TS. DƯƠNG TỬ CƯỜNG

Ngôn ngữ lập trình C++

Từ CƠ BẢN đến HƯỚNG ĐỐI TƯỢNG

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

TS. DƯƠNG TỬ CƯỜNG

NGÔN NGỮ LẬP TRÌNH C++

Từ ***CƠ BẢN***

Đến ***HƯỚNG ĐỐI TƯỢNG***

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
HÀ NỘI

Chịu trách nhiệm xuất bản : **PGS.TS. Tô Đăng Hải**
Biên tập : **ThS. Vũ Thị Minh Luận**
Trình bày bìa : **Hương Lan**

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
70 - Trần Hưng Đạo, Hà Nội

In 1000 bản, khổ 14,5 x 20,5 cm tại Xí nghiệp in NXB Lý luận chính trị
Giấy phép xuất bản số: 150-191, cấp ngày 4/2/2005
In xong và nộp lưu chiểu tháng 10 năm 2005.

LỜI NÓI ĐẦU

Trong những năm 1980, ngôn ngữ C đã khẳng định được vị trí quan trọng trong các ngôn ngữ lập trình có cấu trúc bởi tính đa năng của mình. Một chương trình được thiết kế trên ngôn ngữ C thường phát huy được nhiều tác dụng khó có thể hội tụ ở các ngôn ngữ lập trình khác. Tuy vậy khi độ phức tạp của các bài toán cần giải quyết trên thực tế ngày càng tăng thì ngôn ngữ C cũng đã bộc lộ những điểm yếu, nhất là khi được sử dụng cho các dự án lớn. Để khắc phục những hạn chế còn tồn tại trong ngôn ngữ C nói riêng và của các ngôn ngữ lập trình có cấu trúc nói chung, các nhà thiết kế phần mềm đã phát triển một ý tưởng mới. Các ý tưởng này, mặc dù được xuất hiện từ những năm 1970 nhưng chỉ được sử dụng một cách rộng rãi để xây dựng phần mềm vào những năm 1980. Điểm mấu chốt để xây dựng lên ý tưởng này là khả năng thiết kế những phần mềm mang những đặc tính của thế giới thực bên ngoài. Kỹ thuật lập trình dựa trên ý tưởng mới này có tên "Kỹ thuật lập trình hướng đối tượng" (Object - Oriented - Programing OOP) và trên kỹ thuật mới này nhiều trình biên dịch đã được thiết kế như smalltalk, C^{++}, v.v..

Lập trình định hướng đối tượng được phát triển từ ngôn ngữ lập trình có cấu trúc nhưng thay vì xoay quanh chức năng của nhiệm vụ được đặt ra, OOP lại đặt trọng tâm của mình vào việc xử lý các dữ liệu để thực hiện các chức năng đó. Trong lập trình định hướng đối tượng, khái niệm về object (đối tượng) trở thành

một khái niệm trọng tâm và hầu như mọi công việc trong một chương trình đều được tiến hành trên các đối tượng này.

Các thành phần của một OOP bao gồm: đối tượng, thuộc tính, tác động (phương thức) giao diện và khả năng nhìn thấy của các đối tượng. Mọi đối tượng được thiết lập trong OOP đều là các cấu trúc độc lập bao gồm dữ liệu và các tác động mà đối tượng có thể thực hiện trên các dữ liệu đó. Khái niệm về đối tượng được dùng riêng cho một thực thể riêng biệt hoặc cho một lớp của nhiều thực thể. Một đối tượng chỉ có thể thực hiện các tác động được định nghĩa bên trong nó qua các thông điệp được gửi đến chính bản thân đối tượng này và điều cần lưu ý là chỉ có chính đối tượng này mới có thể thực hiện các tác động đó. Qua thông điệp này đối tượng sẽ nhận dược các nhiệm vụ đòi hỏi nó phải thực hiện. Như vậy, một đối tượng có thể xem như là một đại lượng mà ở đó hội tụ những đặc điểm sau: có tên, có trạng thái, có các tác động mà đối tượng có thể thực hiện và khả năng ẩn đối với các đối tượng khác.

Ngôn ngữ C++ là một trong các ngôn ngữ lập trình được xây dựng từ các ý tưởng mới này và có thể xem C++ là một đại diện điển hình cho phương pháp lập trình mới: lập trình hướng đối tượng. Với ngôn ngữ này, ta có thể làm quen với một số khái niệm mới trong kỹ thuật lập trình liên quan đến thế giới thực: tính đóng góp (encapsulation), tính thừa kế (inheritance) và tính tương ứng bội (polymorphism). Các đặc tính không có trong ngôn ngữ lập trình truyền thống đã làm cho C++ có thể phát huy hết tác dụng của mình khi thiết kế các dự án lớn nhưng cũng đem lại nhiều khó khăn cho các bạn mới bắt đầu với C++. Đã làm quen với C++ thì có thể nhận thấy rằng một chương trình được viết trên ngôn ngữ này sẽ hết sức súc tích, rõ ràng và đặc biệt là

ở một chừng mực nào đó sẽ cho phép phát triển nó theo một suy nghĩ hết sức tự nhiên.

Sự phát triển của C++ đã trải qua nhiều chặng đường với nhiều phiên bản khác nhau. Tài liệu này sử dụng phiên bản của hãng Borland - Borland C 3.1 để giới thiệu về ngôn ngữ C++. Đây là trình biên dịch mà theo chúng tôi rất tiện lợi cho việc nghiên cứu cũng như thiết kế các phần mềm. Cuốn sách này được biên soạn dựa trên nhiều tài liệu khác nhau và trên kinh nghiệm của chính tác giả khi làm việc với C++. Khác với nội dung của một số tài liệu khác, *cuốn sách này được biên soạn để bạn đọc có thể làm việc ngay với C++ mà không cần phải biết về ngôn ngữ C.*

Nội dung của cuốn sách được chia làm 2 phần bao gồm 11 chương

Phần I: C++ - Lập trình cơ bản, bao gồm 6 chương:

Chương I: Các khái niệm cơ bản về C++.

Chương II: Các hàm và các dòng nhập xuất.

Chương III: Các phép toán và câu lệnh điều khiển.

Chương IV: Bộ tiền xử lý.

Chương V: Biến con trỏ, biến tham chiếu và hàm.

Chương VI: Các kiểu dữ liệu phức tạp.

Phần II: Lập trình hướng đối tượng, bao gồm 5 chương:

Chương VII: Lớp và đối tượng.

Chương IIX: Tính thừa kế.

Chương IX: Định nghĩa chồng các hàm và toán tử

Chương X: Tính tương ứng bội.

Chương XI: Thư viện các dòng nhập xuất

Ngôn ngữ lập trình C là một ngôn ngữ lập trình khó và C++ lại còn phức tạp hơn ngôn ngữ C vì vậy đòi hỏi ở chúng ta một tinh thần chịu khó tìm tòi, ham hiểu biết. Tuy vậy, một khi bạn đã nắm chắc được ngôn ngữ này, chúng tôi tin tưởng rằng, trong tay bạn, C++ sẽ trở thành một công cụ hết sức đắc lực và vô cùng tiện lợi giúp bạn giải quyết các bài toán kỹ thuật phức tạp mà trước đó bạn đã phải vất vả khi giải quyết trên ngôn ngữ C và các ngôn ngữ khác. *Chúc các bạn thành công.*

Phần I

C++

LẬP TRÌNH CƠ BẢN

Chương I

CÁC KHÁI NIỆM CƠ BẢN VỀ C++

1.1. CÁC KÝ HIỆU

Các ký hiệu cơ bản bao gồm những ký tự được phép sử dụng trong ngôn ngữ như các chữ cái, số và tổ hợp các ký tự khác. C++ sử dụng một số ký tự sau:

- 52 chữ cái in thường, in hoa và ký tự gạch nối:

  a, b, c, . . .z, A, B, C,. . . Z và _

- Các chữ số:

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- Các ký tự đặc biệt được dùng riêng trong ngôn ngữ:

, . : ; ? | ( ) [ ] + - * / = % & ! ^ ' " { } # $

- Một số ký tự khác như dấu cách (ký tự trắng), dấu xuống hàng và dấu canh theo cột.

Các từ khoá trong C++ được dùng để viết và xây dựng các câu lệnh của chương trình. Cũng giống như C, các từ khoá này luôn được viết bằng chữ thường và không được dùng cho các mục đích khác như đặt tên cho biến, cho hàm v.v.

Có thể liệt kê một số từ khoá được sử dụng trong C++:

auto	*break*	*case*	*char*
continue	*default*	*do*	*double*
else	*entry*	*enum*	*extern*
float	*for*	*goto*	*if*
int	*long*	*register*	*return*
sezeof	*short*	*static*	*struct*
switch	*typedef*	*union*	*class*
unsigned	*while*	*void*	*asm*
fortran	*pascal*	*ada*	*volatile*
const			

1.2. HẰNG

Các thông tin mà giá trị không thể thay đổi trong chương trình được gọi là hằng. Các hằng trong C++ được chia làm một số loại sau:

Hằng số học: Các hằng này có giá trị bằng giá trị của các số mà nó biểu diễn. Hằng số học lại được chia làm hai loại

- *Hằng số nguyên:* Loại hằng số này có thể được biểu diễn dưới dạng cơ số 10, cơ số 8 và cơ số 16.

Các hằng cơ số 10 có thể là số dương hoặc âm. Cần lưu ý là không được dùng số 0 như là chữ số đầu tiên của hằng số nguyên.

Ví dụ

+14 - Hằng số nguyên dương có giá trị là 14;

-12 - Hằng số nguyên âm có giá trị là âm 12;

012 - Không phải là hằng số nguyên

Các hằng số nguyên cơ số 8 là các hằng không có dấu và được biểu diễn bằng các chữ số trong hệ đếm cơ số 8 - các số từ 0 đến 7. Chữ số đầu tiên trong các hằng kiểu này phải là chữ số 0. Đây chính là nguyên nhân chữ số 0 không được sử dụng như chữ số đầu tiên của hằng cơ số 10.

Ví dụ:

014 - Hằng cơ số 8 với giá trị là 12 (cơ số 10)

0114 - Hằng cơ số 8 với giá trị 76 (cơ số 10)

Các hằng số nguyên cơ số 16 được viết không có dấu và luôn được bắt đầu bởi 0X hoặc 0x. Trong hệ đếm cơ số 16 ngoài các chữ số từ 0 đến 9 người ta còn sử dụng các chữ cái từ A đến F để biểu thị các số từ 10 đến 15.

Ví dụ:

0XC hoặc 0xc - Hằng cơ số 16 có giá trị 12 (cơ số 10)

0XFF hoặc 0xff - Hằng cơ số 16 có giá trị 255 (cơ số 10)

- *Hằng số thực:* Là một loại hằng chỉ được được viết dưới

dạng cơ số 10. Một hằng số thực có thể được biểu diễn dưới dạng thập phân hay dưới dạng khoa học

Ví dụ:

12.0 - Hằng số thực dương có giá trị là 12.0

-12.5 - Hằng số thực âm có giá trị là -12.15

1.2E1 hoặc 1.2e1 - Hằng số thực với giá trị 12

Hằng ký tự bao gồm các ký tự đơn được viết trong dấu nháy đơn, ví dụ 'A', 'B', '1'. Mỗi ký tự có giá trị bằng mã ASCII của nó. Ví dụ mã của 'A' là 65 và '0' là 48.

Hằng kiểu chuỗi là một dãy ký tự được viết trong dấu nháy kép, ví dụ: "this is a string" là một hằng kiểu chuỗi. Đối với hằng kiểu chuỗi cần lưu ý:

- Độ dài của hằng kiểu chuỗi là không hạn chế và có thể không chứa ký tự nào.
- Trình biên dịch sẽ tự động đưa thêm một byte có giá trị bằng 0 (NULL) vào sau chuỗi trong quá trình biên dịch. Để phân biệt với ký tự '0', byte này được viết dưới dạng '\0'. Byte này được xem như dấu hiệu kết thúc của chuỗi.
- Nếu dùng mảng một chiều để lưu trữ hằng kiểu chuỗi thì số phần tử của mảng phải lớn hơn số ký tự của chuỗi là một đơn vị (để chứa byte NULL).

Các ký tự dùng riêng (ký tự không in ra được) có thể được sử dụng như những ký tự bình thường nhưng khi viết chúng phải thêm ký tự \ ở trước. Chẳng hạn khi chỉ đường dẫn như một hằng kiểu chuỗi, thay cho "C:\borlandc\bgi" ta phải viết "C:\\borlandc\\bgi".

Có một số ký tự thuộc dạng không in ra được (có giá trị ASCII từ 0 đến 31) nhưng trình biên dịch có thể nhận biết được chúng thông qua một cặp ký tự (được viết bắt đầu bằng ký tự \). Các ký tự này thường được dùng để điều khiển màn hình hoặc máy in. Bảng 1.1 mô tả một số ký tự thường hay được sử dụng:

Bảng 1.1

Tác dụng	Mô tả trong C	Mã ASCII	Ký hiệu
Byte trắng	\0	0	NULL (null byte)
Lùi	\b	8	BS (Backspace)
Về đầu dòng	\r	13	CR (Carier Return)
Canh cột	\t	9	T (Horizontal Tab)
Xuống dòng	\n	10	NL (New line)
Sang trang	\f	12	FF (Form feed)
Tín hiệu chuông	\a	7	bel (Bell)

1.3 BIẾN

Biến là một đại lượng thuộc một kiểu nhất định (số nguyên, số thực v.v...) mà giá trị có thể thay đổi trong quá trình thực hiện chương trình. Mục đích của biến là dùng để lưu trữ dữ liệu và việc sử dụng biến được thực hiện thông qua tên của nó. Khi đặt tên cho biến cần phải đảm bảo các qui tắc sau:

- Tên bao gồm một dãy các chữ cái, số và phải bắt đầu

bằng chữ cái. Không sử dụng các ký tự đặc biệt để đặt tên cho biến kể cả dấu cách.

