỏ ;
```

Con trỏ void có thể chứa các địa chỉ có kiểu bất kỳ, và dùng để trỏ đến vùng nhớ chứa danh sách cần quản lý. Một chú ý quan trọng là mỗi khi gửi vào hay lấy ra một giá trị từ vùng nhớ, thì tùy theo kiểu giá trị mà ta phải dùng phép chuyển kiểu thích hợp đối với con trỏ. Ví dụ sau minh họa cách lập một danh sách gồm một số nguyên, một số thực và một chuỗi ký tự. Chúng ta cần một bộ nhớ để chứa số nguyên, số thực và địa chỉ chuỗi và dùng các con trỏ void để quản lý vùng nhớ này.

```
void *list, *p ; // Con trỏ list trỏ tới đầu danh sách
// p dùng để duyệt qua các phần tử của danh sách
list=malloc(sizeof(int) + sizeof(float)+ sizeof(char*) );
p=list;
*((int*)p) = 12; // Đưa số nguyên 12 vào danh sách
((int*)p)++; // Chuyển sang phần tử tiếp theo
*((float*)p) = 3.14; // Đưa số thực 3.14 vào danh sách
((float*)p)++; // Chuyển sang phần tử tiếp theo
*((char**)p) = "HA NOI"; // Đưa địa chỉ chuỗi "HA NOI"
// vào danh sách
// Nhận các phần tử trong danh sách
p=list; // Về đầu danh sách
int a = *((int*)p); // Nhận phần tử thứ nhất
((int*)p)++; // Chuyển sang phần tử tiếp theo
float x=*((float*)p); // Nhận phần tử thứ hai
((float*)p)++; // Chuyển sang phần tử tiếp theo
char *str = *((char**)p) ; // Nhận phần tử thứ ba
```

3. Hàm với đối số bất định

+ Đối bất định bao giờ cũng đặt sau cùng và được khai báo bằng dấu ba chấm. Ví dụ ví dụ hàm

```
void f(int n, char *s, ...);
```

có 2 đối cố định là n, s và đối bất định.

+ Để nhận được các tham số bất định trong lời gọi hàm ta cần lưu ý các điểm sau:

- Các tham số bất định chứa trong một danh sách. Để nhận được địa chỉ đầu danh sách ta dùng một con trỏ void và phép gán sau:

```
void *list ;
list = ... ;
```

- Dùng một tham số cố định kiểu chuỗi để quy định số lượng và kiểu của mỗi tham số trong danh sách, ví dụ:

“3i” hiểu là : tham số bất định gồm 3 giá trị int

“3f” hiểu là : tham số bất định gồm 3 giá trị float

“fiss” hiểu là có 4 tham số bất định có kiểu lần lượt là float, int, char*, char*

Một khi đã biết được địa chỉ đầu danh sách, biết được số lượng và kiểu của mỗi tham số, thì dễ dàng nhận được giá trị các tham số để sử dụng trong thân hàm.

Ví dụ sau đây minh họa cách xây dựng các hàm với tham số 563 564 ic giá trị kiểu int, float và char. Hàm có một tham số cố định để cho biết có bao nhiêu giá trị và kiểu các giá trị cần in. Kiểu quy định như sau: i là int, f là float, s là char*. Tham số có 2 cách viết: lặp (gồm một hằng số nguyên và một chữ cái định kiểu) và liệt kê (một dãy các chữ cái định kiểu). Ví dụ:

“4s” có nghĩa in 4 chuỗi

“siif” có nghĩa in một chuỗi, 2 giá trị nguyên và một giá trị thực:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <stdarg.h>
void InDanhSachGiaTri(char *st,...)
{
    void *list ;
    int gt_int ;
    float gt_float;
    char *gt_str;
    int n,i ;
    char kieu;
    int lap;
    list = ... ; // list tro toi vung nho chua danh sach dia chi cac
                // tham so
    lap = isdigit(st[0]) ;
    if (lap)
        n=st[0] - '0' ;
    else
        n=strlen(st);
    printf("\n n= %d lap = %d",n,lap); getch();
    for(i=0;i<n;++i)
    {
        if(lap)
            kieu=st[1];
        else
            kieu = st[i];
        printf("\nKieu= %c",kieu); getch();
        switch(kieu)
        {
            case 'i' :
                gt_int = *((int*)list);
                if(!lap)
                    ((int*)list)++ ;
                printf("\nGia tri %d = %d",i,gt_int);
                break;
```

```
case 'f' :
    gt_float = (float) (*((double*)list));
    if(!lap)
        ((double*)list)++ ;
    printf("\nGia tri %d = %0.2f",i,gt_float);
    break;
case 's' :
    gt_str = *((char**)list) ;
    if(!lap)
        ((char**)list)++ ;
    printf("\nGia tri %d = %s",i,gt_str);
}
}
}
void main()
{
    float x=3.14;
    int a=123;
    char *tp="HAI PHONG";
    InDanhSachGiaTri("4i",a);
    InDanhSachGiaTri("4s","HA NOI");
    InDanhSachGiaTri("ifsssfii", a, x, tp, tp,"QUY NHON",
                    x, 6.28, a, 246);
    InDanhSachGiaTri("4f",6.28);
    getch();
}
```

4. Hàm không đối và hàm với đối bất định

Nhiều người nghĩ hàm khai báo như sau

```
void f();
```

là hàm không đối trong C. Trong C++ thì hiểu như thế là đúng, còn trong C thì đó là hàm có đối bất định (hàm không đối trong C khai báo như sau: f(void)). Do không có đối cố định nào cho biết về số lượng và kiểu của các tham số bất định, nên giải pháp ở đây là dùng các biến toàn bộ. Rõ ràng giải pháp này không thuận tiện cho người dùng vì phải khai báo đúng tên biến toàn bộ và phải khởi gán giá trị cho nó trước khi gọi hàm. Ví dụ trình bày một hàm chỉ có đối bất định dùng để tính max và min của các giá trị thực. Các tham số bất định được đưa vào theo trình tự sau: Địa chỉ chứa max, địa chỉ chứa min, các giá trị nguyên cần tính max, min. Chương trình dùng biến toàn bộ N để cho biết số giá trị nguyên cần tính max, min.

```
int N;
void maxmin()
{
    void *lt = ... ;
```

```
float *max, *min , tg;
int i;
max = *((float**)lt)++;
min = *((float**)lt)++;
*max = *min = (float) *((double*)lt)++;
for(i=1;i<N;++i)
{
    tg= (float) *((double*)lt)++;
    if(tg > *max) *max = tg;
    if(tg < *min) *min = tg;
}
}
```

phụ lục 5

Tóm tắt các hàm của Turbo C theo thứ tự ABC

1. _chmod	<io.h>
2. _close	<io.h>
3. _creat	<io.h>
4. _open	<io.h>
5. abort	<process.h>
6. abs	<stdlib.h>
7. acos	<math.h>
8. arc	<graphics.h>
9. asin	<math.h>
10. atan	<math.h>
11. atan2	<math.h>
12. atof	<ctype.h>
13. atoi	<ctype.h>
14. atol	<ctype.h>
15. bar	<graphics.h>
16. bar3d	<graphics.h>
17. cabs	<math.h>
18. calloc	<alloc.h>
19. ceil	<math.h>
20. chdir	<dir.h>
21. chmod	<io.h>
22. circle	<graphics.h>
23. cleardevive	<graphics.h>
24. clearviewport	<graphics.h>
25. close	<io.h>
26. clreol	<conio.h>
27. clrscr	<conio.h>
28. coreleft	<alloc.h>
29. cos	<math.h>
30. cosh	<math.h>
31. cprintf	<conio.h>
32. creat	<io.h>
33. cscanf	<conio.h>
34. delay	<dos.h>
35. delline	<conio.h>
36. disable	<dos.h>
37. drawpoly	<graphics.h>

38. ecvt	<ctype.h>
39. ellipse	<graphics.h>
40. enable	<dos.h>
41. exit	<process.h>
42. exp	<math.h>
43. fabs	<math.h>
44. fclose	<stdio.h>
45. fcloseall	<stdio.h>
46. fevt	<ctype.h>
47. feof	<stdio.h>
48. ferror	<stdio.h>
49. fflush	<stdio.h>
50. fflushall	<stdio.h>
51. fgetc	<stdio.h>
52. fgets	<stdio.h>
53. fillpopy	<graphics.h>
54. findfirst	<dir.h>
55. findnext	<dir.h>
56. floodfill	<graphics.h>
57. floor	<math.h>
58. fmode	<math.h>
59. fopen	<stdio.h>
60. FP_OFF	<dos.h>
61. FP_SEG	<dos.h>
62. fprintf	<stdio.h>
63. fprintf	<stdio.h>
64. fputc	<stdio.h>
65. fputs	<stdio.h>
66. fread	<stdio.h>
67. free	<alloc.h>
68. fscanf	<stdio.h>
69. fseek	<stdio.h>
70. fteel	<stdio.h>
71. fwrite	<stdio.h>
72. gcvt	<ctype.h>
73. geninterrupt	<dos.h>
74. getbkcolor	<graphics.h>
75. getc	<stdio.h>
76. getch	<conio.h>
77. getchar	<stdio.h>

78. getche	<conio.h>
79. getcolor	<graphics.h>
80. getcwd	<dir.h>
81. getdate	<time.h>
82. getimage	<graphics.h>
83. getlinesettings	<graphics.h>
84. getmaxcolor	<graphics.h>
85. getmaxx	<graphics.h>
86. getmaxy	<graphics.h>
87. getpalette	<graphics.h>
88. getpixel	<graphics.h>
89. gets	<stdio.h>
90. gettextinfo	<conio.h>
91. gettime	<dos.h>
92. gettime	<time.h>
93. getvect	<dos.h>
94. getviewport	<graphics.h>
95. getw	<stdio.h>
96. gotoxy	<conio.h>
97. gotoxy	<conio.h>
98. grapherrormsg	<graphics.h>
99. graphresult	<graphics.h>
100. imagesize	<graphics.h>
101. initgraph	<graphics.h>
102. int86	<dos.h>
103. int86x	<dos.h>
104. intdos	<dos.h>
105. intdosx	<dos.h>
106. intr	<dos.h>
107. inxdigit	<ctype.h>
108. isalnum	<ctype.h>
109. isalpha	<ctype.h>
110. iscntrl	<ctype.h>
111. isdigit	<ctype.h>
112. isgraph	<ctype.h>
113. islower	<ctype.h>
114. isprint	<ctype.h>
115. ispunct	<ctype.h>
116. isspace	<ctype.h>
117. isupper	<ctype.h>

118. itoa	<ctype.h>
119. kbhit	<conio.h>
120. keep	<dos.h>
121. labs	<stdlib.h>
122. line	<graphics.h>
123. linerel	<graphics.h>
124. lineto	<graphics.h>
125. log	<math.h>
126. log10	<math.h>
127. lseek	<io.h>
128. ltoa	<ctype.h>
129. malloc	<alloc.h>
130. memccpy	<memory.h> hoặc <string.h>
131. memchr	<memory.h> hoặc <string.h>
132. memcmp	<memory.h> hoặc <string.h>
133. memcpy	<memory.h> hoặc <string.h>
134. memicmp	<memory.h> hoặc <string.h>
135. memset	<memory.h> hoặc <string.h>
136. MK_FP	<dos.h>
137. mkdir	<dir.h>
138. movedata	<mem.h>
139. movedata	<memory.h> hoặc <string.h>
140. moveto	<graphics.h>
141. nosound	<dos.h>
142. open	<io.h>
143. outtext	<graphics.h>
144. outtextxy	<graphics.h>
145. peek	<dos.h>
146. peekb	<dos.h>
147. perror	<stdio.h>
148. pieslice	<graphics.h>
149. poke	<dos.h>
150. pokeb	<dos.h>
151. pow	<math.h>
152. printf	<stdio.h>
153. putc	<stdio.h>
154. putch	<conio.h>
155. putchar	<stdio.h>
156. putimage	<graphics.h>
157. putpixel	<graphics.h>

158. puts	<stdio.h>
159. putw	<stdio.h>
160. rand	<stdlib.h>
161. random	<stdlib.h>
162. randomize	<stdlib.h> và <time.h>
163. read	<io.h>
164. realloc	<alloc.h>
165. rectangle	<graphics.h>
166. remove	<stdio.h>
167. rewind	<stdio.h>
168. rmdir	<dir.h>
169. scanf	<stdio.h>
170. segread	<dos.h>
171. setbkcolor	<graphics.h>
172. setcolor	<graphics.h>
173. setdate	<time.h>
174. setfillstyle	<graphics.h>
175. setlinestyle	<graphics.h>
176. setpalette	<graphics.h>
177. setttextjustify	<graphics.h>
178. setttextstyle	<graphics.h>
179. settime	<time.h>
180. setvect	<dos.h>
181. setviewport	<graphics.h>
182. setwritemode	<graphics.h>
183. sin	<math.h>
184. sinh	<math.h>
185. sleep	<dos.h>
186. sound	<dos.h>
187. sprintf	<stdio.h>
188. sqrt	<math.h>
189. srand	<stdlib.h>
190. strcat	<string.h>
191. strchr	<string.h>
192. strcmp	<string.h>
193. strcmpi	<string.h>
194. strcpy	<string.h>
195. strcspn	<string.h>
196. strdup	<string.h>
197. stricmp	<string.h>

198. strlen	<string.h>
199. strlwr	<string.h>
200. strncat	<string.h>
201. strncmp	<string.h>
202. strncpy	<string.h>
203. strnicmp	<string.h>
204. strnset	<string.h>
205. strpbrk	<string.h>
206. strchr	<string.h>
207. strrev	<string.h>
208. strset	<string.h>
209. strspn	<string.h>
210. strstr	<string.h>
211. strupr	<string.h>
212. system	<process.h>
213. tan	<math.h>
214. tanh	<math.h>
215. textbackground	<conio.h>
216. textcolor	<conio.h>
217. textheight	<graphics.h>
218. textmode	<conio.h>
219. textwidth	<graphics.h>
220. time	<time.h>
221. tolower	<ctype.h>
222. toupper	<ctype.h>
223. ultoa	<ctype.h>
224. unlink	<stdio.h>
225. wherex	<conio.h>
226. wherey	<conio.h>
227. window	<conio.h>
228. write	<io.h>

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG



NGÔN NGỮ LẬP TRÌNH C++

(Dùng cho sinh viên hệ đào tạo đại học từ xa)

Lưu hành nội bộ

HÀ NỘI - 2006

NGÔN NGỮ LẬP TRÌNH C++

PGS.TS. Trần Đình Quế
KS. Nguyễn Mạnh Hùng



Lập trình nâng cao với C++
Lập trình hướng đối tượng với C++



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04).5541221; Fax: (04).5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhtx@o-pit.edu.vn

GIỚI THIỆU

C++ là ngôn ngữ lập trình hướng đối tượng được mở rộng từ ngôn ngữ C. Do vậy, C++ có ưu điểm là kế thừa được các điểm mạnh truyền thống của ngôn ngữ C như uyển chuyển, tương thích với các thiết bị phần cứng. Hiện nay, C++ là một ngôn ngữ lập trình phổ biến, được giảng dạy tại các trường đại học trong nước và trên thế giới và đặc biệt được sử dụng rộng rãi cho nhu cầu phát triển của công nghiệp phần mềm hiện nay. Tài liệu này không những nhằm giới thiệu cho sinh viên ngôn ngữ lập trình C++, mà còn mong muốn qua đó sinh viên có thể hiểu được tư tưởng của phương pháp lập trình hướng đối tượng nói chung. Nội dung của tài liệu bao gồm hai phần chính:

- Phần thứ nhất là lập trình nâng cao với C++, bao gồm lập trình C++ với con trỏ và mảng, các kiểu dữ liệu có cấu trúc cùng các thao tác vào ra trên tệp.
- Phần thứ hai là lập trình hướng đối tượng với C++, bao gồm các định nghĩa và các thao tác trên lớp đối tượng, tính kế thừa và tương ứng bội trong C++, cách sử dụng một số lớp cơ bản trong thư viện C++.

Nội dung tài liệu được tổ chức thành 7 chương:

Chương 1: Giới thiệu tổng quan về các phương pháp lập trình

Trình bày các phương pháp lập trình tuyến tính, lập trình cấu trúc và đặc biệt, làm quen với các khái niệm trong lập trình hướng đối tượng.

Chương 2: Con trỏ và mảng

Trình bày cách khai báo và sử dụng các kiểu con trỏ và mảng trong ngôn ngữ C++.

Chương 3: Kiểu dữ liệu có cấu trúc

Trình bày cách biểu diễn và cài đặt một số kiểu cấu trúc dữ liệu trừu tượng trong C++. Sau đó, trình bày cách áp dụng các kiểu dữ liệu này trong các ứng dụng cụ thể.

Chương 4: Vào ra trên tệp

Trình bày các thao tác đọc, ghi dữ liệu trên các tệp tin khác nhau: tệp tin văn bản và tệp tin nhị phân. Trình bày các cách truy nhập tệp tin trực tiếp.

Chương 5: Lớp đối tượng

Trình bày các khái niệm mở đầu cho lập trình hướng đối tượng trong C++, bao gồm cách khai báo và sử dụng lớp, các thuộc tính của lớp; cách khởi tạo và hủy bỏ đối tượng, các quy tắc truy nhập đến các thành phần của lớp.

Chương 6: Tính kế thừa và tương ứng bội

Trình bày cách thức kế thừa giữa các lớp trong C++, các nguyên tắc truy nhập trong kế thừa, định nghĩa nạp chồng các phương thức và tính đa hình trong lập trình hướng đối tượng với C++.

Chương 7: Một số lớp quan trọng

Trình bày cách sử dụng một số lớp có sẵn trong thư viện chuẩn của C++, bao gồm các lớp: lớp tập hợp, lớp chuỗi, lớp ngăn xếp, lớp hàng đợi và lớp danh sách liên kết.

Để đọc được cuốn sách này, sinh viên phải quen biết các khái niệm cơ bản về lập trình, có một số kỹ năng lập trình với ngôn ngữ C hoặc C++. Cuốn sách này cũng có thể dùng tài liệu tham khảo cho những sinh viên muốn tìm hiểu các kỹ thuật lập trình nâng cao và lập trình hướng đối tượng

với C++. Cuốn sách này có kèm theo một đĩa chương trình chứa toàn bộ các chương trình được lấy làm minh họa và các bài tập trong cuốn sách.

Mặc dù các tác giả đã có nhiều cố gắng trong việc biên soạn tài liệu này, song không thể tránh khỏi những thiếu sót. Rất mong nhận được những ý kiến đóng góp quý báu từ các sinh viên và các bạn đồng nghiệp.



CHƯƠNG 1

GIỚI THIỆU VỀ CÁC PHƯƠNG PHÁP LẬP TRÌNH

Nội dung của chương này tập trung trình bày các phương pháp lập trình:

- Phương pháp lập trình tuyến tính
- Phương pháp lập trình hướng cấu trúc
- Phương pháp lập trình hướng đối tượng.

1.1 LẬP TRÌNH TUYẾN TÍNH

Đặc trưng cơ bản của lập trình tuyến tính là tư duy theo lối tuần tự. Chương trình sẽ được thực hiện theo thứ tự từ đầu đến cuối, lệnh này kế tiếp lệnh kia cho đến khi kết thúc chương trình.

Đặc trưng

Lập trình tuyến tính có hai đặc trưng:

- **Đơn giản:** chương trình được tiến hành đơn giản theo lối tuần tự, không phức tạp.
- **Đơn luồng:** chỉ có một luồng công việc duy nhất, và các công việc được thực hiện tuần tự trong luồng đó.

Tính chất

- **Ưu điểm:** Do tính đơn giản, lập trình tuyến tính được ứng dụng cho các chương trình đơn giản và có ưu điểm dễ hiểu.
- **Nhược điểm:** Với các ứng dụng phức tạp, người ta không thể dùng lập trình tuyến tính để giải quyết.

Ngày nay, lập trình tuyến tính chỉ tồn tại trong phạm vi các modul nhỏ nhất của các phương pháp lập trình khác. Ví dụ trong một chương trình con của lập trình cấu trúc, các lệnh cũng được thực hiện theo tuần tự từ đầu đến cuối chương trình con.

1.2 LẬP TRÌNH HƯỚNG CẤU TRÚC

1.2.1 Đặc trưng của lập trình hướng cấu trúc

Trong lập trình hướng cấu trúc, chương trình chính được chia nhỏ thành các chương trình con và mỗi chương trình con thực hiện một công việc xác định. Chương trình chính sẽ gọi đến chương trình con theo một giải thuật, hoặc một cấu trúc được xác định trong chương trình chính.

Các ngôn ngữ lập trình cấu trúc phổ biến là Pascal, C và C++. Riêng C++ ngoài việc có đặc trưng của lập trình cấu trúc do kế thừa từ C, còn có đặc trưng của lập trình hướng đối tượng. Cho nên C++ còn được gọi là ngôn ngữ lập trình nửa cấu trúc, nửa hướng đối tượng.

Đặc trưng

Đặc trưng cơ bản nhất của lập trình cấu trúc thể hiện ở mối quan hệ:

Chương trình = Cấu trúc dữ liệu + Giải thuật

Trong đó:

- **Cấu trúc dữ liệu** là cách tổ chức dữ liệu cho việc xử lý bởi một hay nhiều chương trình nào đó.
- **Giải thuật** là một quy trình để thực hiện một công việc xác định

Trong chương trình, giải thuật có quan hệ phụ thuộc vào cấu trúc dữ liệu:

- Một cấu trúc dữ liệu chỉ phù hợp với một số hạn chế các giải thuật.
- Nếu thay đổi cấu trúc dữ liệu thì phải thay đổi giải thuật cho phù hợp.
- Một giải thuật thường phải đi kèm với một cấu trúc dữ liệu nhất định.

Tính chất

- Mỗi chương trình con có thể được gọi thực hiện nhiều lần trong một chương trình chính.
- Các chương trình con có thể được gọi đến để thực hiện theo một thứ tự bất kì, tùy thuộc vào giải thuật trong chương trình chính mà không phụ thuộc vào thứ tự khai báo của các chương trình con.
- Các ngôn ngữ lập trình cấu trúc cung cấp một số cấu trúc lệnh điều khiển chương trình.

Ưu điểm

- Chương trình sáng sủa, dễ hiểu, dễ theo dõi.
- Tư duy giải thuật rõ ràng.

Nhược điểm

- Lập trình cấu trúc không hỗ trợ mạnh việc sử dụng lại mã nguồn: Giải thuật luôn phụ thuộc chặt chẽ vào cấu trúc dữ liệu, do đó, khi thay đổi cấu trúc dữ liệu, phải thay đổi giải thuật, nghĩa là phải viết lại chương trình.
- Không phù hợp với các phần mềm lớn: tư duy cấu trúc với các giải thuật chỉ phù hợp với các bài toán nhỏ, nằm trong phạm vi một modul của chương trình. Với dự án phần mềm lớn, lập trình cấu trúc tỏ ra không hiệu quả trong việc giải quyết mối quan hệ vĩ mô giữa các modul của phần mềm.

Vấn đề

Vấn đề cơ bản của lập trình cấu trúc là bằng cách nào để phân chia chương trình chính thành các chương trình con cho phù hợp với yêu cầu, chức năng và mục đích của mỗi bài toán. Thông thường, để phân rã bài toán trong lập trình cấu trúc, người ta sử dụng phương pháp thiết kế trên xuống (top-down).

1.2.2 Phương pháp thiết kế trên xuống (top-down)

Phương pháp thiết kế top-down tiếp cận bài toán theo hướng từ trên xuống dưới, từ tổng quát đến chi tiết. Theo đó, một bài toán được chia thành các bài toán con nhỏ hơn. Mỗi bài toán con lại được chia nhỏ tiếp, nếu có thể, thành các bài toán con nhỏ hơn nữa. Quá trình này còn được gọi là quá trình làm mịn dần. Quá trình này sẽ dừng lại khi các bài toán con không cần chia nhỏ thêm

nữa. Nghĩa là khi mỗi bài toán con đều có thể giải quyết bằng một chương trình con với một giải thuật đơn giản.

Ví dụ, sử dụng phương pháp top-down để giải quyết bài toán xây một căn nhà mới. Chúng ta có thể phân rã bài toán theo các bước như sau:

- Ở mức thứ nhất, chia bài toán xây nhà thành các bài toán nhỏ hơn như *làm móng*, đổ cột, đổ trần, xây tường, lợp mái.
- Ở mức thứ hai, phân rã các công việc ở mức thứ nhất như việc *làm móng* nhà có thể phân rã tiếp thành các công việc *đào móng*, gia cố nền, làm khung sắt, đổ bê tông; công việc đổ cột được phân rã thành ...
- Ở mức thứ ba, phân rã các công việc của mức thứ hai như việc *đào móng* có thể phân chia tiếp thành các công việc như đo đạc, cắm mốc, căng dây, đào và kiểm tra móng. Việc gia cố nền được phân rã thành ...

Quá trình phân rã có thể dừng ở mức này, bởi vì các công việc con thu được như đo đạc, cắm mốc, căng dây, đào... có thể thực hiện được ngay, không cần chia nhỏ thêm nữa.

Lưu ý:

- Cùng sử dụng phương pháp top-down với cùng một bài toán, nhưng có thể cho ra nhiều kết quả khác nhau. Nguyên nhân là do sự khác nhau trong tiêu chí để phân rã một bài toán thành các bài toán con.

Ví dụ, vẫn áp dụng phương pháp top-down để giải quyết bài toán xây nhà, nhưng nếu sử dụng một cách khác để phân chia bài toán, ta có thể thu được kết quả khác biệt so với phương pháp ban đầu:

- Ở mức thứ nhất, chia bài toán xây nhà thành các bài toán nhỏ hơn như *làm phần gỗ*, làm phần sắt, làm phần bê tông và làm phần gạch.
- Ở mức thứ hai, phân rã các công việc ở mức thứ nhất là *làm phần gỗ* có thể chia thành các công việc như xẻ gỗ, gia công gỗ, tạo khung, lắp vào nhà. Việc làm sắt có thể chia nhỏ thành...

Rõ ràng, với cách làm mịn thế này, ta sẽ thu được một kết quả khác hẳn với cách thức đã thực hiện ở phần trên.

1.3 LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

1.3.1 Lập trình hướng đối tượng

Trong lập trình hướng đối tượng:

- Người ta coi các thực thể trong chương trình là các đối tượng và sau đó trừu tượng hoá đối tượng thành lớp đối tượng.
- Dữ liệu được tổ chức thành các thuộc tính của lớp. Người ta ngăn chặn việc thay đổi tùy tiện dữ liệu trong chương trình bằng các cách giới hạn truy nhập như chỉ cho phép truy nhập dữ liệu thông qua đối tượng, thông qua các phương thức mà đối tượng được cung cấp...
- Quan hệ giữa các đối tượng là quan hệ ngang hàng hoặc quan hệ kế thừa: Nếu lớp B kế thừa từ lớp A thì A được gọi là *lớp cơ sở* và B được gọi là *lớp dẫn xuất*.

Ngôn ngữ lập trình hướng đối tượng phổ biến hiện nay là Java, C++, C#... Mặc dù C++ cũng có những đặc trưng cơ bản của lập trình hướng đối tượng nhưng vẫn không phải là ngôn ngữ lập trình thuần hướng đối tượng.

Đặc trưng

Lập trình hướng đối tượng có hai đặc trưng cơ bản:

- **Đóng gói dữ liệu:** dữ liệu luôn được tổ chức thành các thuộc tính của lớp đối tượng. Việc truy nhập đến dữ liệu phải thông qua các phương thức của đối tượng lớp.
- **Sử dụng lại mã nguồn:** việc sử dụng lại mã nguồn được thể hiện thông qua cơ chế kế thừa. Cơ chế này cho phép các lớp đối tượng có thể kế thừa từ các lớp đối tượng khác. Khi đó, trong các lớp dẫn xuất, có thể sử dụng các phương thức (mã nguồn) của các lớp cơ sở mà không cần phải định nghĩa lại.

Ưu điểm

Lập trình hướng đối tượng có một số ưu điểm nổi bật:

- Không còn nguy cơ dữ liệu bị thay đổi tự do trong chương trình. Vì dữ liệu đã được đóng gói vào các đối tượng. Nếu muốn truy nhập vào dữ liệu phải thông qua các phương thức được cho phép của đối tượng.
- Khi thay đổi cấu trúc dữ liệu của một đối tượng, không cần thay đổi mã nguồn của các đối tượng khác, mà chỉ cần thay đổi một số thành phần của đối tượng dẫn xuất. Điều này hạn chế sự ảnh hưởng xấu của việc thay đổi dữ liệu đến các đối tượng khác trong chương trình.
- Có thể sử dụng lại mã nguồn, tiết kiệm tài nguyên, chi phí thời gian. Vì nguyên tắc kế thừa cho phép các lớp dẫn xuất sử dụng các phương thức từ lớp cơ sở như những phương thức của chính nó, mà không cần thiết phải định nghĩa lại.
- Phù hợp với các dự án phần mềm lớn, phức tạp.

1.3.2 Một số khái niệm cơ bản

Trong mục này, chúng ta sẽ làm quen với một số khái niệm cơ bản trong lập trình hướng đối tượng. Bao gồm:

- Khái niệm đối tượng (object)
- Khái niệm đóng gói dữ liệu (encapsulation)
- Khái niệm kế thừa (inheritance)
- Khái niệm đa hình (polymorphism)

Đối tượng (Object)

Trong lập trình hướng đối tượng, đối tượng được coi là đơn vị cơ bản nhỏ nhất. Các dữ liệu và cách xử lý chỉ là thành phần của đối tượng mà không được coi là thực thể. Một đối tượng chứa các dữ liệu của riêng nó, đồng thời có các phương thức (hành động) thao tác trên các dữ liệu đó:

Đối tượng = dữ liệu + phương thức

Lớp (Class)

Khi có nhiều đối tượng giống nhau về mặt dữ liệu và phương thức, chúng được nhóm lại với nhau và gọi chung là lớp:

- Lớp là sự trừu tượng hoá của đối tượng
- Đối tượng là một thể hiện của lớp.

Đóng gói dữ liệu (Encapsulation)

- Các dữ liệu được đóng gói vào trong đối tượng. Mỗi dữ liệu có một phạm vi truy nhập riêng.
- Không thể truy nhập đến dữ liệu một cách tự do như lập trình cấu trúc
- Muốn truy nhập đến các dữ liệu đã được bảo vệ, phải thông qua các đối tượng, nghĩa là phải sử dụng các phương thức mà đối tượng cung cấp mới có thể truy nhập đến dữ liệu của đối tượng đó.

Tuy nhiên, vì C++ chỉ là ngôn ngữ lập trình nửa đối tượng, cho nên C++ vẫn cho phép định nghĩa các biến dữ liệu và các hàm tự do, đây là kết quả kế thừa từ ngôn ngữ C, một ngôn ngữ lập trình thuần cấu trúc.

Kế thừa (Inheritance)

Tính kế thừa của lập trình hướng đối tượng cho phép một lớp có thể kế thừa từ một số lớp đã tồn tại. Khi đó, lớp mới có thể sử dụng dữ liệu và phương thức của các lớp cơ sở như là của mình. Ngoài ra, lớp dẫn xuất còn có thể bổ sung thêm một số dữ liệu và phương thức. Ưu điểm của kế thừa là khi thay đổi dữ liệu của một lớp, chỉ cần thay đổi các phương thức trong phạm vi lớp cơ sở mà không cần thay đổi trong các lớp dẫn xuất.

Đa hình (Polymorphsim)

Đa hình là khái niệm luôn đi kèm với kế thừa. Do tính kế thừa, một lớp có thể sử dụng lại các phương thức của lớp khác. Tuy nhiên, nếu cần thiết, lớp dẫn xuất cũng có thể định nghĩa lại một số phương thức của lớp cơ sở. Đó là sự nạp chồng phương thức trong kế thừa. Nhờ sự nạp chồng phương thức này, ta chỉ cần gọi tên phương thức bị nạp chồng từ đối tượng mà không cần quan tâm đó là đối tượng của lớp nào. Chương trình sẽ tự động kiểm tra xem đối tượng là thuộc kiểu lớp cơ sở hay thuộc lớp dẫn xuất, sau đó sẽ gọi phương thức tương ứng với lớp đó. Đó là tính đa hình.

1.3.3 Lập trình hướng đối tượng trong C++

Vì C++ là một ngôn ngữ lập trình được mở rộng từ một ngôn ngữ lập trình cấu trúc C nên C++ được xem là ngôn ngữ lập trình nửa hướng đối tượng, nửa hướng cấu trúc.

Những đặc trưng hướng đối tượng của C++

- Cho phép định nghĩa lớp đối tượng.
- Cho phép đóng gói dữ liệu vào các lớp đối tượng. Cho phép định nghĩa phạm vi truy nhập dữ liệu của lớp bằng các từ khoá phạm vi: **public**, **protected**, **private**.

- Cho phép kế thừa lớp với các kiểu kế thừa khác nhau tùy vào từ khoá dẫn xuất.
- Cho phép lớp dẫn xuất sử dụng các phương thức của lớp cơ sở (trong phạm vi quy định).
- Cho phép định nghĩa chồng phương thức trong lớp dẫn xuất.

Những hạn chế hướng đối tượng của C++

Những hạn chế này là do C++ được phát triển từ một ngôn ngữ lập trình thuần cấu trúc C.

- Cho phép định nghĩa và sử dụng các biến dữ liệu tự do.
- Cho phép định nghĩa và sử dụng các hàm tự do.
- Ngay cả khi dữ liệu được đóng gói vào lớp, dữ liệu vẫn có thể truy nhập trực tiếp như dữ liệu tự do bởi các hàm bạn, lớp bạn (friend) trong C++.

TỔNG KẾT CHƯƠNG 1

Chương 1 đã trình bày tổng quan về các phương pháp lập trình hiện nay. Nội dung tập trung vào ba phương pháp lập trình có liên quan trực tiếp đến ngôn ngữ lập trình C++:

- Lập trình tuyến tính
- Lập trình hướng cấu trúc
- Lập trình hướng đối tượng.

C++ là ngôn ngữ lập trình được mở rộng từ ngôn ngữ lập trình cấu trúc C. Do đó, C++ vừa có những đặc trưng của lập trình cấu trúc, vừa có những đặc trưng của lập trình hướng đối tượng.

CHƯƠNG 2

CON TRỎ VÀ MẢNG

Nội dung của chương này tập trung trình bày các vấn đề cơ bản liên quan đến các thao tác trên kiểu dữ liệu con trỏ và mảng trong C++:

- Khái niệm con trỏ, cách khai báo và sử dụng con trỏ.
- Mối quan hệ giữa con trỏ và mảng
- Con trỏ hàm
- Cấp phát bộ nhớ cho con trỏ

2.1 KHÁI NIỆM CON TRỎ

2.1.1 Khai báo con trỏ

Con trỏ là một biến đặc biệt chứa địa chỉ của một biến khác. Con trỏ có cùng kiểu dữ liệu với kiểu dữ liệu của biến mà nó trỏ tới. Cú pháp khai báo một con trỏ như sau:

```
<Kiểu dữ liệu> *<Tên con trỏ>;
```

Trong đó:

- **Kiểu dữ liệu:** Có thể là các kiểu dữ liệu cơ bản của C++, hoặc là kiểu dữ liệu có cấu trúc, hoặc là kiểu đối tượng do người dùng tự định nghĩa.
- **Tên con trỏ:** Tuân theo qui tắc đặt tên biến của C++:
 - Chỉ được bắt đầu bằng một kí tự (chữ), hoặc dấu gạch dưới “_”.
 - Bắt đầu từ kí tự thứ hai, có thể có kiểu kí tự số.
 - Không có dấu trống (space bar) trong tên biến.
 - Có phân biệt chữ hoa và chữ thường.
 - Không giới hạn độ dài tên biến.

Ví dụ, để khai báo một biến con trỏ có kiểu là int và tên là pointerInt, ta viết như sau:

```
int *pointerInt;
```

Lưu ý

- Có thể viết dấu con trỏ “*” ngay sau kiểu dữ liệu, nghĩa là hai cách khai báo sau là tương đương:

```
int *pointerInt;  
int* pointerInt;
```

- Các cách khai báo con trỏ như sau là sai cú pháp:

```
*int pointerInt; // Khai báo sai con trỏ  
int pointerInt*; // Khai báo sai con trỏ
```

2.1.2 Sử dụng con trỏ

Con trỏ được sử dụng theo hai cách:

- Dùng con trỏ để lưu địa chỉ của biến để thao tác
- Lấy giá trị của biến do con trỏ trỏ đến để thao tác

Dùng con trỏ để lưu địa chỉ của biến

Bản thân con trỏ sẽ được trỏ vào địa chỉ của một biến có cùng kiểu dữ liệu với nó. Cú pháp của phép gán như sau:

```
<Tên con trỏ> = &<tên biến>;
```

Lưu ý

- Trong phép toán này, tên con trỏ không có dấu “*”.

Ví dụ:

```
int x, *px;
px = &x;
```

sẽ cho con trỏ px có kiểu int trỏ vào địa chỉ của biến x có kiểu nguyên. Phép toán &<Tên biến> sẽ cho địa chỉ của biến tương ứng.

Lấy giá trị của biến do con trỏ trỏ đến

Phép lấy giá trị của biến do con trỏ trỏ đến được thực hiện bằng cách gọi tên:

```
*<Tên con trỏ>;
```

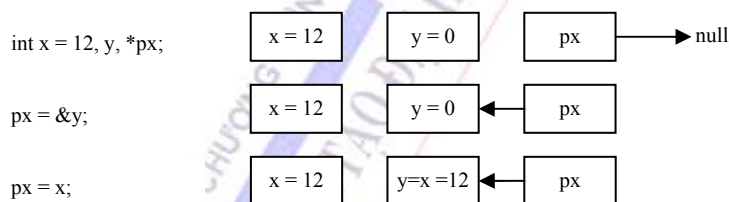
Lưu ý

- Trong phép toán này, phải có dấu con trỏ “*”. Nếu không có dấu con trỏ, sẽ trở thành phép lấy địa chỉ của biến do con trỏ trỏ tới.

Ví dụ:

```
int x = 12, y, *px;
px = &y;
*px = x;
```

Quá trình diễn ra như sau:



con trỏ px vẫn trỏ tới địa chỉ biến y và giá trị của biến y sẽ là 12.

Phép gán giữa các con trỏ

Các con trỏ cùng kiểu có thể gán cho nhau thông qua phép gán và lấy địa chỉ con trỏ:

```
<Tên con trỏ 1> = &<Tên con trỏ 2>;
```

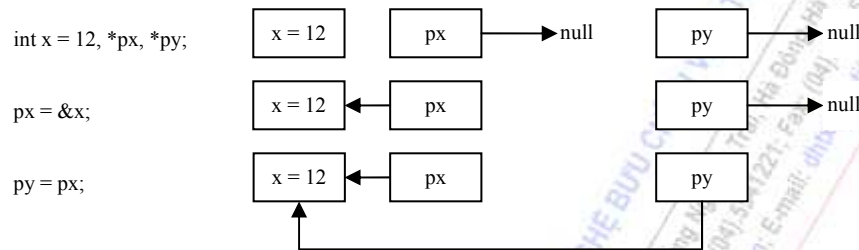
Lưu ý

- Trong phép gán giữa các con trỏ, bắt buộc phải dùng phép lấy **địa chỉ của biến** do con trỏ trỏ tới (không có dấu “*” trong tên con trỏ) mà không được dùng phép lấy **giá trị của biến** do con trỏ trỏ tới.

- Hai con trỏ phải cùng kiểu. Trong trường hợp hai con trỏ khác kiểu, phải sử dụng các phương thức ép kiểu tương tự như trong phép gán các biến thông thường có kiểu khác nhau.

Ví dụ:

```
int x = 12, *px, *py;
px = &x;
py = px;
```



con trỏ py cũng trỏ vào địa chỉ của biến x như con trỏ px. Khi đó *py cũng có giá trị 12 giống như *px và là giá trị của biến x.

Chương trình 2.1 minh họa việc dùng con trỏ giữa các biến của một chương trình C++.

Chương trình 2.1

```
#include <stdio.h>
#include <conio.h>
void main(void) {
    int x = 12, *px, *py;
    cout << "x = " << x << endl;

    px = &x;          // Con trỏ px trỏ tới địa chỉ của x
    cout << "px = &x, *px = " << *px << endl;

    *px = *px + 20;    // Nội dung của px là 32
    cout << "*px = *px+20, x = " << x << endl;

    py = px;          // Cho py trỏ tới chỗ mà px trỏ: địa chỉ của x
    *py += 15;         // Nội dung của py là 47
    cout << "py = px, *py +=15, x = " << x << endl;
}
```

Trong chương trình 2.1, ban đầu biến x có giá trị 12. Sau đó, con trỏ px trỏ vào địa chỉ của biến x nên con trỏ px cũng có giá trị 12. Tiếp theo, ta tăng giá trị của con trỏ px thêm 20, giá trị của con trỏ px là 32. Vì px đang trỏ đến địa chỉ của x nên x cũng có giá trị là 32. Sau đó, ta cho con trỏ py trỏ đến vị trí mà px đang trỏ tới (địa chỉ của biến x) nên py cũng có giá trị 32. Cuối cùng, ta tăng giá trị của con trỏ py thêm 15, py sẽ có giá trị 37. Vì py cũng đang trỏ đến địa chỉ của x nên x cũng có giá trị 37. Do đó, ví dụ 2.1 sẽ in ra kết quả như sau:

```
x = 12
px = &x, *px = 12
*px = *px + 20, x = 32
py = px, *py += 15, x = 37
```

2.2 CON TRỎ VÀ MẢNG

2.2.1 Con trỏ và mảng một chiều

Mảng một chiều

Trong C++, tên một mảng được coi là một kiểu con trỏ hằng, được định vị tại một vùng nhớ xác định và địa chỉ của tên mảng trùng với địa chỉ của phần tử đầu tiên của mảng.