- Tên của biến không được trùng với các từ khoá.
- Có sự phân biệt giữa chữ thường và chữ hoa khi đặt tên cho biến.

1.4. CÁC LOẠI DỮ LIỆU VÀ CÁCH KHAI BÁO

Tất cả các biến trong ngôn ngữ C và C++ đều phải được khái báo trước khi sử dụng. Thông qua việc khai báo này trình biên dịch xác định kích thước của biến (cấp phát bộ nhớ để lưu trữ giá trị của biến trong bộ nhớ).

Khai báo biến được thực hiện qua cú pháp sau:

Kiểu biến Tên biến ;

Dưới đây sẽ chỉ ra một số từ khoá được dành cho việc khai báo biến, kích thước của biến và khoảng giá trị có thể được lưu trữ trong biến khi khai báo bằng các từ khoá này (đối với các loại máy sử dụng 16 bit dữ liệu):

Bảng 1.2

Kiểu	Kích thước (bit)	Giá trị
char	8	[-126 ÷ 125]
unsigned char	8	[0 ÷ 255]
int	16	[-32768 ÷ +32767]
unsigned int	16	[0 ÷ 65535]
long int	32	[-2e9 ÷ +2e9]
float	32	[-10e-37 ÷ 10e37
double	64	[-10e-307 ÷ 10e 307]

Trong các kiểu biến này:

char được dùng khai báo cho dữ liệu kiểu ký tự;

int - dữ liệu kiểu số nguyên;

float - dữ liệu kiểu số thực;

double - dữ liệu kiểu số thực với độ chính xác gấp đôi;

unsigned - dữ liệu không dấu (chỉ dùng cho char và int);

long - Dữ liệu có độ dài lớn hơn gấp hai lần (dùng cho int và float).

Các biến có thể được khai báo trên cùng một dòng (ví dụ int x, y) hoặc trên các dòng khác nhau, chẳng hạn:

```
int x, y;
float z;
int a;
```

Về vị trí khai báo các biến, trong khi ngôn ngữ C đòi hỏi việc khai báo các biến phải được thực hiện ở trước phạm vi mà chúng được sử dụng (thông thường khi bắt đầu định nghĩa hàm) thì việc khai báo biến trong C++ chỉ cần được thực hiện trước khi các biến này được sử dụng.

1.5. MẢNG

Mảng là tập hợp các biến có cùng kiểu được phân bố liên tục trong bộ nhớ. Khi sử dụng mảng cũng cần phải lưu ý:

- Mảng phải được đặt tên trước khi sử dụng. Nguyên tắc đặt tên của mảng giống như biến;
- Mỗi phần tử của mảng được xem là một biến;
- Mảng phải được khai báo trước khi sử dụng

Dưới đây là một số ví dụ về khai báo mảng trong C++:

int x[5]; mảng x với 5 phần tử thuộc kiểu int;

char a[3]; mảng a với 3 phần tử thuộc kiểu char.

Phần tử của mảng có thể được truy cập thông qua chỉ số và phần tử đầu tiên của mảng trong C++ luôn có chỉ số 0. Như vậy *x[1]* là phần tử thứ 2 của mảng và *x[0]* là phần tử đầu của mảng.

Cần lưu ý là thông thường các trình biên dịch của C++ không kiểm tra giá trị được gán cho chỉ số các phần tử của mảng. Điều này đòi hỏi người lập trình phải hết sức chú ý khi làm việc với chỉ số của mảng.

Ngoài mảng một chiều như đã chỉ ra trong các ví dụ trên, C++ còn cho phép định nghĩa nhiều kiểu mảng khác nhau. Nếu để phân biệt các phần tử của mảng mà phải dùng đến 2 chỉ số thì mảng được gọi là hai chiều.

Ví dụ về mảng 2 chiều:

int a[3][3];

Mảng này gồm 9 phần tử và tên của các phần tử lần lượt là:

a[0][0], a[0][1], a[0][2]

a[1][0], a[1][1], a[1][2]

a[2][0], a[2][1], a[2][2]

Như vậy đối với mảng hai chiều thì chỉ số đầu của phần tử dùng để chỉ hàng và chỉ số thứ hai dùng để chỉ cột. Các phần tử của mảng được lưu trữ trong bộ nhớ theo hàng.

Theo định nghĩa ta cũng có thể khai báo một mảng thuộc kiểu *char*. Về thực chất mảng này sẽ được dùng để lưu trữ xâu ký tự. Mảng kiểu *char* được khai báo như sau:

char name[12];

Kích thước của mảng này là 12. Như vậy mảng này có thể dùng để lưu trữ 11 ký tự, ký tự cuối cùng - ký tự NULL sẽ được trình biên dịch tự động đưa thêm.

1.6. CHÚ GIẢI

C++ sử dụng cả hai loại chú giải. Loại thứ nhất dành cho một khối văn bản được viết giữa hai ký hiệu /* (bắt đầu) và */ (kết thúc). Đây cũng là loại chú giải đã được dùng trong C. Loại thứ hai dành cho một đoạn văn bản được tính từ hai ký tự // cho đến cuối dòng. Loại này chỉ dùng cho C++.

1.7. CẤU TRÚC CỦA CHƯƠNG TRÌNH C++

Tất cả các chương trình trong C++ đều được xây dựng từ các hàm. Số lượng hàm trong chương trình là không hạn chế nhưng bao giờ cũng phải có ít nhất là một hàm có tên gọi là *main*. Thông qua hàm này chương trình sẽ thực hiện việc giao tiếp với hệ điều hành. Khái niệm về hàm sẽ được đề cập chi tiết trong chương V.

Sau đây là một ví dụ cụ thể về một chương trình được viết trong C++:

```
#include <iostream.h>
void main ( )
{
    int i;
    i = i+1;
    cout << i;
}
```

Trong khi lập trình ta có thể sử dụng một số hàm của thư

viện chuẩn. Các hàm này thường đã viết sẵn bởi các nhà cung cấp và được khai báo trong *các header file* (file tiêu đề). Các file này thường có phần mở rộng là *h* (header). Chương trình sẽ thực hiện việc nối kết các file tiêu đề này vào file nguồn bằng chỉ thị của bộ tiền xử lý *#include <tên file>* được viết ở đầu chương trình. Trong ví dụ trên, file iostream.*h* có chứa khai báo của dòng xuất *cout* - dùng để hiển thị thông tin lên màn hình; Trong ví dụ, ta chỉ sử dụng một hàm duy nhất là hàm *main*. Hàm này có kiểu *void* có nghĩa là không nhận giá trị trả về. Như vậy từ khoá *void* có tác dụng định nghĩa một hàm như một thủ tục trong Pascal. Cũng cần lưu ý là một hàm bao giờ cũng được bắt đầu bằng ký tự { và kết thúc bằng ký tự }. Các ký tự này cũng dùng cho việc bắt đầu và kết thúc một khối lệnh.

Chương II

CÁC HÀM VÀ CÁC DÒNG NHẬP XUẤT

Để thuận tiện cho việc xây dựng các ví dụ minh hoạ tiếp theo, phần này sẽ đề cập đến các khả năng nhập xuất thông tin trong C++.

2.1. CÁC HÀM NHẬP XUẤT THÔNG TIN

Với mục đích giúp bạn đọc có thể tiếp cận được với những chương trình viết trên C, phần dưới đây sẽ đề cập ngắn gọn đến các hàm nhập và xuất thông tin được sử dụng trong cả C và C++. Khi sử dụng các hàm này, chương trình cần phải sử dụng file tiêu đề *stdio.h*

2.1.1. Hàm xuất thông tin - hàm printf

Hàm *printf* có cú pháp sau:

printf("Tham số định dạng", Các tham số);

Hàm này dùng để hiển thị dữ liệu lên màn hình. Các tham số được truyền cho hàm bao gồm:

Tham số định dạng được đặt trong dấu nháy kép và có thể bao gồm các mã định dạng.

Các tham số được viết trong hàm phải được viết cách nhau bởi dấu phẩy. Đây là các biến hoặc biểu thức hoặc thậm chí là hằng mà giá trị cần hiển thị lên màn hình. Các mã định dạng

chỉ được sử dụng khi cần hiển thị giá trị của các tham số. Đối với mã định dạng cần lưu ý:

- Có bao nhiêu tham số phải có bấy nhiêu mã định dạng và mỗi mã định dạng sẽ lần lượt tương ứng với một tham số tính từ trái qua phải;
- Khi hiển thị, giá trị của các tham số sẽ được hiển thị tại vị trí của mã định dạng tương ứng.
- Kiểu của mã định dạng phải tương ứng với kiểu giá trị cần hiển thị của tham số.

Dưới đây là một số mã định dạng hay được sử dụng:

%d - Hiển thị kiểu int

%c - Hiển thị kiểu char

%f - Hiển thị kiểu float

%lf - Hiển thị kiểu double

%s - Hiển thị kiểu mảng char

%u - Hiển thị kiểu unsigned

Bạn đọc có thể theo dõi việc sử dụng hàm *printf* qua một số ví dụ dưới đây:

Ví dụ 2.1

```
#include <stdio.h>
void main ( )
{
    printf("Ngôn ngữ C là ngôn ngữ khó\n");
    printf("Vì vậy cần chịu khó khi học ngôn ngữ này");
}
```

Trong ví dụ này các mã định dạng không được sử dụng và kết quả chương trình sau khi thực hiện là:

Ngôn ngữ C là ngôn ngữ khó

Vì vậy cần chịu khó khi học ngôn ngữ này

Ký tự \n trong mã định dạng thứ nhất sẽ có tác dụng xuống dòng sau khi đã hiển thị thông tin trong tham số định dạng. (Kết quả này cũng có được nếu ta đặt ký tự này trong câu lệnh thứ hai - printf("\nVì vậy cần chịu khó khi học ngôn ngữ này");)

Ví dụ 2.2

```
#include <stdio.h>
#include <conio.h>
void main( )
{
    clrscr( );
    int a =7;
    float b =4.5;
    char s[20] = "Tổng của a và b là =";
    printf("Giá trị của biến a là %d", a);
    printf("Giá trị của biến b là %f", b);
    float Tong;
    Tong = a + b;
    printf("%s %d",s,Tong);
    getch( );
}
```

Trong ví dụ có sử dụng thêm hàm clrscr( ) và hàm getch( ).

Hàm clrscr( ) dùng để xoá màn hình, trong khi hàm getch ( ) sẽ có tác dụng dừng màn hình cho đến khi người sử dụng gõ phím bất kỳ. Để sử dụng các hàm này, chương trình cần phải nối kết thêm file tiêu đề *conio.h*. Bạn đọc có thể chạy chương trình này và lưu ý cách sử dụng các tham số định dạng trong hàm *printf*.

2.1.2. Hàm nhập dữ liệu - scanf

Hàm được sử dụng qua cú pháp sau:

scanf("Mã định dạng", Danh sách biến);

Trong hàm này, các biến cần được nhập giá trị từ bàn phím được chứa trong *danh sách biến*. Các biến này phải được viết cách nhau bởi dấu phảy và mỗi biến phải được viết sau ký tự & (địa chỉ của biến) trừ các biến thuộc kiểu mảng *char*. Việc sử dụng các mã định dạng trong hàm *scanf* hoàn toàn tương tự như trong câu lệnh *printf*. Khi gặp câu lệnh, chương trình sẽ tạm dừng để người sử dụng nhập dữ liệu từ bàn phím, việc nhập dữ liệu cho từng biến có thể được kết thúc bằng một trong các phím *Enter*, phím *canh cột* (Tab) và phím *cách* (Space bar). Dưới đây là ví dụ về cách sử dụng hàm này.

Ví dụ 2.3

```
#include <conio.h>
#include <stdio.h>
void main( )
{
    clrscr ( );
    int tuoi;
    char ten[30]; // Biến thuộc kiểu mảng char
    float can_nang;
```

```
    printf("Hãy đưa tên tuổi và cân nặng của bạn:\n");
    scanf("%s%d%f",ten, &tuoi, &can_nang);
    printf("Tên của bạn là %s\n", ten);
    printf("Tuổi của bạn là %d\n", tuoi);
    printf("Cân nặng của bạn là %f  kg", can_nang);
    getch( );
}
```

Bạn đọc thực hiện chương trình này, theo dõi kết quả và chú ý cách sử dụng hàm các mã định dạng trong hàm *scanf*.

Chi tiết về cách sử dụng các hàm *printf* và *scanf* có thể tham khảo trong phần trợ giúp của trình biên dịch.

2.1.3. Hàm nhập ký tự - getchar( )

Hàm này cho phép nhập một ký tự từ bàn phím và gán cho biến. Hàm này được khai báo trong file tiêu đề *stdio.h*. Hàm có cú pháp sau:

Tên_biến = getchar( );

Khi gặp hàm *getchar( )*, chương trình sẽ tạm dừng cho đến khi ta gõ một ký tự từ bàn phím. Chương trình sẽ tiếp tục chạy sau khi nhấn phím *Enter*. Như vậy có thể thấy việc sử dụng hàm *getchar* hoàn toàn tương đương với hàm *scanf* để nhập dữ liệu cho biến kiểu ký tự với mã định dạng *%c*.

2.2. CÁC DÒNG NHẬP XUẤT

Ngoài các hàm nhập/ xuất đã được giới thiệu, ngôn ngữ C++ còn sử dụng các kỹ thuật nhập xuất dữ liệu thông qua các dòng nhập xuất. Các dòng nhập xuất (Streams) là một kỹ thuật hoàn toàn mới trong C++ để điều khiển quá trình nhập và xuất dữ liệu

trong quá trình thực hiện chương trình.