Ví dụ khai báo:

```
int A[5];
```

thì địa chỉ của mảng A (cũng viết là A) sẽ trùng với địa chỉ phần tử đầu tiên của mảng A (là &A[0]) nghĩa là:

```
A = &A[0];
```

Quan hệ giữa con trỏ và mảng

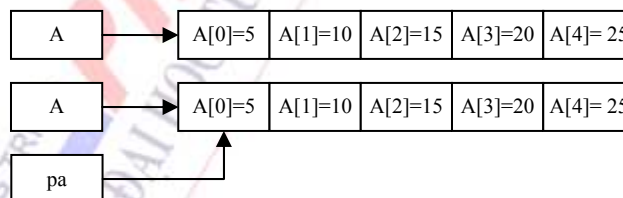
Vì tên của mảng được coi như một con trỏ hằng, nên nó có thể được gán cho một con trỏ có cùng kiểu.

Ví dụ khai báo:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = A;
```

```
int A[5] = {5, 10, 15, 20, 25};
```

```
int *pa = A;
```



thì con trỏ pa sẽ trỏ đến mảng A, tức là trỏ đến địa chỉ của phần tử A[0], cho nên hai khai báo sau là tương đương:

```
pa = A;
pa = &A[0];
```

Với khai báo này, thì địa chỉ trỏ tới của con trỏ pa là địa chỉ của phần tử A[0] và giá trị của con trỏ pa là giá trị của phần tử A[0], tức là *pa = 5;

Phép toán trên con trỏ và mảng

Khi một con trỏ trỏ đến mảng, thì các phép toán tăng hay giảm trên con trỏ sẽ tương ứng với phép dịch chuyển trên mảng.

Ví dụ khai báo:

```
int A[5] = {5, 10, 15, 20, 25};
```

```
int *pa = &A[2];
```

thì con trỏ pa sẽ trỏ đến địa chỉ của phần tử A[2] và giá trị của pa là: $*pa = A[2] = 15$.

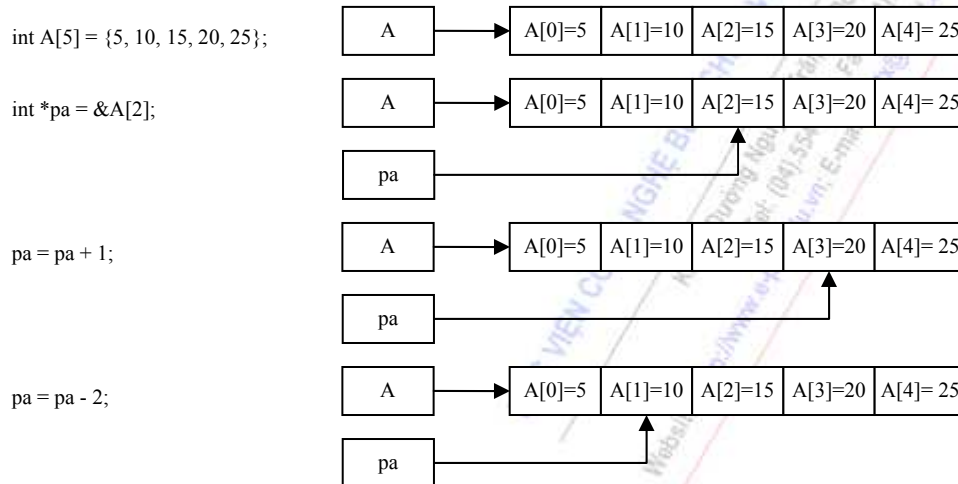
Khi đó, phép toán:

```
pa = pa + 1;
```

sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử tiếp theo của mảng A, đó là địa chỉ của A[3]. Sau đó, phép toán:

```
pa = pa - 2;
```

sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử A[1].



Lưu ý:

- Hai phép toán `pa++` và `*pa++` có tác dụng hoàn toàn khác nhau trên mảng, `pa++` là thao tác trên con trỏ, tức là trên bộ nhớ, nó sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử tiếp theo của mảng. `*pa++` là phép toán trên giá trị, nó tăng giá trị hiện tại của phần tử mảng lên một đơn vị.

Ví dụ:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = &A[2];
```

thì `pa++` là tương đương với `pa = &A[3]` và `*pa = 20`.

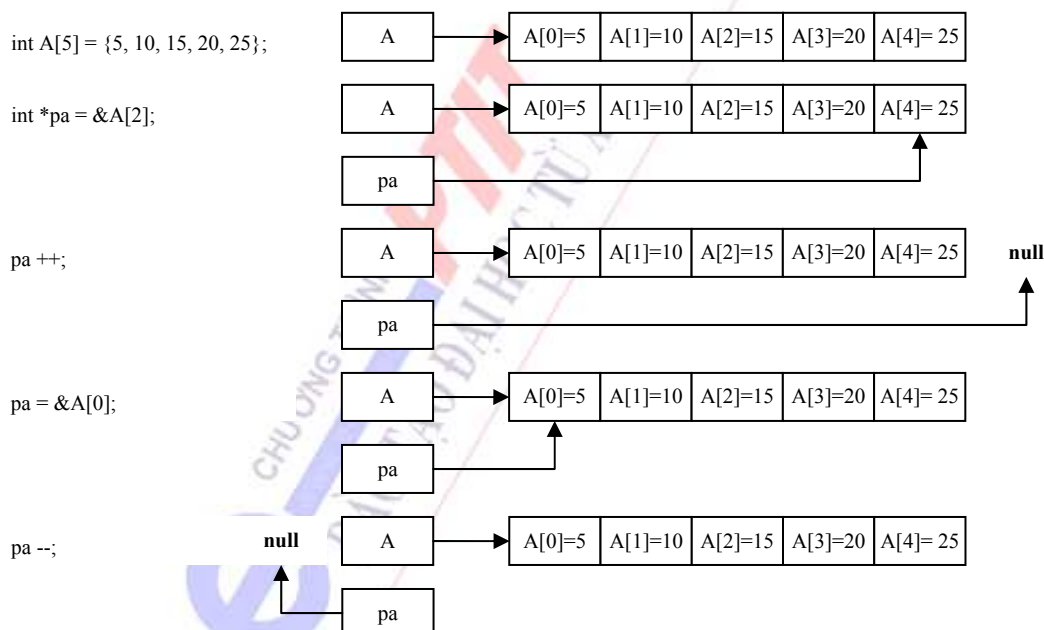
nhưng `*pa++` lại tương đương với `pa = &A[2]` và `*pa = 15+1 = 16`, `A[2] = 16`.



- Trong trường hợp:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = &A[4];
```

thì phép toán pa++ sẽ đưa con trỏ pa trở đến một địa chỉ không xác định. Lí do là A[4] là phần tử cuối của mảng A, nên pa++ sẽ trở đến địa chỉ ngay sau địa chỉ của A[4], địa chỉ này nằm ngoài vùng chỉ số của mảng A nên không xác định. Tương tự với trường hợp pa=&A[0], phép toán pa-- cũng đưa pa trở đến một địa chỉ không xác định.



- Vì mảng A là con trỏ hằng, cho nên không thể thực hiện các phép toán trên A mà chỉ có thể thực hiện trên các con trỏ trỏ đến A: các phép toán pa++ hoặc pa-- là hợp lệ, nhưng các phép toán A++ hoặc A-- là không hợp lệ.

Chương trình 2.2a minh họa việc cài đặt một thủ tục sắp xếp các phần tử của một mảng theo cách thông thường.

Chương trình 2.2a

```
void SortArray(int A[], int n){
    int temp;
    for(int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(A[i] > A[j]){
                temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
}
```

Chương trình 2.2b cài đặt một thủ tục tương tự bằng con trỏ. Hai thủ tục này có chức năng hoàn toàn giống nhau.

Chương trình 2.2b

```
void SortArray(int *A, int n){
    int temp;
    for(int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(*(A+i) > *(A+j)){
                temp = *(A+i);
                *(A+i) = *(A+j);
                *(A+j) = temp;
            }
}
```

Trong chương trình 2.2b, thay vì dùng một mảng, ta dùng một con trỏ để trỏ đến mảng cần sắp xếp. Khi đó, ta có thể dùng các thao tác trên con trỏ thay vì các thao tác trên các phần tử mảng.

2.2.2 Con trỏ và mảng nhiều chiều

Con trỏ và mảng nhiều chiều

Một câu hỏi đặt ra là nếu một ma trận một chiều thì tương đương với một con trỏ, vậy một mảng nhiều chiều thì tương đương với con trỏ như thế nào?

Xét ví dụ:

```
int A[3][3] = {
    {5, 10, 15},
    {20, 25, 30},
    {35, 40, 45}
};
```

Khi đó, địa chỉ của ma trận A chính là địa chỉ của hàng đầu tiên của ma trận A, và cũng là địa chỉ của phần tử đầu tiên của hàng đầu tiên của ma trận A:

- Địa chỉ của ma trận A: $A = A[0] = *(A+0) = \&A[0][0];$
- Địa chỉ của hàng thứ nhất: $A[1] = *(A+1) = \&A[1][0];$
- Địa chỉ của hàng thứ i: $A[i] = *(A+i) = \&A[i][0];$
- Địa chỉ phần tử $\&A[i][j] = (*(A+i)) + j;$
- Giá trị phần tử $A[i][j] = *((*(A+i)) + j);$

Như vậy, một mảng hai chiều có thể thay thế bằng một mảng một chiều các con trỏ cùng kiểu:

```
int A[3][3];
```

có thể thay thế bằng:

```
int (*A)[3];
```

Con trỏ trỏ tới con trỏ

Vì một mảng hai chiều `int A[3][3]` có thể thay thế bằng một mảng các con trỏ `int (*A)[3]`. Hơn nữa, một mảng `int A[3]` lại có thể thay thế bằng một con trỏ `int *A`. Do vậy, một mảng hai chiều có thể thay thế bằng một mảng các con trỏ, hoặc một con trỏ trỏ đến con trỏ. Nghĩa là các cách viết sau là tương đương:

```
int A[3][3];
int (*A)[3];
int **A;
```

2.3 CON TRỎ HÀM

Mặc dù hàm không phải là một biến cụ thể nên không có một địa chỉ xác định. Nhưng trong khi chạy, mỗi một hàm trong C++ cũng có một vùng nhớ xác định, do vậy, C++ cho phép dùng con trỏ để trỏ đến hàm. Con trỏ hàm được dùng để truyền tham số có dạng hàm.

Khai báo con trỏ hàm

Con trỏ hàm được khai báo tương tự như khai báo nguyên mẫu hàm thông thường trong C++, ngoại trừ việc có thêm kí hiệu con trỏ “*” trước tên hàm. Cú pháp khai báo con trỏ hàm như sau:

```
<Kiểu dữ liệu trả về> (*<Tên hàm>) (<Các tham số>);
```

Trong đó:

- **Kiểu dữ liệu trả về:** là các kiểu dữ liệu thông thường của C++ hoặc kiểu do người dùng tự định nghĩa.
- **Tên hàm:** tên do người dùng tự định nghĩa, tuân thủ theo quy tắc đặt tên biến trong C++.
- **Các tham số:** có thể có hoặc không (phần trong dấu “[]” là tùy chọn). Nếu có nhiều tham số, mỗi tham số được phân cách nhau bởi dấu phẩy.

Ví dụ khai báo:

```
int (*Calcul)(int a, int b);
```

là khai báo một con trỏ hàm, tên là `Calcul`, có kiểu `int` và có hai tham số cũng là kiểu `int`.

Lưu ý:

- Dấu “()” bao bọc tên hàm là cần thiết để chỉ ra rằng ta đang khai báo một con trỏ hàm. Nếu không có dấu ngoặc đơn này, trình biên dịch sẽ hiểu rằng ta đang khai báo một hàm thông thường và có giá trị trả về là một con trỏ.

Ví dụ, hai khai báo sau là khác nhau hoàn toàn:

```
// Khai báo một con trỏ hàm
int (*Calcul)(int a, int b);

// Khai báo một hàm trả về kiểu con trỏ
int *Calcul(int a, int b);
```

Sử dụng con trỏ hàm

Con trỏ hàm được dùng khi cần gọi một hàm như là tham số của một hàm khác. Khi đó, một hàm được gọi phải có khuôn mẫu giống với con trỏ hàm đã được khai báo.

Ví dụ, với khai báo:

```
int (*Calcul)(int a, int b);
```

thì có thể gọi các hàm có hai tham số kiểu int và trả về cũng kiểu int như sau:

```
int add(int a, int b);
int sub(int a, int b);
```

nhưng không được gọi các hàm khác kiểu tham số hoặc kiểu trả về như sau:

```
int add(float a, int b);
int add(int a);
char* sub(char* a, char* b);
```

Chương trình 2.3 minh họa việc khai báo và sử dụng con trỏ hàm.

Chương trình 2.3

```
#include <ctype.h>
#include <string>

// Hàm có sử dụng con trỏ hàm như tham số
void Display(char[] str, int (*Xtype)(int c)){
    int index = 0;
    while(str[index] != '\0'){
        cout << (*Xtype)(str[index]); // Sử dụng con trỏ hàm
        index ++;
    }
    return;
}

// Hàm main, dùng lời gọi hàm đến con trỏ hàm
void main(){
    char input[500];
    cout << "Enter the string: ";
    cin >> input;
```



```
char reply;
cout << "Display the string in uppercase or lowercase (u,l): ";
cin >> reply;
if(reply == 'l') // Hiển thị theo dạng lowercase
    Display(str, tolower);
else // Hiển thị theo dạng uppercase
    Display(str, toupper);
return;
}
```

Chương trình 2.3 khai báo hàm Display() có sử dụng con trỏ hàm có khuôn mẫu

```
int (*Xtype)(int c);
```

Trong hàm main, con trỏ hàm này được gọi bởi hai thẻ hiện là các hàm tolower() và hàm toupper(). Hai hàm này được khai báo trong thư viện ctype.h với mẫu như sau:

```
int tolower(int c);
int toupper(int c);
```

Hai khuôn mẫu này phù hợp với con trỏ hàm Xtype trong hàm Display() nên lời gọi hàm Display() trong hàm main là hợp lệ.

2.4 CẤP PHÁT BỘ NHỚ ĐỘNG

Xét hai trường hợp sau đây:

- Trường hợp 1, khai báo một con trỏ và gán giá trị cho nó:
- Trường hợp 2, khai báo con trỏ đến phần tử cuối cùng của mảng rồi tăng thêm một đơn vị cho nó:

```
int *pa = 12;

int A[5] = {5, 10, 15, 20, 25};
int *pa = &A[4];
pa++;
```

Trong cả hai trường hợp, ta đều không biết thực sự con trỏ pa đang trỏ đến địa chỉ nào trong bộ nhớ: trường hợp 1 chỉ ra rằng con trỏ pa đang trỏ tới một địa chỉ không xác định, nhưng lại chứa giá trị là 12 do được gán vào. Trường hợp 2, con trỏ pa đã trỏ đến địa chỉ ngay sau địa chỉ phần tử cuối cùng của mảng A, đó cũng là một địa chỉ không xác định. Các địa chỉ không xác định này là các địa chỉ nằm ở vùng nhớ tự do còn thừa của bộ nhớ. Vùng nhớ này có thể bị chiếm dụng bởi bất kỳ một chương trình nào đang chạy.

Do đó, rất có thể các chương trình khác sẽ chiếm mất các địa chỉ mà con trỏ pa đang trỏ tới. Khi đó, nếu các chương trình thay đổi giá trị của địa chỉ đó, giá trị pa cũng bị thay đổi theo mà ta không thể kiểm soát được. Để tránh các rủi ro có thể gặp phải, C++ yêu cầu phải cấp phát bộ nhớ một cách tường minh cho con trỏ trước khi sử dụng chúng.

2.4.1 Cấp phát bộ nhớ động cho biến

Cấp phát bộ nhớ động

Thao tác cấp phát bộ nhớ cho con trỏ thực chất là gán cho con trỏ một địa chỉ xác định và đưa địa chỉ đó vào vùng đã bị chiếm dụng, các chương trình khác không thể sử dụng địa chỉ đó. Cú pháp cấp phát bộ nhớ cho con trỏ như sau:

```
<tên con trỏ> = new <kiểu con trỏ>;
```

Ví dụ, khai báo:

```
int *pa;
pa = new int;
```

sẽ cấp phát bộ nhớ hợp lệ cho con trỏ pa.

Lưu ý:

- Ta có thể vừa cấp phát bộ nhớ, vừa khởi tạo giá trị cho con trỏ theo cú pháp sau:

```
int *pa;
pa = new int(12);
```

sẽ cấp phát cho con trỏ pa một địa chỉ xác định, đồng thời gán giá trị của con trỏ *pa = 12.

Giải phóng bộ nhớ động

Địa chỉ của con trỏ sau khi được cấp phát bởi thao tác **new** sẽ trở thành vùng nhớ đã bị chiếm dụng, các chương trình khác không thể sử dụng vùng nhớ đó ngay cả khi ta không dùng con trỏ nữa. Để tiết kiệm bộ nhớ, ta phải huỷ bỏ vùng nhớ của con trỏ ngay sau khi không dùng đến con trỏ nữa. Cú pháp huỷ bỏ vùng nhớ của con trỏ như sau:

```
delete <tên con trỏ>;
```

Ví dụ:

```
int *pa = new int(12); // Khai báo con trỏ pa, cấp phát bộ nhớ
                        // và gán giá trị ban đầu cho pa là 12.
delete pa;             // Giải phóng vùng nhớ vừa cấp cho pa.
```

Lưu ý:

- Một con trỏ, sau khi bị giải phóng địa chỉ, vẫn có thể được cấp phát một vùng nhớ mới hoặc trỏ đến một địa chỉ mới:

```
int *pa = new int(12); // Khai báo con trỏ pa, cấp phát bộ nhớ
                        // và gán giá trị ban đầu cho pa là 12.
delete pa;             // Giải phóng vùng nhớ vừa cấp cho pa.
int A[5] = {5, 10, 15, 20, 25};
pa = A;                // Cho pa trỏ đến địa chỉ của mảng A
```

- Nếu có nhiều con trỏ cùng trỏ vào một địa chỉ, thì chỉ cần giải phóng bộ nhớ của một con trỏ, tất cả các con trỏ còn lại cũng bị giải phóng bộ nhớ:

```
int *pa = new int(12); // *pa = 12
int *pb = pa;          // pb trỏ đến cùng địa chỉ pa.
*pb += 5;              // *pa = *pb = 17
delete pa;             // Giải phóng cả pa lẫn pb
```

- Một con trỏ sau khi cấp phát bộ nhớ động bằng thao tác **new**, cần phải phóng bộ nhớ trước khi trỏ đến một địa chỉ mới hoặc cấp phát bộ nhớ mới:

```
int *pa = new int(12); // pa được cấp bộ nhớ và *pa = 12
*pa = new int(15);    // pa trỏ đến địa chỉ khác và *pa = 15.
                      // địa chỉ cũ của pa vẫn bị coi là bận
```

2.4.2 Cấp phát bộ nhớ cho mảng động một chiều

Cấp phát bộ nhớ cho mảng động một chiều

Mảng một chiều được coi là tương ứng với một con trỏ cùng kiểu. Tuy nhiên, cú pháp cấp phát bộ nhớ cho mảng động một chiều là khác với cú pháp cấp phát bộ nhớ cho con trỏ thông thường:

```
<Tên con trỏ> = new <Kiểu con trỏ>[<Độ dài mảng>];
```

Trong đó:

- **Tên con trỏ:** tên do người dùng đặt, tuân thủ theo quy tắc đặt tên biến của C++.
- **Kiểu con trỏ:** Kiểu dữ liệu cơ bản của C++ hoặc là kiểu do người dùng tự định nghĩa.
- **Độ dài mảng:** số lượng các phần tử cần cấp phát bộ nhớ của mảng.

Ví dụ:

```
int *A = new int[5];
```

sẽ khai báo một mảng A có 5 phần tử kiểu int được cấp phát bộ nhớ động.

Lưu ý:

- Khi cấp phát bộ nhớ cho con trỏ có khởi tạo thông thường, ta dùng dấu “()”, khi cấp phát bộ nhớ cho mảng, ta dùng dấu “[]”. Hai lệnh cấp phát sau là hoàn toàn khác nhau:

```
// Cấp phát bộ nhớ và khởi tạo cho một con trỏ int
int *A = new int(5);
// Cấp phát bộ nhớ cho một mảng 5 phần tử kiểu int
int *A = new int[5];
```

Giải phóng bộ nhớ của mảng động một chiều

Để giải phóng vùng nhớ đã được cấp phát cho một mảng động, ta dùng cú pháp sau:

```
delete [] <tên con trỏ>;
```

Ví dụ:

```
// Cấp phát bộ nhớ cho một mảng có 5 phần tử kiểu int
int *A = new int[5];
// Giải phóng vùng nhớ do mảng A đang chiếm giữ.
delete [] A;
```

Chương trình 2.4 minh họa hai thủ tục khởi tạo và giải phóng một mảng động một chiều.

Chương trình 2.4

```
void InitArray(int *A, int length){
    A = new int[length];
    for(int i=0; i<length; i++)
```

```

        A[i] = 0;
    return;
}

void DeleteArray(int *A){
    delete [] A;
    return;
}

```

2.4.3 Cấp phát bộ nhớ cho mảng động nhiều chiều

Cấp phát bộ nhớ cho mảng động nhiều chiều

Một mảng hai chiều là một con trỏ đến một con trỏ. Do vậy, ta phải cấp phát bộ nhớ theo từng chiều theo cú pháp cấp phát bộ nhớ cho mảng động một chiều.

Ví dụ:

```

int **A;
const int length = 10;
A = new int*[length];    // Cấp phát bộ nhớ cho số dòng của ma trận A
for(int i=0; i<length; i++)
    // Cấp phát bộ nhớ cho các phần tử của mỗi dòng
    A[i] = new int[length];

```

sẽ cấp phát bộ nhớ cho một mảng động hai chiều, tương đương với một ma trận có kích thước 10*10.

Lưu ý:

- Trong lệnh cấp phát `A = new int*[length]`, cần phải có dấu “*” để chỉ ra rằng cần cấp phát bộ nhớ cho một mảng các phần tử có kiểu là con trỏ `int` (`int*`), khác với kiểu `int` bình thường.

Giải phóng bộ nhớ của mảng động nhiều chiều

Ngược lại với khi cấp phát, ta phải giải phóng lần lượt bộ nhớ cho con trỏ tương ứng với cột và hàng của mảng động.

Ví dụ:

```

int **A;
...;                                // cấp phát bộ nhớ
...
for(int i=0; i<length; i++)
    delete [] A[i];    // Giải phóng bộ nhớ cho mỗi dòng
delete [] A;           // Giải phóng bộ nhớ cho mảng các dòng

```

sẽ giải phóng bộ nhớ cho một mảng động hai chiều.

Chương trình 2.5 minh họa việc dùng mảng động hai chiều để tính tổng của hai ma trận.

Chương trình 2.5

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

/* Khai báo nguyên mẫu hàm */
void InitArray(int **A, int row, int colum);
void AddArray(int **A, int **B, int row, int colum);
void DisplayArray(int **A, int row, int colum);
void DeleteArray(int **A, int row);

void InitArray(int **A, int row, int colum){
    A = new int*[row];
    for(int i=0; i<row; i++){
        A[i] = new int[colum];
        for(int j=0; j<colum; j++){
            cout << "Phan tu [" << i << ", " << j << "] = ";
            cin >> A[i][j];
        }
    }
    return;
}

void AddArray(int **A, int **B, int row, int colum){
    for(int i=0; i<row; i++)
        for(int j=0; j<colum; j++)
            A[i][j] += B[i][j];
    return;
}

void DisplayArray(int **A, int row, int colum){
    for(int i=0; i<row; i++){
        for(int j=0; j<colum; j++)
            cout << A[i][j] << " ";
        cout << endl; // Xuống dòng
    }
    return;
}

void DeleteArray(int **A, int row){
    for(int i=0; i<row; i++)
        delete [] A[i];
    delete [] A;
    return;
}
```

```
void main() {
    clrscr();
    int **A, **B, row, column;
    cout << "So dong: ";
    cin >> row;
    cout << "So cot: ";
    cin >> column;

    /* Khởi tạo các ma trận */
    cout << "Khoi tao mang A:" << endl;
    InitArray(A, row, column);

    cout << "Khoi tao mang B:" << endl;
    InitArray(B, row, column);

    // Cộng hai ma trận
    AddArray(A, B, row, column);

    // Hiển thị ma trận kết quả
    cout << "Tong hai mang A va mang B:" << endl;
    DisplayArray(A, row, column);

    // Giải phóng bộ nhớ
    DeleteArray(A, row);
    DeleteArray(B, row);
    return;
}
```

TỔNG KẾT CHƯƠNG 2

Nội dung chương 2 đã trình bày các vấn đề liên quan đến việc khai báo và sử dụng con trỏ và mảng trong ngôn ngữ C++:

- Con trỏ là một kiểu biến đặc biệt, nó trỏ đến địa chỉ của một biến khác. Có hai cách truy nhập đến con trỏ là truy nhập đến địa chỉ hoặc truy nhập đến giá trị của địa chỉ mà con trỏ trỏ đến.
- Con trỏ có thể tham gia vào các phép toán như các biến thông thường bằng phép lấy giá trị.
- Một con trỏ có sự tương ứng với một mảng một chiều có cùng kiểu.
- Một ma trận hai chiều có thể thay thế bằng một mảng các con trỏ hoặc một con trỏ trỏ đến con trỏ.
- Một con trỏ có thể trỏ đến một hàm, khi đó, nó được dùng để gọi một hàm như là một tham số cho hàm khác.

- Một con trỏ cần phải trỏ vào một địa chỉ xác định hoặc phải được cấp phát bộ nhớ qua phép toán **new** và giải phóng bộ nhớ sau khi dùng bằng thao tác **delete**.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 2

1. Trong các khai báo con trỏ sau, những khai báo nào là đúng:

- a. `int A*;`
- b. `*int A;`
- c. `int* A, B;`
- d. `int* A, *B;`
- e. `int *A, *B;`

2. Với khai báo:

```
int a = 12;  
int *pa;
```

Các phép gán nào sau đây là hợp lệ:

- a. `pa = &a;`
- b. `pa = a;`
- c. `*pa = &a;`
- d. `*pa = a;`

3. Với khai báo:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A+2;
```

Khi đó, `*pa = ?`

- a. 10
- b. 20
- c. 30
- d. 40
- e. 50

4. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
*pa += 2;
```

Khi đó, `*pa = ?`

- a. 10
- b. 12
- c. 30
- d. 32

5. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;
```

```
pa += 2;
```

Khi đó, *pa = ?

- a. 10
- b. 12
- c. 30
- d. 32

6. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
pa += 2;
```

Khi đó, pa = ?

- a. &A[0]
- b. A[2]
- c. &A[2]
- d. Không xác định

7. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
pa -= 2;
```

Khi đó, pa = ?

- a. &A[0]
- b. &A[2]
- c. &A[4]
- d. Không xác định

8. Với đoạn chương trình:

```
int A[3][3] = {  
    {10, 20, 30},  
    {40, 50, 60},  
    {70, 80, 90}  
};  
  
int *pa;
```

Khi đó, để có được kết quả *pa = 50, các lệnh nào sau đây là đúng?

- a. pa = A + 4;
- b. pa = (*(A+1)) + 1;
- c. pa = &A[1][1];
- d. pa = *((*(A+1)) + 1);

9. Giả sử ta khai báo một hàm có sử dụng con trỏ hàm với khuôn mẫu như sau:

```
int Calcul(int a, int b, int (*Xcalcul)(int x, int y)){}  
Và ta có cài đặt một số hàm như sau:
```



```
int add(int a, int b);  
void cal(int a, int b);  
int square(int a);
```

Khi đó, lời gọi hàm nào sau đây là đúng:

- a. `Calcul(5, 10, add);`
- b. `Calcul(5, 10, add(2, 3));`
- c. `Calcul(5, 10, cal);`
- d. `Calcul(5, 10, square);`

10. Ta muốn cấp phát bộ nhớ cho một con trỏ kiểu `int` và khởi đầu giá trị cho nó là 20. Lệnh nào sau đây là đúng:

- a. `int *pa = 20;`
- b. `int *pa = new int{20};`
- c. `int *pa = new int(20);`
- d. `int *pa = new int[20];`

11. Ta muốn cấp phát bộ nhớ cho một mảng động kiểu `int` có chiều dài là 20. Lệnh nào sau đây là đúng:

- a. `int *pa = 20;`
- b. `int *pa = new int{20};`
- c. `int *pa = new int(20);`
- d. `int *pa = new int[20];`

12. Xét đoạn chương trình sau:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
pa = new int(2);
```

Khi đó, `*pa = ?`

- a. 10
- b. 30
- c. 2
- d. Không xác định

13. Xét đoạn chương trình sau:

```
1> int A[5] = {10, 20, 30, 40, 50};  
2> int *pa = A;  
3> pa += 15;  
4> delete pa;
```

Đoạn chương trình trên có lỗi ở dòng nào?

- a. 1
- b. 2
- c. 3
- d. 4

14. Viết chương trình thực hiện các phép toán cộng, trừ, nhân, chia trên đa thức. Các đa thức được biểu diễn bằng mảng động một chiều. Bậc của đa thức và các hệ số tương ứng được nhập từ bàn phím.
15. Viết chương trình thực hiện các phép toán cộng, trừ, nhân hai ma trận kích thước $m \times n$. Các ma trận được biểu diễn bằng mảng động hai chiều. Giá trị kích cỡ ma trận (m, n) và giá trị các phần tử của ma trận được nhập từ bàn phím.



CHƯƠNG 3

Kiểu dữ liệu cấu trúc

Nội dung chương này tập trung trình bày các vấn đề liên quan đến kiểu dữ liệu có cấu trúc trong C++:

- Định nghĩa một cấu trúc
- Sử dụng một cấu trúc bằng các phép toán cơ bản trên cấu trúc
- Con trỏ cấu trúc, khai báo và sử dụng con trỏ cấu trúc
- Mảng các cấu trúc, khai báo và sử dụng mảng các cấu trúc
- Một số kiểu dữ liệu trừu tượng khác như ngăn xếp, hàng đợi, danh sách liên kết.

3.1 ĐỊNH NGHĨA CẤU TRÚC

Kiểu dữ liệu có cấu trúc được dùng khi ta cần nhóm một số biến dữ liệu luôn đi kèm với nhau. Khi đó, việc xử lý trên một nhóm các biến được thực hiện như trên các biến cơ bản thông thường.

3.1.1 Khai báo cấu trúc

Trong C++, một cấu trúc do người dùng tự định nghĩa được khai báo thông qua từ khoá **struct**:

```
struct <Tên cấu trúc>{  
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;  
    <Kiểu dữ liệu 2> <Tên thuộc tính 2>;  
    ...  
    <Kiểu dữ liệu n> <Tên thuộc tính n>;  
};
```

Trong đó:

- **struct**: là tên từ khoá để khai báo một cấu trúc, bắt buộc phải có khi định nghĩa cấu trúc.
- **Tên cấu trúc**: là tên do người dùng tự định nghĩa, tuân thủ theo quy tắc đặt tên biến trong C++. Tên này sẽ trở thành tên của kiểu dữ liệu có cấu trúc tương ứng.
- **Thuộc tính**: mỗi thuộc tính của cấu trúc được khai báo như khai báo một biến thuộc kiểu dữ liệu thông thường, gồm có kiểu dữ liệu và tên biến tương ứng. Mỗi khai báo thuộc tính phải kết thúc bằng dấu chấm phẩy “;” như một câu lệnh C++ thông thường.

Ví dụ, để quản lý nhân viên của một công ty, khi xử lý thông tin về mỗi nhân viên, ta luôn phải xử lý các thông tin liên quan như:

- Tên
- Tuổi
- Chức vụ
- Lương

Do đó, ta sẽ dùng cấu trúc để lưu giữ thông tin về mỗi nhân viên bằng cách định nghĩa một cấu trúc có tên là Employee với các thuộc tính như sau:

```
struct Employeee{
    char name[20];    // Tên nhân viên
    int age;           // Tuổi nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
};
```

Lưu ý:

- Cấu trúc chỉ cần định nghĩa một lần trong chương trình và có thể được khai báo biến cấu trúc nhiều lần. Khi cấu trúc đã được định nghĩa, việc khai báo biến ở lần khác trong chương trình được thực hiện như khai báo biến thông thường:

<Tên cấu trúc> <tên biến 1>, <tên biến 2>;

Ví dụ, sau khi đã định nghĩa cấu trúc Employeee, muốn có biến myEmployeee, ta khai báo như sau:

```
Employee myEmployeee;
```

3.1.2 Cấu trúc lồng nhau

Các cấu trúc có thể được định nghĩa lồng nhau khi một thuộc tính của một cấu trúc cũng cần có kiểu là một cấu trúc khác. Khi đó, việc định nghĩa cấu trúc cha được thực hiện như một cấu trúc bình thường, với khai báo về thuộc tính đó là một cấu trúc con:

```
struct <Tên cấu trúc cha>{
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;
    // Có kiểu cấu trúc
    <Kiểu cấu trúc con> <Tên thuộc tính 2>;
    ...
    <Kiểu dữ liệu n> <Tên thuộc tính n>;
};
```

Ví dụ, với kiểu cấu trúc Employee, ta không quan tâm đến tuổi nhân viên nữa, mà quan tâm đến ngày sinh của nhân viên. Vì ngày sinh cần có các thông tin luôn đi với nhau là ngày sinh, tháng sinh, năm sinh. Do đó, ta định nghĩa một kiểu cấu trúc con cho kiểu ngày sinh:

```
struct Date{
    int day;
    int month;
    int year;
};
```

khi đó, cấu trúc Employee trở thành:

```
struct Employee{
    char name[20];    // Tên nhân viên
    Date birthDay;  // Ngày sinh của nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
};
```

Lưu ý:

- Trong định nghĩa các cấu trúc lồng nhau, cấu trúc con phải được định nghĩa trước cấu trúc cha để đảm bảo các kiểu dữ liệu của các thuộc tính của cấu trúc cha là tường minh tại thời điểm nó được định nghĩa.

3.1.3 Định nghĩa cấu trúc với từ khoá typedef

Để tránh phải dùng từ khoá **struct** mỗi khi khai báo biến cấu trúc, ta có thể dùng từ khoá **typedef** khi định nghĩa cấu trúc:

```
typedef struct {  
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;  
    <Kiểu dữ liệu 2> <Tên thuộc tính 2>;  
    ...  
    <Kiểu dữ liệu n> <Tên thuộc tính n>;  
} <Tên kiểu dữ liệu cấu trúc>;
```

Trong đó:

- **Tên kiểu dữ liệu cấu trúc:** là tên kiểu dữ liệu của cấu trúc vừa định nghĩa. Tên này sẽ được dùng như một kiểu dữ liệu thông thường khi khai báo biến cấu trúc.

Ví dụ, muốn có kiểu dữ liệu có cấu trúc nhân viên, có tên là Employee, ta dùng từ khoá **typedef** để định nghĩa cấu trúc như sau:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    int age;           // Tuổi nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;
```

Khi đó, muốn có hai biến là myEmployee1 và myEmployee2 có kiểu cấu trúc Employee, ta chỉ cần khai báo như sau mà không cần từ khoá **struct**:

```
Employee myEmployee1, myEmployee2;
```

Trong ví dụ khai báo lồng cấu trúc Employee, dùng từ khoá **typedef** cho kiểu Date:

```
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;
```

cấu trúc Employee trở thành:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    Date birthDay;    // Ngày sinh của nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;
```

Lưu ý:

- Khi không dùng từ khoá **typedef**, tên cấu trúc (nằm sau từ khoá **struct**) được dùng để khai báo biến. Trong khi đó, khi có từ khoá **typedef**, tên kiểu dữ liệu cấu trúc (dòng cuối cùng trong định nghĩa) mới được dùng để khai báo biến.
- Khi dùng từ khoá **typedef** thì không thể khai báo biến đồng thời với định nghĩa cấu trúc.

3.2 THAO TÁC TRÊN CẤU TRÚC

Các thao tác trên cấu trúc bao gồm:

- Khai báo và khởi tạo giá trị ban đầu cho biến cấu trúc
- Truy nhập đến các thuộc tính của cấu trúc

3.2.1 Khởi tạo giá trị ban đầu cho cấu trúc

Khởi tạo biến có cấu trúc đơn

Biến cấu trúc được khai báo theo các cách sau:

```
<Tên kiểu dữ liệu cấu trúc> <tên biến>;
```

Ngoài ra, ta có thể khởi tạo các giá trị cho các thuộc tính của cấu trúc ngay khi khai báo bằng các cú pháp sau:

```
<Tên kiểu dữ liệu cấu trúc> <tên biến> = {  
    <giá trị thuộc tính 1>,  
    <giá trị thuộc tính 2>,  
    ...  
    <giá trị thuộc tính n>  
};
```

Trong đó:

- **Giá trị thuộc tính:** là giá trị khởi đầu cho mỗi thuộc tính, có kiểu phù hợp với kiểu dữ liệu của thuộc tính. Mỗi giá trị của thuộc tính được phân cách bằng dấu phẩy “,”.

Ví dụ, với định nghĩa cấu trúc:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    int age;           // Tuổi nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;
```

thì có thể khai báo và khởi tạo cho một biến như sau:

```
Employee myEmployee1 = {  
    "Nguyen Van A",  
    27,  
    "Nhan vien",  
    300f
```

```
};
```

Khởi tạo các biến có cấu trúc lồng nhau

Trong trường hợp các cấu trúc lồng nhau, phép khởi tạo cũng thực hiện như thông thường với phép khởi tạo cho tất cả các cấu trúc con.

Ví dụ với khai báo cấu trúc như sau:

```
typedef struct {
    int day;
    int month;
    int year;
} Date;
```

và:

```
typedef struct {
    char name[20];    // Tên nhân viên
    Date birthDay;    // Ngày sinh của nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
} Employee;
```

Thì khai báo và khởi tạo một biến có kiểu `Employee` có thể thực hiện như sau:

```
Employee myEmployee1 = {
    "Nguyen Van A",
    {15, 05, 1980},    // Khởi tạo cấu trúc con
    "Nhan vien",
    300f
};
```

3.2.2 Truy nhập đến thuộc tính của cấu trúc

Việc truy nhập đến thuộc tính của cấu trúc được thực hiện bằng cú pháp:

<Tên biến cấu trúc>.<tên thuộc tính>

Ví dụ, với một biến cấu trúc kiểu `Employee` đơn:

```
Employee myEmployee1 = {
    "Nguyen Van A",
    27,
    "Nhan vien",
    300f
};
```

ta có thể truy xuất như sau:

```
cout << myEmployee1.name;    // hiển thị ra "Nguyen Van A"
myEmployee1.age += 1;         // Tăng số tuổi lên 1
```

Đối với kiểu cấu trúc lồng nhau, phép truy nhập đến thuộc tính được thực hiện lần lượt từ cấu trúc cha đến cấu trúc con.

Ví dụ, với một biến cấu trúc kiểu `Employee` lồng nhau:


```
Employee myEmployee1 = {  
    "Nguyen Van A",  
    {15, 05, 1980},  
    "Nhan vien",  
    300f  
};
```

ta có thể truy xuất như sau:

```
cout << myEmployee1.name;           // hiển thị ra "Nguyen Van A"  
myEmployee1.birthDay.day = 16;       // Sửa lại ngày sinh thành 16  
myEmployee1.birthDay.month = 07;     // Sửa lại tháng sinh thành 07
```

Chương trình 3.1a minh hoạ việc tạo lập và sử dụng cấu trúc Employee đơn, không dùng từ khoá typedef.

Chương trình 3.1a

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
struct Employee{  
    char name[20];    // Tên nhân viên  
    int age;          // Tuổi nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
};  
  
/* Khai báo khuôn mẫu hàm */  
void Display(Employee myEmployee);  
  
void Display(Employee myEmployee){  
    cout << "Name: " << myEmployee.name << endl;  
    cout << "Age: " << myEmployee.age << endl;  
    cout << "Role: " << myEmployee.role << endl;  
    cout << "Salary: " << myEmployee.salary << endl;  
    return;  
}  
  
void main(){  
    clrscr();  
    // Hiển thị giá trị mặc định  
    Employee myEmployee =
```



```
        {"Nguyen Van A", 27, "Nhan vien", 300f};  
    cout << "Thông tin mặc định:" << endl;  
    Display(myEmployee);  
  
    // Thay đổi giá trị cho các thuộc tính  
    cout << "Name: ";  
    cin >> myEmployee.name;  
    cout << "Age: ";  
    cin >> myEmployee.age;  
    cout << "Role: ";  
    cin >> myEmployee.role;  
    cout << "Salary: ";  
    cin >> myEmployee.salary;  
  
    cout << "Thông tin sau khi thay đổi:" << endl;  
    Display(myEmployee);  
    return;  
}
```

Chương trình 3.1b minh họa việc tạo lập và sử dụng cấu trúc Employee lồng nhau, có dùng từ khoá **typedef**.