Nhìn lại các hàm *printf(...), scanf(...),* có thể thấy các hàm này đã tỏ ra rất hữu hiệu đối với người sử dụng. Tuy vậy, khi làm việc với ngôn ngữ C++, một ngôn ngữ lập trình theo hướng đối tượng, các hàm trên đã thể hiện một số nhược điểm của chúng. Trong khi C++ cho phép người sử dụng đưa ra các kiểu dữ liệu mới, thì các hàm *printf(...), scanf(...), sprintf(...), sscanf(...)* v.v... lại hạn chế người sử dụng ở một số khuôn dạng nhập xuất cố định, không thể thay đổi. Thêm vào đó, các hàm này không nhất quán về thứ tự và ngữ nghĩa của các tham số.

Trong C++, *streams* được xây dựng theo một ý tưởng hoàn toàn mới - thông qua các lớp (khái niệm về lớp sẽ được đề cập đến ở chương VII). Các lớp này được gọi là thư viện các dòng nhập xuất đã tạo ra một khả năng rất mạnh cho phép sửa đổi người sử dụng định nghĩa. Trong C++, *stream* là một nguồn hoặc một đích của các thao tác nhập và xuất dữ liệu. Các dòng nhập (*ostream*) cho phép nhập dữ liệu vào các *stream* trong khi các dòng xuất (*istream*) lại cho phép dọc dữ liệu. Để thuận tiện trong quá trình trình bày về *stream* tuy vậy chúng tôi xin dùng các thuật ngữ *stream* nhập đối với *istream* vì *stream* này cho phép nhập dữ liệu từ *stream* vào các biến *stream* xuất (*ostream*) vì chúng có tác dụng xuất dữ liệu đến một đích cụ thể - màn hình hoặc là file.

Thư viện các dòng nhập xuất là một cấu trúc cây của các lớp. Cấu trúc cây này sẽ được đề cập chi tiết ở chương XI .

Streams được sử dụng hoàn toàn độc lập với các hàm vào/ ra trong thư viện chuẩn *stdio.h*. Việc sử dụng đồng thời *streams* và các hàm này trong cùng một chương trình có thể làm phát sinh nhiều vấn đề. Nếu trong chương trình đã sử dụng *stream* để

xuất một ký tự, sau đó lại dùng hàm của *stdio.h* để xuất một ký tự nữa thì các ký tự này có thể sẽ không được gửi đi theo thứ tự cần thiết. Điều này phụ thuộc vào nhiều nguyên nhân. Chẳng hạn trong ví dụ sau, việc sử dụng cả *cout* và *printf* có thể sẽ không đảm bảo cho việc gửi kết quả ra màn hình theo thứ tự được đưa ra:

Ví dụ 2.4

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
void main()
    {
    clrscr();
    int value =3;
    cout <<"\n Có";
    printf("tất cả %d",value);
    cout <<"giá trị";
    getch( );
    }
```

2.2.1. Stream với các kiểu dữ liệu chuẩn

Các *stream* này thường được sử dụng với các cú pháp sau:

Input_stream >> Typed_variable;

Output_stream <<Typed_variable;

Trong đó:

Input_stream: các dòng nhập.

Output_stream: các dòng xuất;

Typed_variable: các biến có kiểu

Các toán tử dịch phải (>>) và dịch trái (<<) được định nghĩa chồng để nhập và gửi dữ liệu đến các đối tượng của dòng nhập xuất.

Ví dụ 2.5

```
#include <iostream.h>
#include <conio.h>
void main()
    {
    clrscr( );
    int a;
    char c;
    float f;
    double d;
    // Nhập dữ liệu
    cin >>a;
    cin >>c;
    cin >>f;
    cin >>d;
    // Xuất dữ liệu ra màn hình.
    cout <<a;
    cout <<c;
    cout <<f;
    cout <<d;
    cout <<"Đây là chuỗi";
    getch ( );
}
```

Trong C++, *cin* là một *stream* nhập chuẩn, *stream* này cho phép nhập một ký tự từ bàn phím. Như vậy có thể xem *cin* thay thế cho *stdin* với cùng mục đích trong ANSI C. Stream *cout* cũng tương tự như *stdout* trong ANSI C, dùng để xuất dữ liệu ra các thiết bị như màn hình v.v. Ngoài hai *stream* này còn có hai *stream* chuẩn được định nghĩa trong C++; *cerr* (thay thế cho *stderr*) và *clog* (không có trong ANSI C), *clog* cũng tương tự như *cerr*, tuy vậy khác với *cin, cout, cerr, stream* này có sử dụng bộ đệm khi thao tác với dữ liệu.

Các *stream* chuẩn như *cin, cout, cerr, clog* được tự động mở trước khi hàm *main* được thực hiện và tự động đóng lại trước khi hàm *main* kết thúc công việc.

Để sử dụng các *stream* trên, trong chương trình cần sử dụng File tiêu đề *iostream.h*. Việc truy xuất đến các kiểu dữ liệu khác nhau có thể được thực hiện liên tiếp trên cùng một dòng. Toán tử >> được gọi là *extract operation* và toán tử << được gọi là *insert operation*.

Ví dụ 2.6

```
#include <iostream.h>
void main()
    {
    int a;
    char c;
    float f;
    double d;
    cin >>a>>c>>f>>d; // (1)
    cout<<a<<c<<f<<d; // (2)
    }
```

Khi thực hiện việc nhập và xuất dữ liệu trên cùng một dòng, các toán tử >> và << được thực hiện lần lượt từ trái qua phải. Theo thứ tự này, trong chương trình, dữ liệu được nhập vào biến *a* trước biến *c* v.v... trong câu lệnh (1) và *c* được hiển thị sau *a* trong (2). Cũng nhận thấy rằng, khi sử dụng các *stream* chuẩn *cin* và *cout*, trình biên dịch sẽ tự động thực hiện việc chuyển kiểu cho phù hợp với kiểu dữ liệu đang được xử lý. Đây là một điểm mạnh không có trong các hàm của thư viện *stdio.h*.

2.2.2. Sử dụng stream đối với các dữ liệu kiểu char và char*

Các phép toán làm việc với dữ liệu kiểu ký tự (character) trên *stream* tương đối đơn giản vì chúng hầu như không đòi hỏi phải thiết lập khuôn dạng tương ứng với kiểu của chúng. Khi thực hiện việc đọc dữ liệu từ *stream* (chẳng hạn với *cin*) các ký tự sẽ được đọc cho đến khi gặp một ký tự trắng. Các ký tự trắng đó có thể là dấu cách, dấu canh cột, hoặc xuống dòng. Nếu trong dãy dữ liệu cần nhập có dấu cách, các ký tự đó phải được loại bỏ hoặc phải được xử lý.

Ví dụ 2.7

Nhập một ký tự từ bàn phím

```
#include<iostream.h>
void main ( )
{
    char c;
    cout <<"\n Hãy gõ một ký tự";
    cin >>c;
    cout <<"\n Ký tự được gõ là ::" <<c;
}
```

Trong ví dụ này, chương trình sẽ hiển thị dòng *"Hãy gõ một ký tự"* và chờ người sử dụng gõ một phím bất kỳ qua lệnh *cin* >> c. Giá trị này của *c* sau đó sẽ sử dụng trong dòng lệnh xuất tiếp đó. Cần lưu ý là câu lệnh *cin* >> c sẽ chờ kỳ tự từ bàn phím và kết thúc khi nhập dấu cách (ký tự trắng, dấu canh cột hay xuống dòng). Để có thể nhập dữ liệu mà không cần kết thúc bởi phím *Enter* chúng ta có thể sử dụng các hàm chuẩn của C như *getche()* hoặc *getch()*.

Ví dụ 2.8

Nhập một chuỗi từ bàn phím.

```
#inlude <iostream.h>
void main()
    {
    char buffer[20];
    cin >> buffer;
    cout <<buffer
    }
```

Trong chương trình này, khi thực hiện dòng lệnh *cin* >> *buffer*, nếu ta gõ vào chuỗi ký tự "Wait Please", thì chỉ có chuỗi "Wait" được hiển thị còn "Please" sẽ ở lại trong bộ đệm của vùng nhập.

2.2.3. Sử dụng Stream đối với các dữ liệu kiểu double và float

Các thao tác nhập và xuất trên các kiểu dữ liệu *float* và *double* cũng được thực hiện dễ dàng giống như các kiểu dữ liệu *int* hay *long int*. Quá trình nhập dữ liệu từ *stream* sẽ kết thúc

khi gặp ký tự trắng hoặc các ký tự không hợp lệ đối với *float* và *double*.

Ví dụ 2.9

```
#include <iostream.h>
void main()
    {
    cout <<"\n Hãy nhập một số"
    float f;
    cin >>f;
    cout <<"\n Hãy nhập số khác";
    double d;
    cin >>d;
    cout <<"\n Hai số được nhập vào là "<<f<<" và "<<d;
    }
```

Khi thực hiện chương trình trên, nếu hai số được nhập là 1.234 và 5.6789 thì kết quả của việc xuất dữ liệu bằng *stream* sẽ là:

Hai số được nhập vào là 1.234 và 5.6789.

2.2.4. Các hàm thành phần định dạng

Khi sử dụng các *stream* nhập và xuất dữ liệu, C++ còn đưa ra một số hàm thành phần (hay còn gọi là phương thức) xây dựng sẵn. Các hàm thành phần này là các hàm thành phần của lớp nhập xuất (Xem hàm thành phần chương VII) và dùng để điều khiển khuôn dạng cho các quá trình nhập và xuất dữ liệu. Dưới đây sẽ xem xét một số hàm thành phần hay sử dụng.

2.2.4.1. Hàm thành phần width()

Hàm thành phần này cho phép xác định số ký tự tối đa được cất vào trong một vùng đệm. Ví dụ :

```
char buffer[15];
cin.width(15)
cin >> buffer;
```

Cần chú ý là khi được sử dụng, các hàm thành phần phải luôn gắn liền với một đối tượng nhập xuất cụ thể. Các kiến thức về sử dụng các hàm thành phần (hàm thành phần) sẽ được đề cập chi tiết hơn ở các chương sau. Ở đây chỉ giới thiệu ngắn gọn cách gọi một hàm thành phần của C++. Một hàm thành phần của một đối tượng cần được gọi qua cách sau:

Đối_tượng.Phương_thức;

Trong ví dụ trên đối tượng của *cin* gọi *width* qua dòng lệnh:

```
cin.width(15);
```

Sau khi thực hiện lệnh này, *cin* chỉ được phép nhập 15 ký tự từ bàn phím trong một lần nhập.

Hàm thành phần *width* cũng được áp dụng cho cả dòng xuất, kết quả của việc thực hiện đoạn chương trình:

```
int x=1;
cout.width(5);
cout <<x;
```

sẽ hiển thị giá trị 1 trên màn hình. Ở đây chiều dài của vùng hiển thị kết quả là 5 ký tự. Giá trị này sẽ được căn theo lề phải của vùng hiển thị theo mặc định. Nếu kích thước của *x* (tính theo số ký tự) có độ dài lớn hơn giá trị được thiết lập trong *width(...)* thì toàn bộ giá trị của *x* sẽ được hiển thị.

Giá trị mặc định của *width* cho một dòng xuất nhập là 0. Giá trị này cho phép việc xuất nhập được thực hiện trên chiều dài thực của một số. Sau mỗi lần xuất nhập, *width* sẽ được đặt lại giá trị 0.

Ví dụ đoạn chương trình:

```
int x=3,y=5;
cout.width(5);
cout<<x<<" "<<y;
```

sẽ xuất ra giá trị của *x* trong một vùng có kích thước là 5 ký tự, hiển thị một khoảng trắng và sau đó hiện giá trị của *y* với kích thước đúng của nó.

2.2.4.2. Hàm thành phần precision()

Hàm thành phần này cho phép thiết lập số chữ số xuất hiện sau dấu chấm thập phân khi xuất ra một số thuộc kiểu *float* hay *double*. Giá trị này được truyền qua tham số của hàm thành phần. Sau khi thực hiện, *precision* xác định cách hiển thị dữ liệu mới, đồng thời trả lại giá trị bằng số chữ số sau phần thập phân được sử dụng trước đó. Khi không được truyền tham số, hàm thành phần này sử dụng tham số mặc định với giá trị là 6. Ví dụ:

```
const float pi = 3.1415927;
cout.precision(2);
cout <<pi;          //(1)
int pre_preci = cout.precision(4);
cout<<pi;     //(2)
cout.precison(pre_preci); //(3)
cout<<pi;     //(4)
```

Dòng lệnh (1) sẽ xuất giá trị của *pi* lên màn hình với 2 chữ số thập phân. Dòng lệnh (2) sẽ in giá trị của *pi* lên màn hình với 4 chữ số thập phân. Dòng lệnh (3) thiết lập lại tham số định dạng được sử dụng trước đó (sử dụng 2 chữ số thập phân) và tham số định dạng này lại được sử dụng trong dòng lệnh (4).

2.2.4.3. Hàm thành phần fill()

Hàm thành phần này cho phép xác định ký tự dùng để độn vào khoảng trắng của vùng xuất dữ liệu được xác định bởi hàm thành phần *width()* khi kích thước của dữ liệu nhỏ hơn giá trị được thiết lập bởi *width()*. Ký tự độn mặc định là khoảng trắng (space). Đoạn chương trình sau sẽ minh họa cho việc sử dụng ký tự độn:

```
int x =15;
cout.width(5); //Độ rộng vùng xuất - 5 ký tự
cou.fill(*); //Ký tự độn là '*'
cout<<x;
```

Sau khi sử dụng các hàm thành phần *width()* và *fill()*, lệnh *cout<<x* sẽ đưa ra màn hình kết quả: ***15.

2.2.4.4. Hàm thành phần ignore()

Hàm thành phần này sẽ cho phép dòng xuất nhập bỏ qua một số ký tự, không phụ thuộc các ký tự đó có phải là ký tự trắng hay không. Đối số được truyền cho hàm thành phần sẽ xác định số lượng các ký tự cần bỏ qua. Chẳng hạn lệnh cin.*ignore(7)* sẽ cho phép bỏ qua 7 ký tự trong lần nhập kế tiếp.

2.2.4.5. Hàm thành phần putback()

Hàm thành phần này cho phép trả lại ký tự vừa được đọc trước đó trở lại bộ đệm dùng cho thao tác nhập. Ví dụ, sau khi thực hiện các dòng lệnh:

```
char ch, ch1;
cin >> ch1;
cin >> ch;
cin.putback(ch); //Đưa ch trở lại bộ đệm.
cin >> ch1; //Đọc ký tự trong bộ đệm ch1 = ch.
```

ch1 sẽ có giá trị bằng giá trị của *ch*.

2.2.4.6. Hàm thành phần get()

Hàm thành phần *get* cho phép đọc một ký tự từ bàn phím không phân biệt ký tự đó là dấu cách hay không. Hàm thành phần này có thể được sử dụng ở một số dạng sau:

get(char* str, int len, char delim = ' n');

Hàm đọc các ký tự từ dòng nhập (Input stream) vào mảng ký tự được chỉ ra bởi *str*. Quá trình đọc này sẽ dừng lại khi đã đọc đến ký tự thứ *len* hoặc khi gặp ký tự được chỉ ra bởi *delim* (giá trị ngầm định của ký tự này là ' n'). *Ký tự kết thúc sẽ không* được lấy vào.

get(unsigned char & ch);

Đọc ký tự kế tiếp từ dòng xuất nhập và cất ký tự này vào *ch*.

2.2.4.7. Hàm thành phần getline()

Hàm thành phần được sử dụng dưới dạng sau:

getline(char* ste, int len, char delin = '\n');

Hàm này làm việc giống như hàm *get* ở trên với một khác biệt nhỏ là nó lấy cả ký tự kết thúc.

Ví dụ 2.10

```
#include <iostream.h>
void main()
{
    char s[30];
    char s1[30];
    cout <<"Hãy nhập một xâu ký tự"<<endl;
    cin >>s; //(1)
    cout <<s; //(2)
    cout <<"Hãy nhập một xâu ký tự khác" <<endl;
    cin.getline(s1,30); //(3)
    cout<<s1;
}
```

Trong ví dụ này, khi thực hiện đến dòng lệnh (1), chương trình sẽ đòi hỏi nhập một xâu ký tự từ bàn phím. Chẳng hạn nếu xâu này là *"Ngon ngu lap trinh C++ "* thì nội dung của s sẽ là một chuỗi được nhập cho đến khi gặp dấu cách, trường hợp này là "Ngon". Giá trị này sẽ được xuất lên màn hình bằng dòng lệnh (3). Bằng hàm thành phần *getline*, s1 sẽ nhận nội dung còn lại trong bộ đệm sau lần nhập trước mà không phân biệt dấu cách - trường hợp này là *"ngu lap trinh C++"*.

2.2.4.9. Hàm thành phần put()

Hàm thành phần này dùng để xuất một ký tự ra màn hình.

Ví dụ sau lệnh *cout.put('a')*; ký tự 'a' sẽ được hiển thị lên màn hình.

2.3. XỬ LÝ KHUÔN DẠNG NHẬP XUẤT

2.3.1. Các bộ phận xử lý định dạng

Trong ngôn ngữ C, việc tạo khuôn dạng cho dữ liệu nhập và xuất được thực hiện ngay trong các hàm dùng cho mục đích tương ứng. Các hàm này bao gồm *frintf(...), scanf(...), sprinrf(...), sscanf(...)* v.v và việc sử dụng chúng vẫn có nhiều hạn chế như đã đề cập đến ở trên. Hơn nữa, trong C++ việc sử dụng các hàm của C cùng các *streams* là một điều nên tránh. Để giải quyết vấn đề này, trong C++ có sử dụng các bộ phận xử lý khuôn dạng nhập xuất (*Manipulator*) mà thực chất là các hàm đặc biệt được thiết kế để phù hợp với các *streams*. Trong thư viện *iostream.h* có chứa một số *manipulator* có sẵn đối với các loại dữ liệu chuẩn. Các *manipulator* này có thể được mở rộng đối với các kiểu dữ liệu mới do người sử dụng định nghĩa. Mặc dù một số thao tác định dạng có thể được xác định qua các hàm thành phần, nhưng việc sử dụng chúng thường rườm rà và phức tạp hơn so với sử dụng các *manipulator*. Thông thường các *manipulator* được sử dụng để:

- Xác định độ rộng của vùng chứa dữ liệu.
- Độ chính xác của phần thập phân.

v.v

Các *Manipulator* này sẽ có tác dụng cho các thao tác định dạng từ khi chúng được thiết lập cho đến khi có thiết lập mới. Dưới đây là danh sách của một số *Manipulator* đã được thiết kế dành cho các loại dữ liệu chuẩn. Chi tiết về việc sử dụng các

thao tác định dạng này sẽ được trình bày trong XI.

dec, hex, oct: Xác định cơ số để nhập và hiển thị dữ liệu. Nếu sử dụng *oct*, C++ sẽ dùng cơ số 8, *hex* - cơ số 15 và *dec* - cơ số 10, cơ số này cũng là cơ số mặc định.

ws: Loại các ký tự trắng đầu tiên trong các *Input Stream* (Stream nhập).

Dữ liệu sẽ được đọc cho đến khi gặp ký tự trắng đầu tiên. Thao tác định dạng này chỉ có tác dụng khi cờ *skipws* đã bị tắt, (xem cờ *skipws* ở phần sau).

endl: Đặt ký tự xuống dòng.

ends: Chèn ký tự '' vào các *Output Streams* (dòng xuất) .

flush: Thao tác định dạng này ép tất cả các phép xuất trong *Input* và *Output Stream* phải được viết thật sự về mặt vật lý ra thiết bị tương ứng. *Flush* thông thường dùng để làm việc với các phép xuất có sử dụng bộ đệm. Khi sử dụng bộ đệm, bộ đệm này có thể được làm sạch trước khi thực hiện một thao tác xuất mới bằng *flush.*

setbase(int): Thiết lập việc chuyển đổi cơ số với 1 trong các giá trị sau:

- 0: Cơ số mặc định. Khi sử dụng gá trị này, cơ số 10 được sử dụng cho việc xuất dữ liệu. Trong quá trình nhập dữ liệu, các số được bắt đầu bởi 0 được xem như một số ở hệ cơ số 8, *0x* được xem như các số ở hệ cơ số 16. Trong trường hợp ngược lại, cơ số 10 được xem như cơ số đang sử dụng.
- 8: Sử dụng cơ số 8.
- 16: Sử dụng cơ số 16.

resetiosflagA(long): Xoá một hay nhiều cờ tạo dạng *(formatting flag)* trong *ios::x_flags*).

setiosflangs(long): Thiết lập một hoặc nhiều cờ định dạng.

setfill(int): Thiết lập ký tự độn. Các ký tự này có thể được căn theo lề trái hoặc lề phải của vùng hiển thị dữ liệu. Chế độ này chỉ có tác dụng nếu độ rộng của vùng hiển thị dữ liệu được thiết lập qua hàm *ios:: width()* hoặc qua manipulator *setw(int)*...

setprecision(int): Xác định độ chính xác phần thập phân của số cần hiển thị. Giá trị mặc định là 6. Manipulator này chỉ có tác dụng trên các *Output Stream* (các dòng xuất).

setw(int: Thiết lập độ rộng của biến trong thao tác xuất kế tiếp. Nếu giá trị đòi hỏi độ rộng nhỏ hơn độ rộng được thiết lập, C++ sẽ sử dụng ký tự độn được xác định bởi *setfill.*

Dưới đây xin giới thiệu một số ví dụ minh họa cho việc sử dụng các bộ phận thao tác định dạng.

Ví dụ 2.11

Xuất nhập với cơ số ngầm định :

```
#include <iostream.h>
#include <strstrea.h>
#include <iomanip.h>
void main()
{
    //Tạo chuỗi
    char number[] = "\n 10 010 0x10";
    //Tạo sự liên kết giữa chuỗi và String Stream
    istrstream is(number,sizeof(number));
```

```
    //Nhập giá trị cho các biến từ String Stream
    int v1,v2,v3;
    is>>v1>>v2>>v3;
    //Hiển thị kết quả.
    cout<<"\n"<<v1<<" "<<v2<<" "<<v3;
}
```

Trong ví dụ này, đầu tiên một dòng nhập chuỗi (*is*) được tạo và có chứa 3 giá trị với các cơ số 10, 8 và 16 (các giá trị này được khai báo trong mảng *number[]*). Tiếp đó nội dung của *stream* được đọc vào 3 biến *v1,v2,v3* và sau đó được hiển thị lên màn hình bằng dòng xuất *cout*.

Một điều cần lưu ý là trong chương trình không sử dụng *stream* chuẩn *cin* (*cin* là *stream* được liên kết với bàn phím và cho phép nhập giá trị từ bàn phím). Thay vào *cin*, chương trình có sử dụng một *stream* khác - *is*. Đây là một *stream* nhập và được liên kết với một chuỗi có sẵn - chuỗi *number* bằng dòng lệnh *istrstream is(number,sizeof(number))*. Sau khi liên kết có thể sử dụng *stream* này để nhập giá trị cho các biến từ nội dung của chuỗi. Sau khi thực hiện, trên màn hình sẽ xuất hiện kết quả:

10 8 16

Kết quả này có được là do trình biên dịch tự động phát hiện khuôn dạng của các số trong chuỗi.

Nếu sử dụng thao tác định dạng dec: *is>>dec>>v1>>v2>>v3;* thì khi đó *v1,v2,v3* sẽ có các giá trị lần lượt là 10 10 0. Theo các định dạng này thì việc tự động phát hiện khuôn dạng của các số sẽ bị bỏ qua. Tất cả các số đều được sử dụng cơ số 10 khi nhập. Sau khi nhập, *v3* nhận giá trị 0 do quá trình nhập *v3* bị dừng lại

khi gặp ký tự *x* - ký tự không được chấp nhận trong hệ đếm cơ số 10.

Nếu sử dụng thao tác định dạng hex: *is>hex>v1>v2>v3*; kết quả trên màn hình sẽ là:

16 16 0

Ký tự *x* lúc này cũng không được chấp nhận trong hệ đếm cơ số 16.

Các định dạng *dec,oct,hex* có thể sử dụng để chuyển đổi cơ số ngay trên một dòng lệnh.

Ví dụ 2.12

```
void main()
{
    const int v=100;
    //
    cout<<"\n"<<v<<" "<<oct<<v<<" "<<hec<<endl;
}
```

Kết quả của chương trình sẽ là dòng:

100 144 64

được xuất hiện trên màn hình.

2.3.2. Thiết lập và xoá cờ định dạng

Tất cả các *stream* trong C++ có sử dụng lớp *ios* đều có thể làm việc với các cờ định dạng. Dưới đây xin liệt kê một số cờ hay được sử dụng:

skipws - Cờ có tác dụng đối với các dòng nhập dữ liệu. Khi cờ được bật, các ký tự trắng đầu tiên trong quá trình nhập dữ

liệu sẽ bị bỏ qua. Khi cờ tắt, các dòng nhập sẽ nhập cả các ký tự trắng. Trong trường hợp này, nếu trên dòng nhập xuất có sử dụng thao tác định dạng *ws* thì các ký tự trắng vẫn bị bỏ qua.

left, right, internal - Chỉ có một cờ có thể bật ở mỗi thời điểm. Nếu *left* được bật, dữ liệu sẽ được canh trái trong vùng xuất được xác định bởi hàm thành phần hay thao tác định dạng *width*, khoảng còn lại của vùng sẽ được chèn bởi các ký tự độn. Nếu *right* được bật thì dữ liệu này sẽ được canh theo lề phải của vùng xuất. Nếu *internal* được bật thì dấu của mọi giá trị số sẽ được canh biên trái, số được canh theo biên phải trong khi phần giữa sẽ được chèn bởi các ký trắng.

dect, oct, hex - Các cờ định dạng này cũng có tác dụng tương tự như các thao tác định dạng *dec, oct, hex* đã trình bày ở trên.

showbase - Hiển thị cả các ký tự tương ứng cho cơ số dạng được dụng để xuất dữ liệu. Chẳng hạn số 100 sẽ được xuất trên màn hình dưới dạng 0x64.

showpoint - Cờ này khi được bật sẽ bắt buộc dòng xuất phải hiển thị dấu chấm thập phân khi xuất các số *float* hay *double* thậm chí nếu phần thập phân không có. Mặc định cờ này không được bật.

uppercase - Khi được bật, tất cả các ký tự dùng cho cơ số của số được xuất sẽ được chuyển thành chữ viết in. Chẳng hạn như ký tự 'x' khi hiển thị số 100 trong hệ cơ số 16 - 0X64. Mặc định cờ này không được bật.

showpos - Nếu cờ này được bật, dấu "+" sẽ xuất hiện trước tất cả các số nguyên được xuất ra. Mặc định thì cờ này không được bật.

scientific - Khi cờ này được bật, các giá trị dấu phẩy động được xuất ra theo dạng khoa học sử dụng ký tự "e" làm cơ số cho luỹ thừa. Chẳng hạn số 200.200 sẽ được hiển thị dưới dạng 2.002e+2.

fixed - Khi cờ này được bật, giá trị được xuất ra dưới dạng số thập phân.

unibuf - Khi cờ này được bật, dòng xuất nhập được làm sạch sau mỗi lần xuất ra. Cờ này chỉ sử dụng cho các dòng xuất nhập có dùng bộ đệm.

stdio - Cờ này làm sạch các thiết bị xuất *stdout* và *stder* được định nghĩa trong *stdio.h*.

Để có thể sử dụng các cờ này, trong chương trình cần phải sử dụng File tiêu đề *iomanip.h*. Giá trị của các cờ khi đó có thể được thiết lập lại nhờ các bộ phận định dạng. Có 2 bộ phận định dạng hay được sử dụng:

setiosflags(long): Thiết lập các cờ nhớ.

resetiosflags(long): Xoá bỏ các cờ nhớ.

Ví dụ sau sẽ minh họa cho việc sử dụng các bộ phận định dạng này:

```
cout<<setiosflags(ios::showbase)<<"\n"<<v<<" "<<oct<<v<<""
    <<hex<<v<<endl;
```

Nếu giá trị của v là 100 thì khi in giá trị này lên màn hình, *cout* sẽ chỉ rõ khuôn dạng xuất dữ liệu. Kết quả của dòng lệnh trên sẽ là:

```
100 0140 0x64
```

Các ví dụ dưới đây sẽ mô tả cho việc sử dụng các bộ phận và

các cờ xử lý định dạng.

Ví dụ 2.13

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
    float v1=1300.23;
    float v2 = 320.99;
    float v3 = 54430.00;
    cout <setiosflags(ios::showpoint | ios::fixed)
        <<setprecision(2)<<setfill('*'<<setiosflags(ios::right);
    cout <<"\n check value: S"<<setw(10)<<v1;
    cout <<"\n check value: S"<<setw(10)<<v2;
    cout <<"\n chek value; S"<<setw(10<<v3;
}
```

Chương trình trên sẽ cho kết quả sau trên màn hình:

```
check value: S***1300.23
check value: S****320.99
check value: S**54430.00
```

Trong chương trình có sử dụng một số cờ định dạng sau:

ios:: showpoint: Bắt buộc *stream* phải hiển thị dấu chấm thập phân, thậm chí nếu phần thập phân không có.

ios::fixed: Hiển thị số dưới dạng thập phân, không sử dụng số mũ.

ios::right: Dóng dữ liệu về bên phải vùng cần hiển thị.