Chương trình 3.1b

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;  
  
typedef struct {  
    char name[20]; // Tên nhân viên  
    Date birthDay; // Ngày sinh của nhân viên  
    char role[20]; // Chức vụ của nhân viên  
    float salary; // Lương của nhân viên  
} Employee;  
  
/* Khai báo khuôn mẫu hàm */  
void Display(Employee myEmployee);
```

```
void Display(Employee myEmployee) {
    cout << "Name: " << myEmployee.name << endl;
    cout << "Birth day: " << myEmployee.birthDay.day << "/"
        << myEmployee.birthDay.month << "/"
        << myEmployee.birthDay.year << endl;
    cout << "Role: " << myEmployee.role << endl;
    cout << "Salary: " << myEmployee.salary << endl;
    return;
}

void main() {
    clrscr();
    // Hiển thị giá trị mặc định
    Employee myEmployee =
        {"Nguyen Van A", {15, 5, 1980}, "Nhan vien", 300f};
    cout << "Thông tin mặc định:" << endl;
    Display(myEmployee);

    // Thay đổi giá trị cho các thuộc tính
    cout << "Name: ";
    cin >> myEmployee.name;
    cout << "Day of birth: ";
    cin >> myEmployee.birthDay.day;
    cout << "Month of birth: ";
    cin >> myEmployee.birthDay.month;
    cout << "Year of birth: ";
    cin >> myEmployee.birthDay.year;
    cout << "Role: ";
    cin >> myEmployee.role;
    cout << "Salary: ";
    cin >> myEmployee.salary;

    cout << "Thông tin sau khi thay đổi:" << endl;
    Display(myEmployee);
    return;
}
```

3.3 CON TRỎ CẤU TRÚC VÀ MẢNG CẤU TRÚC

3.3.1 Con trỏ cấu trúc

Con trỏ cấu trúc là một con trỏ trỏ đến địa chỉ của một biến có kiểu cấu trúc. Cách khai báo và sử dụng con trỏ cấu trúc được thực hiện như con trỏ thông thường.

Khai báo con trỏ cấu trúc

Con trỏ cấu trúc được khai báo theo cú pháp:

```
<Tên kiểu cấu trúc> *<Tên biến>;
```

Ví dụ, với kiểu khai báo cấu trúc:

```
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;
```

và:

```
typedef struct {  
    char name[20];    // Tên nhân viên  
    Date birthDay;    // Ngày sinh của nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;    // Lương của nhân viên  
} Employee;
```

thì ta có thể khai báo một con trỏ cấu trúc như sau:

```
Employee *ptrEmployee;
```

Lưu ý:

- Cũng như khai báo con trỏ thông thường, dấu con trỏ “*” có thể nằm ngay trước tên biến hoặc nằm ngay sau tên kiểu cấu trúc.

Cũng giống con trỏ thông thường, con trỏ cấu trúc được sử dụng khi:

- Cho nó trỏ đến địa chỉ của một biến cấu trúc
- Cấp phát cho nó một vùng nhớ xác định.

Gán địa chỉ cho con trỏ cấu trúc

Một con trỏ cấu trúc có thể trỏ đến địa chỉ của một biến cấu trúc có cùng kiểu thông qua phép gán:

```
<Tên biến con trỏ> = &<Tên biến thường>;
```

Ví dụ, khai báo và phép gán:

```
Employee *ptrEmployee, myEmployee;  
ptrEmployee = &myEmployee;
```

sẽ đưa con trỏ ptrEmployee trỏ đến địa chỉ của biến cấu trúc myEmployee.

Cấp phát bộ nhớ động cho con trỏ cấu trúc

Trong trường hợp ta muốn tạo ra một con trỏ cấu trúc mới, không trỏ vào một biến cấu trúc có sẵn nào, để sử dụng con trỏ mới này, ta phải cấp phát vùng nhớ cho nó. Cú pháp cấp phát vùng nhớ cho con trỏ cấu trúc:

```
<Tên biến con trỏ> = new <Kiểu cấu trúc>;
```

Ví dụ, cấu trúc `Employee` được khai báo bằng từ khoá **typedef**, ta có thể cấp phát vùng nhớ cho con trỏ cấu trúc như sau:

```
Employee *ptrEmployee;  
ptrEmployee = new Employee;
```

hoặc cấp phát ngay khi khai báo:

```
Employee *ptrEmployee = new Employee;
```

Sau khi cấp phát vùng nhớ cho con trỏ bằng thao tác **new**, khi con trỏ không được dùng nữa, hoặc cần trỏ sang một địa chỉ khác, ta phải giải phóng vùng nhớ vừa được cấp phát cho con trỏ bằng thao tác:

```
delete <Tên biến con trỏ>;
```

Ví dụ:

```
Employee *ptrEmployee = new Employee;  
...  
// Thực hiện các thao tác trên con trỏ  
...  
delete ptrEmployee;
```

Lưu ý:

- Thao tác **delete** chỉ được thực hiện đối với con trỏ mà trước đó, nó được cấp phát bộ nhớ động thông qua thao tác **new**:

```
Employee *ptrEmployee = new Employee;  
delete ptrEmployee; //đúng
```

mà không thể thực hiện với con trỏ chỉ trỏ đến địa chỉ của một biến cấu trúc khác:

```
Employee *ptrEmployee, myEmployee;  
ptrEmployee = &myEmployee;  
delete ptrEmployee; //lỗi
```

Truy nhập thuộc tính của con trỏ cấu trúc

Thuộc tính của con trỏ cấu trúc có thể được truy nhập thông qua hai cách:

Cách 1:

```
<Tên biến con trỏ> -> <Tên thuộc tính>;
```

Cách 2:

```
(*<Tên biến con trỏ>).<Tên thuộc tính>;
```

Ví dụ, thuộc tính tên nhân viên của cấu trúc `Employee` có thể được truy nhập thông qua hai cách:

```
Employee *ptrEmployee = new Employee;  
cin >> ptrEmployee -> name;
```

hoặc:

```
cin >> (*ptrEmployee).name;
```

Lưu ý:

- Trong cách truy nhập thứ hai, phải có dấu ngoặc đơn “()” quanh tên con trỏ vì phép toán truy nhập thuộc tính “.” có độ ưu tiên cao hơn phép toán lấy giá trị con trỏ “*”.
- Thông thường, ta dùng cách thứ nhất cho đơn giản và thuận tiện.

Chương trình 3.2 cài đặt việc khởi tạo và hiển thị nội dung của một con trỏ cấu trúc.

Chương trình 3.2

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

typedef struct {
    int day;
    int month;
    int year;
} Date;

typedef struct {
    char name[20];    // Tên nhân viên
    Date birthDay;    // Ngày sinh của nhân viên
    char role[20];    // Chức vụ của nhân viên
    float salary;     // Lương của nhân viên
} Employee;

/* Khai báo khuôn mẫu hàm */
void InitStruct(Employee *myEmployee);
void Display(Employee *myEmployee);

void InitStruct(Employee *myEmployee){
    myEmployee = new Employee;
    cout << "Name: ";
    cin >> myEmployee->name;
    cout << "Day of birth: ";
    cin >> myEmployee->birthDay.day;
    cout << "Month of birth: ";
    cin >> myEmployee->birthDay.month;
    cout << "Year of birth: ";
    cin >> myEmployee->birthDay.year;
    cout << "Role: ";
    cin >> myEmployee->role;
    cout << "Salary: ";
```

```
cin >> myEmployee->salary;
}

void Display(Employee myEmployee) {
    cout << "Name: " << myEmployee->name << endl;
    cout << "Birth day: " << myEmployee->birthDay.day << "/"
        << myEmployee->birthDay.month << "/"
        << myEmployee->birthDay.year << endl;
    cout << "Role: " << myEmployee->role << endl;
    cout << "Salary: " << myEmployee->salary << endl;
    return;
}

void main() {
    clrscr();
    Employee *myEmployee;
    InitStruct(myEmployee);
    Display(myEmployee);
    return;
}
```

3.3.2 Mảng cấu trúc

Khi cần xử lý nhiều đối tượng có dùng kiểu dữ liệu cấu trúc, ta có thể sử dụng mảng các cấu trúc. Vì một mảng một chiều là tương đương với một con trỏ có cùng kiểu. Do đó, có thể khai báo mảng theo hai cách: Khai báo mảng tĩnh như thông thường hoặc khai báo mảng động thông qua con trỏ.

Khai báo mảng tĩnh các cấu trúc

Khai báo mảng tĩnh các cấu trúc theo cú pháp:

<Tên kiểu cấu trúc> <Tên biến mảng>[<Số phần tử mảng>];

Ví dụ:

```
Employee employees[10];
```

là khai báo một mảng tên là employees gồm 10 phần tử có kiểu là cấu trúc Employee.

Khai báo mảng động các cấu trúc

Khai báo một mảng động các cấu trúc hoàn toàn tương tự khai báo một con trỏ cấu trúc cùng kiểu:

<Tên kiểu cấu trúc> *<Tên biến>;

Ví dụ, khai báo:

```
Employee *employees;
```

vừa có thể coi là khai báo một con trỏ thông thường có cấu trúc Employee, vừa có thể coi là khai báo một mảng động các cấu trúc có kiểu cấu trúc Employee.

Tuy nhiên, cách cấp phát bộ nhớ động cho mảng các cấu trúc khác với một con trỏ. Đây là cách để chương trình nhận biết ta đang dùng một con trỏ cấu trúc hay một mảng động có cấu trúc. Cách pháp cấp phát bộ nhớ cho mảng động như sau:

```
<Tên biến mảng> = new <Kiểu cấu trúc>[<Số lượng phần tử>];
```

Ví dụ, khai báo:

```
Employee *employees = new Employee[10];
```

sẽ cấp phát bộ nhớ cho một mảng động employees có 10 phần tử kiểu cấu trúc Employee.

Truy nhập đến phần tử của mảng cấu trúc

Việc truy nhập đến các phần tử của mảng cấu trúc được thực hiện như truy cập đến phần tử của mảng thông thường. Ví dụ muốn truy nhập đến thuộc tính tên nhân viên phần tử nhân viên thứ i trong mảng cấu trúc, ta viết như sau:

```
Employee *employees = new Employee[10];  
employees[i].name;
```

Chương trình 3.3 cài đặt việc khởi tạo một mảng các nhân viên của một phòng trong một công ty. Sau đó, chương trình sẽ tìm và in ra thông tin về nhân viên có lương cao nhất và nhân viên có lương thấp nhất trong phòng.

Chương trình 3.3

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;  
  
typedef struct {  
    char name[20];    // Tên nhân viên  
    Date birthDay;    // Ngày sinh của nhân viên  
    char role[20];    // Chức vụ của nhân viên  
    float salary;     // Lương của nhân viên  
} Employee;  
  
/* Khai báo khuôn mẫu hàm */  
void InitArray(Employee *myEmployee, int length);  
Employee searchSalaryMax(Employee *myEmployee, int length);  
Employee searchSalaryMin(Employee *myEmployee, int length);  
void Display(Employee myEmployee);
```



```
void InitArray(Employee *myEmployee, int length){
    myEmployee = new Employee[length];
    for(int i=0; i<length; i++){
        cout << "Nhan vien thu " << i << endl;
        cout << "Name: ";
        cin >> myEmployee[i].name;
        cout << "Day of birth: ";
        cin >> myEmployee[i].birthDay.day;
        cout << "Month of birth: ";
        cin >> myEmployee[i].birthDay.month;
        cout << "Year of birth: ";
        cin >> myEmployee[i].birthDay.year;
        cout << "Role: ";
        cin >> myEmployee[i].role;
        cout << "Salary: ";
        cin >> myEmployee[i].salary;
    }
    return;
}

Employee searchSalaryMax(Employee *myEmployee, int length){
    int index = 0;
    int maxSalary = myEmployee[0].salary;
    for(int i=1; i<length; i++){
        if(myEmployee[i].salary > maxSalary){
            maxSalary = myEmployee[i].salary;
            index = i;
        }
    }
    return myEmployee[index];
}

Employee searchSalaryMin(Employee *myEmployee, int length){
    int index = 0;
    int minSalary = myEmployee[0].salary;
    for(int i=1; i<length; i++){
        if(myEmployee[i].salary < minSalary){
            minSalary = myEmployee[i].salary;
            index = i;
        }
    }
    return myEmployee[index];
}

void Display(Employee myEmployee){
```



```
cout << "Name: " << myEmployee.name << endl;
cout << "Birth day: " << myEmployee.birthDay.day << "/"
    << myEmployee.birthDay.month << "/"
    << myEmployee.birthDay.year << endl;
cout << "Role: " << myEmployee.role << endl;
cout << "Salary: " << myEmployee.salary << endl;
return;
}

void main(){
    clrscr();
    Employee *myEmployee, tmpEmployee;
    int length = 0;
    cout << "So luong nhan vien: ";
    cin >> length;

    // Khởi tạo danh sách nhân viên
    InitArray(myEmployee);

    // Nhân viên có lương cao nhất
    tmpEmployee = searchSalaryMax(myEmployee, length);
    Display(tmpEmployee);

    // Nhân viên có lương thấp nhất
    tmpEmployee = searchSalaryMin(myEmployee, length);
    Display(tmpEmployee);

    // Giải phóng vùng nhớ
    delete [] myEmployee;
    return;
}
```

3.4 MỘT SỐ KIỂU DỮ LIỆU TRỪU TƯỢNG

Nội dung phần này tập trung trình bày việc cài đặt một số cấu trúc dữ liệu trừu tượng, bao gồm:

- Ngăn xếp (stack)
- Hàng đợi (queue)
- Danh sách liên kết (list)

3.4.1 Ngăn xếp

Ngăn xếp (stack) là một kiểu danh sách cho phép thêm và bớt các phần tử ở một đầu danh sách, gọi là đỉnh của ngăn xếp. Ngăn xếp hoạt động theo nguyên lý: phần tử nào được đưa vào sau, sẽ được lấy ra trước.

Định nghĩa cấu trúc ngăn xếp

Vì ta chỉ cần quan tâm đến hai thuộc tính của ngăn xếp là:

- Danh sách các phần tử của ngăn xếp
- Vị trí đỉnh của ngăn xếp

nên ta có thể định nghĩa cấu trúc ngăn xếp như sau (các phần tử của ngăn xếp có kiểu int):

```
typedef SIZE 100;
typedef struct {
    int top;           // Vị trí của đỉnh
    int nodes[SIZE];  // Danh sách các phần tử
} Stack;
```

Tuy nhiên, định nghĩa này tồn tại một vấn đề, đó là kích thước (SIZE) của danh sách chứa các phần tử là tĩnh. Do đó:

- Nếu ta chọn SIZE lớn, nhưng khi gặp ứng dụng chỉ cần một số ít phần tử cho ngăn xếp thì rất tốn bộ nhớ.
- Nếu ta khai báo SIZE nhỏ, thì khi gặp bài toán cần ngăn xếp có nhiều phần tử, ta sẽ không thêm được các phần tử mới vào, chương trình sẽ có lỗi.

Để khắc phục hạn chế này, ta có thể sử dụng bộ nhớ động (mảng động thông qua con trỏ) để lưu danh sách các phần tử của ngăn xếp. Khi đó, định nghĩa cấu trúc ngăn xếp sẽ có dạng như sau:

```
typedef struct {
    int top;           // Vị trí của đỉnh
    int *nodes;        // Danh sách các phần tử
} Stack;
```

Ta sẽ sử dụng định nghĩa này trong các chương trình ứng dụng ngăn xếp.

Các thao tác trên ngăn xếp

Đối với các thao tác trên ngăn xếp, ta quan tâm đến hai thao tác cơ bản:

- Thêm một phần tử mới vào đỉnh ngăn xếp, gọi là **push**.
- Lấy ra một phần tử từ đỉnh ngăn xếp, gọi là **pop**.

Khi thêm một phần tử mới vào ngăn xếp, ta làm các bước như sau:

1. Số phần tử trong ngăn xếp cũ là (top+1). Do đó, ta cấp phát một vùng nhớ mới để lưu được (top+1+1) = (top+2) phần tử.
2. Sao chép (top+1) phần tử cũ sang vùng mới. Nếu danh sách ban đầu rỗng (top = -1) thì không cần thực hiện bước này.
3. Thêm phần tử mới vào cuối vùng nhớ mới
4. Giải phóng vùng nhớ của danh sách cũ

5. Cho danh sách nodes trở vào vùng nhớ mới.

Chương trình 3.4a cài đặt thủ tục thêm một phần tử mới vào ngăn xếp.

Chương trình 3.4a

```
void push(Stack *stack, int node){
    int *tmpNodes = new int[stack->top + 2]; // Cấp phát vùng nhớ mới
    stack->top++; // Tăng chỉ số của node đỉnh
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    tmpNodes[stack->top] = node; // Thêm node mới vào đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return;
}
```

Khi lấy ra một phần tử của ngăn xếp, ta làm các bước như sau:

- Kiểm tra xem ngăn xếp có rỗng ($top = -1$) hay không. Nếu không rỗng thì thực hiện các bước tiếp theo.
- Lấy phần tử ở đỉnh ngăn xếp ra
- Cấp phát một vùng nhớ mới có $(top+1) - 1 = top$ phần tử
- Sao chép top phần tử từ danh sách cũ sang vùng nhớ mới (trừ phần tử ở đỉnh).
- Giải phóng vùng nhớ cũ
- Cho con trỏ danh sách trở vào vùng nhớ mới.
- Trả về giá trị phần tử ở đỉnh đã lấy ra.

Chương trình 3.4b cài đặt thủ tục lấy một phần tử từ ngăn xếp.

Chương trình 3.4b

```
int pop(Stack *stack){
    if(stack->top < 0){ // Kiểm tra ngăn xếp rỗng
        cout << "Stack is empty!" << endl;
        return 0;
    }
    int result = stack->nodes[stack->top]; // Lưu giữ giá trị đỉnh
    int *tmpNodes = new int[stack->top]; // Cấp phát vùng nhớ mới
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    stack->top--; // Giảm chỉ số của node đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return result; // Trả về giá trị node đỉnh
}
```

```
}
```

Áp dụng

Ngăn xếp được sử dụng trong các ứng dụng thỏa mãn nguyên tắc: cái nào đặt vào trước sẽ được lấy ra sau. Chương trình 3.4c minh họa việc dùng ngăn xếp để đảo ngược một chuỗi ký tự được nhập vào từ bàn phím.

Chương trình 3.4c

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

typedef struct {
    int top;                // Vị trí node đỉnh
    int *nodes;             // Danh sách phần tử
} Stack;

/* Khai báo nguyên mẫu hàm */
void init(Stack *stack);
void push(Stack *stack, int node);
int pop(Stack *stack);
void release(Stack *stack);

void init(Stack *stack){
    stack = new Stack;     // Cấp phát vùng nhớ cho con trỏ
    stack->top = -1;        // Khởi tạo ngăn xếp rỗng
}

void push(Stack *stack, int node){
    int *tmpNodes = new int[stack->top + 2]; // Cấp phát vùng nhớ mới
    stack->top++;                // Tăng chỉ số của node đỉnh
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    tmpNodes[stack->top] = node; // Thêm node mới vào đỉnh
    delete [] stack->nodes;     // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes;    // Trỏ vào vùng nhớ mới
    return;
}

int pop(Stack *stack){
    if(stack->top < 0){        // Kiểm tra ngăn xếp rỗng
```

```

        cout << "Stack is empty!" << endl;
        return 0;
    }

    int result = stack->nodes[stack->top]; // Lưu giữ giá trị đỉnh
    int *tmpNodes = new int[stack->top]; // Cấp phát vùng nhớ mới
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    stack->top--; // Giảm chỉ số của node đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return result; // Trả về giá trị node đỉnh
}

void release(Stack *stack){
    delete [] stack->nodes; // Giải phóng vùng danh sách
    delete stack; // Giải phóng con trỏ
    return;
}

void main(){
    clrscr();
    Stack *stack;
    init(stack); // Khởi tạo ngăn xếp
    char strIn[250];
    // Nhập chuỗi kí tự từ bàn phím
    cout << "Nhap chuoi: ";
    cin >> strIn;
    for(int i=0; i<strlen(strIn); i++) // Đặt vào ngăn xếp
        push(stack, strIn[i]);
    while(stack->top > -1) // Lấy ra từ ngăn xếp
        cout << pop(stack);
    release(stack); // Giải phóng bộ nhớ
    return;
}

```

3.4.2 Hàng đợi

Hàng đợi (queue) cũng là một cấu trúc tuyến tính các phần tử. Trong đó, các phần tử luôn được thêm vào ở một đầu, gọi là đầu cuối hàng đợi, và việc lấy ra các phần tử luôn được thực hiện ở đầu còn lại, gọi là đầu mặt của hàng đợi. Hàng đợi hoạt động theo nguyên lý: phần tử nào được đưa vào trước, sẽ được lấy ra trước.

Định nghĩa cấu trúc hàng đợi

Hàng đợi có các thuộc tính:

- Một danh sách các phần tử có mặt trong hàng đợi.
- Chỉ số của phần tử đứng đầu của danh sách (front).
- Chỉ số phần tử cuối của danh sách (rear).

Nếu dùng cấu trúc tĩnh để định nghĩa, hàng đợi có cấu trúc như sau:

```
typedef SIZE 100;
typedef struct {
    int front, rear; // Vị trí của đỉnh đầu, đỉnh cuối
    int nodes[SIZE]; // Danh sách các phần tử
} Queue;
```

Nếu dùng bộ nhớ động để lưu giữ hàng đợi, thì phần tử front luôn là phần tử thứ 0 của danh sách. Và rear sẽ bằng độ dài danh sách trừ đi 1. Cấu trúc động của hàng đợi:

```
typedef struct {
    int front, rear; // Vị trí của đỉnh đầu, đỉnh cuối
    int *nodes; // Danh sách các phần tử
} Queue;
```

Thao tác trên hàng đợi

- Thêm một phần tử vào cuối hàng đợi
- Lấy một phần tử ở vị trí đầu của hàng đợi

Thao tác thêm một phần tử vào cuối hàng đợi với bộ nhớ động được thực hiện tương tự với ngăn xếp. Chương trình 3.5a cài đặt thủ tục thêm một phần tử vào cuối hàng đợi động.

Chương trình 3.5a

```
void insert(Queue *queue, int node){
    int *tmpNodes = new int[queue->rear + 2]; // Cấp phát vùng nhớ mới
    queue->rear++; // Tăng chỉ số của node đuôi
    if(queue->front == -1) // Nếu hàng đợi cũ rỗng
        queue->front = 0; // thì cập nhật front
    for(int i=0; i<queue->rear; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = queue->nodes[i];
    tmpNodes[queue->rear] = node; // Thêm node mới vào đuôi
    delete [] queue->nodes; // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes; // Trỏ vào vùng nhớ mới
    return;
}
```

Thao tác lấy ra một phần tử đầu của hàng đợi thực hiện theo các bước:

1. Kiểm tra tính rỗng (front = rear = -1) của hàng đợi. Nếu không rỗng mới thực hiện tiếp

2. Lấy phần tử nodes[0] ra.
3. Sao chép danh sách còn lại sang vùng nhớ mới
4. Giải phóng vùng nhớ cũ
5. Đưa danh sách trở vào vùng nhớ mới
6. Trả về giá trị phần tử lấy ra

Chương trình 3.5b cài đặt thủ tục lấy ra một phần tử của hàng đợi động.

Chương trình 3.5b

```
int remove(Queue *queue) {
    if((queue->front < 0) || (queue->rear < 0)) { // Kiểm tra hàng đợi rỗng
        cout << "Queue is empty!" << endl;
        return 0;
    }
    // Lưu giữ giá trị phần tử đầu
    int result = queue->nodes[queue->front];
    int *tmpNodes;
    if(queue->rear > 0) { // Nếu có hơn 1 phần tử
        tmpNodes = new int[queue->rear]; // Cấp phát vùng nhớ mới
        for(int i=0; i<queue->rear; i++) // Sao chép sang vùng nhớ mới
            tmpNodes[i] = queue->nodes[i];
    } else // Nếu chỉ có 1 phần tử
        queue->front --; // Hàng đợi thành rỗng
    queue->rear --; // Giảm chỉ số của node đuôi
    delete [] queue->nodes; // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes; // Trở vào vùng nhớ mới
    return result; // Trả về giá trị node đầu
}
```

Áp dụng

Hàng đợi được áp dụng trong các bài toán cần cơ chế quản lý cái nào vào trước sẽ được lấy ra trước. Chương trình 3.5c minh họa cơ chế quản lý tiến trình đơn giản nhất của hệ điều hành: các tiến trình được quản lý theo mã tiến trình, khi xuất hiện, tiến trình được đưa vào cuối của một hàng đợi. Khi nào CPU rảnh thì sẽ lấy tiến trình đầu hàng đợi ra để thực hiện.

Chương trình 3.5c

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

typedef struct {
    int front, rear; // Vị trí của đỉnh đầu, đỉnh cuối
}
```

```

        int *nodes;                                // Danh sách các phần tử
    } Queue;

/* Khai báo các nguyên mẫu hàm */
void init(Queue *queue);
void insert(Queue *queue, int node);
int remove(Queue *queue);
void traverse(Queue *queue);
void release(Queue *queue);

void init(Queue *queue){
    queue = new Queue;                            // Cấp phát bộ nhớ cho con trỏ
    queue->front = -1;                             // Khởi tạo danh sách rỗng
    queue->rear = -1;
    return;
}

void insert(Queue *queue, int node){
    int *tmpNodes = new int[queue->rear + 2];      // Cấp phát vùng nhớ mới
    queue->rear++;                                  // Tăng chỉ số của node đuôi
    if(queue->front == -1)                          // Nếu hàng đợi cũ rỗng
        queue->front = 0;                          // thì cập nhật front
    for(int i=0; i<queue->rear; i++)                // Sao chép sang vùng nhớ mới
        tmpNodes[i] = queue->nodes[i];
    tmpNodes[queue->rear] = node;                  // Thêm node mới vào đuôi
    delete [] queue->nodes;                        // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes;                       // Trỏ vào vùng nhớ mới
    return;
}

int remove(Queue *queue){
    if((queue->front < 0) || (queue->rear < 0)){    // Kiểm tra hàng đợi rỗng
        cout << "Queue is empty!" << endl;
        return 0;
    }
    // Lưu giữ giá trị phần tử đầu
    int result = queue->nodes[queue->front];
    int *tmpNodes;
    if(queue->rear > 0){                            // Nếu có hơn 1 phần tử
        tmpNodes = new int[queue->rear];           // Cấp phát vùng nhớ mới
        for(int i=0; i<queue->rear; i++)           // Sao chép sang vùng nhớ mới
            tmpNodes[i] = queue->nodes[i];
    }else
        // Nếu chỉ có 1 phần tử

```



```

        queue->front --;                // Hàng đợi thành rỗng
queue->rear --;                        // Giảm chỉ số của node cuối
delete [] queue->nodes;                // Giải phóng vùng nhớ cũ
queue->nodes = tmpNodes;                // Trỏ vào vùng nhớ mới
return result;                        // Trả về giá trị node đầu
}

void traverse(Queue *queue){
    if(queue->front < 0){                // Khi danh sách rỗng
        cout << "Danh sach rong!" << endl;
        return;
    }
    for(int i=queue->front; i<=queue->rear; i++)
        cout << queue->nodes[i] << " "; // Liệt kê các phần tử
    cout << endl;
    return;
}

void release(Queue *queue){
    if(queue->front > -1)                //Nếu danh sách không rỗng thì
        delete [] queue->nodes; //giải phóng vùng nhớ của danh sách
    delete queue;                      //Giải phóng vùng nhớ của con trỏ
}

void main(){
    clrscr();
    Queue *queue;
    init(queue);                        // Khởi tạo hàng đợi
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot tien trinh vao hang doi" << endl;
        cout << "2: Dua mot tien trinh trinh vao thuc hien" << endl;
        cout<<"3: Xem tat ca cac tien trinh trong hang doi" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1':                    // Thêm vào hàng đợi
                int maso;
                cout << "Ma so tien trinh vao hang doi: ";

```

```

        cin >> maso;
        insert(queue, maso);
        break;
    case '2':                // Lấy ra khỏi hàng đợi
        cout << "Tien trinh duoc thuc hien: " <<
            remove(queue) << endl;
        break;
    case '3':                // Duyệt hàng đợi
        cout<<"Cac tien trinh dang o trong hang doi la:"
            <<endl;
        traverse(queue);
        break;
    }while(function != '5');
    release(queue);          // Giải phóng hàng đợi
    return;
}

```

3.4.3 Danh sách liên kết

Danh sách liên kết là một kiểu dữ liệu bao gồm một dãy các phần tử có thứ tự, các phần tử có cùng cấu trúc dữ liệu, ngoại trừ node đầu tiên của danh sách là node lưu thông tin về danh sách. Có hai loại danh sách liên kết:

- Danh sách liên kết đơn: mỗi node có một con trỏ trỏ đến node tiếp theo trong danh sách
- Danh sách liên kết kép: mỗi node có hai con trỏ, một trỏ vào node trước, một trỏ vào node tiếp theo trong danh sách.

Trong phần này sẽ trình bày danh sách liên kết đơn. Danh sách liên kết kép được coi như là một bài tập mở rộng từ danh sách liên kết đơn.

Định nghĩa danh sách đơn

Mỗi node của danh sách đơn chứa dữ liệu của nó, đồng thời trỏ đến node tiếp theo trong danh sách, cấu trúc một node như sau:

```

struct simple{
    Employee employee;    // Dữ liệu của node có kiểu Employee
    struct simple *next;  // Trỏ đến node kế tiếp
};
typedef struct simple SimpleNode;

```

Node đầu của danh sách đơn có cấu trúc riêng, nó không chứa dữ liệu như node thường mà chứa các thông tin:

- Số lượng node trong danh sách (không kể bản thân nó – node đầu)
- Con trỏ đến node đầu tiên của danh sách
- Con trỏ đến node cuối cùng của danh sách

Do vậy, cấu trúc node đầu của danh sách đơn là:

```
typedef struct{
    int nodeNumber;          // Số lượng các node
    SimpleNode *front, *rear; // Trỏ đến node đầu và cuối danh sách
} SimpleHeader;
```

Các thao tác trên danh sách liên kết đơn

Các thao tác cơ bản trên danh sách đơn bao gồm:

- Chèn thêm một node vào vị trí thứ n trong danh sách
- Loại ra một node ở vị trí thứ n trong danh sách

Việc chèn thêm một node vào vị trí thứ n trong danh sách được thực hiện theo các bước:

1. Nếu $n \leq 0$, chèn vào đầu. Nếu $n > \text{số phần tử của danh sách}$, chèn vào cuối. Trường hợp còn lại, chèn vào giữa.
2. Tìm node thứ n: giữ vết của hai node thứ n-1 và thứ n.
3. Tạo một node mới: cho node thứ n-1 trỏ tiếp vào node mới và node mới trỏ tiếp vào node thứ n.

Chương trình 3.6a cài đặt thủ tục chèn một node vào vị trí thứ n của danh sách.

Chương trình 3.6a

```
void insert(SimpleHeader *list, int position, int value){
    SimpleNode *newNode = new SimpleNode;
    newNode->value = value;
    if(position <= 0){                // Chèn vào đầu ds
        newNode->next = list->front; // Chèn vào trước node đầu
        list->front = newNode;        // Cập nhật lại node đầu ds
        if(list->nodeNumber == 0)     // Nếu ds ban đầu rỗng thì
            list->rear = newNode;     // node đầu trùng với node cuối
    }else if(position >= list->nodeNumber){ // Chèn vào cuối ds
        list->rear->next = newNode;   // Chèn vào sau node cuối
        list->rear = newNode;        // Cập nhật lại node cuối ds
        if(list->nodeNumber == 0)     // Nếu ds ban đầu rỗng thì
            list->front = newNode;    // node đầu trùng node cuối
    }else{                            // Chèn vào giữa ds
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){      // tìm node n-1 và n
            prev = curr;
            curr = curr->next;
            index++;
        }
        newNode->next = curr;         // chèn vào trước node n
        prev->next = newNode;         // và chèn vào sau node n-1
    }
}
```

```
list->nodeNumber++;           // Cập nhật số lượng node
return;
}
```

Việc xóa một node ở vị trí thứ n trong danh sách được thực hiện theo các bước:

1. Nếu $n < 0$ hoặc $n > \text{số phần tử của danh sách}$, không xóa node nào.
2. Tìm node thứ n: giữ vết của ba node thứ n-1, thứ n và thứ n+1.
3. Cho node thứ n-1 trở tiếp vào node thứ n+1, xóa con trỏ của node thứ n.
4. Trả về node thứ n.

Chương trình 3.6b cài đặt thủ tục xóa một node ở vị trí thứ n của danh sách.

Chương trình 3.6b

```
SimpleNode* remove(SimpleHeader *list, int position){
    if((position < 0) || (position >= list->nodeNumber))
        return NULL;           // Không xóa node nào cả
    SimpleNode* result;
    if(position == 0){           // Xóa node đầu
        result = list->front;     // Giữ node cần xóa
        list->front = list->front->next; // Cập nhật node đầu
        if(list->nodeNumber == 1) // Nếu ds chỉ có 1 node thì
            list->rear = list->front; // Cập nhật node cuối ds
        }else if(position == list->nodeNumber - 1){
            result = list->rear;   // Giữ node cần xóa
            SimpleNode *curr = list->front;
            while(curr->next != list->rear)
                curr = curr->next; // Tìm node trước của node cuối
            curr->next = NULL;     // Xóa node rear hiện tại
            list->rear = curr;     // Cập nhật node cuối ds
        }else{
            SimpleNode *prev = list->front, *curr = list->front;
            int index = 0;
            while(index < position){ // Tìm node n-1 và n
                prev = curr;
                curr = curr->next;
                index++;
            }
            result = curr;         // Giữ node cần xóa
            prev->next = curr->next; // Cho node n-1 trở đến node n+1
        }
        list->nodeNumber --;      // Cập nhật số lượng node
        return result;           // Trả về node cần xóa
    }
```

```
}
```

Áp dụng

Chương trình 3.6c minh họa việc dùng danh sách liên kết đơn để quản lý nhân viên văn phòng với các thông tin rất đơn giản: tên, tuổi và tiền lương của mỗi nhân viên.

Chương trình 3.6c

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

typedef struct{
    char name[25];           // Tên nhân viên
    int age;                 // Tuổi nhân viên
    float salary;            // Lương nhân viên
} Employee;

struct simple{
    Employee employee;       // Dữ liệu của node
    struct simple *next;     // Trỏ đến node kế tiếp
};

typedef struct simple SimpleNode;

typedef struct{
    int nodeNumber;          // Số lượng các node
    SimpleNode *front, *rear; // Trỏ đến node đầu và cuối ds
} SimpleHeader;

/* Khai báo các nguyên mẫu hàm */
void init(SimpleHeader *list);
void insert(SimpleHeader *list, int position, Employee employee);
SimpleNode* remove(SimpleHeader *list);
void traverse(SimpleHeader *list);
void release(SimpleHeader *list);

void init(SimpleHeader *list){
    list = new list;        // Cấp phát bộ nhớ cho con trỏ
    list->front = NULL;      // Khởi tạo danh sách rỗng
    list->rear = NULL;
```

```

    list->nodeNumber = 0;
    return;
}

void insert(SimpleHeader *list, int position, Employee employee){
    SimpleNode *newNode = new SimpleNode;
    newNode->employee = employee;
    if(position <= 0){
        // Chèn vào đầu ds
        newNode->next = list->front; // Chèn vào trước node đầu
        list->front = newNode;       // Cập nhật lại node đầu ds
        if(list->nodeNumber == 0)    // Nếu ds ban đầu rỗng thì
            list->rear = newNode;   // node đầu trùng với node đầu
    }else if(position >= list->nodeNumber){ // Chèn vào cuối ds
        list->rear->next = newNode; // Chèn vào sau node cuối
        list->rear = newNode;       // Cập nhật lại node cuối ds
        if(list->nodeNumber == 0) // Nếu ds ban đầu rỗng thì
            list->front = newNode; // node đầu trùng node cuối
    }else{
        // Chèn vào giữa ds
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){ // tìm node n-1 và n
            prev = curr;
            curr = curr->next;
            index++;
        }
        newNode->next = curr; // chèn vào trước node n
        prev->next = newNode; // và chèn vào sau node n-1
    }
    list->nodeNumber++; // Cập nhật số lượng node
    return;
}

SimpleNode* remove(SimpleHeader *list, int position){
    if((position < 0) || (position >= list->nodeNumber))
        return NULL; // Không xóa node nào cả
    SimpleNode* result;
    if(position == 0){
        // Xóa node đầu
        result = list->front; // Giữ node cần xóa
        list->front = list->front->next; // Cập nhật node đầu
        if(list->nodeNumber == 1) // Nếu ds chỉ có 1 node thì
            list->rear = list->front; // Cập nhật node cuối ds
    }else if(position == list->nodeNumber - 1){
        result = list->rear; // Giữ node cần xóa
    }
}

```

```

        SimpleNode *curr = list->front;
        while(curr->next != list->rear)
            curr = curr->next; // Tìm node trước của node cuối
        curr->next = NULL;      // Xoá node rear hiện tại
        list->rear = curr;      // Cập nhật node cuối ds
    }else{
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){ // Tìm node n-1 và n
            prev = curr;
            curr = curr->next;
            index++;
        }
        result = curr;          // Giữ node cần xoá
        prev->next = curr->next; // Cho node n-1 trở đến node n+1
    }
    list->nodeNumber--;         // Cập nhật số lượng node
    return result;             // Trả về node cần xoá
}

void traverse(SimpleHeader *list){
    if(list->nodeNumber <= 0){ // Khi danh sách rỗng
        cout << "Danh sach rong!" << endl;
        return;
    }
    SimpleNode *curr = list->front;
    while(curr != NULL){
        cout << curr->employee.name << " "
            << curr->employee.age << " "
            << curr->employee.salary << endl; // Liệt kê các phần tử
        curr = curr->next;
    }
    return;
}

void release(SimpleHeader *list){
    SimpleNode* curr = remove(list, 0);
    while(curr != NULL){
        delete curr;          //Giải phóng vùng nhớ của node
        curr = remove(list, 0);
    }
    delete list;             //Giải phóng vùng nhớ của con trỏ
}

```



```

void main() {
    clrscr();
    SimpleHeader *list;
    init(list);          // Khởi tạo ds
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot nhan vien" << endl;
        cout << "2: Xoa mot nhan vien" << endl;
        cout << "3: Xem tat ca cac nhan vien trong phong" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1':          // Thêm vào ds
                int position;
                Employee employee;
                cout << "Vi tri can chen: ";
                cin >> position;
                cout << "Ten nhan vien: ";
                cin >> employee.name;
                cout << "Tuoi nhan vien: ";
                cin >> employee.age;
                cout << "Luong nhan vien: ";
                cin >> employee.salary;
                insert(list, position, employee);
                break;
            case '2':          // Lấy ra khỏi ds
                int position;
                cout << "Vi tri can xoa: ";
                cin >> position;
                SimpleNode* result = remove(list, position);
                if(result != NULL){
                    cout << "Nhan vien bi loai: " << endl;
                    cout << "Ten: " << result->employee.name
                        << endl;
                    cout << "Tuoi: " << result->employee.age
                        << endl;
                    cout << "Luong: "
                        << result->employee.salary << endl;
                }
            }
        }
    } while(function != 5);
}

```



```
        }
        break;
    case '3':          // Duyệt ds
        cout<<"Cac nhan vien cua phong:"<<endl;
        traverse(list);
        break;
    }while(function != '5');
    release(list);      // Giải phóng ds
    return;
}
```

TỔNG KẾT CHƯƠNG 3

Nội dung chương 3 đã trình bày các vấn đề liên quan đến các kiểu dữ liệu có cấu trúc trong C++:

- Khai báo cấu trúc thông qua từ khoá **struct**
- Tự định nghĩa kiểu dữ liệu cấu trúc bằng từ khoá **typedef**.
- Khai báo một biến có kiểu dữ liệu cấu trúc
- Khai báo các cấu trúc lồng nhau.
- Truy nhập đến các thuộc tính của cấu trúc
- Khai báo con trỏ cấu trúc, cấp phát và giải phóng bộ nhớ động của con trỏ cấu trúc. Truy nhập đến các thuộc tính của con trỏ cấu trúc.
- Khai báo và sử dụng mảng cấu trúc
- Khai báo mảng cấu trúc bằng con trỏ cấu trúc. Cấp phát và giải phóng vùng nhớ của mảng động các cấu trúc.
- Cài đặt một số cấu trúc đặc biệt:
 - Ngăn xếp
 - Hàng đợi
 - Danh sách liên kết

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 3

1. Để định nghĩa một cấu trúc sinh viên có tên là Sinhvien, gồm có tên và tuổi sinh viên. Định nghĩa nào sau đây là đúng:

- a.

```
struct Sinhvien{
    char name[20];
    int age;
};
```
- b.

```
struct {
    char name[20];
    int age;
} Sinhvien;
```

```
c.    typedef struct Sinhvien{
        char name[20];
        int age;
    };
```

2. Một cấu trúc được định nghĩa như sau:

```
struct Employee{
    char name[20];
    int age;
};
```

Khi đó, cách khai báo biến nào sau đây là đúng:

- a. struct Employee myEmployee;
- b. struct employee myEmployee;
- c. Employee myEmployee;
- d. employee myEmployee;

3. Một cấu trúc được định nghĩa như sau:

```
typedef struct employee{
    char name[20];
    int age;
} Employee;
```

Khi đó, cách khai báo biến nào sau đây là đúng:

- a. Employee myEmployee;
- b. employee myEmployee;
- c. struct Employee myEmployee;
- d. struct employee myEmployee;

4. Với cấu trúc được định nghĩa như trong bài 3. Khi đó, cách khởi tạo biến nào sau đây là đúng:

- a. Employee myEmployee = {'A', 27};
- b. Employee myEmployee = {"A", 27};
- c. Employee myEmployee = ('A', 27);
- d. Employee myEmployee = ("A", 27);

5. Với cấu trúc được định nghĩa như trong bài 3. Khi đó, các cách cấp phát bộ nhớ cho biến con trỏ nào sau đây là đúng:

- a. Employee *myEmployee = new Employee;
- b. Employee *myEmployee = new Employee();
- c. Employee *myEmployee = new Employee(10);
- d. Employee *myEmployee = new Employee[10];

6. Định nghĩa một cấu trúc về môn học của một học sinh có tên Subject, bao gồm các thông tin:

- Tên môn học, kiểu char[];
 - Điểm tổng kết môn học, kiểu float;
7. Định nghĩa cấu trúc về học sinh tên là Student bao gồm các thông tin sau:
- Tên học sinh, kiểu char[];
 - Tuổi học sinh, kiểu int;
 - Lớp học sinh, kiểu char[];
 - Danh sách điểm các môn học của học sinh, kiểu là một mảng các cấu trúc Subject đã được định nghĩa trong bài tập 6.
 - Xếp loại học lực, kiểu char[];
8. Khai báo một biến có cấu trúc là Student đã định nghĩa trong bài 7. Sau đó, thực hiện tính điểm trung bình của tất cả các môn học của học sinh đó, và viết một thủ tục xếp loại học sinh dựa vào điểm trung bình các môn học:
- Nếu điểm tb nhỏ hơn 5.0, xếp loại kém
 - Nếu điểm tb từ 5.0 đến dưới 6.5, xếp loại trung bình.
 - Nếu điểm tb từ 6.5 đến dưới 8.0, xếp loại khá
 - Nếu điểm tb từ 8.0 trở lên, xếp loại giỏi.
9. Viết một chương trình quản lý các học sinh của một lớp, là một dãy các cấu trúc có kiểu Stupid định nghĩa trong bài 7. Sử dụng thủ tục đã cài đặt trong bài 8 để thực hiện các thao tác sau:
- Khởi tạo danh sách và điểm của các học sinh trong lớp.
 - Tính điểm trung bình và xếp loại cho tất cả các học sinh.
 - Tìm tất cả các học sinh theo một loại nhất định
10. Sử dụng cấu trúc ngăn xếp đã định nghĩa trong bài để đổi một số từ kiểu thập phân sang kiểu nhị phân: Chi số nguyên cho 2, mãi cho đến khi thương < 2 , lưu các số dư vào ngăn xếp. Sau đó, đọc các giá trị dư từ ngăn xếp ra, ta sẽ thu được chuỗi nhị phân tương ứng.
11. Mở rộng cấu trúc hàng đợi đã định nghĩa trong bài để trở thành hàng đợi có độ ưu tiên:
- Cho mỗi node thêm một thuộc tính là độ ưu tiên của node đó
 - Khi thêm một node vào hàng đợi, thay vì thêm vào cuối hàng đợi như thông thường, ta tìm vị trí có độ ưu tiên phù hợp để chèn node vào, sao cho dãy các node trong hàng đợi là một danh sách có độ ưu tiên của các node là giảm dần.
 - Việc lấy ra là không thay đổi: lấy ra phần tử ở đầu hàng đợi, chính là phần tử có độ ưu tiên cao nhất.
12. Áp dụng hàng đợi có độ ưu tiên trong bài 11 để xây dựng chương trình quản lý tiến trình có độ ưu tiên của hệ điều hành, mở rộng ứng dụng trong bài ngăn xếp.
13. Mở rộng cấu trúc danh sách liên kết đơn trong bài thành danh sách liên kết kép:
- Mỗi node có thêm một con trỏ prev để trỏ đến node trước nó
 - Đối với node header, cũng cần 2 con trỏ: trỏ đến node đầu tiên và node cuối cùng của danh sách

- Riêng với node đầu tiên (front) của danh sách, con trỏ prev của nó sẽ trỏ đến NULL. Giống như con trỏ next của node rear.
14. Cài đặt lại hai thao tác thêm vào một node và xóa một node ở một vị trí xác định trong một cấu trúc danh sách liên kết kép định nghĩa trong bài 13.
 15. Áp dụng các định nghĩa và thao tác trong các bài 13 và 14. Cài đặt lại chương trình quản lý nhân viên ở chương trình 3.6c bằng danh sách liên kết kép.