Ngoài cờ này, chương trình còn sử dụng các hàm:

setprecision(2): Chỉ in 2 số sau phần thập phân.

setfill(''):* Sử dụng * làm ký tự độn.

Ví dụ 2.14

```
#include <iostream.h>
#include <strstrea.h>
#include <iomanip.h>
void main()
{
    const float f=3.14159;
    cout <<setiosflags(ios::showpos | ios::scientific)
    <<"\n The value of PI is"
    <<setprecision(3) <<setw(15) <<setfill('-')
    <<setiosflags(ios::right)<<f;
}
```

Sau khi thực hiện, chương trình sẽ hiển thị kết quả sau trên màn hình:

```
The value of PI is ----- 3.142E+00
```

Để làm việc với các cờ, C++ còn sử dụng một số hàm thành phần sau:

long flags(f): Các cờ định dạng bật theo *f*, trả về giá trị cờ trước đó.

long setf(long f): Bật các cờ có bật trong *f*, trả về các giá trị cờ trước đó.

Ví dụ 2.15

```
#include <iostream.h>
int main()
{
    double pi = 3.1415927;
```

```
    int x = 123;
    //Cất các giá trị cờ nguyên thủy.
    long orig_flags = cout.flags();
    //Bật một số cờ
    cout.setf(ios::hex | ios::showbase | ios::uppercase |
              ios::scientific);
    //In theo phép định dạng mới
    cout <<"\n x= "<<x<<"pi ="<<pi;
    //Trả lại các giá trị cờ nguyên thủy
    cout.flags(orig_flags);
    //In theo phép định dạng nguyên thủy
    cout<<"\n x="<<x<<"pi="<<pi;
    return 0;
}
```

Sau khi chương trình được thực hiện, trên màn hình sẽ hiện kết quả:

```
x = 0x4D2       pi = 3.141593E+00
x= 1234         pi = 3.141593.
```

2.3.3. Các dòng nhập xuất làm việc với file

Trong C++ có rất nhiều lớp trong thư viện cho phép người sử dụng làm việc với File. Các lớp hay được sử dụng nhất là *ifstream* và ofstream. Trong hai lớp này, *ifstream* hỗ trợ cho các thao tác mở và đọc dữ liệu từ File trong khi *ofstream* hỗ trợ cho các thao tác đọc mở và ghi dữ liệu lên File. Trong phần này các khái niệm về lớp cũng chưa được đề cập đến. Tuy vậy, cần lưu ý là để sử dụng các *stream* này, file tiêu đề *fstream. h* phải được khai báo ở đầu chương trình.

2.3.3.1. Đọc dữ liệu từ File

Hãy xét chương trình sau:

Ví dụ 2.16

```
#include <iostream.h>
#include <fstream.h>
void main()
{
    //Mở file đã tồn tại trên đĩa.
    ifstream help_file("\\Borlandc\\Readme");
    //Kiểm tra việc mở file
    if(help_file)
    {
     cout<<"\n Không thể mở được file \Borlandc\\Readme";
    return;
    }
 //Hiển thị nội dung file
     while(help_file)
     {
          char buffer[100];
          help_file.getline(buffer,sizeof(buffer));
          count<< "\n"<<buffer;
     }
}
```

Trong ngôn ngữ C và C++, dạng file cho phép nhập xuất dữ liệu đơn giản nhất là dạng file văn bản. Trong chế độ làm việc này, dữ liệu của file được chia làm nhiều dòng khác nhau, mỗi dòng được kết thúc bởi ký tự xuống dòng mới. Nội dung của các

file này bao gồm các ký tự *ASCII*. Các lớp nhập xuất của file cho phép mở file và sau đó làm việc (đọc và ghi) theo từng dòng.

Ở ví dụ trên, để có thể mở một file cho thao tác đọc, chương trình có sử dụng dòng lệnh:

```
ifstream help_file("\\Borlandc\\Readme");
```

Dòng lệnh này định nghĩa một đối tượng thuộc kiểu dòng nhập làm việc với file (*Input file Stream*). Đối tượng này có tên là *help_file*. Sau khi định nghĩa, file *Borlandc\Readme* đã được liên kết với đối tượng này và qua đối tượng đó có thể thực hiện các thao tác đọc dữ liệu trên file tương ứng. Cú pháp chung để tạo một đối tượng thuộc kiểu *Input File Stream* có dạng sau:

ifstream Object_name(file_name);

Trong đó:

ifstream: Lớp hỗ trợ cho việc mở và đọc file

Object_name: Tên của đối tượng thuộc lớp *ifstream* được liên kết với *file_name.*

Sau lệnh trên, đối tượng được tạo sẽ mở file có tên được chỉ trong lệnh ở chế độ đọc. Bởi vì lệnh này không trả lại giá trị sau khi thực hiện do đó để có thể kiểm tra việc mở file có thành công hay không ta phải dùng lệnh tiếp sau:

```
if(!help_file);
```

Như vậy, khi thao tác mở file thực hiện thành công thì *help_file* có giá trị đúng và ngược lại *help_file* nhận giá trị sai nếu việc mở file không thành công.

Việc đọc nội dung một dòng thuộc file vào bộ nhớ được thực hiện qua hàm thành phần *getline()*. Với dòng lệnh *help_file,getline(buffer, sizeof(buferr));* nội dung một dòng của

file sẽ được đọc vào *buffer*.

Nội dung của file tuy vậy có thể được đọc theo từng ký tự hoặc theo từng kiểu dữ liệu có sẵn. Chẳng hạn như trong đoạn chương trình sau:

```
char c;
char* cp;
int i;
long l;
double d;
help_file>>c>>cp>>i>>l>>d;
```

Khi thực hiện việc đọc theo từng kiểu dữ liệu này, ký tự xuống dòng vẫn được xem như bất kỳ một ký tự nào khác. Có thể thấy rằng nếu file được đọc là file văn vản thì việc đọc từng ký tự của file sẽ kém hiệu quả hơn *getline*. Tuy vậy, nếu dữ liệu trong file thuộc nhiều kiểu khác nhau và khi thứ tự xuất hiện của chúng trong file đã được biết trước thì việc đọc theo từng loại dữ liệu như trên sẽ hiệu quả hơn rất nhiều, thậm chí có thể nói *getline* không thể giải quyết được vấn đề này. Việc đọc dữ liệu thuộc các kiểu không phải ký tự sẽ được kết thúc nếu gặp phải ký tự trắng hoặc dữ liệu không cùng kiểu.

Một điều cần lưu ý là nếu file được mở bằng một đối tượng thuộc kiểu *stream* ở trong một phạm vi nào đó thì file sẽ được tự động đóng lại khi chương trình thoát ra khỏi phạm vi này. Đây lại là một điểm mới nữa của C++ và nó tránh cho người lập trình khỏi phải thực hiện một thao tác cần thiết nhưng hay bị quên đó là thao tác đóng file.

2.3.3.2. Kiểm tra lỗi khi làm việc với File

Cũng như các hàm làm việc với file khác trong ngôn ngữ C, khi làm việc với file qua các *stream*, các lỗi có thể phát sinh khi thực hiện các thao tác với file do nhiều nguyên nhân khác nhau. Chẳng hạn, lỗi có thể phát sinh khi mở một file không tồn tại trên đĩa, tiếp tục đọc dữ liệu khi đã đến cuối file, hay nhập và xuất dữ liệu không đúng kiểu v.v. Tất cả các lỗi này đều có thể được phát hiện trong C++. Có thể bạn đã để ý đến một phương pháp đã sử dụng ở trên để phát hiện trạng thái khi mở file:

```
if(Object_name) // Có lỗi khi mở file.
    //Các lệnh thực hiện khi có lỗi.
if(Object_name) // Không có lỗi khi mở file.
    //Các lệnh thực hiện khi không có lỗi.
```

Ngoài việc dùng các toán tử đã được định nghĩa chồng như *!* và *void**. Việc phát hiện lỗi có thể được thực hiện thông qua các hàm thành phần. Dưới đây xin giới thiệu một số hàm thành phần hay được sử dụng.

bad(): Sử dụng để bắt các lỗi không được thực hiện trên file. Ví dụ:

```
if(Object_name.bad()) // Xuất hiện lỗi
// Các lệnh được thực hiện khi có lỗi
```

eof(): Sử dụng để bắt lỗi khi đã đọc đến cuối file. Ví dụ:

```
if(Object_name.eof()) // Đã đọc đến cuối file.
// Các lệnh thực hiện khi có lỗi.
```

good() Sử dụng để xác định thao tác làm việc được tiến hành thành công. Ví dụ:

```
if(Object_name.good()) // Thao tác với file thực hiện tốt.
```

Có thể kiểm tra các lỗi phát sinh khi làm việc với nhiều *stream* bằng một lệnh, chẳng hạn như ví dụ sau:

```
while(Input_file && Output_file)
    {
    // Sao chép vào Output_file
    }
```

trong đó *Input_file* là một đối tượng thuộc dòng nhập và *Output_file* là một đối tượng thuộc dòng xuất của file. Vòng lặp trên sẽ được thực hiện cho đến khi xảy ra một lỗi nào đó khi làm việc với file; thông thường là đã đạt đến điểm cuối của một trong hai file.

Một số cờ trong lớp *ios* cho phép xác định các lỗi cụ thể bị phát sinh khi làm việc với *File Stream*. Các cờ này bao gồm:

ios::eofbit: Cờ này có giá trị bằng 1 khi đã đạt đến cuối file.

ios::goodbit: Có giá trị 0x00 khi thao tác trên file không có lỗi.

ios::failbit: Có giá trị là 0x02 khi thao tác nhập xuất cuối cùng trên file thực hiện không thành công.

ios::badbit: Có giá trị là 0x04 khi thực hiện một thao tác không được phép trên file.

ios::hardfail: Có giá trị là 0x08 khi thực hiện một thao tác nguy hiểm trên file (*Fatal Stream error).*

Các cờ này là các giá trị của biến *io_state* thuộc kiểu *enum* trong lớp *ios*. Các giá trị này có thể được đọc bởi hàm thành phần *ios::rdstate()*. Một khi một trong các cờ này được bật thì tất

cả các thao tác đọc và ghi file bằng *stream* đều không được phép cho đến khi cờ đó bị xoá đi. Để có thể xoá các cờ này, C++ cung cấp cho người sử dụng hàm thành phần *ios:: clear(int)*. Hàm thành phần này tuy vậy không chỉ được dùng để xoá các cờ mà còn có thể dùng để thiết lập giá trị của các cờ này. Giá trị mặc định được truyền cho tham số là 0 và khi đó tất cả các cờ đều bị xoá. Chẳng hạn, *help_file.clear()* sẽ có tác dụng xoá tất cả các cờ. Giá trị của các cờ cũng có thể được thiết lập bởi người sử dụng, chẳng hạn dòng lệnh: *help_file.clear(ios::failbit);* sẽ có tác dụng thiết lập *failbit* và xoá các cờ bit còn lại.

Một cờ nào đó có thể được thiết lập trong khi các cờ khác vẫn giữ nguyên giá trị của chúng. Chẳng hạn ta có thể thiết lập *failbit:*

```
help_file(help_file.rdstate() | ios::failbit);
```

Ngược lại, để xoá một cờ (chẳng hạn *failbit*) mà không làm ảnh hưởng đến các cờ khác, ta có thể dùng lệnh:

```
help_file.clear(file.rdstate() &ios::faiblit);
```

2.3.3.3. Ghi dữ liệu vào File

Chương trình sau mô tả việc ghi dữ liệu lên file sử dụng *File stream*

Ví dụ 2.17

```
#include <iostream.h>
#include <fstream.h>
    void main()
    {
        // Mở một file đã tồn tại để đọc
```

```
    ifstream help_file("\\Borlandc\\Readme");
    //Mở một file để ghi và xoá nội dung cũ của file
    ofstream copy_file("copy");
    //Các thao tác kiểm tra khi mở file
    if(help_file)
    {
        cout<<"\n không mở được file
                \\Borlandc\\Readme";
        return;
    }
    if(!copy_file)
    {
        cout<<"n Không mở được file Copy";
        return;
    }
    //Sao chép 4 dòng đầu của file Readme sang file Copy
      int line_count =4;
    while(help_file &&copy_file)
    {
        char buffer[80];
        help_file.getline(buffer,sizeof(buferr));
        copy_file<buffer<<"\n";
        if(++line_count ==4) break;
    }
}
```

Trong ví dụ trên nếu ta không dùng biến điều khiển *line_count* để thoát ra khỏi vòng lặp sau khi ghi 4 dòng vào file thì toàn bộ nội dung của file *help_file* sẽ được sao chép sang file *Copy* bởi vì vòng lặp chỉ kết thúc khi đọc đến cuối file *help_file*.

Cũng giống như trường hợp đọc dữ liệu từ file, thao tác ghi dữ liệu lên file cũng có thể được thực hiện theo từng ký tự hoặc từng kiểu dữ liệu có sẵn theo cách sau:

```
char c;
char* cp;
int i;
float f;
double d;
copy_file<<c<<cp<<i<<f<<d;
```

Tuy nhiên, khi dữ liệu được ghi lên file theo cách này thì chúng cũng phải được đọc từ file theo cách tương ứng để đảm bảo giá trị của dữ liệu được đọc một cách chính xác.

C++ còn cho phép làm việc với file với một chế độ khác nhau. Các chế độ này được thiết lập qua các cờ khi khởi tạo một đối tượng làm việc với dòng xuất nhập File:

ios::app: Cộng dữ liệu vào cuối file khi thực hiện thao tác ghi.

ios::trunc: Xoá nội dung cũ của file nếu file đã tồn tại trên đĩa.

ios::nocreate: Mở một file đã tồn tại trên đĩa.

ios::norplace: Mở một file mới chưa tồn tại trên đĩa.

ios::binary: Mở một file trong chế độ nhị phân, nếu cờ này không được thiết lập thì file sẽ được mở trong chế độ text.

ios::in: Mở file trong chế độ chỉ cho phép đọc.

ios::out: Mở file trong chế độ chỉ cho phép ghi.

ios::ate: Đặt con trỏ file vào cuối nội dung của file khi mở file.

Dưới đây là các ví dụ đơn giản minh họa cho việc thiết lập

các chế độ làm việc khác nhau với File:

ofstream copy_file(Filename,ios:app); - Mở file có tên là *FileName* trong chế độ ghi cho phép cộng thêm dữ liệu vào cuối nội dung của file đó.

ofstream copy_file(FileName,ios::noreplace); - Mở file có tên là *FileName* trong chế độ ghi và đảm bảo rằng file đó chưa tồn tại trên đĩa.

ofstream copy_file(FileName,ios::nocreate); - Mở file có tên là *FileName* trong chế độ ghi và đảm bảo rằng file đó đã tồn tại trên đĩa.

Để kiểm tra các thao tác xuất trên *stream,* chúng ta có thể sử dụng các phương pháp đã trình bày ở phần đọc dữ liệu từ file.

Khi mở một file ở chế độ ghi chúng ta có thể kiểm tra xem file đó đã tồn tại trên đĩa hay chưa bằng cách sau:

```
ofstream copy_file("Copy",ios::nocreate);
if(copy_file) // File chưa tồn tại trên đĩa
{
// Các lệnh khi file chưa tồn tại trên đĩa.
    }
```

2.3.4. Làm việc với File trong chế độ nhị phân

Khác với các file văn bản, các file làm việc trong chế độ nhị phân không phân biệt các ký tự trắng và cũng không được tổ chức theo từng dòng. Việc mở và đóng các file trong chế độ nhị phân cũng được thực hiện gần giống như trong trường hợp file văn bản, ngoại trừ việc sử dụng cờ *ios:: binary* như tham số truyền khi mở file. Hãy xét một số ví dụ sau:

2.3.4.1. Đọc file văn bản theo chế độ nhị phân

Ví dụ 2.18

```
#include <iostream.h>
#include <fstream.h>
void main()
{
   //Mở file trong chế độ nhị phân.
   ifstream help_file("\\Borlandc\\Readme",ios::binary);
   if(help_file)//Kiểm tra lỗi khi mở file
    {
      cout<<"n Không mở được file
      \\Borlandc\\Readme";
    return;
    }
    // Hiển thị nội dung file
    while(help_file)// Thực hiện cho đến khi gặp cuối file
{
    char c;
    help_file.get(c); //Đọc 1 ký tự từ file
    cout <<c;
}
 }
```

Bởi vì khi làm việc trong chế độ nhị phân, C++ không phân biệt dấu cách và xuống dòng do đó hàm thành phần đọc dữ liệu theo từng dòng không thể thực hiện được. Trong chế độ này, để đọc nội dung của file chỉ có phương pháp đọc dữ liệu theo từng

byte là làm việc chính xác và hiệu quả nhất. Tuy vậy, nếu sử dụng hàm thành phần *read*, ta vẫn có thể đọc dữ liệu với một kích thước nhất định vào bộ nhớ. Chẳng hạn các dòng lệnh sau:

```
char buffer[100];
help_file.read(buffer,sizeof(buffer));
```

sẽ cho phép đọc dữ liệu từ file được liên kết với đối tượng *help_file* với kích thước bằng kích thước của mảng *buffer*.

2.3.4.2. Đọc file nhị phân

Ví dụ 2.19

```
#include <iostream.h>
#include<fstream.h>
void main()
    {
    // Mở file đã tồn tại để đọc
    ifstream help_file("\\Personal",ios::binary);
    // Kiểm tra lỗi khi mở file
    if(help_file)
    {
        cout <<"\n Không mở được file \\ Personal";
        return;
    }
     struct {
        char name[10];
        char surname[20];
        int age;
        float salary;
```

```
        } data;
        while(help_file)
        help_file.read((char*)&data, sizeof(data));
        cout<<"Trong lần đọc cuối cùng chỉ có"
        <<help_file.gcount()<<"ký tự được đọc trong tổng số"
        << sizeof(data);
    }
```

Với vòng lặp này nội dung của file *Personal* được đọc vào biến cấu trúc *data* với kích thước mỗi lần đọc bằng kích thước của chính cấu trúc này. Tất nhiên là với vòng lặp này thì chỉ có dữ liệu trong lần đọc cuối cùng được lưu trữ trong *data*. Lệnh *cout* sau đó sẽ hiển thị số ký tự trong lần đọc cuối cùng từ file vào cấu trúc này.

2.3.4.3. Ghi file nhị phân

Ví dụ 2.20

```
#include <iostream.h>
#include <fstream.h>
struct Data{
    char name[10];
    char surname[20];
    int age;
    float salary;
    };
    void main()
    {
        // Mở file đã tồn tại để ghi ofstream
    company_records("RECORDS.DAT",ios::binary);
        // Kiểm tra lỗi khi mở file
```

```
        if(company_records)
        {
          cout<<"\n Không mở được file RECORDS.DAT";
          return;
        }
        Data employee;
        int number =0;
        while(number <10 &&company_records)
        {
        company_records.put((char )number++); (1)
    company_records.write((char*)&Data,sizeof(Data));(2)
        }
}
```

Chương trình trên ghi dữ liệu của 10 bản ghi vào file. Dữ liệu được ghi vào file bao gồm: số thứ tự của bản ghi - dòng lệnh (1) và dữ liệu của từng bản ghi - dòng lệnh (2).

2.4. CÁC DÒNG XUẤT NHẬP LÀM VIỆC VỚI CHUỖI

Trong rất nhiều trường hợp, dữ liệu sau khi được tạo khuôn dạng không cần thiết phải hiển thị lên màn hình mà chỉ cần đưa vào bộ đệm, dữ liệu này sau đó lại có thể được đọc vào biến để phục vụ cho các thao tác xử lý dữ liệu tiếp theo. Để phục vụ cho mục đích này, ngôn ngữ C sử dụng hàm *sprintf(...)* và *sscanf(...)* trong khi C++ sử dụng lớp *strstream* cho các thao tác nhập và xuất dữ liệu. Về phương pháp sử dụng có thể nhận thấy các thao tác nhập xuất dữ liệu với các *String Stream* (dòng nhập xuất kiểu chuỗi) cũng tương tự như khi thao tác với các *File Stream* (dòng nhập xuất File). Khi sử dụng các dòng xuất nhập với

chuỗi, chương trình cần sử dụng file tiêu đề *strstrea.h*.

2.4.1. Các thao tác xuất dữ liệu

Các thao tác xuất dữ liệu bằng *String Stream* được minh họa qua ví dụ sau:

Ví dụ 2.21

```
#include <isotream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
void main()
    {
        //Tạo một strstream rỗng.
    //Kích thước của bộ nhớ sẽ được cấp phát động strstream
    buffer;
    // Khai báo một số biến
    int number =30;
    char* color = "Đỏ";
    float weight = 135.56;
    // Xuất dữ liệu vào buffer và thiết lập các chuỗi định
        // dạng trong Stream.
    buffer <<"\n Có tất cả"<<number;
    buffer <<" chiếc ghế có màu" <<color
    buffer <<"với tổng khối lượng là"
    buffer << setprecision(2) <<weight <<"kilo";
    // mở file report.doc
    ofstream report("report.doc");
    if(!report)
```

```
    {
        cerr<<"Không mở được file report";
        exit(1);
    }
        // Ghi dữ liệu vào file
report<buffer.rdbuf();
}
```

Trong ví dụ trên, dòng lệnh *strstream buffer* đã tạo ra một đối tượng *buffer* thuộc dòng nhập xuất kiểu chuỗi. Sau khi được tạo, dữ liệu kiểu chuỗi có thể được xuất vào *buffer* qua toán tử <<. Nội dung của *buffer* sau đó lại được xuất vào file *report.doc* qua hàm thành phần *rdbuf()*. Như vậy sau khi thực hiện chương trình thì file *report.doc* sẽ chứa chuỗi "Có tất cả 30 chiếc ghế với màu đỏ với tổng khối lượng là 135.96. kilo".

Ví dụ 2.22

```
#includ <iostream.h>
#include <strstream.h>
#include <iomanip.h>
void main()
{
    const float pi = 3.14159;
    strstream buffer;
// Thiết lập các thao tác định dạng và xuất dữ liệu vào buffer.
    buffer << setiosflags(ios::showpos | ios:scientific)
    << "Giá trị của PI là"
    << setprecision(3)
    << setw(15) < setfill(_)
    << setiosflags(ios::right)
    << pi <<ends;
```

```
// Chèn ký tự trắng vào buffer - kết thúc chuỗi.
cout <<buffer.rdbuf();

// Xuất nội dung của buffer lên màn hình

}
```

Cũng cần lưu ý là lớp *strstream* sẽ cho phép tạo ra một đối tượng để thực hiện cả hai thao tác xuất và nhập dữ liệu do đó có thể nhập dữ liệu vào một biến kiểu chuỗi từ *buffer* qua hàm thành phần *getline*. Chẳng hạn, nội dung của mảng *ch* có thể được nhập từ *buffer* qua dòng *lệnh buffer.getline(ch,50);*

Ngoài lớp strstream dùng cho cả hai thao tác nhập xuất, C++ còn đưa ra các lớp riêng dành cho việc nhập - istrstrea và xuất dữ liệu - ostrstrea để làm việc với mảng.

2.4.2. Sử dụng istrstream

Ví dụ 2.23

```
#include <iostream.h>
#include <strstrea.h>
void main(int arg, char** argv)
{
    istrstrea argument(argf[0]);
    cout <<"\n Có "<<arg <<"tham số. Tham số thứ
    nhất là"  <<argument.rdbuf( );

}
```

Trong chương trình trên, đối tượng *argument* là đối tượng thuộc lớp nhập của dòng nhập xuất kiểu chuỗi. Đối tượng này được khởi tạo bằng giá trị của *argv[0]* (tham số thứ nhất của dòng lệnh). Giá trị này sau đó được xuất ra màn hình thông qua

hàm thành phần *rdbuf()*.

2.4.3. Xử lý nhiều phép định dạng bằng stream

Ví dụ 2.24

```
#include<strstrea.h>
#include <iomanip.h>
void main()
{
    char buffer[80];
    ostrstream text(buffer,sizeof(buffer));
    int i=10;
    char* code = "Không biết";
//Chèn ký tự kết thúc chuỗi vào buffer;
        text <<"\n Có"<< i <<"Đối tượng"<<ends;
        cout<<buffer;
        //Đưa con trỏ của strstream về lại vị trí đầu.
        text.seekp(0));
        //Chèn ký tự kết thúc chuỗi vào buffer.
        text<<"\n Mã của lỗi là"<< code<< '\0';
        cout<<buffer;
    }
```

Trong ví dụ, đối tượng *text* là đối tượng thuộc lớp xuất của dòng nhập xuất kiểu chuỗi. Đối tượng này được liên kết với mảng *buffer*. Nội dung của mảng, sau đó có thể được khởi tạo thông qua đối tượng *text*. Một điểm cần lưu ý là khi khởi tạo nội dung cho mảng, mảng phải được kết thúc bằng ký tự *NULL*. Trong chương trình, ký tự này được chèn vào bằng hai cách: sử dụng bộ phận định dạng *ends* hoặc chèn trực tiếp ký tự '\0' qua phép nhập dữ liệu << *'\0 '*. Trong chương trình cũng sử dụng

hàm thành phần *seekp* để đưa con trỏ về vị trí đầu của đối tượng.

2.5. SỬ DỤNG PRINTER NHƯ STREAM

Trong C++, dòng nhập chuỗi (*String Stream*) cũng có thể được sử dụng để xuất dữ liệu ra *Printer*. Có thể so sánh hai chương trình với mục đích gửi dữ liệu ra Printer trong C và C++.