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhkx@pit.edu.vn

CHƯƠNG 4

VÀO RA TRÊN TỆP

Nội dung chương này tập trung trình bày các vấn đề liên quan đến các thao tác trên tệp dữ liệu trong ngôn ngữ C++:

- Khái niệm tệp, tệp văn bản và tệp nhị phân
- Các thao tác vào ra trên tệp
- Phương thức truy nhập tệp trực tiếp

4.1 KHÁI NIỆM TỆP

4.1.1 Tệp dữ liệu

Trong C++, khi thao tác với một tệp dữ liệu, cần thực hiện tuần tự theo các bước như sau:

1. Mở tệp tin
2. Thực hiện các thao tác đọc, ghi trên tệp tin đang mở
3. Đóng tệp tin

Để thực hiện các thao tác liên quan đến tệp dữ liệu, C++ cung cấp một thư viện `<fstream.h>` chứa các lớp và các hàm phục vụ cho các thao tác này. Do vậy, trong các chương trình làm việc với tệp tin, ta cần khai báo chỉ thị dùng thư viện này ngay từ đầu chương trình:

```
#include<fstream.h>
```

Khai báo biến tệp

Trong C++, khi khai báo một biến tệp, đồng thời ta sẽ mở tệp tương ứng theo cú pháp tổng quát bằng cách dùng kiểu **`fstream`** như sau:

```
fstream <Tên biến tệp>(<Tên tệp>, <Chế độ mở tệp>);
```

Trong đó:

- **Tên biến tệp:** có tính chất như một tên biến thông thường, nó sẽ được dùng để thực hiện các thao tác với tệp gắn với nó. Tên biến tệp cũng phải tuân thủ theo quy tắc đặt tên biến trong C++.
- **Tên tệp:** là tên tệp dữ liệu mà ta cần thao tác trên nó.
- **Chế độ mở tệp:** là các hằng kiểu bit đã được định nghĩa sẵn bởi C++. Nó chỉ ra rằng ta đang mở tệp tin ở chế độ nào: đọc hoặc ghi, hoặc cả đọc lẫn ghi.

Ví dụ, khai báo:

```
fstream myFile("abc.txt", ios::in);
```

là khai báo một biến tệp, có tên là `myFile`, dùng để mở tệp tin có tên là `abc.txt` và tệp tin này được mở ở chế độ để đọc dữ liệu (bit chỉ thị `ios::in`).

Lưu ý:

- Tên tệp tin có dạng một chuỗi kí tự, nếu khai báo tên tệp có đường dẫn thư mục “\” thì mỗi dấu “\” phải được viết thành “\\” để tránh bị nhầm lẫn với các kí tự đặc biệt trong C như “\n”, “\d”...

Ví dụ, muốn mở một tệp tên là abc.txt trong thư mục myDir để đọc, ta phải khai báo như sau:

```
fstream myFile("myDir\\abc.txt", ios::in);
```

Các chế độ mở tệp tin

Các chế độ mở tệp tin được định nghĩa bởi các bit chỉ thị:

- **ios::in:** Mở một tệp tin để đọc.
- **ios::out:** Mở một tệp tin có sẵn để ghi.
- **ios::app:** Mở một tệp tin có sẵn để thêm dữ liệu vào cuối tệp.
- **ios::ate:** Mở tệp tin và đặt con trỏ tệp tin vào cuối tệp.
- **ios::trunc:** Nếu tệp tin đã có sẵn thì dữ liệu của nó sẽ bị mất.
- **ios::nocreate:** Mở một tệp tin, tệp tin này bắt buộc phải tồn tại.
- **ios::noreplace:** Chỉ mở tệp tin khi tệp tin chưa tồn tại.
- **ios::binary:** Mở một tệp tin ở chế độ nhị phân.
- **ios::text:** Mở một tệp tin ở chế độ văn bản.

Lưu ý:

- Khi muốn mở một tệp tin đồng thời ở nhiều chế độ khác nhau, ta kết hợp các bit chỉ thị tương ứng bằng phép toán hợp bit “|”.

Ví dụ, muốn mở một tệp tin abc.txt để đọc (ios::in) đồng thời với để ghi (ios::out) dưới chế độ văn bản (ios::text), ta khai báo như sau:

```
fstream myFile("abc.txt", ios::in|ios::out|ios::text);
```

4.1.2 Tệp văn bản

Để mở một tệp tin dưới chế độ văn bản, ta dùng cú pháp sau:

```
fstream <Tên biến tệp>(<Tên tệp>, ios::text);
```

Khi đó, các thao tác đọc, ghi trên biến tệp được thực hiện theo đơn vị là các từ, được phân cách bởi dấu trống (space bar) hoặc dấu xuống dòng (enter).

Ví dụ, muốn mở tệp tin baitho.txt dưới chế độ văn bản, ta khai báo như sau:

```
fstream myBaiTho("baitho.txt", ios::text);
```

4.1.3 Tệp nhị phân

Để mở một tệp tin dưới chế độ nhị phân, ta dùng cú pháp sau:

```
fstream <Tên biến tệp>(<Tên tệp>, ios::binary);
```

Khi đó, các thao tác đọc, ghi trên biến tệp được thực hiện theo đơn vị byte theo kích thước các bản ghi (cấu trúc) được ghi trong tệp.

Ví dụ, muốn mở tệp tin baitho.txt dưới chế độ nhị phân, ta khai báo như sau:

```
fstream myBaiTho("baitho.txt", ios::binary);
```


4.2 VÀO RA TRÊN TỆP

4.2.1 Vào ra tệp văn bản bằng “>>” và “<<”

Ghi tệp văn bản bằng “<<”

Các bước thực hiện để ghi dữ liệu vào một tệp tin như sau:

1. Mở tệp tin theo chế độ để ghi bằng đối tượng ofstream (mở tệp tin chỉ để ghi):

```
ofstream <Tên biến tệp>(<Tên tệp tin>, ios::out);
```

2. Ghi dữ liệu vào tệp bằng thao tác “<<”:

```
<Tên biến tệp> << <Dữ liệu>;
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Chương trình 4.1 minh họa việc ghi dữ liệu vào tệp tin:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ ghi vào tệp các kí tự do người dùng gõ vào từ bàn phím, mỗi kí tự được phân cách nhau bởi dấu trống (space bar).
- Chương trình dừng lại khi người dùng nhập kí tự ‘e’. Và tệp tin được kết thúc bằng một dấu xuống dòng “endl”.

Chương trình 4.1

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

const int length = 25;           // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], input;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName;    // Nhập tên tệp tin

    /* Mở tệp tin */
    ofstream fileOut(fileName, ios::out); // Khai báo và mở tệp tin
    if(!fileOut){                       // Không mở được tệp
        cout << "Khong the tao duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Ghi dữ liệu vào tệp tin */
```

```
do{
    cin >> input;                // Đọc kí tự từ bàn phím
    fileOut << input << ' ';    // Ghi kí tự vào tệp tin
}while((input != 'e') && (fileOut));
fileOut << endl;                // Xuống dòng cuối tệp tin

/* Đóng tệp tin */
fileOut.close();                // Đóng tệp tin
return;
}
```

Đọc dữ liệu từ tệp văn bản bằng ">>"

Các bước thực hiện để đọc dữ liệu từ một tệp tin như sau:

1. Mở tệp tin theo chế độ để đọc bằng đối tượng ifstream (mở tệp tin chỉ để đọc):

```
ifstream <Tên biến tệp>(<Tên tệp tin>, ios::in);
```

2. Đọc dữ liệu từ tệp bằng thao tác ">>":

```
<Tên biến tệp> >> <Biến dữ liệu>;
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Chương trình 4.2 minh họa việc đọc dữ liệu từ tệp tin vừa sử dụng trong chương trình 4.1 ra màn hình:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ đọc các kí tự trong tệp và hiển thị ra màn hình, mỗi kí tự được phân cách nhau bởi dấu trống (space bar).
- Chương trình dừng lại khi kết thúc tệp tin.

Chương trình 4.2

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

const int length = 25;                // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], output;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName;    // Nhập tên tệp tin

    /* Mở tệp tin */
```



```
ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin
if(!fileIn){                        // Không mở được tệp
    cout << "Khong the mo duoc tep tin " << fileName << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra màn hình */
while(fileIn){
    fileIn >> output;                // Đọc kí tự từ tệp tin
    cout << output;                  // Ghi kí tự ra màn hình
}
cout << endl;                        // Xuống dòng trên màn hình

/* Đóng tệp tin */
fileIn.close();                      // Đóng tệp tin
return;
}
```

Chương trình 4.3 minh họa việc copy toàn bộ nội dung của một tệp tin sang một tệp tin mới:

- Tên tệp tin nguồn và tệp tin đích được nhập từ bàn phím bởi người dùng.
- Tệp tin nguồn được mở ở chế độ đọc.
- Tệp tin đích được mở ở chế độ ghi.
- Đọc từng kí tự từ tệp tin nguồn và ghi ngay vào tệp tin đích.
- Đóng các tệp tin khi kết thúc.

Chương trình 4.3

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

const int length = 25;                // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char sourceFile[length], targetFile[length], data;
    cout << "Ten tep tin nguon: ";
    cin >> setw(length) >> sourceFile; // Nhập tên tệp tin nguồn

    cout << "Ten tep tin dich: ";
    cin >> setw(length) >> targetFile; // Nhập tên tệp tin đích
```

```

/* Mở tệp tin nguồn */
ifstream fileIn(sourceFile, ios::in); // Khai báo và mở tệp nguồn
if(!fileIn){                          // Không mở được tệp nguồn
    cout << "Khong the mo duoc tep tin nguon "
         << sourceFile << endl;
    exit(1);
}

/* Mở tệp tin đích */
ofstream fileOut(targetFile, ios::out); // Khai báo và mở tệp đích
if(!fileOut){                          // Không mở được tệp đích
    cout << "Khong the tao duoc tep tin dich "
         << targetFile << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra tệp đích */
while(fileIn){
    fileIn >> data;                    // Đọc kí tự từ tệp nguồn
    fileOut << data;                   // Ghi kí tự ra tệp đích
}

/* Đóng các tệp tin */
fileIn.close();                       // Đóng tệp tin nguồn
fileOut.close();                      // Đóng tệp tin đích
return;
}

```

Lưu ý:

- Tên biến tệp, sau khi dùng xong với một tệp xác định, có thể sử dụng để mở một tệp khác, với một chế độ mở tệp khác bằng phép toán **open()** của biến tệp.

<Tên biến tệp>.open(<Tên tệp mới>, <chế độ mở mới>);

Ví dụ, đoạn chương trình:

```

ofstream myFile("abc.txt", ios::out);
...      // ghi vào file abc.txt
myFile.close();
myFile.open("xyz.txt", ios::out|ios::app);
...      // Thêm vào cuối file xyz.txt
myFile.close();

```

sẽ dùng biến tệp myFile (có kiểu ofstream) hai lần: một lần là dùng với tệp tin abc.txt ở chế độ mở để ghi từ đầu. Một lần khác là với tệp tin xyz.txt ở chế độ mở để ghi thêm vào cuối.

4.2.2 Vào ra tệp nhị phân bằng read và write

Ghi vào tệp nhị phân bằng write

Các bước thực hiện để ghi dữ liệu vào một tệp nhị phân như sau:

1. Mở tệp tin theo chế độ để ghi nhị phân bằng đối tượng fstream:

```
fstream <Tên biến tệp>(<Tên tệp tin>, ios::out|ios::binary);
```

2. Ghi dữ liệu vào tệp bằng thao tác “**write()**”:

```
<Tên biến tệp>.write(char* <Dữ liệu>,  
int <Kích thước dữ liệu>);
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Trong đó, thao tác **write** nhận hai tham số đầu vào như sau:

- Tham số thứ nhất là con trỏ kiểu char trỏ đến vùng dữ liệu cần ghi vào tệp. Vì con trỏ bắt buộc có kiểu char nên khi muốn ghi dữ liệu có kiểu khác vào tệp, ta dùng hàm chuyển kiểu:

```
reinterpret_cast<char*>(<Dữ liệu>);
```

- Tham số thứ hai là kích cỡ dữ liệu được ghi vào tệp. Kích cỡ này được tính theo byte, nên thông thường ta dùng toán tử:

```
sizeof(<Kiểu dữ liệu>);
```

Lưu ý:

- Khi muốn đọc, ghi các dữ liệu có cấu trúc (struct) vào tệp thì ta phải dùng ở chế độ đọc/ghi tệp nhị phân mà không thể dùng chế độ đọc/ghi ở chế độ văn bản.
- Khi đọc/ghi dữ liệu có kiểu cấu trúc, để toán tử sizeof() thực hiện chính xác thì các thành viên của cấu trúc không được là kiểu con trỏ. Vì toán tử sizeof() đối với con trỏ chỉ cho kích cỡ của con trỏ mà không cho kích cỡ thật của vùng dữ liệu mà con trỏ trỏ tới.

Chương trình 4.4 minh họa việc ghi dữ liệu vào tệp tin nhị phân, dữ liệu là kiểu cấu trúc:

- Tên tệp tin và số lượng bản ghi được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ ghi vào tệp các bản ghi có cấu trúc do người dùng gõ vào từ bàn phím.

Chương trình 4.4

```
#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<type.h>

const int length = 25; // Độ dài tối đa tên tệp tin

typedef struct {
    int day; // Ngày
    int month; // Tháng
```

```

        int year;                                // Năm
    } Date;

typedef struct {
    char name[20];                                // Tên nhân viên
    Date birthDay;                                // Ngày sinh của nhân viên
    char role[20];                                // Chức vụ của nhân viên
    float salary;                                // Lương của nhân viên
} Employee;

void main(){
    clrscr();
    char fileName[length];                        // Tên tệp tin
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName;            // Nhập tên tệp tin

    int recordNumber;                             // Số lượng bản ghi
    cout << "So luong ban ghi: ";
    cin >> recordNumber;                         // Nhập số lượng bản ghi

    /* Mở tệp tin */
    // Khai báo và mở tệp tin
    fstream fileOut(fileName, ios::out|ios::binary);
    if(!fileOut){                                // Không mở được tệp
        cout << "Khong the tao duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Ghi dữ liệu vào tệp tin */
    Employee myEmployee;
    for(int i=0; i<recordNumber; i++){
        cout << "Ban ghi thu " << i+1 << endl;
        cout << "Name: ";
        cin >> myEmployee.name;                  // Nhập tên nhân viên
        cout << "Day of birth: ";
        cin >> myEmployee.birthDay.day;          // Nhập ngày sinh
        cout << "Month of birth: ";
        cin >> myEmployee.birthDay.month;        // Nhập tháng sinh
        cout << "Year of birth: ";
        cin >> myEmployee.birthDay.year;        // Nhập năm sinh
        cout << "Role: ";
        cin >> myEmployee.role;                  // Nhập chức vụ
        cout << "Salary: ";
    }
}

```

```
        cin >> myEmployee.salary;        // Nhập tiền lương

        // Ghi dữ liệu vào tệp
        fileOut.write(reinterpret_cast<char *>(&myEmployee),
                        sizeof(Employee));

    }

    /* Đóng tệp tin */
    fileOut.close();                        // Đóng tệp tin
    return;
}
```

Đọc dữ liệu từ tệp nhị phân bằng read

Các bước thực hiện để đọc dữ liệu từ một tệp tin nhị phân như sau:

1. Mở tệp tin theo chế độ để đọc nhị phân bằng đối tượng fstream (mở tệp tin chỉ để ghi):

```
fstream <Tên biến tệp>(<Tên tệp tin>, ios::in|ios::binary);
```

2. Đọc dữ liệu từ tệp bằng thao tác “read()”:

```
<Tên biến tệp>.read(char* <Dữ liệu ra>,
                    int <Kích thước dữ liệu>);
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Chương trình 4.5 minh họa việc đọc dữ liệu từ tệp tin vào biến có cấu trúc:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ đọc các cấu trúc nhân viên trong tệp và hiển thị ra màn hình.
- Chương trình dừng lại khi kết thúc tệp tin.

Chương trình 4.5

```
#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<type.h>

const int length = 25;                // Độ dài tối đa tên tệp tin

typedef struct {
    int day;                          // Ngày
    int month;                        // Tháng
    int year;                         // Năm
} Date;
```

```
typedef struct {
    char name[20];                // Tên nhân viên
    Date birthDay;                // Ngày sinh của nhân viên
    char role[20];                // Chức vụ của nhân viên
    float salary;                 // Lương của nhân viên
} Employee;

void main(){
    clrscr();
    char fileName[length];       // Tên tệp tin
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin

    /* Mở tệp tin */
    // Khai báo và mở tệp tin
    fstream fileIn(fileName, ios::in|ios::binary);
    if(!fileIn){                 // Không mở được tệp
        cout << "Khong the mo duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Đọc dữ liệu từ tệp tin ra màn hình */
    Employee myEmployee;
    while(fileIn){
        fileIn.read(reinterpret_cast<char *>(&myEmployee),
                    sizeof(Employee)); // Đọc kí tự từ tệp tin
        cout << myEmployee.name << " "
             << myEmployee.birthDay.day << "/"
             << myEmployee.birthDay.month << "/"
             << myEmployee.birthDay.year << " "
             << myEmployee.role << " "
             << myEmployee.salary << endl; // Ghi kí tự ra màn hình
    }

    /* Đóng tệp tin */
    fileIn.close();              // Đóng tệp tin
    return;
}
```

4.3 TRUY NHẬP TỆP TRỰC TIẾP

4.3.1 Con trỏ tệp tin

Con trỏ tệp tin có vai trò như một đầu đọc trỏ vào một vị trí xác định của tệp và thao tác truy nhập tệp diễn ra tuần tự:

- Tại mỗi thời điểm, con trỏ tệp tin xác định một vị trí trên tệp mà tại đó, thao tác truy nhập tệp (đọc/ghi) được thực hiện.
- Sau thao tác truy nhập, con trỏ tệp tự động chuyển đến vị trí tiếp theo dựa vào kích thước đơn vị dữ liệu được truy nhập.

Cách truy nhập tệp tuần tự có nhược điểm là bao giờ cũng phải bắt đầu từ đầu tệp tin, đi tuần tự cho đến vị trí cần truy nhập. Khi tệp tin có kích thước lớn thì cách truy nhập này rất tốn thời gian.

Để tránh nhược điểm này, C++ cho phép truy nhập trực tiếp đến một vị trí xác định trên tệp tin bằng các phép toán:

- Truy nhập vị trí hiện tại của con trỏ tệp tin
- Dịch chuyển con trỏ tệp tin đến một vị trí xác định

4.3.2 Truy nhập vị trí hiện tại của con trỏ tệp

Cú pháp truy nhập đến vị trí hiện thời của con trỏ tệp phụ thuộc vào kiểu biến tệp đang dùng là để đọc hay ghi.

- Nếu biến tệp là kiểu mở tệp để đọc **ifstream** thì cú pháp là:

```
<Tên biến tệp>.tellg();
```

- Nếu biến tệp là kiểu mở tệp để ghi **ofstream** thì cú pháp là:

```
<Tên biến tệp>.tellp();
```

Chương trình 4.6a minh họa việc xác định vị trí hiện thời của con trỏ tệp sau một số thao tác đọc tệp trước đó.

Chương trình 4.6a

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

const int length = 25; // Độ dài tối đa tên tệp tin

void main() {
    clrscr();
    char fileName[length], output;
    cout << "Tên tệp tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin
```



```
/* Mở tệp tin */
ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin
if(!fileIn){                          // Không mở được tệp
    cout << "Khong the mo duoc tep tin " << fileName << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra màn hình
 * Ghi vị trí con trỏ tệp ra màn hình cứ sau 5 lần đọc kí tự */
int index = 0;
while(fileIn){
    fileIn >> output;                // Đọc kí tự từ tệp tin
    cout << output;                  // Ghi kí tự ra màn hình
    if(index % 5 == 0)                // Ghi ra vị trí con trỏ tệp
        cout<< endl << "Vi tri con tro tep: "
            << fileIn.tellg() << endl;
    index ++;
}
cout << endl;                        // Xuống dòng trên màn hình

/* Đóng tệp tin */
fileIn.close();                      // Đóng tệp tin
return;
}
```

Chương trình 4.6b minh họa việc xác định vị trí hiện thời của con trỏ tệp sau một số thao tác ghi vào tệp trước đó.

Chương trình 4.6b

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

const int length = 25;                // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], input;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin
```



```

/* Mở tệp tin */
ofstream fileOut(fileName, ios::out); // Khai báo và mở tệp tin
if(!fileOut){                          // Không mở được tệp
    cout << "Khong the tao duoc tep tin " << fileName << endl;
    exit(1);
}

/* Ghi dữ liệu vào tệp tin
* Hiện ra màn hình vị trí con trỏ tệp sau khi ghi được 5 kí tự*/
int index = 0;
do{
    cin >> input;                      // Đọc kí tự từ bàn phím
    fileOut << input << ' ';           // Ghi kí tự vào tệp tin
    if(index%5 == 0)                   // Hiện thị vị trí con trỏ tệp
        cout << "Vi tri con tro tep: "
              << fileOut.tellp() << endl;
    index++;
}while((input != 'e') && (fileOut));
fileOut << endl;                       // Xuống dòng cuối tệp tin

/* Đóng tệp tin */
fileOut.close();                       // Đóng tệp tin
return;
}

```

4.3.3 Dịch chuyển con trỏ tệp

Ngoài việc xác định vị trí hiện thời của con trỏ tệp, C++ còn cho phép dịch chuyển con trỏ tệp đến một vị trí bất kì trên tệp. Cú pháp dịch chuyển phụ thuộc vào kiểu biến tệp là đọc hay ghi.

- Nếu biến tệp có kiểu là mở tệp tin để đọc **ifstream**, cú pháp sẽ là:
`<Tên biến tệp>.seekg(<Kích thước>, <Mốc dịch chuyển>);`
- Nếu biến tệp có kiểu là mở tệp để ghi **ofstream**, cú pháp sẽ là:
`<Tên biến tệp>.seekp(<Kích thước>, <Mốc dịch chuyển>);`

Trong đó:

- **Kích thước:** là tham số mô tả khoảng cách dịch chuyển so với vị trí mốc dịch chuyển. Đơn vị tính của kích thước là byte, có kiểu là số nguyên.
- **Mốc dịch chuyển:** là vị trí gốc để xác định khoảng cách dịch chuyển của con trỏ tệp. Có ba tham số hằng về kiểu mốc dịch chuyển:
 - **ios::beg:** Mốc dịch chuyển là đầu tệp tin.
 - **ios::cur:** Mốc dịch chuyển là vị trí hiện thời của con trỏ tệp.
 - **ios::end:** Mốc dịch chuyển là vị trí cuối cùng của tệp tin.

Ví dụ:

```
ifstream fileIn("abc.txt", ios::in);  
fileIn.seekg(sizeof(char)*7, ios::beg);
```

sẽ dịch chuyển con trỏ tệp tin đến kí tự (kiểu char) thứ $7+1 = 8$ trong tệp tin abc.txt để đọc (giả sử tệp tin abc.txt lưu các kí tự kiểu char).

Lưu ý:

- Vì khoảng cách cách dịch chuyển có kiểu số nguyên (int) cho nên có thể nhận giá trị âm hoặc dương. Nếu giá trị dương, dịch chuyển về phía sau vị trí làm mốc, nếu giá trị âm, dịch chuyển về phía trước vị trí làm mốc.
- Nếu vị trí dịch chuyển đến nằm ngoài phạm vi tệp tin (phía sau vị trí cuối cùng của tệp hoặc phía trước vị trí đầu tiên của tệp) sẽ nảy sinh lỗi, khi đó <Tên biến tệp> = false.

Chương trình 4.7 cài đặt chương trình truy nhập tệp tin trực tiếp để đọc giá trị kí tự (kiểu char) trong tệp:

- Tên tệp tin (chứa dữ liệu kiểu char) do người dùng nhập vào từ bàn phím.
- Sau đó, mỗi khi người dùng nhập vào một số nguyên, chương trình sẽ dịch chuyển đến vị trí mới, cách vị trí cũ đúng bằng từng ấy kí tự, tính từ vị trí hiện thời của con trỏ tệp.
- Chương trình sẽ kết thúc khi người dùng nhập vào số 0.

Chương trình 4.7

```
#include<stdlib.h>  
#include<iostream.h>  
#include<fstream.h>  
#include<conio.h>  
  
const int length = 25;           // Độ dài tối đa tên tệp tin  
  
void main() {  
    clrscr();  
    char fileName[length], output;  
    cout << "Tên tệp tin: ";  
    cin >> setw(length) >> fileName;    // Nhập tên tệp tin  
  
    /* Mở tệp tin */  
    ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin  
    if(!fileIn) {                     // Không mở được tệp  
        cout << "Khong the mo duoc tep tin " << fileName << endl;  
        exit(1);  
    }  
  
    /* Đọc dữ liệu từ tệp tin ra màn hình  
    * Ghi vị trí con trỏ tệp ra màn hình cứ sau 5 lần đọc kí tự */
```

```
int index = 1;
do{
    cout << "Số kí tự dịch chuyển: ";
    cin >> index;

    // Dịch chuyển con trỏ tệp từ vị trí hiện thời
    fileIn.seekg(sizeof(char)*index, ios::cur);
    if(fileIn){ // Đúng
        fileIn >> output; // Đọc kí tự từ tệp tin

        // Ghi kí tự ra màn hình
        cout << "Vị trí: " << fileIn.tellg() << output;
    }else{ // Ra khỏi phạm vi tệp
        fileIn.clear(); // Về vị trí đầu tệp
    }
}while(index);

/* Đóng tệp tin */
fileIn.close(); // Đóng tệp tin
return;
}
```

TỔNG KẾT CHƯƠNG 4

Nội dung chương 4 đã tập trung trình bày các vấn đề liên quan đến các thao tác trên tệp tin trong ngôn ngữ C++. Bao gồm:

- Các bước tuần tự khi thao tác với một tệp tin:
 - Mở tệp tin
 - Đọc/ghi dữ liệu trên tệp tin
 - Đóng tệp tin
- Thao tác mở tệp tin với nhiều chế độ bằng kiểu fstream.
- Thao tác mở tệp tin chỉ để đọc với kiểu ifstream
- Thao tác mở tệp tin chỉ để ghi với thao tác ofstream.
- Đọc dữ liệu từ tệp tin văn bản với thao tác ">>".
- Ghi dữ liệu vào tệp tin văn bản bằng thao tác "<<".
- Đọc tệp tin nhị phân bằng thao tác read().
- Ghi vào tệp tin nhị phân bằng thao tác write().
- Xác định vị trí hiện thời của con trỏ tệp tin với các thao tác tellg() và tellp().
- Dịch chuyển vị trí của con trỏ tệp tin với các thao tác seekg() và seekp().
- Thiết lập lại trạng thái cho con trỏ tệp tin bằng thao tác clear().

- Đóng tệp tin đã sử dụng bằng thao tác `close()`.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 4

1. Muốn mở một tệp tin tên là `abc.txt` để đọc dữ liệu, lệnh mở tệp nào sau đây là đúng:

- a. `fstream myFile("abc.txt", ios::in);`
- b. `fstream myFile("abc.txt", ios::out);`
- c. `fstream myFile("abc.txt", ios::app);`
- d. `fstream myFile("abc.txt", ios::ate);`

2. Muốn mở một tệp tin `abc.txt` nằm trong thư mục `xyz` để ghi dữ liệu vào. Lệnh mở nào sau đây là đúng:

- a. `fstream myFile("xyz\\abc.txt", ios::out);`
- b. `fstream myFile("xyz\\abc.txt", ios::out);`
- c. `fstream myFile("xyz/abc.txt", ios::out);`
- d. `fstream myFile("xyz//abc.txt", ios::out);`

3. Muốn mở một tệp tin `abc.txt` để ghi thêm dữ liệu vào cuối tệp, lệnh nào sau đây là đúng:

- a. `fstream myFile("abc.txt", ios::out);`
- b. `fstream myFile("abc.txt", ios::app);`
- c. `fstream myFile("abc.txt", ios::out|ios::app);`
- d. `fstream myFile("abc.txt", ios::out||ios::app);`

4. Xét hai lệnh khai báo sau:

```
fstream myFile1("abc.txt", ios::out);  
ofstream myFile2("abc.txt", ios::out);
```

Nhận định nào sau đây là đúng:

- a. `myFile1` và `myFile2` có chức năng giống nhau.
- b. `myFile1` và `myFile2` có chức năng khác nhau

5. Xét hai lệnh khai báo sau:

```
fstream myFile1("abc.txt", ios::in);  
ifstream myFile2("abc.txt", ios::in);
```

Nhận định nào sau đây là đúng:

- a. `myFile1` và `myFile2` có chức năng giống nhau.
- b. `myFile1` và `myFile2` có chức năng khác nhau

6. Xét đoạn chương trình sau:

```
ofstream myFile("abc.txt", ios::out);  
if(myFile) myFile << "abc.txt";
```

Chương trình sẽ làm gì?

- a. Ghi ra màn hình dòng chữ "abc.txt"
- b. Ghi vào tệp tin `abc.txt` dòng chữ "abc.txt"
- c. Đọc từ tệp tin `abc.txt` dòng chữ "abc.txt"

d. Chương trình sẽ báo lỗi.

7. Xét đoạn chương trình sau:

```
ifstream myFile("abc.txt", ios::in);
char text[20];
if(myFile) myFile >> text;
```

Chương trình sẽ làm gì, nếu tệp tin abc.txt có nội dung là dòng chữ "abc.txt"?

- a. Ghi ra màn hình dòng chữ "abc.txt"
- b. Ghi vào tệp tin abc.txt dòng chữ "abc.txt"
- c. Đọc từ tệp tin abc.txt dòng chữ "abc.txt"
- d. Chương trình sẽ báo lỗi.

8. Xét đoạn chương trình sau:

```
fstream myFile("abc.txt", ios::out);
if(myFile) myFile << "abc.txt";
myFile.close();
myFile.open("abc.txt", ios::in);
char text[20];
if(myFile) myFile >> text;
cout << text;
```

Chương trình sẽ làm gì, nếu tệp tin abc.txt có nội dung là dòng chữ "abc.txt"?

- a. Ghi vào tệp tin abc.txt dòng chữ "abc.txt"
- b. Đọc từ tệp tin abc.txt dòng chữ "abc.txt"
- c. Ghi ra màn hình dòng chữ "abc.txt"
- d. Cả ba đáp án trên.
- e. Chương trình sẽ báo lỗi.

9. Xét đoạn chương trình sau:

```
ifstream myFile("abc.txt", ios::in);
if(myFile) cout << myFile.tellg();
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0
- b. 1
- c. 8
- d. 16

10. Xét đoạn chương trình sau, nếu tệp abc.txt chứa một số lượng kí tự đủ lớn:

```
ifstream myFile("abc.txt", ios::in);
if(myFile) {
    char c;
    myFile >> c;
    cout << myFile.tellg();
}
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0

- b. 1
- c. 8
- d. 16

11. Xét đoạn chương trình sau, nếu tệp abc.txt chứa một số lượng kí tự đủ lớn:

```
ifstream myFile("abc.txt", ios::in);  
if(myFile) {  
    myFile.seekg(sizeof(char)*5, ios::beg);  
    myFile.seekg(sizeof(char)*5, ios::cur);  
    cout << myFile.tellg();  
}
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0
 - b. 5
 - c. 10
 - d. 80
12. Viết một chương trình gộp nội dung của hai tệp tin có sẵn vào một tệp tin thứ ba. Tên các tệp tin được nhập vào từ bàn phím.
13. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có tên là X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
14. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có năm sinh là X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
15. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có lương cao hơn hoặc bằng một giá trị X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
16. Viết một chương trình sao chép một đoạn đầu nội dung của một tệp tin vào một tệp tin thứ hai. Tên các tệp tin và độ dài đoạn nội dung cần sao chép được nhập từ bàn phím.
17. Viết một chương trình sao chép một đoạn cuối nội dung của một tệp tin vào một tệp tin thứ hai. Tên các tệp tin và độ dài đoạn nội dung cần sao chép được nhập từ bàn phím.

CHƯƠNG 5

LỚP

Nội dung chương này tập trung trình bày các vấn đề liên quan đến lớp đối tượng trong C++:

- Khái niệm, khai báo và sử dụng lớp
- Khai báo và sử dụng các thành phần của lớp: các thuộc tính và các phương thức của lớp
- Phạm vi truy nhập lớp
- Khai báo và sử dụng các phương thức khởi tạo và hủy bỏ của lớp
- Sử dụng lớp thông qua con trỏ đối tượng, mảng các đối tượng.

5.1 KHÁI NIỆM LỚP ĐỐI TƯỢNG

C++ coi lớp là sự trừu tượng hóa các đối tượng, là một khuôn mẫu để biểu diễn các đối tượng thông qua các thuộc tính và các hành động đặc trưng của đối tượng.

5.1.1 Định nghĩa lớp đối tượng

Để định nghĩa một lớp trong C++, ta dùng từ khóa **class** với cú pháp:

```
class <Tên lớp>{  
    };  
};
```

Trong đó:

- **class**: là tên từ khóa bắt buộc để định nghĩa một lớp đối tượng trong C++.
- **Tên lớp**: do người dùng tự định nghĩa. Tên lớp có tính chất như tên kiểu dữ liệu để sử dụng sau này. Cách đặt tên lớp phải tuân thủ theo quy tắc đặt tên biến trong C++.

Ví dụ:

```
class Car{  
    };  
};
```

là định nghĩa một lớp xe ô tô (Car). Lớp này chưa có bất kì một thành phần nào, việc định nghĩa các thành phần cho lớp sẽ được trình bày trong mục 5.2.

Lưu ý:

- Từ khóa **class** là bắt buộc để định nghĩa một lớp đối tượng trong C++. Hơn nữa, C++ có phân biệt chữ hoa chữ thường trong khai báo cho nên chữ **class** phải được viết bằng chữ thường.

Ví dụ:

```
class Car{ // Định nghĩa đúng  
    };  
};
```

nhưng:

```
Class Car{ // Lỗi từ khóa  
    };  
};
```

- Bắt buộc phải có dấu chấm phẩy “;” ở cuối định nghĩa lớp vì C++ coi định nghĩa một lớp như định nghĩa một kiểu dữ liệu, cho nên phải có dấu chấm phẩy cuối định nghĩa (tương tự định nghĩa kiểu dữ liệu kiểu cấu trúc).
- Để phân biệt với tên biến thông thường, ta nên (nhưng không bắt buộc) đặt *tên lớp bắt đầu bằng một chữ in hoa và các tên biến bắt đầu bằng một chữ in thường*.

5.1.2 Sử dụng lớp đối tượng

Lớp đối tượng được sử dụng khi ta khai báo các thể hiện của lớp đó. Một thể hiện của một lớp chính là một đối tượng cụ thể của lớp đó. Việc khai báo một thể hiện của một lớp được thực hiện như cú pháp khai báo một biến có kiểu lớp:

```
<Tên lớp> <Tên biến lớp>;
```

Trong đó:

- **Tên lớp:** là tên lớp đối tượng đã được định nghĩa trước khi khai báo biến.
- **Tên biến lớp:** là tên đối tượng cụ thể. Tên biến lớp sẽ được sử dụng như các biến thông thường trong C++, ngoại trừ việc nó có kiểu lớp đối tượng.

Ví dụ, muốn khai báo một thể hiện (biến) của lớp Car đã được định nghĩa trong mục 5.1.1, ta khai báo như sau:

```
Car myCar;
```

Sau đó, ta có thể sử dụng biến myCar trong chương trình như các biến thông thường: truyền tham số cho hàm, gán cho biến khác ...

Lưu ý:

- Khi khai báo biến lớp, ta không dùng lại từ khóa **class** nữa. Từ khóa **class** chỉ được sử dụng khi định nghĩa lớp mà không dùng khi khai báo biến lớp.

Ví dụ, khai báo:

```
Car myCar; // đúng
```

là đúng, nhưng khai báo:

```
class Car myCar; // Lỗi cú pháp
```

là sai cú pháp.

5.2 CÁC THÀNH PHẦN CỦA LỚP

Việc khai báo các thành phần của lớp có dạng như sau:

```
class <Tên lớp>{  
private:  
    <Khai báo các thành phần riêng>  
protected:  
    <Khai báo các thành phần được bảo vệ>  
public:  
    <Khai báo các thành phần công cộng>  
};
```

Trong đó:

- **private:** là từ khóa chỉ tính chất của C++ để chỉ ra rằng các thành phần được khai báo trong phạm vi từ khóa này là riêng tư đối với lớp đối tượng. Các đối tượng của các lớp khác không truy nhập được các thành phần này.
- **protected:** các thành phần được khai báo trong phạm vi từ khóa này đều được bảo vệ. Qui định loại đối tượng nào được truy nhập đến các thành phần được bảo vệ sẽ được mô tả chi tiết trong mục 5.3.
- **public:** các thành phần công cộng. Các đối tượng của các lớp khác đều có thể truy nhập đến các thành phần công cộng của một đối tượng bất kì.

Các thành phần của lớp được chia làm hai loại:

- Các thành phần chỉ dữ liệu của lớp, được gọi là *thuộc tính* của lớp
- Các thành phần chỉ hành động của lớp, được gọi là *phương thức* của lớp.

5.2.1 Thuộc tính của lớp

Khai báo thuộc tính

Thuộc tính của lớp là thành phần chứa dữ liệu, đặc trưng cho các tính chất của lớp. Thuộc tính của lớp được khai báo theo cú pháp sau:

<Kiểu dữ liệu> <Tên thuộc tính>;

Trong đó:

- **Kiểu dữ liệu:** có thể là các kiểu dữ liệu cơ bản của C++, cũng có thể là các kiểu dữ liệu phức tạp do người dùng tự định nghĩa như struct, hoặc kiểu là một lớp đã được định nghĩa trước đó.
- **Tên thuộc tính:** là tên thuộc tính của lớp, có tính chất như một biến thông thường. Tên thuộc tính phải tuân theo quy tắc đặt tên biến của C++.

Ví dụ, khai báo:

```
class Car{
private:
    int speed;
public:
    char mark[20];
};
```

là khai báo một lớp xe ô tô (Car), có hai thuộc tính: thuộc tính tốc độ (speed) có tính chất private, thuộc tính nhãn hiệu xe (mark) có tính chất public.

Lưu ý:

- Không được khởi tạo giá trị ban đầu cho các thuộc tính ngay trong lớp. Vì các thuộc tính chỉ có giá trị khi nó gắn với một đối tượng cụ thể, là một thể hiện (biến) của lớp.