Ví dụ 2.25

Sử dụng C

```
#include <stdio.h>
void main()
{
    FILE* fp;
    fp = fopen("LPT1", "w");
    if(!fp)
    {
        printf("Không thể truy nhập đến Printer");
        return;
     }
     // In ra máy in
     fprintf(fp,"In thử ra máy in");
     fclosef(fp);
    }
```

Ví dụ 2.26

Sử dụng C++

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

```
#include <strstrea.h>
#include <fstream.h>
#include <stdlib.h>
    void main(int arg, char ** argv)
    {
        strstream record;
        record <<"Có <<arg <<"tham số. Tham số thứ nhất
        là"<<argv[0]<<endl<<ends;
        ofstream printer("PRN");
       if(!printer)
       {
           cerr<<"\n Không thể truy nhập đến Printer";
           exit(1);
       }
      // In dữ liệu ra Printer
      printer <<"Ngắn gọn về dòng lệnh" <<record.rdbuf();
      }
```

Trong cả hai ví dụ trên chúng ta đều dùng "PRN" để truy nhập đến *Printer* thông qua file (trong C) và *Printer Stream* (trong C++). Thay cho "PRN", trong chương trình cũng có thể sử dụng các tên khác để làm việc với *Printer* như "LPT1" hoặc "LPT2". Sau khi tạo các *Stream* hoặc file liên kết với *Printer*, việc gửi dữ liệu đến *Printer* sẽ được thực hiện thông qua các *Stream* hoặc file này.

Chương III

CÁC PHÉP TOÁN VÀ CÂU LỆNH ĐIỀU KHIỂN

3.1. CÁC PHÉP TOÁN

3.1.1. Phép chuyển kiểu

C++ sử dụng cả hai dạng ép kiểu:

(Kiểu) Biểu thức;

hoặc:

Kiểu (Biểu thức);

Sau khi thực hiện việc ép kiểu, *Biểu thức* sẽ được chuyển sang kiểu được chỉ trong câu lệnh.

Dạng thứ nhất của phép ép kiểu có thể được sử dụng cả trong C và C++.

Ví dụ:

```
int a = int(10.8) + int (2.5);
```

Sau khi thực hiện câu lệnh này *a* sẽ có giá trị là 10.

3.1.2. Các phép toán số học

C++ sử dụng các phép toán số học sau:

Phép toán	Cách viết	Ví dụ
Cộng	+	a = a + b;
Trừ	-	a = a - 2;
Nhân	*	a = b* 2;
Chia	/	a = b/2;
Chia lấy phần dư	%	a = a%3;

Trong các phép toán số học này có hai phép toán mà cần quan tâm: phép chia (/) và phép chia lấy phần dư (%).

Phép chia lấy phần dư (%) chỉ được thực hiện trên hai số nguyên. Kết quả của phép toán *a % b* là phần dư của phép chia của *a* cho *b*.

Ví dụ

5 % 3 =2; 7 % 3 =1;

Đối với phép chia *a / b*, kết quả của phép toán lại phụ thuộc vào kiểu của số bị chia và số chia. Có thể chia làm hai trường hợp:

- Nếu cả *a* và *b* đều là số nguyên thì kết quả của phép toán *a/b* là phần nguyên của phép chia của *a* cho *b*.

Ví dụ:

7 /3 =2; 5 /3 =1;

- Nếu ít nhất một trong hai số *a* và b là số thực thì kết quả của phép chia a / *b* là một số thực.

Ví dụ:

5.0 / 2 = 2.5; 7 / 2.0 = 3.5;

Như vậy, nếu muốn lấy kết quả đúng của phép toán *a / b*

với *a* và *b* là số nguyên thì ta cần phải sử dụng phép ép kiểu.

Chẳng hạn nếu khai báo:

```
int a =7, b =2;
```

thì a/ b =3;

trong khi float(a) /b = 3.5;

Ví dụ 3.1

Chuyển đơn vị thời gian được tính bằng giây sang phút và giây (Chẳng hạn 62 giây thành 1 phút và 2 giây)

Dễ dàng nhận thấy nếu gọi *sec* là thời gian tính bằng giây và *minute* và *left* là thời gian được tính bằng phút và giây sau khi chuyển thì ta có:

```
minute = sec / 60;
left = minute % 60;
```

Chương trình được viết như sau:

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr( );
    int sec, minute, left;
    cout <<"Hãy nhập thời gian tính bằng giây"<<endl;
    cin >> sec;
    minute = sec / 60;
    left = sec % 60;
    cout <<sec <<"giây bằng "<< minute <<"phút và "
```

```
                << left << "giây;
        getch();
    }
```

Ví dụ 3.2

Hãy nhập một số nguyên không âm và tính giá trị byte thấp và byte cao của nó.

Số nguyên không âm *x* (unsigned int) được lưu trữ trong 2 byte của bộ nhớ và yêu cầu của bài toán ở đây là in giá trị riêng biệt từng byte của số nguyên này.

Nếu để ý rằng giá trị lớn nhất có thể chứa trong một byte là 255 (từ 0 đến 255) thì ta sẽ có:

Giá trị byte thấp = x % 256;

Giá trị byte cao = x /256;

Như vậy chương trình có thể được viết như sau:

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr( );
    unsigned int x;
    cout << "Hãy nhập số nguyên không âm: ";
    cin >> x;
    cout <<"Giá trị của byte thấp là "<<x %256 <<endl;
    cout << " Giá trị của byte cao là " << x / 256 ;
    getch( );
}
```

3.1.3. Các phép toán quan hệ và logic

Các phép toán này thường được sử dụng cùng nhau và dùng để so sánh giá trị của các biểu thức và sau đó chọn phương pháp cần giải quyết tiếp theo.

Các phép toán quan hệ (so sánh) bao gồm:

Phép toán	Ý nghĩa
>	Lớn hơn
<	Nhỏ hơn
> =	Lớn hơn hoặc bằng
< =	Nhỏ hơn hoặc bằng
!=	Khác

Kết quả của phép toán quan hệ là đúng (bằng 1) nếu phép so sánh được đánh giá là đúng và bằng không nếu phép so sánh là sai.

Các phép toán logic bao gồm:

Phép và : &&

Phép hoặc : | |

Phép phủ định : !

Kết quả của các phép toán được mô tả trong bảng dưới:

Bảng 3.1

Giá trị của toán hạng		Kết quả		
Toán hạng 1	Toán hạng 2	&&	||	!
TRUE	TRUE	1	1	0
TRUE	FALSE	0	1	0
FALSE	TRUE	0	1	1
FALSE	FALSE	0	0	1

Ở đây TRUE và FALSE lần lượt là các giá trị khác và bằng không của toán hạng.

3.1.4. Phép gán

Phép toán này cho phép gán giá trị cho một biến và có dạng:

Tên biến = Toán hạng;

trong đó, toán hạng có thể là một biến hoặc một biểu thức.

Có một số điều cần chú ý đối với phép gán:

- Các phép gán có thể sử dụng liên tiếp nhau. Khi đó phép gán sẽ được thực hiện từ phải qua trái

  Ví dụ: X1 = X2 = 100;

- Phép gán (=) và phép so sánh (==) có ý nghĩa hoàn toàn khác nhau.
- Các phép toán có sử dụng phép gán có thể viết ở dạng ngắn gọn:

Bảng 3.2

Phép toán	Viết ngắn gọn
a = a + b;	a += b;
a = a - b;	a -= b;
a = a / b;	a /=b;
a = a*b;	a *= b;
a = a %b;	a % =b;

3.1.5. Các phép toán tăng giảm một đơn vị

Đối với các phép toán tăng giảm một đơn vị, ngôn ngữ C++ lại đưa ra cách viết ở dạng ngắn gọn hơn. Các phép toán này

thường được sử dụng ở dạng tiền tố và hậu tố.

Ở dạng tiền tố, phép toán được viết như sau:

++a; và --a;

Ở dạng hậu tố, phép toán có dạng:

a++ và a --

trong đó *a* là một biến.

Các phép toán tiền tố *++a* và hậu tố *a++* đều tương đương với cách viết *a = a + 1* (--a và a-- tương đương với a = a -1). Tuy nhiên, nếu tham gia vào biểu thức thì các phép toán tăng giảm tiền tố và hậu tố sẽ cho các kết quả khác nhau do nguyên nhân sau:

- Đối với phép toán tăng giảm được viết dưới dạng tiền tố (++a hoặc --a) thì trước tiên giá trị của *a* tăng hoặc giảm một đơn vị sau đó mới tham gia vào biểu thức.
- Đối với phép toán tăng giảm được viết dưới dạng hậu tố (a++ hoặc a--) thì trước tiên giá trị của *a* được sử dụng để tính giá trị của biểu thức sau đó mới được tăng hoặc giảm một đơn vị.

Ví dụ 3.3

```
#inclde < iostream.h>
#include < conio.h>
void main ( )
{
    clrscr ( );
    int a =2, b =3;
    int c;
```

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

Theo qui tắc thực hiện của phép toán tăng giảm một đơn vị tiền tố và hậu tố thì giá trị của *a* trước tiên sẽ tham gia vào biểu thức sau đó mới tăng một đơn vị trong khi giá trị của *b* sẽ tăng lên một đơn vị trước khi tham gia vào biểu thức. Kết quả thực hiện của chương trình sẽ là:

```
c = 6
a = 3
b = 4
```

Để tránh các kết quả sai do thứ tự thực hiện các phép toán ++ và -- không hợp lý ta cần chú ý:

- Không sử dụng phép toán tăng giảm một đơn vị cho các biến có mặt ở nhiều đối số của hàm.

  Ví dụ: printf("a = %d a* a = %d\n", a, a*a++);

- Không sử dụng phép toán tăng giảm một đơn vị cho các biến có mặt nhiều lần trong cùng một biểu thức.

  Ví dụ b = a /2 +5 * (1 + a ++);

Nguyên nhân là do trình biên dịch có thể thực hiện phép toán không theo thứ tự như ta mong muốn. Chẳng hạn ở trường hợp thứ nhất, khi thực hiện hàm *printf*, giá trị của tham số thứ hai có thể được tính trước khi in giá trị của tham số thứ nhất lên màn hình.

3.1.6. Tính kích thước của đối tượng

Kích thước của đối tượng chính là số byte mà trình biên dịch phân bố cho đối tượng đó khi biên dịch. Ở đây, đối tượng được tính kích thước có thể là biến, mảng, cấu trúc, biểu thức hoặc đơn giản là một từ khoá được dùng cho một kiểu dữ liệu nhất định. Phép toán được viết dưới dạng sau:

sizeof toán hạng;

Có thể đưa ra một ví dụ đơn giản về cách sử dụng phép toán này.

Ví dụ 3.4

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr ( );
    char name[10] = "Nguyen Van A";
    int a;
    float b;
    cout << "Kích thước của a là " << sizeof a << endl;
    cout << Kích thước của b là " << sizeof b <<endl;
    cout << Kích thước kiểu char là "<< sizeof char
            <<endl;
    cout<< "Kích thước của name là " << sizeof name ;
    getch();
}
```

Sau khi thực hiện kết quả của chương trình sẽ là:

Kích thước của a là 2

Kích thước của b là 4

Kích thước kiểu char là 1

Kích thước của name là 20

3.1.7. Phép toán có điều kiện

Phép toán có điều kiện được ký hiệu bởi ?: và bao gồm 3 toán hạng. Phép toán có thể được mô tả ngắn gọn dưới dạng:

Biểu thức 1? Biểu thức 2: Biểu thức 3;

Phép toán được thực hiện thông qua các bước sau:

- Tính giá trị của biểu thức 1.
- Nếu giá trị của biểu thức 1 khác 0 (Biểu thức 1 có giá trị đúng), kết quả của phép toán là biểu thức 2.
- Nếu giá trị của biểu thức 1 bằng 0 (Biểu thức 1 có giá trị sai), kết quả của phép toán là biểu thức 3.

Cần lưu ý là kiểu kết quả của phép toán sẽ là kiểu cao nhất trong các biểu thức 2 và 3. Chẳng hạn nếu biểu thức 2 có kiểu là *int* và biểu thức 3 có kiểu là *float* thì kiểu của kết quả sẽ là *float*.

Ví dụ dưới đây sẽ mô tả cách sử dụng của phép toán có điều kiện.

Ví dụ 3.5

Tìm và in giá trị lớn nhất của 3 số được nhập từ bàn phím.

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