Ví dụ:

```
class Car{
private:
    int speed;           // đúng
    int weight = 500;    // lỗi
```

};

- Khả năng truy nhập thuộc tính của lớp là phụ thuộc vào thuộc tính ấy được khai báo trong phạm vi của từ khóa nào: private, protected hay public.
- Thông thường, do yêu cầu đóng gói dữ liệu của hướng đối tượng, ta nên khai báo các thuộc tính có tính chất riêng tư (private). Nếu muốn các đối tượng khác truy nhập được vào các thuộc tính này, ta xây dựng các hàm public truy nhập (get / set) đến thuộc tính đó.

Sử dụng thuộc tính

Thuộc tính có thể được sử dụng cho các chương trình nằm ngoài lớp thông qua tên biến lớp hoặc sử dụng ngay trong lớp bởi các phương thức của lớp.

- Nếu thuộc tính được dùng bên ngoài phạm vi lớp, cú pháp phải thông qua tên biến lớp (cách này chỉ sử dụng được với các biến có tính chất public):

<Tên biến lớp>.<tên thuộc tính>;

- Nếu thuộc tính được dùng bên trong lớp, cú pháp đơn giản hơn:

<Tên thuộc tính>;

Ví dụ, với định nghĩa lớp:

```
class Car{
private:
    int speed;
public:
    char mark[20];
};
```

ta khai báo một biến lớp:

```
Car myCar;
```

Thì có thể sử dụng thuộc tính nhãn hiệu xe khi in ra màn hình như sau:

```
cout << myCar.mark;
```

Lưu ý:

- Khi dùng thuộc tính bên trong các phương thức của lớp, mà tên thuộc tính lại bị trùng với tên biến toàn cục (tự do) của chương trình, ta phải chỉ rõ việc dùng tên thuộc tính của lớp (mà không phải tên biến toàn cục) bằng cách dùng chỉ thị phạm vi lớp "::" với cú pháp:

<Tên lớp>::<Tên thuộc tính>;

5.2.2 Phương thức của lớp

Khai báo khuôn mẫu phương thức

Một phương thức là một thao tác thực hiện một số hành động đặc trưng của lớp đối tượng. Phương thức được khai báo tương tự như các hàm trong C++:

<Kiểu trả về> <Tên phương thức>([<Các tham số>]);

Trong đó:

- **Kiểu trả về:** là kiểu dữ liệu trả về của phương thức. Kiểu có thể là các kiểu dữ liệu cơ bản của C++, cũng có thể là kiểu do người dùng định nghĩa, hoặc kiểu lớp đã được định nghĩa.

- **Tên phương thức:** do người dùng tự đặt tên, tuân theo quy tắc đặt tên biến của C++.
- **Các tham số:** Các tham số đầu vào của phương thức, được biểu diễn bằng kiểu dữ liệu tương ứng. Các tham số được phân cách bởi dấu phẩy “,”. Các tham số là tùy chọn (Phần trong dấu ngoặc vuông “[]” là tùy chọn).

Ví dụ, khai báo:

```
class Car{
private:
    int speed;
    char mark[20];
public:
    void show();
};
```

là định nghĩa một lớp Car có hai thuộc tính cục bộ là speed và mark, và khai báo một phương thức show() để mô tả đối tượng xe tương ứng. Show() là một phương thức không cần tham số và kiểu trả về là void.

Lưu ý:

- Khả năng truy nhập phương thức từ bên ngoài là phụ thuộc vào phương thức được khai báo trong phạm vi của từ khóa nào: private, protected hay public.

Định nghĩa phương thức

Trong C++, việc cài đặt chi tiết nội dung của phương thức có thể tiến hành ngay trong phạm vi lớp hoặc bên ngoài phạm vi định nghĩa lớp. Cú pháp chỉ khác nhau ở dòng khai báo tên phương thức.

- Nếu cài đặt phương thức ngay trong phạm vi định nghĩa lớp, cú pháp là:

```
<Kiểu trả về> <Tên phương thức>(<[<Các tham số>]>){
    ... // Cài đặt chi tiết
}
```
- Nếu cài đặt phương thức bên ngoài phạm vi định nghĩa lớp, ta phải dùng chỉ thị phạm vi “::” để chỉ ra rằng đây là một phương thức của lớp mà không phải là một hàm tự do trong chương trình:

```
<Kiểu trả về> <Tên lớp>::<Tên phương thức>(<[<Các tham số>]>){
    ... // Cài đặt chi tiết
}
```

Ví dụ, nếu cài đặt phương thức show() của lớp Car ngay trong phạm vi định nghĩa lớp, ta cài đặt như sau:

```
class Car{
private:
    int    speed;        // Tốc độ
    char   mark[20];     // Nhãn hiệu
public:
    void show(){         // Khai báo phương thức ngay trong lớp
        cout << "This is a " << mark << " having a speed of "
```

```

        << speed << "km/h!" << endl;
    return;
}

};

```

Nếu muốn cài đặt bên ngoài lớp, ta cài đặt như sau:

```

class Car{
private:
    int    speed;        // Tốc độ
    char   mark[20];     // Nhãn hiệu
public:
    void show();         // Giới thiệu xe
};
/* Khai báo phương thức bên ngoài lớp */
void Car::show() {
    cout << "This is a " << mark << " having a speed of "
         << speed << "km/h!" << endl;
    return;
}

```

Lưu ý:

- Nếu phương thức được cài đặt ngay trong lớp thì các tham số phải tường minh, nghĩa là mỗi tham số phải được biểu diễn bằng một cặp <Kiểu dữ liệu> <Tên tham số> như khi cài đặt chi tiết một hàm tự do trong chương trình.
- Thông thường, chỉ các phương thức ngắn (trên một dòng) là nên cài đặt ngay trong lớp. Còn lại nên cài đặt các phương thức bên ngoài lớp để chương trình được sáng sủa, rõ ràng và dễ theo dõi.

Sử dụng phương thức

Cũng tương tự như các thuộc tính của lớp, các phương thức cũng có thể được sử dụng bên ngoài lớp thông qua tên biến lớp, hoặc có thể được dùng ngay trong lớp bởi các phương thức khác của lớp định nghĩa nó.

- Nếu phương thức được dùng bên ngoài phạm vi lớp, cú pháp phải thông qua tên biến lớp (cách này chỉ sử dụng được với các phương thức có tính chất public):

```
<Tên biến lớp>.<Tên phương thức>([<Các đối số>]);
```

- Nếu thuộc tính được dùng bên trong lớp, cú pháp đơn giản hơn:

```
<Tên phương thức>([<Các đối số>]);
```

Ví dụ, với định nghĩa lớp:

```

class Car{
private:
    int    speed;        // Tốc độ
    char   mark[20];     // Nhãn hiệu
public:
    void show();         // Giới thiệu xe

```

```
};  
/* Khai báo phương thức bên ngoài lớp */  
void Car::show() {  
    cout << "This is a " << mark << " having a speed of "  
        << speed << "km/h!" << endl;  
    return;  
}
```

ta khai báo một biến lớp:

```
Car myCar;
```

Thì có thể sử dụng phương thức giới thiệu xe như sau:

```
myCar.show();
```

Lưu ý:

- Khi dùng phương thức bên trong các phương thức khác của lớp, mà phương thức lại bị trùng với các phương thức tự do của chương trình, ta phải chỉ rõ việc dùng phương thức của lớp (mà không phải dùng phương thức tự do) bằng cách dùng chỉ thị phạm vi lớp "::" với cú pháp:

```
<Tên lớp>::<Tên phương thức>([<Các đối số>]);
```

Chương trình 5.1 cài đặt đầy đủ một lớp xe ô tô (Car) với các thuộc tính có tính chất cục bộ:

- Tốc độ xe (speed)
- Nhân hiệu xe (mark)
- Giá xe (price)

Và các phương thức có tính chất public:

- Khởi tạo các tham số (init)
- Giới thiệu xe (show)
- Các phương thức truy nhập (get/set) các thuộc tính

Sau đó, chương trình main sẽ sử dụng lớp Car này để định nghĩa các đối tượng cụ thể và sử dụng các phương thức của lớp này.

Chương trình 5.1

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp */  
class Car{  
    private:  
        int    speed;           // Tốc độ  
        char   mark[20];       // Nhân hiệu  
        float  price;          // Giá xe  
    public:  
        void   setSpeed(int);   // Gán tốc độ cho xe
```

```

        int    getSpeed();           // Đọc tốc độ xe
        void   setMark(char);       // Gán nhãn cho xe
        char[] getMark();           // Đọc nhãn xe
        void   setPrice(float);     // Gán giá cho xe
        float  getPrice();          // Đọc giá xe
        void   init(int, char[], float); // Khởi tạo thông tin về xe
        void   show();              // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
void Car::setSpeed(int speedIn){    // Gán tốc độ cho xe
    speed = speedIn;
}
int Car::getSpeed(){               // Đọc tốc độ xe
    return speed;
}
void Car::setMark(char markIn){    // Gán nhãn cho xe
    strcpy(mark, markIn);
}
char[] Car::getMark(){            // Đọc nhãn xe
    return mark;
}
void Car::setPrice(float priceIn){ // Gán giá cho xe
    price = priceIn;
}
float Car::getPrice(){            // Đọc giá xe
    return price;
}

void Car::init(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
    return;
}

void Car::show(){                 // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính

```



```
void main(){
    clrscr();
    Car myCar;                                // Khai báo biến lớp

    // Khởi tạo lần thứ nhất
    cout << "Xe thu nhat: " << endl;
    myCar.init(100, "Ford", 3000);
    cout << "Toc do (km/h): " << myCar.getSpeed() << endl;
    cout << "Nhan hieu : " << myCar.getMark() << endl;
    cout << "Gia ($) : " << myCar.getPrice() << endl;

    // Thay đổi thuộc tính xe
    cout << "Xe thu hai: " << endl;
    myCar.setSpeed(150);
    myCar.setMark("Mercedes");
    myCar.setPrice(5000);
    myCar.show();
    return;
}
```

Chương trình 5.1 sẽ in ra kết quả như sau:

```
Xe thu nhat:
Toc do (km/h): 100
Nhan hieu: Ford
Gia ($) : 3000
Xe thu hai:
This is a Mercedes having a speed of 150km/h and its price is $5000
```

5.3 PHẠM VI TRUY NHẬP LỚP

5.3.1 Phạm vi truy nhập lớp

Trong C++, có một số khái niệm về phạm vi, xếp từ bé đến lớn như sau:

- **Phạm vi khối lệnh:** Trong phạm vi giữa hai dấu giới hạn "{}" của một khối lệnh. Ví dụ các lệnh trong khối lệnh lặp while(){} sẽ có cùng phạm vi khối lệnh.
- **Phạm vi hàm:** Các lệnh trong cùng một hàm có cùng mức phạm vi hàm.
- **Phạm vi lớp:** Các thành phần của cùng một lớp có cùng phạm vi lớp với nhau: các thuộc tính và các phương thức của cùng một lớp.
- **Phạm vi chương trình (còn gọi là phạm vi tệp):** Các lớp, các hàm, các biến được khai báo và định nghĩa trong cùng một tệp chương trình thì có cùng phạm vi chương trình.

Trong phạm vi truy nhập lớp, ta chỉ quan tâm đến hai phạm vi lớn nhất, đó là phạm vi lớp và phạm vi chương trình. Trong C++, phạm vi truy nhập lớp được quy định bởi các từ khóa về thuộc tính truy nhập:

- **private**: Các thành phần của lớp có thuộc tính private thì chỉ có thể được truy nhập trong phạm vi lớp.
- **protected**: Trong cùng một lớp, thuộc tính protected cũng có ảnh hưởng tương tự như thuộc tính private: các thành phần lớp có thuộc tính protected chỉ có thể được truy nhập trong phạm vi lớp. Ngoài ra nó còn có thể được truy nhập trong các lớp con khi có kế thừa (sẽ được trình bày trong chương 6).
- **public**: các thành phần lớp có thuộc tính public thì có thể được truy nhập trong phạm vi chương trình, có nghĩa là nó có thể được truy nhập trong các hàm tự do, các phương thức bên trong các lớp khác...

Ví dụ, thuộc tính price của lớp Car có tính chất private nên chỉ có thể truy nhập bởi các phương thức của lớp Car. Không thể truy nhập từ bên ngoài lớp (phạm vi chương trình), chẳng hạn trong một hàm tự do ngoài lớp Car.

```
void Car::setPrice(float priceIn){  
    price = priceIn; // Đúng, vì setPrice là một phương thức  
                    // của lớp Car  
}
```

nhưng:

```
void freeFunction(Car myCar){  
    myCar.price = 3000; // Lỗi, vì freeFunction là một hàm tự do  
                      // nằm ngoài phạm vi lớp Car  
}
```

Khi đó, hàm freeFunction phải truy nhập gián tiếp đến thuộc tính price thông qua phương thức truy nhập có tính chất public như sau:

```
void freeFunction(Car myCar){  
    myCar.setPrice(3000); // Đúng, vì setPrice là một phương thức  
                          // của  
                          // lớp Car có thuộc tính public  
}
```

Tuy nhiên, C++ cho phép một cách đặc biệt để truy nhập đến các thành phần private và protected của một lớp bằng khái niệm *hàm bạn* và *lớp bạn* của một lớp: trong các hàm bạn và lớp bạn của một lớp, có thể truy nhập đến các thành phần private và protected như bên trong phạm vi lớp đó.

5.3.2 Hàm bạn

Có hai kiểu hàm bạn cơ bản trong C++:

- Một hàm tự do là hàm bạn của một lớp
- Một hàm thành phần (phương thức) của một lớp là bạn của một lớp khác

Ngoài ra còn có một số kiểu hàm bạn mở rộng từ hai kiểu này:

- Một hàm là bạn của nhiều lớp

- Tất cả các hàm của một lớp là bạn của lớp khác (lớp bạn)

Hàm tự do bạn của một lớp

Một hàm bạn của một lớp được khai báo bằng từ khóa **friend** khi khai báo khuôn mẫu hàm trong lớp tương ứng.

```
class <Tên lớp>{
...    // Khai báo các thành phần lớp như thông thường
// Khai báo hàm bạn
friend <Kiểu trả về> <Tên hàm bạn>([<Các tham số>]);
};
```

Khi đó, định nghĩa chi tiết hàm bạn được thực hiện như định nghĩa một hàm tự do thông thường:

```
<Kiểu trả về> <Tên hàm bạn>([<Các tham số>]){
...    // Có thể truy nhập trực tiếp các thành phần private
        // của lớp đã khai báo
}
```

Lưu ý:

- Mặc dù hàm bạn được khai báo khuôn mẫu hàm trong phạm vi lớp, nhưng hàm bạn tự do lại không phải là một phương thức của lớp. Nó là hàm tự do, việc định nghĩa và sử dụng hàm này hoàn toàn tương tự như các hàm tự do khác.
- Việc khai báo khuôn mẫu hàm bạn trong phạm vi lớp ở vị trí nào cũng được: hàm bạn không bị ảnh hưởng bởi các từ khóa private, protected hay public trong lớp.
- Trong hàm bạn, có thể truy nhập trực tiếp đến các thành phần private và protected của đối tượng có kiểu lớp mà nó làm bạn (truy nhập thông qua đối tượng cụ thể).

Chương trình 5.2 minh họa việc định nghĩa một hàm bạn của lớp Car, hàm này so sánh xem hai chiếc xe, chiếc nào đắt hơn.

Chương trình 5.2

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

/* Định nghĩa lớp */
class Car{
    private:
        int    speed;                // Tốc độ
        char   mark[20];             // Nhãn hiệu
        float  price;                // Giá xe
    public:
        void   init(int, char[], float); // Khởi tạo thông tin về xe
        // Khai báo hàm bạn của lớp
        friend void moreExpensive(Car, Car);
};
```

```

/* Khai báo phương thức bên ngoài lớp */
void Car::init(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
    return;
}

/* Định nghĩa hàm bạn tự do */
void moreExpensive(Car car1, Car car2){
    if(car1.price > car2.price)//Truy nhập đến các thuộc tính private
        cout << "xe thu nhat dat hon" << endl;
    else if(car1.price < car2.price)
        cout << "xe thu nhat dat hon" << endl;
    else
        cout << "hai xe dat nhu nhau" << endl;
    return;
}

// Hàm main, chương trình chính
void main(){
    clrscr();
    Car car1, car2;                                // Khai báo biến lớp

    // Khởi tạo xe thứ nhất, thứ hai
    car1.init(100, "Ford", 3000);
    car2.init(150, "Mercedes", 3500);

    // So sánh giá hai xe
    moreExpensive(car1, car2);                      // Sử dụng hàm bạn tự do
    return;
}

```

Chương trình 5.2 sẽ in ra thông báo:

xe thu hai dat hon

vì xe thứ hai có giá \$3500, trong khi xe thứ nhất được khởi tạo giá là \$3000.

Phương thức lớp là bạn của một lớp khác

Trong C++, một phương thức của lớp này cũng có thể làm bạn của một lớp kia. Để khai báo một phương thức f của lớp B là bạn của lớp A và f nhận một tham số có kiểu lớp A, ta phải khai báo tuần tự như sau (trong cùng một chương trình):

- Khai báo khuôn mẫu lớp A, để làm tham số cho hàm f của lớp B:

```
class A;
```

- Khai báo lớp B với hàm f như khai báo các lớp thông thường:

```
class B{
    ...           // Khai báo các thành phần khác của lớp B
    void f(A);
};
```

- Khai báo chi tiết lớp A với hàm f của lớp B là bạn

```
class A{
    ...           // Khai báo các thành phần khác của lớp A
    friend void B::f(A);
};
```

- Định nghĩa chi tiết hàm f của lớp B:

```
void B::f(A) {
    ...           // Định nghĩa chi tiết hàm f
}
```

Lưu ý:

- Trong trường hợp này, hàm f chỉ được định nghĩa chi tiết một khi lớp A đã được định nghĩa chi tiết. Do vậy, chỉ có thể định nghĩa chi tiết hàm f ngay trong lớp A (ở bước 3) hoặc sau khi định nghĩa lớp A (ở bước 4), mà không thể định nghĩa chi tiết hàm f ngay trong lớp B (ở bước 2).
- Hàm f có thể truy nhập đến các thành phần private và protected của cả hai lớp A và B. Tuy nhiên, muốn f truy nhập đến các thành phần của lớp A thì phải thông qua một đối tượng cụ thể có kiểu lớp A.

Chương trình 5.3 minh họa việc cài đặt và sử dụng một hàm permission() của lớp Person, là hàm bạn của lớp Car. Hàm này thực hiện việc kiểm tra xem một người có đủ quyền điều khiển xe hay không, theo luật sau:

- Với các loại xe thông thường, người điều khiển phải đủ 18 tuổi.
- Với các loại xe có tốc độ cao hơn 150km/h, người điều khiển phải đủ 21 tuổi.

Chương trình 5.3

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

class Car;                               // Khai báo nguyên mẫu lớp

/* Định nghĩa lớp Person */
class Person{
    private:
        char   name[25];                 // Tên
        int    age;                      // Tuổi
```

```

        public:
            void init(char[], int);           // Khởi tạo thông tin về người
            int permission(Car);             // Xác định quyền điều khiển xe
};

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;                       // Tốc độ
        char   mark[20];                   // Nhãn hiệu
        float  price;                      // Giá xe
    public:
        void  init(int, char[], float);    // Khởi tạo thông tin về xe
        // Khai báo hàm bạn của lớp
        friend int Person::permission(Car);
};

/* Khai báo phương thức bên ngoài lớp */
void Person::init(char nameIn[], int ageIn){
    strcpy(name, nameIn);
    age = ageIn;
    return;
}

void Car::init(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
    return;
}

/* Định nghĩa hàm bạn */
int Person::permission(Car car){
    if(age < 18)
        return 0;

    //Truy nhập thuộc tính private thông qua đối tượng car
    if((age < 21)&&(car.speed > 150))
        return 0;
    return 1;
}

// Hàm main, chương trình chính
void main(){

```

```
clrscr();
// Khai báo các biến lớp
Car car;
Person person;

// Khởi tạo các đối tượng
car.init(100, "Ford", 3000);
person.init("Vinh", 20);

// Xác định quyền điều khiển xe
if(person.permission(car))           // Sử dụng hàm bạn
    cout << "Co quyen dieu khien" << endl;
else
    cout << "Khong co quyen dieu khien" << endl;
return;
}
```

Chương trình 5.3 sẽ hiển thị thông báo:

Co quyen dieu khien

Vì người chủ xe đã 20 tuổi và xe chỉ có tốc độ 100km/h.

5.3.3 Lớp bạn

Khi tất cả các phương thức của một lớp là bạn của một lớp khác, thì lớp của các phương thức đó cũng trở thành lớp bạn của lớp kia. Muốn khai báo một lớp B là lớp bạn của lớp A, ta khai báo theo tuần tự sau:

- Khai báo khuôn mẫu lớp B:

```
class B;
```

- Định nghĩa lớp A, với khai báo B là lớp bạn:

```
class A{
    ...           // Khai báo các thành phần của lớp A
    // Khai báo lớp bạn B
    friend class B;
};
```

- Định nghĩa chi tiết lớp B:

```
class B{
    ...           // Khai báo các thành phần của lớp B
};
```

Lưu ý:

- Trong trường hợp này, lớp B là lớp bạn của lớp A nhưng không có nghĩa là lớp A cũng là bạn của lớp B: tất cả các phương thức của lớp B có thể truy nhập các thành phần private của lớp A (thông qua các đối tượng có kiểu lớp A) nhưng các phương thức của lớp A lại không thể truy nhập đến các thành phần private của lớp B.

- Muốn các phương thức của lớp A cũng truy nhập được đến các thành phần private của lớp B, thì phải khai báo thêm là lớp A cũng là lớp bạn của lớp B.

5.4 HÀM KHỞI TẠO VÀ HUỖ BỎ

5.4.1 Hàm khởi tạo

Hàm khởi tạo được gọi mỗi khi khai báo một đối tượng của lớp. Ngay cả khi không được khai báo tường minh, C++ cũng gọi hàm khởi tạo ngầm định khi đối tượng được khai báo.

Khai báo hàm khởi tạo

Hàm khởi tạo của một lớp được khai báo tường minh theo cú pháp sau:

```
class <Tên lớp>{
    public:
        <Tên lớp>([<Các tham số>]);    // Khai báo hàm khởi tạo
};
```

Ví dụ:

```
class Car{
    int    speed;
    char   mark[20];
    float  price;
    public:
        Car(int speedIn, char markIn[], float priceIn){
            speed = speedIn;
            strcpy(mark, markIn);
            price = priceIn;
        }
};
```

là khai báo một hàm khởi tạo với ba tham số của lớp Car.

Lưu ý:

- Hàm khởi tạo phải có tên trùng với tên của lớp
- Hàm khởi tạo không có giá trị trả về
- Hàm khởi tạo có tính chất public
- Có thể có nhiều hàm khởi tạo của cùng một lớp

Sử dụng hàm khởi tạo của lớp

Hàm khởi tạo được sử dụng khi khai báo biến lớp. Khi đó, ta có thể vừa khai báo, vừa khởi tạo giá trị các thuộc tính của đối tượng lớp theo cú pháp sau:

```
<Tên lớp> <Tên đối tượng>([<Các đối số khởi tạo>]);
```

Trong đó:

- **Tên lớp:** là tên kiểu lớp đã được định nghĩa
- **Tên đối tượng:** là tên biến có kiểu lớp, tuân thủ theo quy tắc đặt tên biến của C++

- **Các đối số khởi tạo:** Là các đối số tương ứng với hàm khởi tạo của lớp tương ứng. Tương tự như việc truyền đối số khi dùng các lời gọi hàm thông thường.

Ví dụ, nếu lớp Car có hàm khởi tạo Car(int, char[], float) thì khi khai báo biến có kiểu lớp Car, ta có thể sử dụng hàm khởi tạo này như sau:

```
Car myCar(100, "Ford", 3000);
```

Lưu ý:

- Khi sử dụng hàm khởi tạo, phải truyền đúng số lượng và đúng kiểu của các tham số của các hàm khởi tạo đã được định nghĩa của lớp.
- Khi một lớp đã có ít nhất một hàm khởi tạo tường minh, thì không được sử dụng hàm khởi tạo ngầm định của C++. Do đó, khi đã khai báo ít nhất một hàm khởi tạo, nếu muốn khai báo biến mà không cần tham số, lớp tương ứng phải có ít nhất một hàm khởi tạo không có tham số.

Ví dụ, nếu lớp Car chỉ có duy nhất một hàm khởi tạo như sau:

```
class Car{
public:
    Car(int, char[], float);
};
```

thì không thể khai báo một biến như sau:

```
Car myCar; // Khai báo lỗi
```

- Trong trường hợp dùng hàm khởi tạo không có tham số, ta không cần phải sử dụng cặp dấu ngoặc đơn “()” sau tên biến đối tượng. Việc khai báo trở thành cách khai báo thông thường.

Chương trình 5.4a minh họa việc định nghĩa và sử dụng lớp Car với hai hàm khởi tạo khác nhau.

Chương trình 5.4a

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

/* Định nghĩa lớp */
class Car{
private:
    int    speed;           // Tốc độ
    char   mark[20];        // Nhãn hiệu
    float  price;           // Giá xe
public:
    Car();                  // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đầy đủ tham số
    void   show();          // Giới thiệu xe
};
```



```
/* Khai báo phương thức bên ngoài lớp */
Car::Car() {                                // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo có đầy đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn) {
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::show() {                          // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main() {
    clrscr();
    Car myCar1;                             // Sử dụng hàm khởi tạo không tham số
    Car myCar2(150, "Mercedes", 5000);      // Dùng hàm khởi tạo đủ tham số

    // Giới thiệu xe thứ nhất
    cout << "Xe thu nhât: " << endl;
    myCar1.show();

    // Giới thiệu xe thứ hai
    cout << "Xe thu hai: " << endl;
    myCar2.show();
    return;
}
```

Chương trình 5.4a sẽ hiển thị các thông tin như sau:

Xe thu nhât:

This is a having a speed of 0km/h and its price is \$0

Xe thu hai:

This is a Mercedes having a speed of 150km/h and its price is \$5000

Lí do là xe thứ nhất sử dụng hàm khởi tạo không có tham số nên xe không có tên, tốc độ và giá đều là mặc định (0). Trong khi đó, xe thứ hai được khởi tạo đầy đủ cả ba tham số nên thông tin giới thiệu xe được đầy đủ.

Tuy nhiên, khi đối tượng có nhiều thuộc tính riêng, để tránh trường hợp phải định nghĩa nhiều hàm khởi tạo cho các trường hợp thiếu vắng một vài tham số khác nhau. Ta có thể sử dụng hàm khởi tạo với các giá trị khởi đầu ngầm định. Chương trình 5.4b cho kết quả hoàn toàn giống chương trình 5.4a, nhưng đơn giản hơn vì chỉ cần định nghĩa một hàm khởi tạo với các tham số có giá trị ngầm định.

Chương trình 5.4b

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

/* Định nghĩa lớp */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhân hiệu
        float  price;           // Giá xe
    public:
        // Khởi tạo với các giá trị ngầm định cho các tham số
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void  show();           // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::show() {                // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
         << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main(){
    clrscr();
    Car myCar1;                   // Các tham số nhận giá trị mặc định
```

```
Car myCar2(150, "Mercedes", 5000); // Dùng hàm khởi tạo đủ tham số

// Giới thiệu xe thứ nhất
cout << "Xe thu nhât: " << endl;
myCar1.show();

// Giới thiệu xe thứ hai
cout << "Xe thu hai: " << endl;
myCar2.show();
return;
}
```

5.4.2 Hàm hủy bỏ

Hàm hủy bỏ được tự động gọi đến khi mà đối tượng được giải phóng khỏi bộ nhớ. Nhiệm vụ của hàm hủy bỏ là dọn dẹp bộ nhớ trước khi đối tượng bị giải phóng. Cú pháp khai báo hàm hủy bỏ như sau:

```
class <Tên lớp>{
    public:
        ~<Tên lớp>(<[Các tham số]>); // Khai báo hàm khởi tạo
};
```

Ví dụ:

```
class Car{
    int    speed;
    char   *mark;
    float  price;
    public:
        ~Car(){
            delete [] mark;
        };
};
```

là khai báo một hàm hủy bỏ của lớp Car với thuộc tính mark có kiểu con trỏ. Hàm này sẽ giải phóng vùng nhớ đã cấp phát cho con trỏ kiểu char của thuộc tính nhãn hiệu xe.

Lưu ý:

- Hàm hủy bỏ phải có tên bắt đầu bằng dấu “~”, theo sau là tên của lớp tương ứng.
- Hàm hủy bỏ không có giá trị trả về.
- Hàm hủy bỏ phải có tính chất public
- Mỗi lớp chỉ có nhiều nhất một hàm hủy bỏ. Trong trường hợp không khai báo tường minh hàm hủy bỏ, C++ sẽ sử dụng hàm hủy bỏ ngầm định.

- Nói chung, khi có ít nhất một trong các thuộc tính của lớp là con trỏ, thì nên dùng hàm hủy bỏ tường minh để giải phóng triệt để các vùng nhớ của các thuộc tính, trước khi đối tượng bị giải phóng khỏi bộ nhớ.

Chương trình 5.5 minh họa việc định nghĩa lớp Car với một hàm khởi tạo có các tham số với giá trị mặc định và một hàm hủy bỏ tường minh.

Chương trình 5.5

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

/* Định nghĩa lớp */
class Car{
    private:
        int    speed;           // Tốc độ
        char   *mark;           // Nhãn hiệu
        float  price;           // Giá xe
    public:
        // Khởi tạo với các giá trị ngầm định cho các tham số
        Car(int speedIn=0, char *markIn=NULL, float priceIn=0);
        void  show();           // Giới thiệu xe
        ~Car();                 // Hàm hủy bỏ tường minh
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char *markIn, float priceIn){
    speed = speedIn;
    mark = markIn;
    price = priceIn;
}

void Car::show(){              // Phương thức giới thiệu xe
    cout << "This is a " << *mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

Car::~~Car(){                  // Hàm hủy bỏ tường minh
    delete [] mark;
    cout << "The object has been destroyed!" << endl;
}
```

```
// Hàm main, chương trình chính
void main() {
    clrscr();
    Car myCar(150, "Mercedes", 5000); // Dùng hàm khởi tạo đủ tham số

    // Giới thiệu xe
    cout << "Gioi thieu xe: " << endl;
    myCar.show();
    return;
}
```

Chương trình 5.5 sẽ in ra thông báo như sau:

```
Gioi thieu xe:
This is a Mercedes having a speed of 150km/h and its price is $5000
The object has been destroyed!
```

Dòng cuối cùng là của hàm hủy bỏ, mặc dù ta không gọi hàm hủy bỏ trực tiếp, nhưng khi thoát khỏi hàm main() của chương trình chính, đối tượng myCar bị giải phóng khỏi bộ nhớ và khi đó, C++ tự động gọi đến hàm hủy bỏ mà ta đã định nghĩa trước. Do vậy, mà có dòng thông báo cuối cùng này.

5.5 CON TRỎ ĐỐI TƯỢNG VÀ MẢNG ĐỐI TƯỢNG

5.5.1 Con trỏ đối tượng

Con trỏ đối tượng là con trỏ trỏ đến địa chỉ của một đối tượng có kiểu lớp. Các thao tác liên quan đến con trỏ đối tượng bao gồm:

- Khai báo con trỏ đối tượng
- Cấp phát bộ nhớ cho con trỏ đối tượng
- Sử dụng con trỏ đối tượng
- Giải phóng bộ nhớ cho con trỏ đối tượng

Khai báo con trỏ đối tượng

Con trỏ đối tượng được khai báo tương tự như khai báo các con trỏ có kiểu thông thường:

```
<Tên lớp> *<Tên con trỏ đối tượng>;
```

Ví dụ, muốn khai báo một con trỏ đối tượng có kiểu của lớp Car, ta khai báo như sau:

```
Car *myCar;
```

Khi đó, myCar là một con trỏ đối tượng có kiểu lớp Car.

Cấp phát bộ nhớ cho con trỏ đối tượng

Con trỏ đối tượng cũng cần phải cấp phát bộ nhớ hoặc trỏ vào một địa chỉ của một đối tượng lớp xác định trước khi được sử dụng. Cấp phát bộ nhớ cho con trỏ đối tượng cũng bằng thao tác **new**:

```
<Tên con trỏ đối tượng> = new <Tên lớp>([<Các đối số>]);
```

Ví dụ, nếu lớp Car có hai hàm khởi tạo như sau:

```
class Car{
public:
    Car();
    Car(int, char[], float);
};
```

thì ta có thể cấp phát bộ nhớ theo hai cách, tương ứng với hai hàm khởi tạo của lớp:

```
myCar = new Car(); // Khởi tạo không tham số
myCar = new Car(100, "Ford", 3000); // Khởi tạo đủ tham số
```

Lưu ý:

- Các đối số truyền phải tương ứng với ít nhất một trong các hàm khởi tạo của lớp.
- Khi sử dụng hàm khởi tạo không có tham số, ta vẫn phải sử dụng cặp ngoặc đơn “()” trong thao tác **new**.
- Khi lớp không có một hàm khởi tạo tường minh nào, sẽ dùng hàm khởi tạo ngầm định của C++ và cú pháp tương tự như sử dụng hàm khởi tạo tường minh không có tham số.
- Có thể vừa khai báo, vừa cấp phát bộ nhớ cho con trỏ đối tượng.

Ví dụ:

```
Car myCar = new Car(); // Khởi tạo không tham số
```

Sử dụng con trỏ đối tượng

Con trỏ đối tượng được sử dụng qua các thao tác:

- Trỏ đến địa chỉ của một đối tượng cùng lớp
- Truy nhập đến các phương thức của lớp

Con trỏ đối tượng có thể trỏ đến địa chỉ của một đối tượng có sẵn, cùng lớp theo cú pháp sau:

```
<Tên con trỏ đối tượng> = &<Tên đối tượng có sẵn>;
```

Ví dụ, ta có một con trỏ và một đối tượng của lớp Car:

```
Car *ptrCar, myCar(100, "Ford", 3000);
```

Khi đó, có thể cho con trỏ ptrCar trỏ vào địa chỉ của đối tượng myCar như sau:

```
ptrCar = &myCar;
```

Khi muốn truy nhập đến các thành phần của con trỏ đối tượng, ta dùng cú pháp sau:

```
<Tên con trỏ đối tượng> -> <Tên thành phần lớp>([<Các đối số>]);
```

Ví dụ, đoạn chương trình sau sẽ thực hiện phương thức giới thiệu xe của lớp Car thông qua con trỏ ptrCar:

```
Car *ptrCar = new Car(100, "Ford", 3000);
ptrCar->show();
```

Lưu ý:

- Danh sách các đối số phải tương thích với tên phương thức tương ứng.
- Các quy tắc phạm vi truy nhập vẫn áp dụng trong truy nhập các thành phần lớp thông qua con trỏ.

Giải phóng bộ nhớ cho con trỏ đối tượng

Con trỏ đối tượng cũng được giải phóng thông qua thao tác **delete**:

```
delete <Tên con trỏ đối tượng>;
```

Ví dụ:

```
Car *ptrCar = new Car(); // Khai báo và cấp phát bộ nhớ
...                      // Sử dụng con trỏ ptrCar
delete ptrCar;           // Giải phóng bộ nhớ.
```

Lưu ý:

- Thao tác **delete** chỉ được dùng khi trước đó, con trỏ được cấp phát bộ nhớ qua thao tác **new**:

```
Car *ptrCar = new Car();
delete ptrCar;           // Đúng.
```

Nhưng không được dùng **delete** khi trước đó, con trỏ chỉ trỏ vào một địa chỉ của đối tượng có sẵn (tĩnh):

```
Car *ptrCar, myCar(100, "Ford", 3000);
ptrCar = &myCar;
delete ptrCar;           // Không được
```

Chương trình 5.6 minh họa việc dùng con trỏ đối tượng có kiểu lớp là Car.

Chương trình 5.6

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

/* Định nghĩa lớp */
class Car{
private:
    int    speed;           // Tốc độ
    char   mark[20];        // Nhãn hiệu
    float  price;           // Giá xe
public:
    // Khởi tạo với các giá trị ngầm định cho các tham số
    Car(int speedIn=0, char markIn[]="", float priceIn=0);
    void  show();           // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
```

```
}

void Car::show() {                                // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main() {
    clrscr();
    // Khai báo con trỏ, cấp phát bộ nhớ dùng hàm khởi tạo đủ tham số
    Car *myCar = new Car(150, "Mercedes", 5000);

    // Giới thiệu xe
    cout << "Gioi thieu xe: " << endl;
    myCar->show();

    // Giải phóng con trỏ
    delete myCar;
    return;
}
```

Chương trình 5.6 hiển thị ra thông báo là một lời giới thiệu xe;

```
Gioi thieu xe:
This is a Mercedes having a speed of 150km/h and its price is $5000
```

5.5.2 Mảng các đối tượng

Mảng các đối tượng cũng có thể được khai báo và sử dụng như mảng của các biến có kiểu thông thường.

Khai báo mảng tĩnh các đối tượng

Mảng các đối tượng được khai báo theo cú pháp:

```
<Tên lớp> <Tên biến mảng>[<Số lượng đối tượng>];
```

Ví dụ:

```
Car cars[10];
```

Là khai báo một mảng có 10 đối tượng có cùng kiểu lớp Car.

Lưu ý:

- Có thể khai báo mảng tĩnh các đối tượng mà chưa cần khai báo độ dài mảng, cách này thường dùng khi chưa biết chính xác độ dài mảng:

```
Car cars[];
```


- Muốn khai báo được mảng tĩnh các đối tượng, lớp tương ứng phải có hàm khởi tạo không có tham số. Vì khi khai báo mảng, tương đương với khai báo một dãy các đối tượng với hàm khởi tạo không có tham số.

Khai báo mảng động với con trỏ

Một mảng các đối tượng cũng có thể được khai báo và cấp phát động thông qua con trỏ đối tượng như sau:

```
<Tên lớp> *<Tên biến mảng động> = new <Tên lớp>[<Độ dài mảng>];
```

Ví dụ:

```
Car *cars = new Car[10];
```

Sau khi được sử dụng, mảng động các đối tượng cũng cần phải giải phóng bộ nhớ:

```
delete [] <Tên biến mảng động>;
```

Ví dụ:

```
Car *cars = new Car[10]; // Khai báo và cấp phát động
...                      // Sử dụng biến mảng động
delete [] cars;          // Giải phóng bộ nhớ của mảng động
```

Sử dụng mảng đối tượng

Khi truy nhập vào các thành phần của một đối tượng có chỉ số xác định trong mảng đã khai báo, ta có thể sử dụng cú pháp:

```
<Tên biến mảng>[<Chỉ số đối tượng>].<Tên thành phần>([<Các đối số>]);
```

Ví dụ:

```
Car cars[10];
cars[5].show();
```

sẽ thực hiện phương thức show() của đối tượng có chỉ số thứ 5 (tính từ chỉ số 0) trong mảng cars.

Chương trình 5.7 sẽ cài đặt một chương trình, trong đó nhập vào độ dài mảng, sau đó yêu cầu người dùng nhập thông tin về mảng các xe. Cuối cùng, chương trình sẽ tìm kiếm và hiển thị thông tin về chiếc xe có giá đắt nhất trong mảng.

Chương trình 5.7

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        void   setSpeed(int);    // Gán tốc độ cho xe
```



```

        int    getSpeed();           // Đọc tốc độ xe
        void   setMark(char);        // Gán nhãn cho xe
        char[] getMark();           // Đọc nhãn xe
        void   setPrice(float);      // Gán giá cho xe
        float  getPrice();          // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void   show();              // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::setSpeed(int speedIn){    // Gán tốc độ cho xe
    speed = speedIn;
}

int Car::getSpeed(){               // Đọc tốc độ xe
    return speed;
}

void Car::setMark(char markIn){    // Gán nhãn cho xe
    strcpy(mark, markIn);
}

char[] Car::getMark(){            // Đọc nhãn xe
    return mark;
}

void Car::setPrice(float priceIn){ // Gán giá cho xe
    price = priceIn;
}

float Car::getPrice(){            // Đọc giá xe
    return price;
}

void Car::show(){                 // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính

```

```

void main() {
    clrscr();
    int length;           // Chiều dài mảng
    float maxPrice = 0;   // Giá đắt nhất
    int index = 0;        // Chỉ số của xe đắt nhất
    Car *cars;            // Khai báo mảng đối tượng

    // Nhập số lượng xe, tức là chiều dài mảng
    cout << "So luong xe: ";
    cin >> length;

    // Cấp phát bộ nhớ động cho mảng
    cars = new Car[length];

    // Khởi tạo các đối tượng trong mảng
    for(int i=0; i<length; i++){
        int speed;        // (Biến tạm) tốc độ
        char mark[20];     // (Biến tạm) nhãn hiệu
        float price;      // (Biến tạm) giá xe
        cout << "Xe thu " << i << ": " << endl;
        cout << "Toc do (km/h): ";
        cin >> speed; cars[i].setSpeed(speed);    // Nhập tốc độ
        cout << "Nhan hieu : ";
        cin >> mark; cars[i].setMark(mark);       // Nhập nhãn xe
        cout << "Gia ($): ";
        cin >> price; cars[i].setPrice(price);    // Nhập giá xe

        if(maxPrice < price){
            maxPrice = price;
            index = i;
        }
    }

    // Tìm xe đắt nhất
    for(int i=0; i<length; i++){
        if(i == index){
            cars[i].show(); // Giới thiệu xe đắt nhất
            break;
        }
    }

    // Giải phóng bộ nhớ của mảng
    delete [] cars;
    return;
}

```

```
}
```

TỔNG KẾT CHƯƠNG 5

Nội dung chương 5 đã tập trung trình bày các vấn đề cơ bản về lớp đối tượng trong ngôn ngữ C++:

- Khai báo, định nghĩa lớp bằng từ khóa `class`
- Sử dụng biến lớp như những đối tượng cụ thể
- Khai báo, định nghĩa các thuộc tính và các phương thức của lớp: định nghĩa các phương thức trong hoặc ngoài phạm vi khai báo lớp
- Khai báo phạm vi truy nhập lớp bằng các từ khóa chỉ phạm vi: `private`, `protected` và `public`. Giới thiệu các kiểu hàm có thể truy nhập phạm vi bất quy tắc: hàm bạn và lớp bạn với từ khóa `friend`.
- Khai báo, định nghĩa tường minh hàm khởi tạo của lớp và sử dụng khi khai báo biến lớp
- Khai báo, định nghĩa tường minh hàm hủy bỏ của lớp, nhằm dọn dẹp, giải phóng bộ nhớ trước khi đối tượng bị giải phóng khỏi bộ nhớ.
- Khai báo, cấp phát bộ nhớ, sử dụng và giải phóng bộ nhớ cho con trỏ đối tượng
- Khai báo, cấp phát bộ nhớ động, sử dụng và giải phóng vùng nhớ của mảng các đối tượng.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 5

1. Trong các khai báo lớp sau, những khai báo nào là đúng:
 - a. `class MyClass;`
 - b. `Class MyClass;`
 - c. `class MyClass{...};`
 - d. `Class MyClass{...};`
2. Giả sử ta đã định nghĩa lớp `MyClass`, bây giờ ta khai báo một đối tượng thuộc kiểu lớp này. Khai báo nào là đúng:
 - a. `class MyClass me;`
 - b. `MyClass me;`
 - c. `MyClass me();`
 - d. `MyClass me{};`
3. Trong các khai báo thuộc tính ngay trong phạm vi của khai báo lớp như sau, những khai báo nào là đúng:
 - a. `int myAge;`
 - b. `private int myAge;`
 - c. `public int myAge;`
 - d. `private: int myAge;`

4. Trong các khai báo phương thức ngay trong phạm vi của khai báo lớp MyClass như sau, những khai báo nào là đúng:

a. public: void show();
 b. void show();
 c. void show(){cout << "hello!";};
 d. void MyClass::show(){cout << "hello!";}

5. Xét đoạn chương trình sau:

```
class MyClass{
    int age;
    public:
        int getAge();
};
MyClass me;
```

Khi đó, trong các lệnh sau, lệnh nào có thể thêm vào cuối đoạn chương trình trên:

a. cout << MyClass::age;
 b. cout << me.age;
 c. cout << me.getAge();
 d. cin >> me.age;

6. Trong các khai báo hàm khởi tạo cho lớp MyClass như sau, những khai báo nào là đúng:

a. myClass();
 b. MyClass();
 c. MyClass(MyClass);
 d. void MyClas();
 e. MyClass MyClass();

7. Trong các khai báo hàm hủy bỏ cho lớp MyClass như sau, những khai báo nào là đúng:

a. ~myClass();
 b. ~MyClass();
 c. ~MyClass(MyClass);
 d. void ~MyClas();
 e. MyClass ~MyClass();

8. Giả sử ta khai báo lớp MyClass có một thuộc tính age và một hàm khởi tạo:

```
class MyClass{
    int age;
    public:
        MyClass(MyClass me){
            ...
        };
        int getAge(){return age};
};
```

Trong các dòng lệnh sau, những dòng nào có thể xuất hiện trong hàm khởi tạo trên:

- a. `age = MyClass.age;`
- b. `age = me.age;`
- c. `age = MyClass.getAge();`
- d. `age = me.getAge();`
- e. `age = 10;`

9. Xét khai báo lớp như sau:

```
class MyClass{
    public:
        MyClass(MyClass);
        MyClass(int);
};
```

Khi đó, trong số các khai báo đối tượng sau, những khai báo nào là đúng:

- a. `MyClass me;`
- b. `MyClass me();`
- c. `MyClass me(10);`
- d. `MyClass me1(10), me2(me1);`

10. Xét khai báo lớp như sau:

```
class MyClass{
    public:
        MyClass();
};
```

Khi đó, trong số các khai báo con trỏ đối tượng sau, những khai báo nào là đúng:

- a. `MyClass *me;`
- b. `MyClass *me();`
- c. `MyClass *me = new me();`
- d. `MyClass *me = new MyClass();`

11. Xét khai báo lớp như sau:

```
class MyClass{
    public:
        MyClass(int i=0, int j=0, float k=0);
};
```

Khi đó, trong số các khai báo con trỏ đối tượng sau, những khai báo nào là đúng:

- a. `MyClass *me;`
- b. `MyClass *me = new MyClass();`
- c. `MyClass *me = new MyClass(1, 20);`
- d. `MyClass *me = new MyClass(1, 20, 30);`

12. Khai báo một lớp nhân viên, có tên là `Employee`, với ba thuộc tính có tính chất `private`:

- Tên nhân viên, có dạng một con trỏ kiểu `char`

- Tuổi nhân viên, có kiểu int
 - Lương nhân viên, có kiểu float.
13. Thêm vào lớp Employee trong bài 12 các phương thức có tính chất public:
- set/get giá trị thuộc tính tên nhân viên
 - set/get giá trị thuộc tính tuổi nhân viên
 - set/get giá trị thuộc tính lương nhân viên.
14. Viết một hàm khởi tạo không có tham số cho lớp Employee trong bài 13, các thuộc tính của lớp nhận giá trị mặc định:
- giá trị thuộc tính tên nhân viên, mặc định là chuỗi kí tự rỗng ""
 - giá trị thuộc tính tuổi nhân viên, mặc định là 18
 - giá trị thuộc tính lương nhân viên, mặc định là \$100.
15. Viết thêm một hàm khởi tạo với đầy đủ ba tham số tương ứng với ba thuộc tính của lớp Employee trong bài 14.
16. Thêm vào lớp Employee một phương thức show() để giới thiệu về tên, tuổi và lương của đối tượng nhân viên.
17. Viết một hàm hủy bỏ tường minh cho lớp Employee nhằm giải phóng vùng nhớ của con trỏ char, là kiểu của thuộc tính tên nhân viên.
18. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo không tham số
 - Dùng hàm show() để giới thiệu về đối tượng đó
19. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo không tham số
 - Nhập từ bàn phím giá trị các thuộc tính tên, tuổi, lương nhân viên.
 - Gán các giá trị này cho các thuộc tính của đối tượng đã khai báo, dùng các hàm set
 - Dùng hàm show() để giới thiệu về đối tượng đó
20. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo với đủ 3 tham số
 - Dùng hàm show() để giới thiệu về đối tượng đó
21. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một con trỏ đối tượng kiểu Employee
 - Cấp phát bộ nhớ, dùng hàm khởi tạo với đủ 3 tham số
 - Dùng hàm show() để giới thiệu về đối tượng mà con trỏ này đang trỏ tới
22. Viết một chương trình nhập dữ liệu cho một mảng động các đối tượng của lớp Employee trong bài 17. Chiều dài mảng động cũng được nhập từ bàn phím.

23. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên trẻ nhất và nhân viên già nhất trong mảng đó.
24. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên có lương cao nhất và nhân viên có lương thấp nhất trong mảng đó.
25. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên có tên xác định, do người dùng nhập từ bàn phím.



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhkx@pit.edu.vn

CHƯƠNG 6

TÍNH KẾ THỪA VÀ ĐA HÌNH

Nội dung chương này tập trung trình bày các vấn đề liên quan đến tính kế thừa và tương ứng bội (đa hình) trong ngôn ngữ C++:

- Khái niệm kế thừa, dẫn xuất và các kiểu dẫn xuất
- Khai báo, định nghĩa các hàm khởi tạo và hàm hủy bỏ trong lớp dẫn xuất
- Truy nhập tới các thành phần của lớp cơ sở và các lớp dẫn xuất
- Việc một lớp được kế thừa từ nhiều lớp cơ sở khác nhau
- Khai báo và sử dụng các lớp cơ sở trừu tượng trong kế thừa
- Tính đa hình trong kế thừa

6.1 KHÁI NIỆM KẾ THỪA

Lập trình hướng đối tượng có hai đặc trưng cơ bản:

- Đóng gói dữ liệu, được thể hiện bằng cách dùng khái niệm lớp để biểu diễn đối tượng với các thuộc tính private, chỉ cho phép bên ngoài truy nhập vào thông qua các phương thức get/set.
- Dùng lại mã, thể hiện bằng việc thừa kế giữa các lớp. Việc thừa kế cho phép các lớp thừa kế (gọi là lớp dẫn xuất) sử dụng lại các phương thức đã được định nghĩa trong các lớp gốc (gọi là lớp cơ sở).

6.1.1 Khai báo thừa kế

Cú pháp khai báo một lớp kế thừa từ một lớp khác như sau:

```
class <Tên lớp dẫn xuất>: <Từ khóa dẫn xuất> <Tên lớp cơ sở>{
    ...           // Khai báo các thành phần lớp
};
```

Trong đó:

- **Tên lớp dẫn xuất:** là tên lớp được cho kế thừa từ lớp khác. Tên lớp này tuân thủ theo quy tắc đặt tên biến trong C++.
- **Tên lớp cơ sở:** là tên lớp đã được định nghĩa trước đó để cho lớp khác kế thừa. Tên lớp này cũng tuân thủ theo quy tắc đặt tên biến của C++.
- **Từ khóa dẫn xuất:** là từ khóa quy định tính chất của sự kế thừa. Có ba từ khóa dẫn xuất là private, protected và public. Mục tiếp theo sẽ trình bày ý nghĩa của các từ khóa dẫn xuất này.

Ví dụ:

```
class Bus: public Car{
    ...           // Khai báo các thành phần
};
```


là khai báo một lớp Bus (xe buýt) kế thừa từ lớp Car (xe ô tô) với tính chất kế thừa là public.

6.1.2 Tính chất dẫn xuất

Sự kế thừa cho phép trong lớp dẫn xuất có thể sử dụng lại một số mã nguồn của các phương thức và thuộc tính đã được định nghĩa trong lớp cơ sở. Nghĩa là lớp dẫn xuất có thể truy nhập trực tiếp đến một số thành phần của lớp cơ sở. Tuy nhiên, phạm vi truy nhập từ lớp dẫn xuất đến lớp cơ sở không phải bao giờ cũng giống nhau: chúng được quy định bởi các từ khóa dẫn xuất private, protected và public.

Dẫn xuất private

Dẫn xuất private quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần private của lớp dẫn xuất
- Các thành phần public của lớp cơ sở cũng trở thành các thành phần private của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Dẫn xuất protected

Dẫn xuất protected quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần protected của lớp dẫn xuất
- Các thành phần public của lớp cơ sở cũng trở thành các thành phần protected của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Dẫn xuất public

Dẫn xuất public quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần protected của lớp dẫn xuất.
- Các thành phần public của lớp cơ sở vẫn là các thành phần public của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Bảng 6.1 tóm tắt lại các quy tắc truy nhập được quy định bởi các từ khóa dẫn xuất.

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất
private	private	Không truy nhập được

	protected public	private private
protected	private protected public	Không truy nhập được protected protected
public	private protected public	Không truy nhập được protected public

6.2 HÀM KHỞI TẠO VÀ HUỖ BỎ TRONG KẾ THỪA

6.2.1 Hàm khởi tạo trong kế thừa

Khi khai báo một đối tượng có kiểu lớp được dẫn xuất từ một lớp cơ sở khác. Chương trình sẽ tự động gọi tới hàm khởi tạo của lớp dẫn xuất. Tuy nhiên, thứ tự được gọi sẽ bắt đầu từ hàm khởi tạo tương ứng của lớp cơ sở, sau đó đến hàm khởi tạo của lớp dẫn xuất. Do đó, thông thường, trong hàm khởi tạo của lớp dẫn xuất phải có hàm khởi tạo của lớp cơ sở.

Cú pháp khai báo hàm khởi tạo như sau:

```
<Tên hàm khởi tạo dẫn xuất>([<Các tham số>]):
    <Tên hàm khởi tạo cơ sở>([<Các đối số>]){
    ...    // Khởi tạo các thuộc tính mới bổ sung của lớp dẫn xuất
    };
```

Vì tên hàm khởi tạo là trùng với tên lớp, nên có thể viết lại thành:

```
<Tên lớp dẫn xuất>([<Các tham số>]):
    <Tên lớp cơ sở>([<Các đối số>]){
    ...    // Khởi tạo các thuộc tính mới bổ sung của lớp dẫn xuất
    };
```

Ví dụ:

```
Bus() : Car() {
    ...    // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
}
```

là một định nghĩa một hàm khởi tạo của lớp Bus kế thừa từ lớp Car. Định nghĩa này được thực hiện trong phạm vi khai báo lớp Bus. Đây là một hàm khởi tạo không tham số, nó gọi tới hàm khởi tạo không tham số của lớp Car.

Lưu ý:

- Nếu định nghĩa hàm khởi tạo bên ngoài phạm vi lớp thì phải thêm tên lớp dẫn xuất và toán tử phạm vi "::" trước tên hàm khởi tạo.
- Giữa tên hàm khởi tạo của lớp dẫn xuất và hàm khởi tạo của lớp cơ sở, chỉ có một dấu hai chấm ".", nếu là hai dấu "::" thì trở thành toán tử phạm vi lớp.

- Nếu không chỉ rõ hàm khởi tạo của lớp cơ sở sau dấu hai chấm “:” chương trình sẽ tự động gọi hàm khởi tạo ngầm định hoặc hàm khởi tạo không có tham số của lớp cơ sở nếu hàm đó được định nghĩa tường minh trong lớp cơ sở.

Ví dụ, định nghĩa hàm khởi tạo:

```
Bus():Car() {
    ...           // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
};
```

Có thể thay bằng:

```
Bus() {           // Gọi hàm khởi tạo không tham số của lớp Car
    ...           // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
};
```

Chương trình 6.1 định nghĩa lớp Car có 3 thuộc tính với hai hàm khởi tạo, sau đó định nghĩa lớp Bus có thêm thuộc tính label là số hiệu của tuyến xe buýt. Lớp Bus sẽ được cài đặt hai hàm khởi tạo tường minh, gọi đến hai hàm khởi tạo tương ứng của lớp Car.

Chương trình 6.1

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    int    speed;           // Tốc độ
    char   mark[20];        // Nhân hiệu
    float  price;           // Giá xe
public:
    Car();                 // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đủ tham số
};

Car::Car() {               // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
```

```
class Bus: public Car{
    int label; // Số hiệu tuyến xe
public:
    Bus(); // Khởi tạo không tham số
    Bus(int, char[], float, int); // Khởi tạo đủ tham số
};

Bus::Bus():Car(){ // Khởi tạo không tham số
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}
```

Trong hàm khởi tạo của lớp Bus, muốn khởi tạo các thuộc tính của lớp Car, ta phải khởi tạo gián tiếp thông qua hàm khởi tạo của lớp Car mà không thể gán giá trị trực tiếp cho các thuộc tính speed, mark và price. Lí do là các thuộc tính này có tính chất private, nên lớp dẫn xuất không thể truy nhập trực tiếp đến chúng.

6.2.2 Hàm hủy bỏ trong kế thừa

Khi một đối tượng lớp dẫn xuất bị giải phóng khỏi bộ nhớ, thứ tự gọi các hàm hủy bỏ ngược với thứ tự gọi hàm thiết lập: gọi hàm hủy bỏ của lớp dẫn xuất trước khi gọi hàm hủy bỏ của lớp cơ sở. Vì mỗi lớp chỉ có nhiều nhất là một hàm hủy bỏ, nên ta không cần phải chỉ ra hàm hủy bỏ nào của lớp cơ sở sẽ được gọi sau khi hủy bỏ lớp dẫn xuất. Do vậy, hàm hủy bỏ trong lớp dẫn xuất được khai báo và định nghĩa hoàn toàn giống với các lớp thông thường:

```
<Tên lớp>::~~<Tên lớp>([<Các tham số>]){
    ... // giải phóng phần bộ nhớ cấp phát cho các thuộc tính bổ sung
}
```

Lưu ý:

- Hàm hủy bỏ của lớp dẫn xuất chỉ giải phóng phần bộ nhớ được cấp phát động cho các thuộc tính mới bổ sung trong lớp dẫn xuất, nếu có, mà không được giải phóng bộ nhớ được cấp cho các thuộc tính trong lớp cơ sở (phần này là do hàm hủy bỏ của lớp cơ sở đảm nhiệm).
- Không phải gọi tường minh hàm hủy bỏ của lớp cơ sở trong hàm hủy bỏ của lớp dẫn xuất.
- Ngay cả khi lớp dẫn xuất không định nghĩa tường minh hàm hủy bỏ (do không cần thiết) mà lớp cơ sở lại có định nghĩa tường minh. Chương trình vẫn gọi hàm hủy bỏ ngầm định của lớp dẫn xuất, sau đó vẫn gọi hàm hủy bỏ tường minh của lớp cơ sở.

Chương trình 6.2 cài đặt lớp Bus kế thừa từ lớp Car: lớp Car có một thuộc tính có dạng con trỏ nên cần giải phóng bằng hàm hủy bỏ tường minh. Lớp Bus có thêm một thuộc tính có dạng con

trở là danh sách các đường phố mà xe buýt đi qua (mảng động các chuỗi kí tự `*char[]`) nên cũng cần giải phóng bằng hàm hủy bỏ tường minh.

Chương trình 6.2

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    char *mark;                // Nhãn hiệu xe
public:
    ~Car();                    // Hủy bỏ tường minh
};

Car::~Car() {                  // Hủy bỏ tường minh
    delete [] mark;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    char *voyage[];            // Hành trình tuyến xe
public:
    ~Bus();                    // Hủy bỏ tường minh
};

Bus::~Bus() {                  // Hủy bỏ tường minh
    delete [] voyage;
}
```

Trong hàm hủy bỏ của lớp Bus, ta chỉ được giải phóng vùng nhớ được cấp phát cho thuộc tính voyage (hành trình của xe buýt), là thuộc tính được bổ sung thêm của lớp Bus. Mà không được giải phóng vùng nhớ cấp phát cho thuộc tính mark (nhãn hiệu xe), việc này là thuộc trách nhiệm của hàm hủy bỏ của lớp Car vì thuộc tính mark được khai báo trong lớp Car.

6.3 TRUY NHẬP TỚI CÁC THÀNH PHẦN TRONG KẾ THỪA LỚP

6.3.1 Phạm vi truy nhập

Mối quan hệ giữa các thành phần của lớp cơ sở và lớp dẫn xuất được quy định bởi các từ khóa dẫn xuất, như đã trình bày trong mục 6.1.2, được tóm tắt trong bảng 6.2

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất
private	private protected	Không truy nhập được private

	public	private
protected	private protected public	Không truy nhập được protected protected
public	private protected public	Không truy nhập được protected public

Ta xét phạm vi truy nhập theo hai loại:

- Phạm vi truy nhập từ các hàm bạn, lớp bạn của lớp dẫn xuất
- Phạm vi truy nhập từ các đối tượng có kiểu lớp dẫn xuất

Truy nhập từ các hàm bạn và lớp bạn của lớp dẫn xuất

Nhìn vào bảng tổng kết 6.2, phạm vi truy nhập của hàm bạn, lớp bạn của lớp dẫn xuất vào lớp cơ sở như sau:

- Với dẫn xuất private, hàm bạn có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần private của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Với dẫn xuất protected, hàm bạn cũng có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần protected của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Với dẫn xuất public, hàm bạn cũng có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần protected và public của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Đối với cả ba loại dẫn xuất, hàm bạn đều không truy nhập được các thành phần private của lớp cơ sở, vì các thành phần này cũng không truy nhập được từ lớp dẫn xuất.

Truy nhập từ các đối tượng tạo bởi lớp dẫn xuất

Nhìn vào bảng tổng kết 6.2, phạm vi truy nhập của các đối tượng của lớp dẫn xuất vào lớp cơ sở như sau:

- Với dẫn xuất private, đối tượng của lớp dẫn xuất không truy nhập được bất cứ thành phần nào của lớp cơ sở vì chúng trở thành các thành phần private của lớp dẫn xuất, không truy nhập được từ bên ngoài.
- Với dẫn xuất protected, đối tượng của lớp dẫn xuất không truy nhập được bất cứ thành phần nào của lớp cơ sở vì chúng trở thành các thành phần protected của lớp dẫn xuất, không truy nhập được từ bên ngoài.
- Với dẫn xuất public, đối tượng của lớp dẫn xuất có thể truy nhập được các thành phần public của lớp cơ sở vì chúng trở thành các thành phần public của lớp dẫn xuất, có thể truy nhập được từ bên ngoài.

Bảng 6.3 tổng kết phạm vi truy nhập từ hàm bạn và đối tượng của lớp dẫn xuất vào các thành phần của lớp cơ sở, được quy định bởi các từ khóa dẫn xuất.

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất	Truy nhập từ hàm bạn của lớp dẫn xuất	Truy nhập từ đối tượng của lớp dẫn xuất
private	private protected public	--- private private	--- ok ok	--- --- ---
protected	private protected public	--- protected protected	--- ok ok	--- --- ---
public	private protected public	--- protected public	--- ok ok	--- --- ok

6.3.2 Sử dụng các thành phần của lớp cơ sở từ lớp dẫn xuất

Từ bảng tổng kết phạm vi truy nhập, ta thấy rằng chỉ có dẫn xuất theo kiểu public thì đối tượng của lớp dẫn xuất mới có thể truy nhập đến các thành phần (thuộc loại public) của lớp cơ sở. Khi đó, việc gọi đến các thành phần của lớp cơ sở cũng tương tự như gọi các thành phần lớp thông thường:

- Đối với biến đối tượng thông thường:
`<Tên đối tượng>.<Tên thành phần>([Các đối số]);`
- Đối với con trỏ đối tượng:
`<Tên đối tượng>-><Tên thành phần>([Các đối số]);`

Lưu ý:

- Cách gọi hàm thành phần này được áp dụng khi trong lớp dẫn xuất, ta không định nghĩa lại các hàm thành phần của lớp cơ sở. Trường hợp định nghĩa lại hàm thành phần của lớp cơ sở sẽ được trình bày trong mục 6.3.3.

Chương trình 6.3 minh họa việc sử dụng các thành phần lớp cơ sở từ đối tượng lớp dẫn xuất: lớp Bus kế thừa từ lớp Car. Lớp Bus có định nghĩa bổ sung một số phương thức và thuộc tính mới. Khi đó, đối tượng của lớp Bus có thể gọi các hàm public của lớp Bus cũng như của lớp Car.

Chương trình 6.3

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;                // Tốc độ
```

```

        char mark[20];                // Nhãn hiệu
        float price;                 // Giá xe

    public:
        void setSpeed(int);           // Gán tốc độ cho xe
        int getSpeed();               // Đọc tốc độ xe
        void setMark(char);           // Gán nhãn cho xe
        char[] getMark();             // Đọc nhãn xe
        void setPrice(float);          // Gán giá cho xe
        float getPrice();             // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void show();                  // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::setSpeed(int speedIn){      // Gán tốc độ cho xe
    speed = speedIn;
}

int Car::getSpeed(){                  // Đọc tốc độ xe
    return speed;
}

void Car::setMark(char markIn){       // Gán nhãn cho xe
    strcpy(mark, markIn);
}

char[] Car::getMark(){                // Đọc nhãn xe
    return mark;
}

void Car::setPrice(float priceIn){    // Gán giá cho xe
    price = priceIn;
}

float Car::getPrice(){                // Đọc giá xe
    return price;
}

void Car::show(){                     // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
}

```



```

        return;
    }

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label;                // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void setLabel(int);       // Gán số hiệu tuyến xe
    int getLabel();           // Đọc số hiệu tuyến xe
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

void Bus::setLabel(int labelIn){    // Gán số hiệu tuyến xe
    label = labelIn;
}

int Bus::getLabel(){                // Đọc số hiệu tuyến xe
    return label;
}

// Chương trình chính
void main(){
    clrscr();
    Bus myBus;                      // Biến đối tượng của lớp Bus
    int speedIn, labelIn;
    float priceIn;
    char markIn[20];

    // Nhập giá trị cho các thuộc tính
    cout << "Tốc độ xe bus:";
    cin >> speedIn;
    cout << "Nhập hiệu xe bus:";
    cin >> markIn;
    cout << "Giá xe bus:";
    cin >> priceIn;
    cout << "Số hiệu tuyến xe bus:";
    cin >> labelIn;

    myBus.setSpeed(speedIn);        // Phương thức của lớp Car

```

```

myBus.setMark(markIn);           // Phương thức của lớp Car
myBus.setPrice(priceIn);         // Phương thức của lớp Car
myBus.setLabel(labelIn);         // Phương thức của lớp Bus
myBus.show();                     // Phương thức của lớp Car
return;
}

```

Trong chương trình 6.3, đối tượng myBus có kiểu lớp Bus, là lớp dẫn xuất của lớp cơ sở Car, có thể sử dụng các phương thức của lớp Car và lớp Bus một cách bình đẳng. Khi đó, lệnh myBus.show() sẽ gọi đến phương thức show() của lớp Car, do vậy, chương trình trên sẽ in ra màn hình kết quả như sau (tùy theo dữ liệu nhập vào ở 4 dòng đầu):

```

Toc do xe bus: 80
Nhan hieu xe bus: Mercedes
Gia xe bus: 5000
So hieu tuyen xe bus: 27
This is a Mercedes having a speed of 80km/h and its price is $5000

```

Trong dòng giới thiệu xe bus (vì ta đang dùng đối tượng myBus của lớp Bus), không có giới thiệu số hiệu tuyến xe. Lí do là vì ta đang dùng hàm show của lớp Car. Muốn có thêm phần giới thiệu về số hiệu tuyến xe buýt, ta phải định nghĩa lại hàm show trong lớp Bus. Mục 6.3.3 sẽ trình bày nội dung này.

6.3.3 Định nghĩa chồng các phương thức của lớp cơ sở

Định nghĩa chồng phương thức của lớp cơ sở

Một phương thức của lớp cơ sở bị coi là nạp chồng nếu ở lớp dẫn xuất cũng định nghĩa một phương thức có cùng tên.

Ví dụ, trong lớp Car, đã có phương thức show(), bây giờ, trong lớp Bus kế thừa từ lớp Car, ta cũng định nghĩa lại phương thức show():

```

class Car{
    ...
public:
    ...
    void show();           // Phương thức của lớp cơ sở
};

class Bus: public Car{
    ...
public:
    ...
    void show();           // Phương thức nạp chồng
};

```

khi đó, phương thức show() của lớp Bus được coi là phương thức nạp chồng từ phương thức show() của lớp Car.

Sử dụng các phương thức nạp chồng

Từ một đối tượng của lớp dẫn xuất, việc truy nhập đến phương thức đã được định nghĩa lại trong lớp dẫn xuất được thực hiện như lời gọi một phương thức thông thường:

- Đối với biến đối tượng thông thường:
`<Tên đối tượng>.<Tên thành phần>([Các đối số]);`
- Đối với con trỏ đối tượng:
`<Tên đối tượng>-><Tên thành phần>([Các đối số]);`

Ví dụ:

```
Bus myBus;  
myBus.show();
```

sẽ gọi đến phương thức `show()` được định nghĩa trong lớp `Bus`.

Trong trường hợp, từ một đối tượng của lớp dẫn xuất, muốn truy nhập đến một phương thức của lớp cơ sở (đã bị định nghĩa lại ở lớp dẫn xuất) thì phải sử dụng chỉ thị phạm vi lớp trước phương thức được gọi:

- Đối với biến đối tượng thông thường:
`<Tên đối tượng>.<Tên lớp cơ sở>::<Tên thành phần>([Các đối số]);`
- Đối với con trỏ đối tượng:
`<Tên đối tượng>-><Tên lớp cơ sở>::<Tên thành phần>([Các đối số]);`

Ví dụ:

```
Bus myBus;  
myBus.Car::show();
```

sẽ gọi đến phương thức `show()` được định nghĩa trong lớp `Car` từ một đối tượng của lớp `Bus`.

Chương trình 6.4 minh họa việc định nghĩa chồng hàm `show()` trong lớp `Bus` và việc sử dụng hai phương thức `show()` của hai lớp từ một đối tượng của lớp dẫn xuất.

Chương trình 6.4

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp Car */  
class Car{  
    private:  
        int    speed;           // Tốc độ  
        char   mark[20];        // Nhãn hiệu  
        float  price;           // Giá xe  
    public:  
        int    getSpeed();       // Đọc tốc độ xe  
        char[] getMark();        // Đọc nhãn xe  
        float  getPrice();       // Đọc giá xe  
        // Khởi tạo thông tin về xe
```

```

        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void show(); // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::setSpeed(int speedIn){ // Gán tốc độ cho xe
    speed = speedIn;
}

int Car::getSpeed(){ // Đọc tốc độ xe
    return speed;
}

char[] Car::getMark(){ // Đọc nhãn xe
    return mark;
}

float Car::getPrice(){ // Đọc giá xe
    return price;
}

void Car::show(){ // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label; // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void show(); // Giới thiệu xe bus
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

```

```
// Định nghĩa nạp chồng phương thức
void Bus::show() { // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl;
    return;
}

// Chương trình chính
void main() {
    clrscr();
    Bus myBus(80, "Mercedes", 5000, 27); // Biến đối tượng của lớp Bus

    cout << "Gioi thieu xe:" << endl;
    myBus.Car::show(); // Phương thức của lớp Car
    cout << "Gioi thieu xe bus:" << endl;
    myBus.show(); // Phương thức của lớp Bus
    return;
}
```

Chương trình 6.4 sẽ hiển thị các thông báo như sau:

```
Gioi thieu xe:
This is a Mercedes having a speed of 80km/h and its price is $5000
Gioi thieu xe bus:
This is a bus of type Mercedes, on the line 27, having a speed of
80km/h and its price is $5000
```

Lưu ý:

- Trong phương thức show() của lớp Bus, ta phải dùng các hàm get để truy nhập đến các thuộc tính của lớp Car. Không được truy nhập trực tiếp đến tên các thuộc tính (speed, mark và price) vì chúng có dạng private của lớp Car.

6.3.4 Chuyển đổi kiểu giữa lớp cơ sở và lớp dẫn xuất

Về mặt dữ liệu, một lớp dẫn xuất bao giờ cũng chứa toàn bộ dữ liệu của lớp cơ sở. Ta luôn tìm thấy lớp cơ sở trong lớp dẫn xuất, nhưng không phải bao giờ cũng tìm thấy lớp dẫn xuất trong lớp cơ sở. Do vậy:

- Có thể gán một đối tượng lớp dẫn xuất cho một đối tượng lớp cơ sở:
`<Đối tượng lớp cơ sở> = <Đối tượng lớp dẫn xuất>; // Đúng`
- Nhưng không thể gán một đối tượng lớp cơ sở cho một đối tượng lớp dẫn xuất:
`<Đối tượng lớp dẫn xuất> = <Đối tượng lớp cơ sở>; // Không được`

Ví dụ, ta có lớp Bus kế thừa từ lớp Car và:

```
Bus myBus;
```

```
Car myCar;
```

khi đó, phép gán:

```
myCar = myBus;    // đúng
```

thì chấp nhận được, nhưng phép gán:

```
myBus = myCar;    // không được
```

thì không chấp nhận được.

Lưu ý:

- Nguyên tắc chuyển kiểu này cũng đúng với các phép gán con trỏ: một con trỏ đối tượng lớp cơ sở có thể trỏ đến địa chỉ của một đối tượng lớp dẫn xuất. Nhưng một con trỏ đối tượng lớp dẫn xuất không thể trỏ đến địa chỉ một đối tượng lớp cơ sở.
- Nguyên tắc chuyển kiểu này cũng đúng với truyền đối số cho hàm: có thể truyền một đối tượng lớp dẫn xuất vào vị trí của tham số có kiểu lớp cơ sở. Nhưng không thể truyền một đối tượng lớp cơ sở vào vị trí một tham số có kiểu lớp dẫn xuất.

Chương trình 6.5 minh họa việc chuyển kiểu giữa các đối tượng của lớp cơ sở và lớp dẫn xuất.

Chương trình 6.5

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        int    getSpeed();       // Đọc tốc độ xe
        char[] getMark();        // Đọc nhãn xe
        float  getPrice();       // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void   show();           // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```

```

int Car::getSpeed(){ // Đọc tốc độ xe
    return speed;
}
char[] Car::getMark(){ // Đọc nhãn xe
    return mark;
}
float Car::getPrice(){ // Đọc giá xe
    return price;
}

void Car::show(){ // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label; // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void show(); // Định nghĩa chồng phương thức
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

// Định nghĩa nạp chồng phương thức
void Bus::show(){ // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl;
    return;
}

// Chương trình chính
void main(){
    clrscr();
    Car myCar(100, "Ford", 3000); // Biến đối tượng lớp Car
    Bus myBus(80, "Mercedes", 5000, 27); // Biến đối tượng lớp Bus
}

```



```

        cout << "Gioi thieu xe o to lan 1:" << endl;
        myCar.show();
        cout << "Gioi thieu xe o to lan 2:" << endl;
        myCar = myBus;
        myCar.show();
        cout << "Gioi thieu xe bus:" << endl;
        myBus.show();
        return;
    }

```

Chương trình 6.5 sẽ in ra kết quả thông báo như sau:

Goi thieu xe o to lan 1:

This is a Ford having a speed of 100km/h and its price is \$3000

Gioi thieu xe lan o to lan 2:

This is a Mercedes having a speed of 80km/h and its price is \$5000

Goi thieu xe bus:

This is a bus of type Mercedes, on the line 27, having a speed of 80km/h and its price is \$5000

Ở thông báo thứ nhất, đối tượng myCar gọi phương thức show() của lớp Car với các dữ liệu được khởi đầu cho myCar: (100, "Ford", 3000). Ở thông báo thứ hai, myCar gọi phương thức show() của lớp Car, nhưng với dữ liệu vừa được gán từ đối tượng myBus: (80, "Mercedes", 5000). Ở thông báo thứ ba, myBus gọi phương thức show() của lớp Bus với các dữ liệu của myBus: (80, "Mercedes", 5000, 27).

6.4 ĐA KẾ THỪA

C++ cho phép đa kế thừa, tức là một lớp có thể được dẫn xuất từ nhiều lớp cơ sở khác nhau, với những kiểu dẫn xuất khác nhau.

6.4.1 Khai báo đa kế thừa

Đa kế thừa được khai báo theo cú pháp:

```

class <Tên lớp dẫn xuất>: <Từ khoá dẫn xuất> <Tên lớp cơ sở 1>,
    <Từ khoá dẫn xuất> <Tên lớp cơ sở 2>,
    ...
    <Từ khoá dẫn xuất> <Tên lớp cơ sở n>{
    ...    // Khai báo thêm các thành phần lớp dẫn xuất
};

```

Ví dụ:

```

class Bus: public Car, public PublicTransport{
    ...    // Khai báo các thành phần bổ sung
};

```


là khai báo lớp Bus (xe buýt) kế thừa từ hai lớp xe Car (ô tô) và PublicTransport (phương tiện giao thông công cộng) theo cùng một kiểu dẫn xuất là public.

Lưu ý:

- Trong đa kế thừa, mỗi lớp cơ sở được phân cách nhau bởi dấu phẩy “,”.
- Mỗi lớp cơ sở cơ thể có một kiểu dẫn xuất bởi một từ khoá dẫn xuất khác nhau.
- Nguyên tắc truy nhập vào các thành phần lớp cơ sở cũng hoàn toàn tương tự như trong kế thừa đơn.

6.4.2 Hàm khởi tạo và hàm huỷ bỏ trong đa kế thừa

Hàm khởi tạo trong đa kế thừa

Hàm khởi tạo trong đa kế thừa được khai báo tương tự như trong đơn kế thừa, ngoại trừ việc phải sắp xếp thứ tự gọi tới hàm khởi tạo của các lớp cơ sở: thông thường, thứ tự gọi đến hàm khởi tạo của các lớp cơ sở nên tuân theo thứ tự dẫn xuất từ các lớp cơ sở trong đa kế thừa.

Chương trình 6.6 minh hoạ việc định nghĩa hàm khởi tạo tường minh trong đa kế thừa: thứ tự gọi hàm khởi tạo của các lớp cơ sở trong hàm khởi tạo của lớp Bus là tương tự thứ tự dẫn xuất: hàm khởi tạo của lớp Car trước hàm khởi tạo của lớp PublicTransport.

Chương trình 6.6

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    int    speed;           // Tốc độ
    char   mark[20];        // Nhãn hiệu
    float  price;           // Giá xe
public:
    Car();                  // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đủ tham số
};

Car::Car() {                // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```

```

}

/* Định nghĩa lớp PublicTransport */
class PublicTransport{
    float ticket;                // Giá vé phương tiện

public:
    PublicTransport();           // Khởi tạo không tham số
    PublicTransport(float);      // Khởi tạo đủ tham số
};

PublicTransport::PublicTransport(){ // Khởi tạo không tham số
    ticket = 0;
}

// Khởi tạo đủ tham số
PublicTransport::PublicTransport(float ticketIn){
    ticket = ticketIn;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */
class Bus: public Car, public PublicTransport{ // Thứ tự khai báo
    int label;                                // Số hiệu tuyến xe

public:
    Bus();                                    // Khởi tạo không tham số
    Bus(int, char[], float, float, int);    // Khởi tạo đủ tham số
};

// Khởi tạo không tham số
Bus::Bus(): Car(), Transport(){              // Theo thứ tự dẫn xuất
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, float tIn, int lIn):
    Car(sIn, mIn, pIn), PublicTransport(tIn){ // Theo thứ tự dẫn xuất
    label = lIn;
}

```

Lưu ý:

- Trong trường hợp dùng hàm khởi tạo ngầm định hoặc không có tham số, ta có thể không cần gọi tường minh các hàm khởi tạo của các lớp cơ sở, trình biên dịch sẽ tự động gọi tới chúng theo đúng thứ tự dẫn xuất

Ví dụ, trong chương trình 6.6, hai cách định nghĩa hàm khởi tạo không tham số của lớp Bus sau là tương đương:

```
Bus::Bus(): Car(), Transport() { // Theo thứ tự dẫn xuất
    label = 0;
}
```

là tương đương với:

```
Bus::Bus() { // Theo thứ tự dẫn xuất ngầm định
    label = 0;
}
```

Hàm hủy bỏ trong đa kế thừa

Vì hàm hủy bỏ là duy nhất của mỗi lớp, hơn nữa hàm hủy bỏ của lớp cơ sở sẽ được tự động gọi đến khi giải phóng đối tượng của lớp dẫn xuất. Cho nên hàm hủy bỏ trong đa kế thừa hoàn toàn tương tự hàm hủy bỏ trong đơn kế thừa:

- Hàm hủy bỏ của lớp dẫn xuất chỉ giải phóng bộ nhớ cho các thành phần bổ sung, nếu có, của lớp dẫn xuất.
- Hàm hủy bỏ của lớp dẫn xuất sẽ được gọi đến sớm nhất. Sau đó các hàm hủy bỏ của các lớp cơ sở sẽ được gọi đến.
- Quá trình này được trình biên dịch thực hiện tự động.

6.4.3 Truy nhập các thành phần lớp trong đa kế thừa

Việc truy nhập đến các thành phần của các lớp trong đa kế thừa được dựa trên các nguyên tắc sau:

- Việc truy nhập từ đối tượng lớp dẫn xuất đến các thành phần của mỗi lớp cơ sở được tuân theo quy tắc phạm vi tương tự như trong đơn kế thừa.
- Trong trường hợp các lớp cơ sở đều có các thành phần cùng tên, việc truy xuất đến thành phần của lớp nào phải được chỉ rõ bằng toán tử phạm vi: “<Tên lớp>::” đối với thành phần lớp cơ sở đó.

Ví dụ, ta định nghĩa lớp Bus kế thừa từ hai lớp cơ sở: Car và PublicTransport. Nhưng cả ba lớp này đều định nghĩa một phương thức show() để tự giới thiệu:

```
class Car{
public:
    void show();
};
class PublicTransport{
public:
    void show();
};
class Bus: public Car, public PublicTransport{
public:
    void show();
};
```

Khi đó, khai báo:

```
Bus myBus;
```

và lời gọi hàm:

```
myBus.show(); // Gọi đến hàm của lớp Bus
myBus.Car::show(); // Gọi đến hàm của lớp Car
myBus.PublicTransport::show(); // Gọi đến hàm của lớp PublicTransport
```

Chương trình 6.7 minh họa việc truy nhập đến các thành phần trùng nhau trong các lớp cơ sở và được định nghĩa lại trong lớp dẫn xuất.

Chương trình 6.7

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

/* Định nghĩa lớp Car */
class Car{
    int    speed;           // Tốc độ
    char   mark[20];        // Nhãn hiệu
    float  price;           // Giá xe
public:
    Car();                  // Khởi tạo không tham số
    Car(int, char[], float); // Khởi tạo đủ tham số
    void show();            // Giới thiệu
    float getSpeed(){return speed;};
    char[] getMark(){return mark;};
    float getPrice(){return price;};
};

Car::Car() {                // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```

```
// Giới thiệu
void Car::show(){
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

/* Định nghĩa lớp PublicTransport */
class PublicTransport{
    float ticket;                // Giá vé phương tiện
public:
    PublicTransport();           // Khởi tạo không tham số
    PublicTransport(float);      // Khởi tạo đủ tham số
    void show();                // Giới thiệu
    float getTicket(){return ticket;};
};

PublicTransport::PublicTransport(){ // Khởi tạo không tham số
    ticket = 0;
}

// Khởi tạo đủ tham số
PublicTransport::PublicTransport(float ticketIn){
    ticket = ticketIn;
}

// Giới thiệu
void PublicTransport::show(){
    cout << "This public transport had a ticket of $"
        << ticket << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */
class Bus: public Car, public PublicTransport{ // Thứ tự khai báo
    int label;                                // Số hiệu tuyến xe
public:
    Bus();                                    // Khởi tạo không tham số
    Bus(int, char[], float, float, int);    // Khởi tạo đủ tham số
    void show();                             // Giới thiệu
};

// Khởi tạo không tham số
```

```

Bus::Bus(): Car(), Transport() { // Theo thứ tự dẫn xuất
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, float tIn, int lIn):
    Car(sIn, mIn, pIn), PublicTransport(tIn) { // Theo thứ tự dẫn xuất
    label = lIn;
}

// Giới thiệu
void Bus::show() {
    cout << "This is a bus on the line " << label
        << ", its speed is " << getSpeed()
        << "km/h, mark is " << getMark()
        << ", price is $" << getPrice()
        << " and ticket is " << getTicket() << endl;
    return;
}

// phương thức main
void main() {
    clrscr();
    Bus myBus(100, "Mercedes", 3000, 1.5, 27);

    myBus.Car::show(); // Hàm của lớp Car
    myBus.PublicTransport::show(); // Hàm của lớp PublicTransport
    myBus.show(); // Hàm của lớp Bus
    return;
}

```

Chương trình 6.7 sẽ in ra thông báo như sau:

```

This is a Mercedes having a speed of 100km/h and its price is $3000
This public transport had a ticket of $1.5
This is a bus on the line 27, its speed is 100km/h, mark is Mercedes,
price is $3000 and ticket is $1.5

```

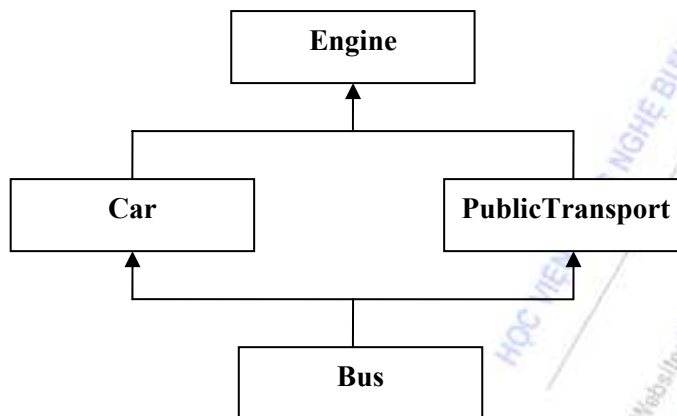
Dòng thứ nhất là kết quả của phương thức show() của lớp Car, dòng thứ hai, tương ứng là kết quả phương thức show() của lớp PublicTransport, dòng thứ ba là kết quả phương thức show() của lớp Bus.

6.5 LỚP CƠ SỞ TRỪU TƯỢNG

6.5.1 Đặt vấn đề

Sự cho phép đa kế thừa trong C++ dẫn đến một số hậu quả xấu, đó là sự đụng độ giữa các thành phần của các lớp cơ sở, khi có ít nhất hai lớp cơ sở lại cùng được kế thừa từ một lớp cơ sở khác. Xét trường hợp:

- Lớp Bus kế thừa từ lớp Car và lớp PublicTransport.
- Nhưng lớp Car và lớp PublicTransport lại cùng được thừa kế từ lớp Engine (động cơ).
Lớp Engine có một thuộc tính là power (công suất của động cơ).



Khi đó, nảy sinh một số vấn đề như sau:

- Các thành phần dữ liệu của lớp Engine bị lặp lại trong lớp Bus hai lần: một lần do kế thừa theo đường Bus::Car::Engine, một lần theo đường Bus::PublicTransport::Engine. Điều này là không an toàn.
- Khi khai báo một đối tượng của lớp Bus, hàm khởi tạo của lớp Engine cũng được gọi hai lần: một lần do gọi truy hồi từ hàm khởi tạo lớp Car, một lần do gọi truy hồi từ hàm khởi tạo lớp PublicTransport.
- Khi giải phóng một đối tượng của lớp Bus, hàm huỷ bỏ của lớp Engine cũng sẽ bị gọi tới hai lần.

Để tránh các vấn đề này, C++ cung cấp một khái niệm là kế thừa từ *lớp cơ sở trừu tượng*. Khi đó, ta cho các lớp Car và PublicTransport kế thừa trừu tượng từ lớp Engine. Bằng cách này, các thành phần của lớp Engine chỉ xuất hiện trong lớp Bus đúng một lần. Lớp Engine được gọi là lớp cơ sở trừu tượng của các lớp Car và PublicTransport.

6.5.2 Khai báo lớp cơ sở trừu tượng

Việc chỉ ra một sự kế thừa trừu tượng được thực hiện bằng từ khoá **virtual** khi khai báo lớp cơ sở:

```

class <Tên lớp cơ sở>: <Từ khoá dẫn xuất> virtual <Tên lớp cơ sở>{
    ...           // Khai báo các thành phần bổ sung
};
    
```

Ví dụ:

```

class Engine{
    ...           // Các thành phần lớp Engine
};
    
```



```
};
class Car: public virtual Engine{
    ...    // Khai báo các thành phần bổ sung
};
```

là khai báo lớp Car, kế thừa từ lớp cơ sở trừu tượng Engine, theo kiểu dẫn xuất public.

Lưu ý:

- Từ khoá virtual được viết bằng chữ thường.
- Từ khoá virtual không ảnh hưởng đến phạm vi truy nhập thành phần lớp cơ sở, phạm vi này vẫn được quy định bởi từ khoá dẫn xuất như thông thường.
- Từ khoá virtual chỉ ra một lớp cơ sở là trừu tượng nhưng lại được viết trong khi khai báo lớp dẫn xuất.
- Một lớp dẫn xuất có thể được kế thừa từ nhiều lớp cơ sở trừu tượng

6.5.3 Hàm khởi tạo lớp cơ sở trừu tượng

Khác với các lớp cơ sở thông thường, khi có một lớp dẫn xuất từ một lớp cơ sở trừu tượng, lại được lấy làm cơ sở cho một lớp dẫn xuất khác thì trong hàm khởi tạo của lớp dẫn xuất cuối cùng, vẫn phải gọi hàm khởi tạo tường minh của lớp cơ sở trừu tượng. Hơn nữa, hàm khởi tạo của lớp cơ sở trừu tượng phải được gọi sớm nhất.

Ví dụ, khi lớp Car và lớp PublicTransport được kế thừa từ lớp cơ sở trừu tượng Engine. Sau đó, lớp Bus được kế thừa từ hai lớp Car và PublicTransport. Khi đó, hàm khởi tạo của lớp Bus cũng phải gọi tường minh hàm khởi tạo của lớp Engine, theo thứ tự sớm nhất, sau đó mới gọi đến hàm khởi tạo của các lớp Car và PublicTransport.

```
class Engine{
    public:
        Engine() {... };
};

class Car: public virtual Engine{                //Lớp cơ sở virtual
    public:
        Car(): Engine() {... };
};

class PublicTransport: public virtual Engine{    //Lớp cơ sở virtual
    public:
        PublicTransport():Engine() {... };
};

class Bus: public Car, public PublicTransport{
    public:
        // Gọi hàm khởi tạo tường minh của lớp cơ sở trừu tượng
        Bus():Engine(), Car(), PublicTransport() {... };
};
```

Lưu ý:

- Trong trường hợp lớp Engine không phải là lớp cơ sở trừu tượng của các lớp Car và PublicTransport, thì trong hàm khởi tạo của lớp Bus không cần gọi hàm khởi tạo của lớp

Engine, mà chỉ cần gọi tới các hàm khởi tạo của các lớp cơ sở trực tiếp của lớp Bus là lớp Car và lớp PublicTransport.

Chương trình 6.8 minh họa việc khai báo và sử dụng lớp cơ sở trừu tượng: lớp Engine là lớp cơ sở trừu tượng của các lớp Car và lớp PublicTransport. Hai lớp này, sau đó, lại làm lớp cơ sở của lớp Bus.

Chương trình 6.8

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

/* Định nghĩa lớp Engine */
class Engine{
    int    power;                // Công suất
public:
    Engine(){power = 0;};        // Khởi tạo không tham số
    Engine(int pIn){power = pIn;}; // Khởi tạo đủ tham số
    void show();                // Giới thiệu
    float getPower(){return power;};
};

// Giới thiệu
void Engine::show(){
    cout << "This is an engine having a power of "
        << power << "KWH" << endl;
    return;
}

/* Định nghĩa lớp Car dẫn xuất từ lớp cơ sở trừu tượng Engine*/
class Car: public virtual Engine{
    int    speed;                // Tốc độ
    char    mark[20];            // Nhãn hiệu
    float price;                // Giá xe
public:
    Car();                      // Khởi tạo không tham số
    Car(int, int, char[], float); // Khởi tạo đủ tham số
    void show();                // Giới thiệu
    float getSpeed(){return speed;};
    char[] getMark(){return mark;};
    float getPrice(){return price;};
};
```

```

Car::Car(): Engine() {                                     // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int pwIn, int sIn, char mIn[], float prIn): Engine(pwIn){
    speed = sIn;
    strcpy(mark, mIn);
    price = prIn;
}

// Giới thiệu
void Car::show() {
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h, its power is" << getPower()
        << "KWh and price is $" << price << endl;
    return;
}

/* Định nghĩa lớp PublicTransport dẫn xuất trừu tượng từ lớp Engine */
class PublicTransport: public virtual Engine{
    float ticket;                                           // Giá vé phương tiện
public:
    PublicTransport();                                     // Khởi tạo không tham số
    PublicTransport(int, float);                          // Khởi tạo đủ tham số
    void show();                                           // Giới thiệu
    float getTicket(){return ticket;};
};

// Khởi tạo không tham số
PublicTransport::PublicTransport(): Engine(){
    ticket = 0;
}

// Khởi tạo đủ tham số
PublicTransport::PublicTransport(int pwIn, float tIn): Engine(pwIn){
    ticket = tIn;
}

// Giới thiệu
void PublicTransport::show() {

```

```

        cout << "This is a public transport havin a ticket of $"
            << ticket << " and its power is " << getPower()
            << "KWh" << endl;
    return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */
class Bus: public Car, public PublicTransport{ // Thứ tự khai báo
    int label; // Số hiệu tuyến xe
public:
    Bus(); // Khởi tạo không tham số
    Bus(int,int,char[],float,float,int); // Khởi tạo đủ tham số
    void show(); // Giới thiệu
};

// Khởi tạo không tham số
Bus::Bus(): Engine(), Car(), Transport(){ // Theo thứ tự dẫn xuất
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int pwIn, int sIn, char mIn[], float prIn, float tIn, int lIn):
    Engine(pwIn), Car(sIn, mIn, prIn), PublicTransport(tIn){
    label = lIn;
}

// Giới thiệu
void Bus::show(){
    cout << "This is a bus on the line " << label
        << ", its speed is " << getSpeed()
        << "km/h, power is" << Car::getPower()
        << "KWh, mark is " << getMark()
        << ", price is $" << getPrice()
        << " and ticket is " << getTicket() << endl;
    return;
}

// phương thức main
void main(){
    clrscr();
    Bus myBus(250, 100, "Mercedes", 3000, 1.5, 27);

    myBus.Car::Engine::show(); // Hàm của lớp Engine
}

```

```
myBus.PublicTransport::Engine::show();// Hàm của lớp Engine
myBus.Car::show();                      // Hàm của lớp Car
myBus.PublicTransport:: show();         // Hàm của lớp PublicTransport
myBus.show();                           // Hàm của lớp Bus
return;
}
```

Chương trình 6.8 sẽ in ra thông báo như sau:

```
This is an engine having a power of 250KWh
This is an engine having a power of 250KWh
This is a Mercedes having a speed of 100km/h, its power is 250KWh and
price is $3000
This is a public transport having a ticket of $1.5 and its power is
250KWh
This is a bus on the line 27, its speed is 100km/h, power is 250KWh,
mark is Mercedes, price is $3000 and ticket is $1.5
```

Hai dòng đầu là kết quả của phương thức show() của lớp Engine: một lần gọi qua lớp Car, một lần gọi qua lớp PublicTransport, chúng cho kết quả như nhau. Dòng thứ ba là kết quả phương thức show() của lớp Car. Dòng thứ tư, tương ứng là kết quả phương thức show() của lớp PublicTransport. Dòng thứ năm là kết quả phương thức show() của lớp Bus.

6.6 ĐA HÌNH

6.6.1 Đặt vấn đề

Sự kế thừa trong C++ cho phép có sự tương ứng giữa lớp cơ sở và các lớp dẫn xuất trong sơ đồ thừa kế:

- Một con trỏ có kiểu lớp cơ sở luôn có thể trỏ đến địa chỉ của một đối tượng của lớp dẫn xuất.
- Tuy nhiên, khi thực hiện lời gọi một phương thức của lớp, trình biên dịch sẽ quan tâm đến kiểu của con trỏ chứ không phải đối tượng mà con trỏ đang trỏ tới: phương thức của lớp mà con trỏ có kiểu được gọi chứ không phải phương thức của đối tượng mà con trỏ đang trỏ tới được gọi.

Ví dụ, lớp Bus kế thừa từ lớp Car, cả hai lớp này đều định nghĩa phương thức show():

```
class Car{
public:
    void show();
};
class Bus: public Car{
public:
    void show();
};
```

khi đó, nếu ta khai báo một con trỏ lớp Bus, nhưng lại trỏ vào địa chỉ của một đối tượng lớp Car:

```
Bus myBus;  
Car *ptrCar = &myBus;    // đúng
```

nhưng khi gọi:

```
ptrCar->show();
```

thì chương trình sẽ gọi đến phương thức `show()` của lớp `Car` (là kiểu của con trỏ `ptrCar`), mà không gọi tới phương thức `show()` của lớp `Bus` (là kiểu của đối tượng `myBus` mà con trỏ `ptrCar` đang trỏ tới).

Để giải quyết vấn đề này, C++ đưa ra một khái niệm là phương thức trừu tượng. Bằng cách sử dụng phương thức trừu tượng. Khi gọi một phương thức từ một con trỏ đối tượng, trình biên dịch sẽ xác định kiểu của đối tượng mà con trỏ đang trỏ đến, sau đó nó sẽ gọi phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới.

6.6.2 Khai báo phương thức trừu tượng

Phương thức trừu tượng (còn gọi là phương thức ảo, hàm ảo) được khai báo với từ khoá **virtual**:

- Nếu khai báo trong phạm vi lớp:

```
virtual <Kiểu trả về> <Tên phương thức>([<Các tham số>]);
```

- Nếu định nghĩa ngoài phạm vi lớp:

```
virtual <Kiểu trả về> <Tên lớp>::<Tên phương thức>([<Các tham số>]) {...}
```

Ví dụ:

```
class Car{  
    public:  
        virtual void show();  
};
```

là khai báo phương thức trừu tượng `show()` của lớp `Car`: phương thức không có tham số và không cần giá trị trả về (`void`).

Lưu ý:

- Từ khoá `virtual` có thể đặt trước hay sau kiểu trả về của phương thức.
- Với cùng một phương thức được khai báo ở lớp cơ sở lẫn lớp dẫn xuất, chỉ cần dùng từ khoá `virtual` ở một trong hai lần định nghĩa phương thức đó là đủ: hoặc ở lớp cơ sở, hoặc ở lớp dẫn xuất.
- Trong trường hợp cây kế thừa có nhiều mức, cũng chỉ cần khai báo phương thức là trừu tượng (`virtual`) ở một mức bất kì. Khi đó, tất cả các phương thức trùng tên với phương thức đó ở tất cả các mức đều được coi là trừu tượng.
- Đôi khi không cần thiết phải định nghĩa chồng (trong lớp dẫn xuất) một phương thức đã được khai báo trừu tượng trong lớp cơ sở.

6.6.3 Sử dụng phương thức trừu tượng – đa hình

Một khi phương thức được khai báo là trừu tượng thì khi một con trỏ gọi đến phương thức đó, chương trình sẽ thực hiện phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới, thay vì

thực hiện phương thức của lớp cùng kiểu với con trỏ. Đây được gọi là hiện tượng đa hình (tương ứng bội) trong C++.

Chương trình 6.9 minh họa việc sử dụng phương thức trừu tượng: lớp Bus kế thừa từ lớp Car, hai lớp này cùng định nghĩa phương thức trừu tượng show().

- Khi ta dùng một con trỏ có kiểu lớp Car trỏ vào địa chỉ của một đối tượng kiểu Car, nó sẽ gọi phương thức show() của lớp Car.
- Khi ta dùng cùng con trỏ đó, trỏ vào địa chỉ của một đối tượng kiểu Bus, nó sẽ gọi phương thức show() của lớp Bus.

Chương trình 6.9

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

/* Định nghĩa lớp Car */
class Car{
    private:
        int    speed;           // Tốc độ
        char   mark[20];        // Nhãn hiệu
        float  price;           // Giá xe
    public:
        int    getSpeed(){return speed;}// Đọc tốc độ xe
        char[] getMark(){return mark;}// Đọc nhãn xe
        float  getPrice(){return price;}// Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        virtual void show();      // Giới thiệu xe, trừu tượng
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

// Phương thức trừu tượng giới thiệu xe
virtual void Car::show(){
    cout << "This is a " << mark << " having a speed of "
         << speed << "km/h and its price is $" << price << endl;
    return;
}
```



```

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label;                // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void show();              // Giới thiệu xe
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

// Định nghĩa nạp chồng phương thức trừu tượng
void Bus::show() {           // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl;
    return;
}

// Chương trình chính
void main(){
    clrscr();
    Car *ptrCar, myCar(100, "Ford", 3000);
    Bus myBus(150, "Mercedes", 5000, 27); // Biến đổi tượng của lớp Bus

    ptrCar = &myCar;           // Trỏ đến đối tượng lớp Car
    ptrCar->show();             // Phương thức của lớp Car

    ptrCar = &myBus;           // Trỏ đến đối tượng lớp Bus
    ptrCar->show();             // Phương thức của lớp Bus
    return;
}

```

Chương trình 6.9 hiển thị kết quả thông báo như sau:

```

This is a Ford having a speed of 100km/h and its price is $3000
This is a bus of type Mercedes, on the line 27, having a speed of
150km/h and its price is $5000

```

Dòng thứ nhất là kết quả khi con trỏ ptrCar trỏ đến địa chỉ của đối tượng myCar, thuộc lớp Car nên sẽ gọi phương thức show() của lớp Car với các dữ liệu của đối tượng myCar: (100, Ford,

3000). Dòng thứ hai tương ứng là kết quả khi con trỏ ptrCar trỏ đến địa chỉ của đối tượng myBus, thuộc lớp Bus nên sẽ gọi phương thức show() của lớp Bus, cùng với các tham số của đối tượng myBus: (150, Mercedes, 5000, 27).

Lưu ý:

- Trong trường hợp ở lớp dẫn xuất không định nghĩa lại phương thức trừu tượng, thì chương trình sẽ gọi phương thức của lớp cơ sở, nhưng với dữ liệu của lớp dẫn xuất.

Ví dụ, nếu trong chương trình 6.9, lớp Bus không định nghĩa chồng phương thức trừu tượng show() thì kết quả hiển thị sẽ là hai dòng thông báo giống nhau, chỉ khác nhau ở dữ liệu của hai đối tượng khác nhau:

```
This is a Ford having a speed of 100km/h and its price is $3000
```

```
This is a Mercedes having a speed of 150km/h and its price is $5000
```

TỔNG KẾT CHƯƠNG 6

Nội dung chương 6 đã trình bày các vấn đề cơ bản liên quan đến thừa kế và tương ứng bội trong C++ như sau:

- Khai báo một lớp dẫn xuất kế thừa từ một lớp cơ sở bằng khai báo kế thừa “:” đi kèm với một từ khoá dẫn xuất.
- Có ba loại dẫn xuất khác nhau, được quy định bởi ba từ khoá dẫn xuất khác nhau: private, protected và public. Kiểu dẫn xuất phổ biến là dẫn xuất public.
- Sự kế thừa tạo ra mối quan hệ tương ứng giữa lớp cơ sở và lớp dẫn xuất: có thể chuyển kiểu ngầm định (trong phép gán, phép truyền đối số, phép trỏ địa chỉ) từ một đối tượng lớp cơ sở đến một đối tượng lớp dẫn xuất. Nhưng không thể chuyển kiểu ngược lại từ lớp dẫn xuất vào lớp cơ sở.
- Hàm khởi tạo của lớp dẫn xuất có thể gọi tường minh hoặc gọi ngầm định hàm khởi tạo của lớp cơ sở. Hàm khởi tạo lớp cơ sở bao giờ cũng được thực hiện trước hàm khởi tạo lớp dẫn xuất.
- Hàm huỷ bỏ của lớp dẫn xuất luôn gọi ngầm định hàm huỷ bỏ của lớp cơ sở. Trái với hàm khởi tạo, hàm huỷ bỏ lớp cơ sở luôn được thực hiện sau hàm huỷ bỏ của lớp dẫn xuất.
- Có thể truy nhập các phương thức của lớp cơ sở từ lớp dẫn xuất, phạm vi truy nhập là phụ thuộc vào kiểu dẫn xuất: private, protected hoặc public. Điều này cho phép sử dụng lại mã nguồn của lớp cơ sở, mà không cần định nghĩa lại ở lớp dẫn xuất.
- Trong lớp dẫn xuất, có thể định nghĩa chồng một số phương thức của lớp cơ sở. Khi có định nghĩa chồng, muốn truy nhập vào phương thức lớp cơ sở, phải dùng toán tử phạm vi lớp “<Tên lớp>::”.
- C++ còn cho phép một lớp có thể được dẫn xuất từ nhiều lớp cơ sở khác nhau, gọi là đa kế thừa. Trong đa kế thừa, quan hệ giữa lớp dẫn xuất với mỗi lớp cơ sở là tương tự như trong đơn kế thừa.
- Trong đa kế thừa, hàm khởi tạo lớp dẫn xuất sẽ gọi tường minh (hoặc ngầm định) hàm khởi tạo các lớp cơ sở, theo thứ tự khai báo kế thừa. Hàm huỷ bỏ lớp dẫn xuất lại gọi ngầm định các hàm huỷ bỏ của các lớp cơ sở.

- C++ cung cấp khái niệm kế thừa từ lớp cơ sở trừu tượng để tránh trường hợp trùng lặp dữ liệu ở lớp dẫn xuất, khi các lớp cơ sở lại cùng được dẫn xuất từ một lớp khác.
- C++ cũng cho phép cơ chế tương ứng bội (đa hình) bằng cách định nghĩa một phương thức là trừu tượng trong sơ đồ thừa kế. Khi đó, một con trỏ lớp cơ sở có thể trỏ đến địa chỉ của một đối tượng lớp dẫn xuất, và phương thức được thực hiện là tùy thuộc vào kiểu của đối tượng mà con trỏ đang trỏ tới.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 6

1. Trong các khai báo sau, khai báo nào là đúng cú pháp kế thừa lớp:
 - a. `class A: public class B{...};`
 - b. `class A: public B{...};`
 - c. `class A: class B{...};`
 - d. `class A:: public B{...};`
2. Trong các kiểu dẫn xuất sau, từ các phương thức lớp dẫn xuất, không thể truy nhập đến các thành phần private của lớp cơ sở:
 - a. `private`
 - b. `protected`
 - c. `public`
 - d. Cả ba kiểu trên
3. Trong các kiểu dẫn xuất sau, từ đối tượng của lớp dẫn xuất, có thể truy nhập đến các thành phần của lớp cơ sở:
 - a. `private`
 - b. `protected`
 - c. `public`
 - d. Cả ba kiểu trên
4. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử có các kiểu khai báo:
`A myA, *ptrA;`
`B myB, *ptrB;`
Khi đó, các lệnh nào sau đây là không có lỗi:
 - a. `myA = myB;`
 - b. `myB = myA;`
 - c. `ptrA = &myB;`
 - d. `ptrB = &myA;`
 - e. `ptrA = ptrB;`
 - f. `ptrB = ptrA;`
5. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử có các kiểu khai báo và nguyên mẫu hàm:
`A myA;`
`B myB;`
`void show(A, B);`

Khi đó, các lệnh gọi hàm nào sau đây là không có lỗi:

- a. `show(myA, myA);`
- b. `show(myA, myB);`
- c. `show(myB, myA);`
- d. `show(myB, myB);`

6. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có một hàm khởi tạo:

```
B(int, float);
```

Khi đó, định nghĩa hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

- a. `A::A() {...};`
- b. `A::A(): B() {...};`
- c. `A::A(int x, float y): B() {...};`
- d. `A::A(int x, float y): B(x, y) {...};`

7. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có hai hàm khởi tạo:

```
B();  
B(int, float);
```

Khi đó, những định nghĩa hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

- a. `A::A() {...};`
- b. `A::A(): B() {...};`
- c. `A::A(int x, float y): B() {...};`
- d. `A::A(int x, float y): B(x, y) {...};`

8. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có hàm hủy bỏ tường minh:

```
~B();
```

Khi đó, những định nghĩa hàm hủy bỏ nào sau đây của lớp A là chấp nhận được:

- a. `A::~~A() {...};`
- b. `A::~~A(): ~B() {...};`
- c. `A::~~A(int x) {...};`
- d. `A::~~A(int x): ~B() {...};`

9. Giả sử B là một lớp được khai báo:

```
class B{  
    int x;  
    public: int getx();  
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu private:

```
class A: private B{  
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. `myA.x;`
- b. `myA.getx();`
- c. Cả hai lệnh trên.

d. Không lệnh nào cả.

10. Giả sử B là một lớp được khai báo:

```
class B{
    int x;
    public: int getX();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu protected:

```
class A: protected B{
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. myA.x;
- b. myA.getX();
- c. Cả hai lệnh trên.
- d. Không lệnh nào cả.

11. Giả sử B là một lớp được khai báo:

```
class B{
    int x;
    public: int getX();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu public:

```
class A: public B{
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. myA.x;
- b. myA.getX();
- c. Cả hai lệnh trên.
- d. Không lệnh nào cả.

12. Giả sử B là một lớp được khai báo:

```
class B{
    public: void show();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu public, có định nghĩa chồng hàm show():

```
class A: public B{
    public: void show();
};
```

khi đó, nếu myA là một đối tượng lớp A, muốn thực hiện phương thức show() của lớp B thì lệnh nào sau đây là chấp nhận được:

- a. myA.show();
- b. myA.B::show();
- c. B::myA.show();

d. `A::B::show();`

13. Muốn khai báo một lớp A kế thừa từ hai lớp cơ sở B và C, những lệnh nào là đúng:

a. `class A: B, C{...};`

b. `class A: public B, C{...};`

c. `class A: public B, protected C{...};`

d. `class A: public B, public C{...};`

14. B là một lớp có hai hàm khởi tạo:

`B();`

`B(int);`

C cũng là một lớp có hai hàm khởi tạo:

`C();`

`C(int, int);`

Và A là một lớp kế thừa từ B và C:

`class A: public B, public C{...};`

Khi đó, hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

a. `A::A() {...};`

b. `A::A(): B(), C() {...};`

c. `A::A(int x, int y): C(x, y) {...};`

d. `A::A(int x, int y, int z): B(x), C(y, z) {...};`

e. `A::A(int x, int y, int z): C(x, y), B(z) {...};`

15. Muốn khai báo lớp A kế thừa từ lớp cơ sở trừu tượng B, những khai báo nào sau đây là đúng:

a. `virtual class A: public B{...};`

b. `class virtual A: public B{...};`

c. `class A: virtual public B{...};`

d. `class A: public virtual B{...};`

16. Lớp A là một lớp dẫn xuất, được kế thừa từ lớp cơ sở B. Hai lớp này đều định nghĩa hàm `show()`. Muốn hàm này trở thành trừu tượng thì những định nghĩa nào sau đây là đúng:

a. `void A::show() {...}` và `void B::show() {...}`

b. `virtual void A::show() {...}` và `void B::show() {...}`

c. `void A::show() {...}` và `virtual void B::show() {...}`

d. `virtual void A::show() {...}` và `virtual void B::show() {...}`

17. Khai báo lớp người (Human) bao gồm các thuộc tính sau:

- Tên người (name)
- Tuổi của người đó (age)
- Giới tính của người đó (sex)

Sau đó khai báo lớp Cá nhân (Person) kế thừa từ lớp Human vừa được định nghĩa ở trên.

18. Bổ sung các phương thức truy nhập các thuộc tính của lớp Human, các phương thức này có tính chất public.
19. Bổ sung thêm các thuộc tính của lớp Person: địa chỉ và số điện thoại. Thêm các phương thức truy nhập các thuộc tính này trong lớp Person.
20. Xây dựng hai hàm khởi tạo cho lớp Human: một hàm không tham số, một hàm với đủ ba tham số tương ứng với ba thuộc tính của nó. Sau đó, xây dựng hai hàm khởi tạo cho lớp Person có sử dụng các hàm khởi tạo của lớp Human: một hàm không tham số, một hàm đủ năm tham số (ứng với hai thuộc tính của lớp Person và ba thuộc tính của lớp Human).
21. Xây dựng một hàm main, trong đó có yêu cầu nhập các thuộc tính để tạo một đối tượng có kiểu Human và một đối tượng có kiểu Person, thông qua các hàm set thuộc tính đã xây dựng.
22. Xây dựng hàm show() cho hai lớp Human và Person. Thay đổi hàm main: dùng một đối tượng có kiểu lớp Person, gọi hàm show() của lớp Person, sau đó lại gọi hàm show() của lớp Human từ chính đối tượng đó.
23. Khai báo thêm một lớp người lao động (Worker), kế thừa từ lớp Human, có thêm thuộc tính là số giờ làm việc trong một tháng (hour) và tiền lương của người đó (salary). Sau đó, khai báo thêm một lớp nhân viên (Employee) kế thừa đồng thời từ hai lớp: Person và Worker. Lớp Employee có bổ sung thêm một thuộc tính là chức vụ (position).
24. Chuyển lớp Human thành lớp cơ sở trừu tượng của hai lớp Person và Worker. Xây dựng thêm hai hàm show() của lớp Worker và lớp Employee. Trong hàm main, khai báo một đối tượng lớp Employee, sau đó gọi đến các hàm show() của các lớp Employee, Person, Worker và Human.
25. Chuyển hàm show() trong các lớp trên thành phương thức trừu tượng. Trong hàm main, khai báo một con trỏ kiểu Human, sau đó, cho nó trỏ đến lần lượt các đối tượng của các lớp Human, Person, Worker và Employee, mỗi lần đều gọi phương thức show() để hiển thị thông báo ra màn hình.

CHƯƠNG 7

MỘT SỐ LỚP QUAN TRỌNG

Nội dung chương này tập trung trình bày một số lớp cơ bản trong C++:

- Lớp vật chứa (Container)
- Lớp tập hợp (Set)
- Lớp chuỗi kí tự (String)
- Lớp ngăn xếp (Stack) và hàng đợi (Queue)
- Lớp danh sách liên kết (Lists)

7.1 LỚP VẬT CHỨA

Lớp vật chứa (Container) bao gồm nhiều lớp cơ bản của C++: lớp Vector, lớp danh sách (List) và các kiểu hàng đợi (Stack và Queue), lớp tập hợp (Set) và lớp ánh xạ (Map). Trong chương này sẽ trình bày một số lớp cơ bản của Container là: Set, Stack, Queue và List

7.1.1 Giao diện của lớp Container

Các lớp cơ bản của Container có một số toán tử và phương thức có chức năng giống nhau, bao gồm:

- **==**: Toán tử so sánh bằng
- **<**: Toán tử so sánh nhỏ hơn
- **begin()**: Giá trị khởi đầu của con chạy iterator
- **end()**: Giá trị kết thúc của con chạy iterator
- **size()**: Số lượng phần tử đối tượng của vật chứa
- **empty()**: Vật chứa là rỗng
- **front()**: Phần tử thứ nhất của vật chứa
- **back()**: Phần tử cuối của vật chứa
- **[]**: Toán tử truy nhập đến phần tử của vật chứa
- **insert()**: Thêm vào vật chứa một (hoặc một số) phần tử
- **push_back()**: Thêm một phần tử vào cuối vật chứa
- **push_front()**: Thêm một phần tử vào đầu vật chứa
- **erase()**: Loại bỏ một (hoặc một số) phần tử khỏi vật chứa
- **pop_back()**: Loại bỏ phần tử cuối của vật chứa
- **pop_front()**: Loại bỏ phần tử đầu của vật chứa.

Ngoài ra, tùy vào các lớp cụ thể mà có một số toán tử và phương thức đặc trưng của lớp đó. Các toán tử và phương thức này sẽ được trình bày chi tiết trong nội dung từng lớp tiếp theo.

7.1.2 Con chạy Iterator

Iterator là một con trỏ trỏ đến các phần tử của vật chứa. Nó đóng vai trò là một con chạy cho phép người dùng di chuyển qua từng phần tử có mặt trong vật chứa. Mỗi phép tăng giảm một đơn vị của con chạy này tương ứng với một phép dịch đến phần tử tiếp theo hay phần tử trước của phần tử hiện tại mà con chạy đang trỏ tới.

Khai báo con chạy

Cú pháp chung để khai báo một biến con chạy iterator như sau:

```
Tên_lớp<T>::iterator Tên_con_chạy;
```

Trong đó:

- **Tên lớp:** là tên của lớp cơ bản ta đang dùng, ví dụ lớp Set, lớp List...
- **T:** là tên kiểu lớp của các phần tử chứa trong vật chứa. Kiểu có thể là các kiểu cơ bản trong C++, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.
- **Tên con chạy:** là tên biến sẽ được sử dụng làm biến chạy trong vật chứa.

Ví dụ:

```
Set<int>::iterator iter;
```

là khai báo một biến con chạy iter cho lớp tập hợp (Set), trong đó, các phần tử của lớp vật chứa có kiểu cơ bản int. Hoặc:

```
List<Person>::iterator iter;
```

là khai báo một biến con chạy iter cho lớp danh sách (List), trong đó, các phần tử có kiểu lớp do người dùng tự định nghĩa là Person.

Sử dụng con chạy

Con chạy được sử dụng khi cần duyệt lần lượt các phần tử có mặt trong vật chứa. Ví dụ sau sẽ in ra các phần tử có kiểu int của một tập hợp:

```
Set<int> mySet; // Khai báo một đối tượng của lớp Set,
               // các phần tử có kiểu int
Set<int>::iterator i; // Khai báo con chạy của lớp Set,
                   // các phần tử có kiểu int
... // Thêm phần tử vào
for(i=mySet.begin(); i<mySet.end(); i++)
    cout << mySet[i] << " ";
```

Lưu ý:

- Biến con chạy phải được khởi đầu bằng phương thức begin() và kết thúc bằng phương thức end() của đối tượng cần duyệt tương ứng.
- Một con chạy có thể được sử dụng nhiều lần cho nhiều đối tượng nếu chúng có cùng kiểu lớp vật chứa và các phần tử của chúng cũng cùng kiểu.

7.2 LỚP TẬP HỢP

Lớp tập hợp (Set) chứa các phần tử có cùng kiểu, không phân biệt thứ tự giữa các phần tử nhưng lại phân biệt giữa các phần tử: các phần tử là khác nhau từng đôi một. Muốn sử dụng lớp tập hợp, phải có chỉ thị đầu tệp:

```
#include<set.h> // Thư viện riêng của lớp tập hợp
```

hoặc:

```
#include<stl.h> // Thư viện chung cho các lớp vật chứa
```

7.2.1 Hàm khởi tạo

Lớp tập hợp có ba kiểu khởi tạo chính:

- Khởi tạo không tham số:

```
Set<T> Tên_đối_tượng;
```

- Khởi tạo bằng một mảng các đối tượng phần tử:

```
Set<T> Tên_đối_tượng(T*, chiều_dài_mảng);
```

- Khởi tạo bằng một đối tượng thuộc lớp tập hợp khác:

```
Set<T> Tên_đối_tượng(Set<T>);
```

Trong đó:

- **T**: là tên kiểu của các phần tử của tập hợp. Kiểu này có thể là kiểu cơ bản của C++, cũng có thể là các kiểu cấu trúc (struct) hoặc lớp (class) do người dùng tự định nghĩa.

Ví dụ:

```
Set<int> mySet;
```

là khai báo một đối tượng mySet của lớp tập hợp, mySet khởi đầu chưa có phần tử nào, các phần tử của đối tượng này có kiểu cơ bản int. Hoặc:

```
Person *myPerson = new Person[10]; // Khởi tạo mảng đối tượng
Set<Person> mySet(myPerson, 10); // Khai báo tập hợp
```

là khai báo một đối tượng mySet của lớp tập hợp, có 10 phần tử tương ứng với các phần tử trong mảng động myPerson, các phần tử có kiểu lớp Person.

7.2.2 Toán tử

Các toán tử trên lớp tập hợp là tương ứng với các toán tử số học trên tập hợp thông thường: OR “|”, AND “&”, XOR “^” và phép trừ “-”.

Phép toán “|” và “|=

Phép toán này trả về phép hợp (OR) của hai đối tượng của lớp tập hợp, kết quả cũng là một đối tượng của lớp tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> | <Đối_tượng_3>;
<Đối_tượng_1> |= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 | mySet2;
```

thì mySet3 sẽ chứa các phần tử kiểu int là {1,2,3,4}.

Phép toán "&" và "&="

Phép toán này trả về phép giao (AND) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> & <Đối_tượng_3>;  
<Đối_tượng_1> &= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 & mySet2;
```

thì mySet3 sẽ chứa các phần tử có kiểu int là {2}.

Phép toán "^" và "^="

Phép toán này trả về phép hợp ngoại (XOR) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> ^ <Đối_tượng_3>;  
<Đối_tượng_1> ^= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 ^ mySet2;
```

thì mySet3 sẽ chứa các phần tử có kiểu int là {1,3,4}.

Phép toán "-" và "-="

Phép toán này trả về phép hiệu (loại trừ) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> - <Đối_tượng_3>;  
<Đối_tượng_1> -= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 - mySet2;
```

thì mySet3 sẽ chứa các phần tử có kiểu int là {1}.

7.2.3 Phương thức

Lớp tập hợp có một số phương thức cơ bản sau:

- Thêm một phần tử vào tập hợp
- Loại một phần tử khỏi tập hợp
- Tìm kiếm một phần tử trong tập hợp

Thêm một phần tử vào tập hợp

Có hai cú pháp để thêm một phần tử vào tập hợp:

```
pair<iterator, bool> insert(T&);  
iterator insert(<Vị trí con chạy>, T&);
```

Trong đó:

- Phương thức thứ nhất thêm một phần tử vào tập hợp, nếu phần tử đã có mặt trong tập hợp, trả về false và vị trí con chạy của phần tử đó. Nếu phần tử chưa tồn tại, trả về true và vị trí con chạy của phần tử mới thêm vào.
- Phương thức thứ hai cũng thêm vào một phần tử, nhưng chỉ kiểm tra xem phần tử đã tồn tại hay chưa bắt đầu từ vị trí con chạy được chỉ ra trong biến <vị trí con chạy>. Phương thức này trả về vị trí con chạy của phần tử mới thêm vào (nếu thành công) hoặc vị trí của phần tử đã có mặt.

Ví dụ, mySet là một tập hợp kiểu int:

```
mySet.insert(10);
```

sẽ thêm một phần tử có giá trị 10 vào tập hợp.

Lưu ý:

- Do tuân thủ theo lý thuyết tập hợp, nên khi chèn vào tập hợp nhiều lần với cùng một giá trị phần tử. Trong tập hợp chỉ tồn tại duy nhất một giá trị phần tử đó, không được trùng lặp.

Loại một phần tử khỏi tập hợp

Có ba cú pháp để loại bỏ phần tử khỏi tập hợp:

```
int erase(T&);  
void erase(<vị trí con chạy>);  
void erase(<vị trí bắt đầu>, <vị trí kết thúc>);
```

Trong đó:

- T: là tên kiểu các phần tử của tập hợp.
- Phương thức thứ nhất xoá phần tử có giá trị được chỉ rõ trong tham số đầu vào.
- Phương thức thứ hai loại bỏ phần tử ở vị trí của con chạy được xác định bởi tham số đầu vào.
- Phương thức thứ ba loại bỏ một số phần tử nằm trong phạm vi từ <vị trí bắt đầu> cho đến <vị trí kết thúc> của con chạy.

Ví dụ, mySet là một tập hợp các phần tử có kiểu int:

```
mySet.erase(10);
```

sẽ xoá phần tử có giá trị 10, hoặc:

```
mySet.erase((Set<int>::iterator)10);
```

sẽ xoá phần tử ở vị trí thứ 10 trong tập hợp, hoặc:

```
mySet.erase(mySet.begin(), mySet.end());
```

sẽ xoá toàn bộ các phần tử hiện có của tập hợp mySet.

Tìm kiếm một phần tử trong tập hợp

Có hai cú pháp để tìm kiếm một phần tử trong tập hợp:

```
iterator find(T&);  
int count(T&);
```

Trong đó:

- Phương thức thứ nhất tìm phần tử có giá trị xác định bởi tham số đầu vào, kết quả là vị trí con chạy của phần tử đó.
- Phương thức thứ hai chỉ để kiểm tra xem phần tử có xuất hiện trong tập hợp hay không: trả về 1 nếu có mặt, trả về 0 nếu không có mặt.

Ví dụ, mySet là một tập hợp các phần tử có kiểu int:

```
Set<int>::iterator index = mySet.find(10);  
cout << index;
```

sẽ hiển thị vị trí con chạy của phần tử có giá trị 10 trong tập hợp mySet.

7.2.4 Áp dụng

Chương trình 7.1 minh họa một số thao tác trên một tập hợp các phần tử có kiểu char: thêm phần tử, loại bỏ phần tử, duyệt các phần tử.

Chương trình 7.1

```
#include<stdio.h>  
#include<conio.h>  
#include<set.h>  
void main() {  
    clrscr();  
    Set<char> mySet;  
    int function;  
    do{  
        clrscr();  
        cout << "CAC CHUC NANG:" << endl;  
        cout << "1: Them mot phan tu vao tap hop" << endl;  
        cout << "2: Loai bo mot phan tu khoi tap hop" << endl;  
        cout << "3: Xem tat ca cac phan tu cua tap hop" << endl;  
        cout << "5: Thoat!" << endl;  
        cout << "======" << endl;  
        cout << "Chon chuc nang: " << endl;  
        cin >> function;  
        switch(function){  
            case '1':                // Thêm vào  
                char phantu;  
                cout << "Ki tu them vao: ";  
                cin >> phantu;  
                mySet.insert(phantu);  
                break;  
            case '2':                // Loại ra  
                char phantu;  
                cout << "Loai bo ki tu: " << endl;  
                cin >> phantu;
```

```

        mySet.erase(phantu);
        break;
    case '3': // Duyệt
        cout<<"Cac phan tu cua tap hop la:"<<endl;
        Set<char>::iterator i;
        for(i=mySet.begin(); i<mySet.end(); i++)
            cout << mySet[i] << " ";
        break;
    }while(function != '5');
    return;
}

```

7.3 LỚP CHUỖI

Lớp chuỗi (String) cũng là một loại lớp chứa, nó chứa một tập các phần tử là một dãy các kí tự có phân biệt thứ tự, các phần tử không nhất thiết phải phân biệt nhau. Muốn sử dụng lớp String, cần thêm vào chỉ thị đầu tệp:

```
#include <string.h>
```

7.3.1 Hàm khởi tạo

Lớp String có ba hàm khởi tạo chính:

- Hàm khởi tạo không tham số:
`String <tên biến>;`
- Hàm khởi tạo từ một string khác:
`String <tên biến>(const String &);`
- Hàm khởi tạo từ một mảng các kí tự:
`String <tên biến>(char*, <chiều dài mảng>);`

Ví dụ:

```
String myStr;
```

là khai báo một chuỗi myStr chưa có phần tử nào (rỗng). Hoặc:

```
String myStr("hello!");
```

là khai báo một chuỗi myStr có các phần tử theo thứ tự là {'h', 'e', 'l', 'l', 'o', '!'}. Hoặc:

```
char* myChar = new char[10];
String myStr(myChar, 10);
```

là khai báo một chuỗi myStr có 10 phần tử tương ứng với các kí tự trong mảng myChar.

Lưu ý:

- Trong trường hợp khởi tạo bằng một chuỗi khác, ta có thể truyền vào một đối số có kiểu cơ bản khác, trình biên dịch sẽ tự động chuyển đổi số đó sang dạng string.

Ví dụ:

```
String myStr(12);
```

thì chuỗi myStr sẽ chứa hai phần tử kí tự {'1', '2'}. Hoặc:

```
String myStr(15.55);
```

thì chuỗi myStr sẽ chứa năm phần tử kí tự {'1', '5', '.', '5', '5'}.

7.3.2 Toán tử

Lớp String có các toán tử cơ bản là:

- Phép gán chuỗi
- Phép cộng chuỗi
- Phép so sánh chuỗi
- Phép vào/ra

Phép gán chuỗi “=”

Cú pháp phép gán chuỗi là tương tự cú pháp gán các đối tượng cơ bản:

```
<Tên biến 1> = <Tên biến 2>;
```

Ví dụ:

```
String s1(12), s2;  
s2 = s1;
```

thì chuỗi s2 cũng chứa hai phần tử như s1 {'1', '2'}.

Lưu ý:

- Có thể gán trực tiếp các đối tượng cơ bản cho chuỗi:

```
String myStr = 12; // myStr có hai phần tử {'1', '2'}
```

- Nhưng phép gán lại có độ ưu tiên thấp hơn phép toán học:

```
String myStr = 12+1.5; // tương đương myStr = 13.5
```

Phép cộng chuỗi “+” và “+=”

Phép cộng chuỗi sẽ nối chuỗi thứ hai vào sau chuỗi thứ nhất, kết quả cũng là một chuỗi:

```
<Tên biến 1> = <Tên biến 2> + <Tên biến 3>;  
<Tên biến 1> += <Tên biến 2>;
```

Ví dụ:

```
String s1(12), s2(3);  
s1+= s2;
```

thì s1 sẽ có ba phần tử kí tự {'1', '2', '3'}.

Phép so sánh chuỗi

Các phép so sánh chuỗi đều là các phép toán hai ngôi, trả về kết quả ở dạng bool (true/false):

- Phép so sánh lớn hơn “>”: chuỗi_1 > chuỗi_2;
- Phép so sánh lớn hơn hoặc bằng “>=”: chuỗi_1 >= chuỗi_2;
- Phép so sánh nhỏ hơn “<”: chuỗi_1 < chuỗi_2;
- Phép so sánh nhỏ hơn hoặc bằng “<=”: chuỗi_1 <= chuỗi_2;

- Phép so sánh bằng “==”: chuỗi_1 == chuỗi_2;
- Phép so sánh khác (không bằng) “!=”: chuỗi_1 != chuỗi_2;

Lưu ý:

- Phép so sánh chuỗi thực hiện so sánh mã ASCII của từng ký tự ở hai chuỗi theo thứ tự tương ứng cho đến khi có sự khác nhau đầu tiên giữa hai ký tự.
- Phép so sánh là phép so sánh dựa trên từ điển, có phân biệt chữ hoa và chữ thường.

Ví dụ:

```
"12" < "a";           // có giá trị đúng
"a" <= "A";           // có giá trị sai
```

Phép vào/ra

- Phép xuất ra “<<”: cout << biến_chuỗi;
- Phép nhập vào “>>”: cin >> biến_chuỗi;

Ví dụ:

```
String s("hello!");
cout << s;
```

sẽ in ra màn hình dòng chữ “hello!”.

7.3.3 Phương thức

Lớp chuỗi có một số phương thức cơ bản:

- Lấy chiều dài chuỗi
- Tìm một chuỗi con
- Thêm một chuỗi con
- Xoá một chuỗi con
- Chuyển kiểu ký tự

Lấy chiều dài chuỗi

Cú pháp:

```
<biến_chuỗi>.length();
```

trả về chiều dài của chuỗi (số lượng phần tử ký tự trong chuỗi).

Ví dụ:

```
String s("hello!");
cout << s.length();
```

sẽ in ra màn hình độ dài chuỗi s là 6.

Tìm một chuỗi con

Cú pháp:

```
<biến_chuỗi>.find(<Chuỗi con>, <Vị trí bắt đầu>, <Kiểu so khớp>);
```

Trong đó:

- Tham số thứ nhất là chuỗi con cần tìm.
- Tham số thứ hai là vị trí để bắt đầu tìm, mặc định là bắt đầu tìm từ phần tử có chỉ số 0.
- Tham số thứ ba chỉ ra cách so khớp có phân biệt chữ hoa với chữ thường: SM_IGNORE là không phân biệt, SM_SENSITIVE là có phân biệt.
- Phương thức này trả về kết quả dạng bool, tương ứng là có tìm thấy hay không.

Ví dụ:

```
s.find("12", 0, SM_IGNORE);
```

sẽ tìm trong chuỗi s xem có sự xuất hiện của chuỗi "12" hay không, vị trí bắt đầu tìm là 0, với cách tìm không phân biệt chữ hoa chữ thường.

Thêm một chuỗi con

Cú pháp:

```
<biến_chuỗi>.insert(<vị trí chèn>, <chuỗi con>);
```

Trong đó:

- Tham số thứ nhất là vị trí chỉ số mà tại đó, chuỗi con sẽ được chèn vào
- Tham số thứ hai là chuỗi con cần chèn, chuỗi con này cũng có thể là một kí tự.

Ví dụ:

```
s.insert(0, "12");
```

sẽ chèn vào đầu chuỗi s một chuỗi con có hai phân tử "12".

Xoá một chuỗi con

Cú pháp:

```
<biến_chuỗi>.delete(<vị trí bắt đầu>, <độ dài chuỗi xoá>);
```

Trong đó:

- Tham số thứ nhất là vị trí bắt đầu xoá chuỗi con
- Tham số thứ hai là độ dài chuỗi con bị xoá, giá trị mặc định là 1.

Ví dụ:

```
s.delete(0, 2);
```

sẽ xoá hai kí tự đầu của chuỗi s.

Chuyển kiểu kí tự

- Đổi chuỗi thành các kí tự hoa: `<biến_chuỗi>.toUpperCase();`
- Đổi chuỗi thành các kí tự thường: `<biến_chuỗi>.toLowerCase();`

Ví dụ:

```
s.toUpperCase(); // chuyển chuỗi s thành kí tự hoa  
s.toLowerCase(); // chuyển chuỗi s thành kí tự thường
```

7.3.4 Áp dụng

Chương trình 7.2 minh họa một số thao tác cơ bản trên lớp chuỗi, có sử dụng thư viện lớp chuỗi chuẩn của C++: cộng thêm một chuỗi, chèn thêm một chuỗi con, xóa một chuỗi con, tìm một chuỗi con...

Chương trình 7.2

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main() {
    clrscr();
    String myStr;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Cong them mot chuoi" << endl;
        cout << "2: Chen them mot chuoi" << endl;
        cout << "3: Xoa di mot chuoi" << endl;
        cout << "4: Tim mot chuoi con" << endl;
        cout << "5: Chuyen thanh chu hoa" << endl;
        cout << "6: Chuyen thanh chu thuong" << endl;
        cout << "7: Xem noi dung chuoi" << endl;
        cout << "8: Xem chieu dai chuoi" << endl;
        cout << "9: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1': // Thêm vào cuối
                String subStr;
                cout << "Chuoi them vao: ";
                cin >> subStr;
                myStr += subStr;
                break;
            case '2': // Chèn vào chuỗi
                String subStr;
                int position;
                cout << "Chuoi them vao: ";
                cin >> subStr;
                cout << "Vi tri chen:";
                cin >> position;
```



```

        myStr.insert(position, subStr);
        break;
    case '3': // Xoá đi một chuỗi con
        int position, count;
        cout << "Vi tri bat dau xoa:";
        cin >> position;
        cout << "Do dai xoa:";
        cin >> count;
        myStr.delete(position, count);
        break;
    case '4': // Tìm chuỗi con
        String subStr;
        int position;
        cout << "Chuoi con can tim:";
        cin >> subStr;
        cout << "Vi tri bat dau tim:";
        cin >> position;
        if(myStr.find(position, subStr))
            cout << "Co xuat hien!" << endl;
        else
            cout << "Khong xuat hien!" << endl;
        break;
    case '5': // Chuyển thành chữ hoa
        myStr.toUpperCase();
        cout << myStr << endl;
        break;
    case '6': // Chuyển thành chữ thường
        myStr.toLowerCase();
        cout << myStr << endl;
        break;
    case '7': // Duyệt
        cout << "Noi dung chuoi:" << endl;
        cout << myStr << endl;
        break;
    case '8': // Duyệt
        cout << "Chieu dai chuoi:"
            << myStr.length() << endl;
        break;
    }while(function != '9');
return;
}

```

7.4 LỚP NGĂN XẾP VÀ HÀNG ĐỢI

7.4.1 Lớp ngăn xếp

Lớp ngăn xếp (stack) cũng là một loại lớp vật chứa, nó chứa các phần tử cùng kiểu, không bắt buộc phải phân biệt nhau nhưng có phân biệt về thứ tự: các thao tác thêm phần tử và lấy phần tử ra đều được thực hiện ở một đầu ngăn xếp. Phần tử nào được thêm vào trước thì sẽ bị lấy ra sau.

Muốn dùng lớp Stack phải dùng chỉ thị đầu tệp:

```
#include<stack.h>
```

Hàm khởi tạo

Lớp Stack có hai cách khởi tạo:

- Khởi tạo không tham số:

```
Stack<T> biến_ngăn_xếp;
```

- Khởi tạo bằng một ngăn xếp khác, có cùng kiểu phần tử:

```
Stack<T> biến_ngăn_xếp(Stack<T>);
```

Trong đó:

- **T**: là kiểu của các phần tử chứa trong ngăn xếp. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
Stack<int> myStack;
```

là khai báo một biến myStack, chứa các phần tử có kiểu cơ bản int.

Toán tử

Lớp Stack chỉ dùng đến các toán tử gán “=” và toán tử so sánh bằng “==”:

- Phép gán “=”:

```
<ngăn xếp 1> = <ngăn xếp 2>;
```

Dùng để gán hai đối tượng ngăn xếp.

- Phép so sánh bằng “==”:

```
<ngăn xếp 1> == <ngăn xếp 2>;
```

Dùng để kiểm tra xem hai đối tượng ngăn xếp có bằng nhau hay không. Kết quả trả về có kiểu bool (true/false).

Phương thức

- Thêm một phần tử:

```
<biến ngăn xếp>.push(T);
```

sẽ thêm một phần tử có kiểu T vào đỉnh ngăn xếp.

- Loại một phần tử:

```
<biến ngăn xếp>.pop();
```

sẽ trả về phần tử đang nằm ở đỉnh ngăn xếp.

- Kiểm tra tính rỗng:

```
<biên ngăn xếp>.empty();
```

sẽ trả về kết quả kiểu bool, tương ứng với trạng thái của ngăn xếp có rỗng hay không.

- Kích thước ngăn xếp:

```
<biên ngăn xếp>.size();
```

sẽ trả về số lượng các phần tử hiện đang có mặt trong ngăn xếp.

Áp dụng

Trong phần này, ta sẽ viết lại chương trình 3.4c đã được minh họa trong phần viết về cấu trúc ngăn xếp (chương 3). Nhưng thay vì phải định nghĩa cấu trúc ngăn xếp, ta dùng lớp ngăn xếp của thư viện C++: đảo ngược một chuỗi ký tự được nhập vào từ bàn phím.

Chương trình 7.3

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stack.h>

void main() {
    clrscr();
    Stack<char> myStack;
    char strIn[250];
    cout << "Nhập chuỗi: ";
    cin >> strIn;      // Nhập chuỗi ký tự từ bàn phím
    for(int i=0; i<strlen(strIn); i++) // Đặt vào ngăn xếp
        myStack.push(strIn[i]);
    while(!myStack.empty()) // Lấy ra từ ngăn xếp
        cout << myStack.pop();
    return;
}
```

7.4.2 Lớp hàng đợi

Lớp hàng đợi (queue) cũng là một loại lớp vật chứa, nó chứa các phần tử cùng kiểu, không bắt buộc phải phân biệt nhau nhưng có phân biệt về thứ tự: các thao tác thêm phần tử được thực hiện ở một đầu, các thao tác lấy phần tử ra được thực hiện ở một đầu còn lại của hàng đợi. Phần tử nào được thêm vào trước thì sẽ bị lấy ra trước.

Muốn dùng lớp Queue phải dùng chỉ thị đầu tệp:

```
#include<queue.h>
```

Hàm khởi tạo

Lớp Queue có hai cách khởi tạo:

- Khởi tạo không tham số:

```
Queue<T> biến_hàng_đợi;
```

- Khởi tạo bằng một hàng đợi khác, có cùng kiểu phần tử:

```
Queue<T> biến_hàng_đợi (Queue<T>);
```

Trong đó:

- **T**: là kiểu của các phần tử chứa trong hàng đợi. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
Queue<int> myQueue;
```

là khai báo một biến myQueue, chứa các phần tử có kiểu cơ bản int.

Toán tử

Lớp Queue chỉ dùng đến các toán tử gán “=” và toán tử so sánh bằng “==”:

- Phép gán “=”:

```
<hàng_đợi_1> = <hàng_đợi_2>;
```

Dùng để gán hai đối tượng hàng đợi.

- Phép so sánh bằng “==”:

```
<hàng_đợi_1> == <hàng_đợi_2>;
```

Dùng để kiểm tra xem hai đối tượng hàng đợi có bằng nhau hay không. Kết quả trả về có kiểu bool (true/false).

Phương thức

- Thêm một phần tử:

```
<biến_hàng_đợi>.push(T);
```

sẽ thêm một phần tử có kiểu T vào cuối hàng đợi.

- Loại một phần tử:

```
<biến_hàng_đợi>.pop();
```

sẽ trả về phần tử đang nằm ở đỉnh đầu hàng đợi.

- Kiểm tra tính rỗng:

```
<biến_hàng_đợi>.empty();
```

sẽ trả về kết quả kiểu bool, tương ứng với trạng thái của hàng đợi có rỗng hay không.

- Kích thước hàng đợi:

```
<biến_hàng_đợi>.size();
```

sẽ trả về số lượng các phần tử hiện đang có mặt trong hàng đợi.

Áp dụng

Trong phần này, ta sẽ cài đặt lại chương trình trong phần cấu trúc hàng đợi (chương 3). Nhưng thay vì phải tự định nghĩa cấu trúc hàng đợi, ta dùng lớp Queue của thư viện C++ để mô phỏng chương trình quản lý tiến trình của hệ điều hành.

Chương trình 7.4

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

void main(){
    clrscr();
    Queue<int> myQueue;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot tien trinh vao hang doi" << endl;
        cout << "2: Dua mot tien trinh trinh vao thuc hien" << endl;
        cout<<"3: Xem tat ca cac tien trinh trong hang doi"<< endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1':                // Thêm vào hàng đợi
                int maso;
                cout << "Ma so tien trinh vao hang doi: ";
                cin >> maso;
                myQueue.push(maso);
                break;
            case '2':                // Lấy ra khỏi hàng đợi
                cout << "Tien trinh duoc thuc hien: " <<
                    myQueue.pop() << endl;
                break;
            case '3':                // Duyệt hàng đợi
                Queue<int>::iterator i;
                for(i=myQueue.begin(); i<myQueue.end(); i++)
                    cout << myQueue[i] << " ";
                break;
        }while(function != '5');
    }
    return;
}
```

7.5 LỚP DANH SÁCH LIÊN KẾT

Lớp danh sách liên kết (List) cũng là một kiểu lớp vật chứa, nó chứa các phần tử cùng kiểu, có tính đến thứ tự. Muốn sử dụng lớp List của thư viện C++, phải khai báo chỉ thị đầu tệp:

```
#include<list.h> // Dành riêng cho lớp List
```

hoặc:

```
#include<stl.h> // Dùng chung cho các lớp vật chứa
```

7.5.1 Hàm khởi tạo

Lớp List có ba kiểu khởi tạo:

- Khởi tạo không tham số:
`List<T> bienn_danh_sach;`
- Khởi tạo từ một danh sách cùng kiểu:
`List<T> bienn_danh_sach(List<T>);`
- Khởi tạo từ một mảng các phần tử:
`List<T> bienn_danh_sach(T* <Mảng phần tử>, int <chiều dài mảng>);`

Trong đó:

- **T**: là kiểu của các phần tử chứa trong danh sách liên kết. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
List<int> myList;
```

sẽ khai báo một danh sách liên kết myList, các phần tử của nó có kiểu cơ bản int.

7.5.2 Toán tử

Toán tử gán “=”

Cú pháp:

```
<danh sách 1> = <danh sách 2>;
```

sẽ copy toàn bộ các phần tử của <danh sách 2> vào <danh sách 1>.

Toán tử so sánh bằng “==”

Cú pháp:

```
<danh sách 1> == <danh sách 2>;
```

sẽ trả về một giá trị kiểu bool, tương ứng với việc hai danh sách này có bằng nhau hay không. Việc so sánh được tiến hành trên từng phần tử ở vị trí tương ứng nhau.

Lưu ý:

- Ngoài ra còn có các phép toán so sánh khác cũng có thể thực hiện trên danh sách: <, >, <=, >=, !=.

7.5.3 Phương thức

Thêm một phần tử vào danh sách

Cú pháp:

```
<biến danh sách>.insert(<vị trí chèn>, <phần tử>);  
<biến danh sách>.push_front(<phần tử>);  
<biến danh sách>.push_back(<phần tử>);
```

Trong đó:

- Phương thức thứ nhất chèn một phần tử vào một vị trí bất kì của danh sách, vị trí chèn được chỉ ra bởi tham số thứ nhất.
- Phương thức thứ hai chèn một phần tử vào đầu danh sách
- Phương thức thứ ba chèn một phần tử vào cuối danh sách.

Ví dụ:

```
List<int> myList;  
myList.push_front(7);
```

sẽ chèn vào đầu danh sách myList một phần tử có giá trị là 7.

Xoá đi một phần tử

Cú pháp:

```
<biến danh sách>.erase(<vị trí xoá>);  
<biến danh sách>.pop_front();  
<biến danh sách>.pop_back();
```

Trong đó:

- Phương thức thứ nhất xoá một phần tử ở một vị trí bất kì của danh sách, vị trí xoá được chỉ ra bởi tham số thứ nhất.
- Phương thức thứ hai xoá một phần tử ở đầu danh sách
- Phương thức thứ ba xoá một phần tử ở cuối danh sách.

Ví dụ:

```
List<int> myList;  
cout << myList.pop_front();
```

sẽ in ra màn hình phần tử đầu của danh sách myList.

Kiểm tra tính rỗng của danh sách

Cú pháp:

```
<biến danh sách>.empty();
```

trả về giá trị bool, tương ứng với trạng thái hiện tại của biến danh sách là rỗng hay không.

Xem kích thước danh sách

Cú pháp:

```
<biến danh sách>.size();
```

Ví dụ:


```
cout << myList.size();
```

sẽ in ra màn hình kích cỡ (số lượng các phần tử) của danh sách.

Xem nội dung một phần tử

Cú pháp:

```
<biến danh sách>.get();  
<biến danh sách>.next();
```

Trong đó:

- Phương thức thứ nhất trả về phần tử ở vị trí hiện tại của con chạy
- Phương thức thứ hai trả về phần tử tiếp theo, và con chạy cũng di chuyển sang phần tử đó.

Ví dụ:

```
List<int> myList;  
cout << myList.get();
```

sẽ in ra màn hình nội dung phần tử thứ nhất của danh sách myList.

7.5.4 Áp dụng

Trong phần này, ta cài đặt lại chương trình 3.6c đã được trình bày trong phần cấu trúc danh sách liên kết (chương 3). Nhưng thay vì tự tạo danh sách liên kết, ta dùng lớp List trong thư viện của C++ để quản lý nhân viên văn phòng.

Chương trình 7.5

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
#include<list.h>  
  
typedef struct{  
    char name[25];           // Tên nhân viên  
    int age;                 // Tuổi nhân viên  
    float salary;            // Lương nhân viên  
} Employee;  
void main(){  
    clrscr();  
    List<Employee> myList;  
    int function;  
    do{  
        clrscr();  
        cout << "CAC CHUC NANG:" << endl;  
        cout << "1: Them mot nhan vien" << endl;  
        cout << "2: Xoa mot nhan vien" << endl;  
        cout << "3: Xem tat ca cac nhan vien trong phong" << endl;
```



```

cout << "5: Thoat!" << endl;
cout << "===== " << endl;
cout << "Chon chuc nang: " << endl;
cin >> function;
switch(function){
    case '1':                // Thêm vào ds
        int position;
        Employee employee;
        cout << "Vi tri can chen: ";
        cin >> position;
        cout << "Ten nhan vien: ";
        cin >> employee.name;
        cout << "Tuoi nhan vien: ";
        cin >> employee.age;
        cout << "Luong nhan vien: ";
        cin >> employee.salary;
        myList.insert(position, employee);
        break;
    case '2':                // Lấy ra khỏi ds
        int position;
        cout << "Vi tri can xoa: ";
        cin >> position;
        Employee result = myList.erase(position);
        if(result != NULL){
            cout << "Nhan vien bi loai: " endl;
            cout << "Ten: " << result.name << endl;
            cout << "Tuoi: " << result.age << endl;
            cout << "Luong: " << result.salary << endl;
        }
        break;
    case '3':                // Duyệt ds
        cout << "Cac nhan vien cua phong:" << endl;
        Employee result = myList.front();
        do{
            cout << result.name << " "
                << result.age << " "
                << result.salary << endl;
            result = myList.next();
        }while(result != NULL);
        break;
    }while(function != '5');
return;
}

```

TỔNG KẾT CHƯƠNG 7

Nội dung chương 7 đã trình bày một số lớp cơ bản của lớp vật chứa (Container) trong thư viện của C++:

- Lớp tập hợp (Set)
- Lớp chuỗi (String)
- Lớp ngăn xếp (Stack)
- Lớp hàng đợi (Queue)
- Lớp danh sách liên kết (List)

Hầu hết các lớp vật chứa này đều có thể chứa các phần tử với kiểu bất kì: có thể là kiểu cơ bản, cũng có thể là kiểu phức tạp do người dùng tự định nghĩa. Ngoại trừ lớp chuỗi, chỉ chứa các phần tử có kiểu char.

Các lớp vật chứa này có một số phương thức tương tự nhau, nhưng cách thực hiện lại khác nhau:

- Thêm vào một phần tử
- Lấy ra một phần tử
- Kiểm tra tính rỗng
- Kiểm tra số lượng phần tử.

Người dùng có thể áp dụng các lớp có sẵn trong thư viện này của C++ để giải quyết các bài toán khác nhau mà không phải tự định nghĩa lại các lớp.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 7

1. Dùng thư viện lớp stack để kiểm tra một số tự nhiên nhập vào từ bàn phím có phải là một số palindrom hay không: Một số tự nhiên là palindrom nếu đảo ngược thứ tự các chữ số, ta vẫn thu được chính số đó, ví dụ, 87278 là một số như vậy.
2. Viết lại chương trình 7.1 dùng lớp tập hợp, nhưng dùng kiểu của các phần tử là lớp Car đã được định nghĩa trong chương 5.
3. Dùng thư viện lớp String để viết chương trình chia nhỏ một chuỗi ban đầu thành một số chuỗi con, ranh giới phân chia là một chuỗi con được nhập vào từ bàn phím. Ví dụ “abc acb”, mà chia nhỏ theo chuỗi con “c” sẽ thu được các chuỗi: “ab”, “ca”, “cb”.
4. Dùng thư viện lớp String để viết chương trình thay thế các chuỗi con của một chuỗi ban đầu bằng một chuỗi con khác. Các chuỗi con được nhập từ bàn phím.
5. Viết lại chương trình 7.5, vẫn dùng thư viện lớp List, nhưng thay thế kiểu phần tử bằng kiểu lớp Car đã được định nghĩa trong chương 5.

HƯỚNG DẪN TRẢ LỜI CÂU HỎI VÀ BÀI TẬP

Chương 1

Chương 2

1. c và e.
2. a và d.
3. c.
4. b.
5. c.
6. c.
7. d.
8. b và c.
9. b.
10. c.
11. d.
12. c.
13. d.

Chương 3

1. a.
2. c.
3. a.
4. b.
5. b và d.
6. Gợi ý:

```
struct Subject{  
    char subject[20];  
    float note;  
};
```

Hoặc:

```
typedef struct {  
    char subject[20];  
    float note;  
}Subject;
```

7. Gợi ý:

```
typedef struct {  
    char name[20];
```

```
int age;
char class[20];
Subject* subjects;
char type[20];
}Student;
```

Chương 4

1. a.
2. b.
3. c.
4. a.
5. a.
6. b.
7. c.
8. d.
9. a.
10. b.
11. c.

Chương 5

1. a và c.
2. b.
3. d.
4. a, b và c.
5. c.
6. b và c.
7. b và c.
8. b, d và e.
9. c và d.
10. a và d.
11. a, b, c và d.
12. Gợi ý (từ bài 12 đến bài 17):

```
class Employee{
    char* name;
    int age;
    float salary;
public:
    Employee();
    Employee(char* nameIn="", int ageIn=18, float salaryIn=100);
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.o-pit.edu.vn>; E-mail: dhdx@pit.edu.vn

CHƯƠNG TRÌNH
ĐÀO TẠO ĐẠI HỌC TỪ XA
PTIT

```
void setName(char*);  
char* getName();  
void setAge(int);  
int getAge();  
void setSalary(float);  
float getSalary();  
void show();  
~Employee();  
  
};
```

Chương 6

1. b.
2. d.
3. c.
4. b, d và f.
5. b và d.
6. d.
7. a, b, c và d.
8. a và c.
9. d.
10. d.
11. b.
12. b.
13. c và d.
14. a, b, c và d.
15. c và d.
16. b và c.
17. Gợi ý (từ bài 17 đến bài 25):

```
class Human{  
    char* name;  
    int age;  
    int sex;  
public:  
    Human();  
    Human(char*, int, int);  
    void setName(char*);  
    char* getName();  
    void setAge(int);  
    int getAge();  
    void setSex(int);  
};
```

```
        int getSex();
        void show();
        ~Human();
};

class Person: public virtual Human{
    char* address;
    char* phone;
public:
    Person();
    Person(char*, int, int, char*, char*);
    void setAddress(char*);
    char* getAddress();
    void setPhone(char*);
    char* getPhone();
    void show();
    ~Person();
};

class Worker: public virtual Human{
    int hour;
    float salary;
public:
    Worker();
    Worker(int, float);
    void setHour(int);
    int getHour();
    void setSalary(float);
    float getSalary();
    void show();
};

class Employee: public Person, public Worker{
    char* position;
public:
    Employee();
    Employee(char*, int, int, char*, char*, int, float,
char*);
    void setPosition(char*);
    char* getPosition();
    void virtual show();
    ~Employee();
};
```

Chương 7

2. Gợi ý

```
void main() {
    clrscr();
    Set<Car> mySet;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot phan tu vao tap hop" << endl;
        cout << "2: Loai bo mot phan tu khoi tap hop" << endl;
        cout << "3: Xem tat ca cac phan tu cua tap hop" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1': // Thêm vào
                Car car = new Car();
                cout << "O to them vao: " << endl;
                cout << "Toc do: ";
                int speed;
                cin >> speed;
                car.setSpeed(speed);
                cout << "Nhan hieu:";
                char[] mark;
                cin >> mark;
                car.setMark(mark);
                float price;
                cout << "Gia xe:";
                cin >> price;
                car.setPrice(price);
                mySet.insert(car);
                break;
            case '2': // Loại ra
                Set<Car>::iterator index;
                cout << "Loai bo xe o vi tri: ";
                cin >> index;
                if(index >= mySet.begin()) && (index < mySet.end())
                    mySet.erase(mySet[index]);
                break;
            case '3': // Duyệt
                cout<<"Cac phan tu cua tap hop la:"<<endl;
                Set<char>::iterator i;
                for(i=mySet.begin(); i<mySet.end(); i++){
                    cout << mySet[i].getSpeed() << " ";
                    cout << mySet[i].getMark() << " ";
                    cout << mySet[i].getPrice() << " ";
                }
            }
        }
    }
}
```

```
        break;
    }while(function != '5');
    return;
}
```

5. Gọi ý

```
void main() {
    clrscr();
    List<Car> myList;
    int function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot xe o to" << endl;
        cout << "2: Xoa mot xe o to" << endl;
        cout << "3: Xem tat ca cac o to" << endl;
        cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl;
        cin >> function;
        switch(function){
            case '1': // Thêm vào ds
                int possition;
                Car car = new Car();
                cout << "Vi tri can chen: ";
                cin >> possition;
                cout << "Toc do: ";
                int speed;
                cin >> speed;
                car.setSpeed(speed);
                cout << "Nhan hieu:";
                char[] mark;
                cin >> mark;
                car.setMark(mark);
                float price;
                cout << "Gia xe:";
                cin >> price;
                car.setPrice(price);
                myList.insert(possition, car);
                break;
            case '2': // Lấy ra khỏi ds
                int possition;
                cout << "Vi tri can xoa: ";
                cin >> possition;
                Car result = myList.erase(possition);
                if(result != NULL){
                    cout << "O to bi loai: " << endl;
                    cout << "Toc do:" << result.getSpeed() << endl;
                }
            }
        }
    } while(function != '5');
```



```
        cout << "Nhan hieu: " << result.getMark()
            << endl;
        cout<<"Gia: "<<result.getPrice()<<endl;
    }
    break;
case '3':
    // Duyệt ds
    cout << "Cac o to hien co:" << endl;
    Car result = myList.front();
    do{
        cout << result.getSpeed() << " "
            << result.getMark() << " "
            << result.getPrice()<< endl;
        result = myList.next();
    }while(result != NULL);
    break;
}while(function != '5');
return;
}
```

TÀI LIỆU THAM KHẢO

Tài liệu tiếng Anh

- [1] James P. Cohoon and Jack W. Davidson, **C++ Program Design – An Introduction to Programming and Object-Oriented Design**, 2nd edition, WCB McGraw-Hill, 1999.
- [2] Nell Dale, Chip Weems and Mark Headington, **Programming and Problem Solving with C++**, John & Barlett Publisher, 1996.
- [3] Michael T. Goodrich, Roberto Tamassia and David Mount, **Data Structures and Algorithms in C++**, John Wiley & Sons Inc, 2004.
- [4] Scott R. Ladd, **C++ Components and Algorithms**, 2nd edition, M&T Books, 1994.
- [5] Scott R. Ladd, **C++ Templates and Tools**, M&T Books, 1994
- [6] Robert Lafore, **Object – Oriented Programming in C++**, Fourth edition, SAMS, 2001.

Tài liệu tiếng Việt

- [1] Lê Đ. Hưng, Tạ T. Anh, Nguyễn H. Đức và Nguyễn T. Thuý, **Lập trình hướng đối tượng với C++**, NXB Khoa học và Kỹ thuật, 2005.
- [2] Nguyễn T. Thuý, Tạ T. Anh, Nguyễn Q. Huy và Nguyễn H. Đức, **Bài tập lập trình hướng đối tượng với C++**, NXB Khoa học và Kỹ thuật, 2004.

Các địa chỉ web

- 1. <http://www.angelfire.com/country/aldev0/cpphowto/>
- 2. <http://www.gnacademy.org/text/cc/Tutorial/tutorial.html>
- 3. <http://sophia.dtp.fmph.uniba.sk/cpptut/tutorial.us.html>
- 4. <http://www.brpreiss.com/books/opus4/html/book.html>
- 5. <http://www.fredosaurus.com/notes-cpp/index.html>

MỤC LỤC

GIỚI THIỆU	3
CHƯƠNG 1.....	5
GIỚI THIỆU VỀ CÁC PHƯƠNG PHÁP LẬP TRÌNH	5
1.1 LẬP TRÌNH TUYẾN TÍNH.....	5
1.2 LẬP TRÌNH HƯỚNG CẤU TRÚC	5
1.2.1 Đặc trưng của lập trình hướng cấu trúc.....	5
1.2.2 Phương pháp thiết kế trên xuống (top-down)	6
1.3 LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG	7
1.3.1 Lập trình hướng đối tượng.....	7
1.3.2 Một số khái niệm cơ bản.....	8
1.3.3 Lập trình hướng đối tượng trong C++	9
TỔNG KẾT CHƯƠNG 1.....	10
CHƯƠNG 2.....	11
CON TRỞ VÀ MẢNG	11
2.1 KHÁI NIỆM CON TRỞ	11
2.1.1 Khai báo con trở.....	11
2.1.2 Sử dụng con trở.....	11
2.2 CON TRỞ VÀ MẢNG	14
2.2.1 Con trở và mảng một chiều.....	14
2.2.2 Con trở và mảng nhiều chiều	17
2.3 CON TRỞ HÀM.....	18
2.4 CẤP PHÁT BỘ NHỚ ĐỘNG.....	20
2.4.1 Cấp phát bộ nhớ động cho biến	21
2.4.2 Cấp phát bộ nhớ cho mảng động một chiều.....	22
2.4.3 Cấp phát bộ nhớ cho mảng động nhiều chiều.....	23
TỔNG KẾT CHƯƠNG 2.....	25
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 2.....	26
CHƯƠNG 3.....	30
KIỂU DỮ LIỆU CẤU TRÚC	30
3.1 ĐỊNH NGHĨA CẤU TRÚC.....	30
3.1.1 Khai báo cấu trúc	30
3.1.2 Cấu trúc lồng nhau.....	31
3.1.3 Định nghĩa cấu trúc với từ khoá typedef.....	32
3.2 THAO TÁC TRÊN CẤU TRÚC	33
3.2.1 Khởi tạo giá trị ban đầu cho cấu trúc	33
3.2.2 Truy nhập đến thuộc tính của cấu trúc.....	34
3.3 CON TRỞ CẤU TRÚC VÀ MẢNG CẤU TRÚC	38
3.3.1 Con trở cấu trúc.....	38
3.3.2 Mảng cấu trúc	41
3.4 MỘT SỐ KIỂU DỮ LIỆU TRỪU TƯỢNG.....	44
3.4.1 Ngăn xếp.....	45
3.4.2 Hàng đợi.....	48
3.4.3 Danh sách liên kết.....	53
TỔNG KẾT CHƯƠNG 3.....	60
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 3.....	60
CHƯƠNG 4.....	64

VÀO RA TRÊN TỆP	64
4.1 KHÁI NIỆM TỆP	64
4.1.1 Tệp dữ liệu	64
4.1.2 Tệp văn bản	65
4.1.3 Tệp nhị phân	65
4.2 VÀO RA TRÊN TỆP	66
4.2.1 Vào ra tệp văn bản bằng ">>" và "<<"	66
4.2.2 Vào ra tệp nhị phân bằng read và write	70
4.3 TRUY NHẬP TỆP TRỰC TIẾP	74
4.3.1 Con trỏ tệp tin	74
4.3.2 Truy nhập vị trí hiện tại của con trỏ tệp	74
4.3.3 Dịch chuyển con trỏ tệp	76
TỔNG KẾT CHƯƠNG 4	78
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 4	79
CHƯƠNG 5	82
LỚP	82
5.1 KHÁI NIỆM LỚP ĐỐI TƯỢNG	82
5.1.1 Định nghĩa lớp đối tượng	82
5.1.2 Sử dụng lớp đối tượng	83
5.2 CÁC THÀNH PHẦN CỦA LỚP	83
5.2.1 Thuộc tính của lớp	84
5.2.2 Phương thức của lớp	85
5.3 PHẠM VI TRUY NHẬP LỚP	90
5.3.1 Phạm vi truy nhập lớp	90
5.3.2 Hàm bạn	91
5.3.3 Lớp bạn	96
5.4 HÀM KHỞI TẠO VÀ HUỖY BỎ	97
5.4.1 Hàm khởi tạo	97
5.4.2 Hàm huỷ bỏ	101
5.5 CON TRỎ ĐỐI TƯỢNG VÀ MẢNG ĐỐI TƯỢNG	103
5.5.1 Con trỏ đối tượng	103
5.5.2 Mảng các đối tượng	106
TỔNG KẾT CHƯƠNG 5	110
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 5	110
CHƯƠNG 6	115
TÍNH KẾ THỪA VÀ ĐA HÌNH	115
6.1 KHÁI NIỆM KẾ THỪA	115
6.1.1 Khai báo thừa kế	115
6.1.2 Tính chất dẫn xuất	116
6.2 HÀM KHỞI TẠO VÀ HUỖY BỎ TRONG KẾ THỪA	117
6.2.1 Hàm khởi tạo trong kế thừa	117
6.2.2 Hàm huỷ bỏ trong kế thừa	119
6.3 TRUY NHẬP TỚI CÁC THÀNH PHẦN TRONG KẾ THỪA LỚP	120
6.3.1 Phạm vi truy nhập	120
6.3.2 Sử dụng các thành phần của lớp cơ sở từ lớp dẫn xuất	122
6.3.3 Định nghĩa chồng các phương thức của lớp cơ sở	125
6.3.4 Chuyển đổi kiểu giữa lớp cơ sở và lớp dẫn xuất	128
6.4 ĐA KẾ THỪA	131
6.4.1 Khai báo đa kế thừa	131
6.4.2 Hàm khởi tạo và hàm huỷ bỏ trong đa kế thừa	132
6.4.3 Truy nhập các thành phần lớp trong đa kế thừa	134
	183

6.5 LỚP CƠ SỞ TRỪU TƯỢNG	138
6.5.1 Đặt vấn đề	138
6.5.2 Khai báo lớp cơ sở trừu tượng	138
6.5.3 Hàm khởi tạo lớp cơ sở trừu tượng	139
6.6 ĐA HÌNH	143
6.6.1 Đặt vấn đề	143
6.6.2 Khai báo phương thức trừu tượng	144
6.6.3 Sử dụng phương thức trừu tượng – đa hình	144
TỔNG KẾT CHƯƠNG 6	147
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 6	148
CHƯƠNG 7	153
MỘT SỐ LỚP QUAN TRỌNG	153
7.1 LỚP VẬT CHỨA	153
7.1.1 Giao diện của lớp Container	153
7.1.2 Con chạy Iterator	154
7.2 LỚP TẬP HỢP	155
7.2.1 Hàm khởi tạo	155
7.2.2 Toán tử	155
7.2.3 Phương thức	156
7.2.4 Áp dụng	158
7.3 LỚP CHUỖI	159
7.3.1 Hàm khởi tạo	159
7.3.2 Toán tử	160
7.3.3 Phương thức	161
7.3.4 Áp dụng	163
7.4 LỚP NGĂN XẾP VÀ HÀNG ĐỢI	165
7.4.1 Lớp ngăn xếp	165
7.4.2 Lớp hàng đợi	166
7.5 LỚP DANH SÁCH LIÊN KẾT	169
7.5.1 Hàm khởi tạo	169
7.5.2 Toán tử	169
7.5.3 Phương thức	170
7.5.4 Áp dụng	171
TỔNG KẾT CHƯƠNG 7	173
CÂU HỎI VÀ BÀI TẬP CHƯƠNG 7	173
HƯỚNG DẪN TRẢ LỜI CÂU HỎI VÀ BÀI TẬP	174
Chương 1	174
Chương 2	174
Chương 3	174
Chương 4	175
Chương 5	175
Chương 6	176
Chương 7	177
TÀI LIỆU THAM KHẢO	181
MỤC LỤC	182

NGÔN NGỮ LẬP TRÌNH C++

Mã số : 412LTC340

Chịu trách nhiệm bản thảo

TRUNG TÂM ĐÀO TẠO BƯU CHÍNH VIỄN THÔNG 1