```
{
    clrscr ( );
    float a;
    int a,b;
    float max;
    cout << "Nhập giá trị của a, b, c:";
    cin >> a >> b >>c;
    max = a >b ? a: b;
    max = max > c? max: c;
    cout << "Giá trị lớn nhất phải tìm là "<< max;
    getch ();
}
```

3.2. CÁC CÂU LỆNH ĐIỀU KHIỂN

Việc sử dụng các câu lệnh điều khiển trong ngôn ngữ nhằm đảm bảo các vấn đề sau khi chạy chương trình:

- Cho phép các câu lệnh viết tiếp nhau được thực hiện trong khi một điều kiện nào đó được đánh giá là đúng.
- Cho phép sử dụng các điều kiện để từ đó chọn các khả năng có thể xảy ra nhằm đề ra biện pháp giải quyết.

3.2.1. Câu lệnh if - else

Câu lệnh *if - else* được dùng để rẽ nhánh trong chương trình. Phụ thuộc vào điều kiện của câu lệnh, quyền điều khiển của chương trình sẽ được chuyển đến một nhóm lệnh nào đó.

Câu lệnh này thường gặp ở 3 dạng sau:

Dạng 1:

if(Biểu thức)

Khối lệnh;

Các lệnh tiếp theo

Ở đây khối lệnh bao gồm các lệnh được bắt đầu bởi ký tự { và kết thúc bởi ký tự }

Khi gặp câu lệnh ở dạng này, chương trình sẽ thực hiện theo các bước sau:

- Tính giá trị của biểu thức
- Nếu biểu thức có giá trị sai, bỏ qua *khối lệnh* và thực hiện *các lệnh tiếp theo*. Nếu biểu thức có giá trị đúng thì lần lượt thực hiện *khối lệnh* và *các lệnh tiếp theo*.

Dạng 2:

if(Biểu thức)

Khối lệnh 1;

else

Khối lệnh 2;

Các lệnh tiếp theo

Các bước thực hiện:

- Tính giá trị của biểu thức.
- Nếu biểu thức có giá trị đúng thực hiện *khối lệnh 1* và *các câu lệnh tiếp theo.*
- Nếu biểu thức có giá trị sai thực hiện *khối lệnh 2* và *các câu lệnh tiếp theo.*

Dạng 3:

if(Biểu thức1)

Khối lệnh 1;

else if(Biểu thức 2)

Khối lệnh 2;

else if(Biểu thức 3)

Khối lệnh 3;

.....

else

Khối lệnh n;

Thứ tự thực hiện của chương trình khi gặp cấu trúc này sẽ là:

- Nếu *biểu thức 1* có giá trị đúng, thực hiện *khối lệnh.1*
- Nếu *biểu thức 1* có giá trị sai, kiểm tra *biểu thức 2*. Nếu biểu thức này có giá trị đúng sẽ thực hiện *khối lệnh 2* v.v.
- Nếu tất cả các biểu thức đều có giá trị sai thì thực hiện *khối lệnh n*
- Sau khi thực hiện một trong các khối lệnh, quyền điều khiển chương trình sẽ được trao cho lệnh đứng ngay sau cấu trúc if - else.

Các ví dụ dưới đây sẽ mô tả cho việc sử dụng cấu trúc if - else.

Ví dụ 3.6.

Từ bàn phím hãy đưa 2 số nguyên và xác định số lớn nhất trong hai số đó.

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

```
#include < conio.h>
void main ( )
{
    clrscr ( );
    int n1, n2;
    cout << "Hãy nhập số thứ nhất" <<endl;
    cin >> n1;
    cout << "Hãy nhập số thứ hai" <<endl;
    cin >> n2;
    int Max;
    if(n1 > n2)
        Max = n1;
    else
        Max = n2;
  cout << "Giá trị lớn nhất là: "<< Max;
  getch( );}
```

Ví dụ 3.7

Từ bàn phím nhập 3 số thực a, b và c. Hãy kiểm tra giá trị của 3 số này có thỏa mãn là 3 cạnh của tam giác hay không và nếu phải hãy kiểm tra dạng của tam giác là vuông, cân hay đều.

Việc kiểm tra 3 số có thoả mãn là 3 cạnh của tam giác hay không được thực hiện bằng cách kiểm tra tổng của hai số bất kỳ luôn lớn hơn số còn lại. Nếu gọi số lớn nhất trong ba số là Max thì điều kiện này sẽ tương đương với:

2*Max < a + b + c;

Khi tam giác là vuông thì giá trị của các cạnh phải thoả

mãn định lý Pitago (bình phương của cạnh huyền bằng tổng bình phương của các cạnh còn lại). Ở đây, do sai số khi tính toán, tam giác sẽ là vuông nếu hiệu của bình phương cạnh huyền và tổng bình phương của hai cạnh còn lại có giá trị tuyệt đối nhỏ hơn một giá trị *d* rất nhỏ nào đó. Điều kiện này sẽ tương đương với:

$$|2*Max*Max - a*a - b*b - c*c| < d ;$$

Như vậy chương trình trên có thể được viết như sau:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
void main ( )
{
    clrscr( );
    float a, b, c, Max;
    float d = 1e-9; // Sai số
    cout << "Hãy nhập ba số a, b, c :"<<endl;
    cin >> a >> b >> c;
    Max = a > b? a: b;
    if (Max  <c )
        Max = c;
    if((2*Max - a - b - c > = 0)
    cout << a  << "  "<< b << " và " << c <<" không phải là
            ba cạnh của tam giác";
    else
    {
```

```
            if(a==b && b == c)
                cout << " Tam giác đều";
            else if(a==b || b ==c || c ==a)
                cout <<" Tam giác cân";
            else
            {
                int e = fabs(2*Max - a*a - b*b -c*c);
                if( e < d)
                    cout << " Tam giác vuông";
                else
                    cout << "Tam giác thường";
            }
        }
}
```

Trong chương trình trên có sử dụng hàm *fabs* để tính giá trị tuyệt đối của một số thực. Điều này đòi hỏi phải nối kết file tiêu đề math.h với file nguồn.

Ví dụ 3.8

Hãy tính tiền điện phải trả hàng tháng theo quy tắc:

1. *Sử dụng đến 100Kw ; giá mỗi Kw là 450 đồng.*
2. *Từ 101 đến 150 Kw giá mỗi kw là 750 đồng;*
3. *Trên 150 Kw; giá là 950 đồng cho mỗi Kw.*

Ví dụ này được dành như bài tập cho bạn đọc.

3.2.2. Câu lệnh for

Với câu lệnh *for* ta có thể lặp lại việc thực hiện của một

hoặc khối lệnh với một số lần cho trước.

Câu lệnh *for* có dạng sau:

for (Biểu thức 1; Biểu thức 2; Biểu thức 3)

Khối lệnh;

Trong câu lệnh này:

- Biểu thức 1 dùng để khởi tạo biến điều khiển của vòng lặp.
- Biểu thức 2 là điều kiện dùng để kiểm tra việc thực hiện của vòng lặp.
- Biểu thức 3 dùng để thay đổi giá trị của biến điều khiển. Biểu thức này thường chứa các công thức để thay đổi giá trị của các biến điều khiển sau mỗi chu trình của vòng lặp.

Câu lệnh for thực hiện qua các bước sau:

- Bước 1. Tính giá trị của *biểu thức 1* (Khởi tạo các biến điều khiển).
- Bước 2. Xác định giá trị của *biểu thức 2*.
- Bước 3. Nếu giá trị của *biểu thức 2* khác 0, thực hiện các lệnh trong thân for, sau đó tính giá trị của *biểu thức 3*, thiết lập giá trị mới cho các biến điều khiển và quay về bước 2. Nếu giá trị của biểu thức 2 bằng 0, thoát khỏi vòng lặp for và tiếp tục thực hiện các lệnh được viết sau thân vòng lặp.

Ví dụ 3.9

Từ bàn phím hãy nhập một số nguyên n và kiểm tra số đó có phải là số nguyên tố hay không.

Vì số nguyên tố là số nguyên chỉ chia hết cho số 1 và chính nó nên để xác định một số nguyên có phải là số nguyên tố hay không ta sẽ dùng thuật toán đơn giản nhất: chia số nguyên đó cho tất cả các số nguyên nằm giữa số 2 và chính nó. Điều này sẽ được thực hiện thông qua vòng lặp for:

for(int divisor =2; n%divisor !=0; divisor ++);

Rõ ràng vòng lặp for sẽ kết thúc khi *n%divisor == 0* (tức là *n* chia hết cho *divisor*). Khi đó:

- Nếu *divisor < n* thì *n* không phải là số nguyên tố.
- Nếu *divisor ==n* thì *n* chính là số nguyên tố.

Chương trình kiểm tra số nguyên tố được viết như sau:

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr ( );
    int n;
    cout << "Hãy nhập số nguyên n: ";
    cin >> n;
    for(int divisor =2; n % divisor !=0; divisor ++);
    if(divisor == n)
        cout << n << "là số nguyên tố";
    else
        cout << n << " không phải là số nguyên tố";
    getch ( );
}
```

Phương pháp tìm số nguyên tố được đưa ra trong ví dụ trên không phải là phương pháp hiệu quả nhất, đây chỉ là thuật toán đơn giản dùng để minh hoạ cho việc sử dụng vòng lặp *for*. Bạn đọc có thể tìm hiểu thêm những thuật toán khác hiệu quả hơn để xác định số nguyên có phải là nguyên tố hay không.

Ví dụ 3.10

Xác định một số nguyên n có phải là số chính phương hay không.

Để xác định một số nguyên có phải là số chính phương hay không ta phải tìm xem số nguyên đó có bằng bình phương của một số nguyên nào đó hay không. Để thực hiện mục đích này ta sẽ sử dụng vòng lặp for:

```
for(int x =2; x*x < n; x++);
```

Rõ ràng vòng lặp *for* sẽ kết thúc nếu $x*x >= n$. Khi đó nếu $x*x == n$ thì n sẽ là số chính phương và trong trường hợp ngược lại n không phải là số chính phương.

Chương trình được viết như sau:

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr ( );
    unsigned int n;
    cout << " Nhập số nguyên n: ";
    for(int x =2; x*x < n; x++);
```

```
        if( x*x == n)
            cout<< n << " là số chính phương";
        else
            cout << n << " không phải là số chính phương;
        getch ( );
    }
```

Trong cú pháp của câu lệnh *for* không nhất thiết phải tồn tại cả 3 biểu thức. Tuy vậy, trong trường hợp một biểu thức nào đó bị bỏ qua, dấu chấm phẩy (;) đi kèm sau nó vẫn buộc phải có. Chẳng hạn, ta có thể viết:

for(; biểu thức 2; biểu thức 3)

hay

for(; ; biểu thức 3)

hoặc thậm chí:

for(; ;);

Ví dụ dưới đây sẽ mô tả trường hợp *biểu thức 1* bị bỏ qua.

Ví dụ 3.11

Từ bàn phím nhập một số nguyên a kiểu long. Hãy tính tổng các chữ số của số nguyên này.

Giả sử số nguyên a được viết dưới dạng $\overline{a_n a_{n-1}....a_1 a_0}$. Số nguyên này có thể được viết dưới dạng:

$$(...(a_n * 10 + a_{n-1}) * 10...) * 10 + a_0$$

Tổng các chữ số của số nguyên dễ dàng được xác định nếu ta xác định được từng chữ số của số nguyên đó. Theo công thức trên, nếu ta chia số nguyên cho 10 thì phần dư của phép chia sẽ là a_0 và nếu ta lấy phần nguyên của phép chia, chia tiếp cho 10

thì phần dư sẽ là a_1. Quá trình này sẽ được tiếp tục cho đến khi phần nguyên của phép chia cho 10 bằng 0. Dưới đây là chương trình giải quyết yêu cầu của bài toán.

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr ( );
    long n;
    cout << " Nhập số nguyên : ";
    cin >> n;
    long number = n; // Lưu lại số n;
    int Kq =0;
    for( ; number !=0; number /=10)
        Kq += number %10;
    cout <<" Tổng các chữ số của " << n << " là " << Kq;
    getch( );
}
```

Các biểu thức 1, biểu thức 2 và biểu thức 3... không chỉ là một biểu thức đơn giản mà còn có thể bao gồm nhiều biểu thức khác nhau. Khi đó dãy các biểu thức trong cùng nhóm phải được viết cách nhau bằng dấu phẩy

Ví dụ 3.12

Tính tổng n số tự nhiên đầu tiên.

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

```
void main ( )
{
    clrscr ( );
    unsigned int n, i;
    cout << "Nhập số n: ";
    cin >> n;
    long int Tong;
    for( i=0, Tong =0; i < n; Tong +=i++);
    cout << " Tổng là: " << Tong;
    getch ( );
}
```

3.2.3. Câu lệnh while và do - while

Bằng nhóm lệnh *while* và *do - while* ta có thể lặp lại việc thực hiện của một hoặc nhóm lệnh tuỳ theo điều kiện được đưa ra trong câu lệnh.

Câu lệnh *while* có dạng sau:

while (biểu thức)

Khối lệnh;

Trong câu lệnh này, biểu thức là điều kiện quyết định số lần thực hiện của câu lệnh. Các bước thực hiện của câu lệnh while bao gồm:

- *Bước 1*. Tính giá trị của biểu thức.
- *Bước 2*. Nếu giá trị của biểu thức khác 0, thực hiện khối lệnh và quay về bước 1. Nếu giá trị của biểu thức bằng 0, thoát khỏi vòng lặp và chuyển tới câu lệnh sau thân while.

Nhìn chung tất cả những gì mà câu lệnh *for* làm được đều có thể dùng vòng lặp *while* để thay thế và ngược lại. Chẳng hạn, vòng lặp *for* tổng quát:

```
for(biểu thức 1; biểu thức 2; biểu thức 3)
    lệnh;
```

có thể được thay thế bằng vòng lặp *while* bằng cách sau:

```
Biểu thức 1;
while(biểu thức 2)
{
    lệnh;
    Biểu thức 3;
}
```

Một vấn đề đặt ra là khi nào nên dùng câu lệnh *for* và khi nào nên dùng câu lệnh *while*. Như đã đề cập đến ở trên, hai câu lệnh này hoàn toàn có thể thay thế cho nhau. Tuy vậy, vòng lặp *for* thường được chọn trong trường hợp cần thiết tạo các biến điều khiển vòng lặp và khi đã biết trước số lần cần thực hiện các câu lệnh trong vòng lặp.

Bạn đọc có thể sử dụng vòng lặp *while* để thực hiện tất cả các ví dụ đã được minh hoạ cho vòng *lặp for*.

Câu lệnh *do - while* có dạng sau:

```
do
{
Khối lệnh;
}
while (biểu thức);
```

Câu lệnh này được thực hiện qua các bước sau:

- Bước 1. Thực hiện khối lệnh.
- Bước 2. Tính giá trị của biểu thức.
- Bước 3. Nếu giá trị của biểu thức khác 0, quay lại bước 1, trong trường hợp ngược lại, thoát ra khỏi vòng lặp.

So với câu lệnh *while*, điểm khác cơ bản của câu lệnh *do - while* là khối lệnh bao giờ cũng được thực hiện ít nhất một lần do điều kiện của vòng lặp được kiểm tra sau khi thực hiện khối lệnh. Một điểm cần lưu ý khi sử dụng *do - while* là nó phải được kết thúc bởi dấu ; được viết sau *while(biểu thức)*.

Ví dụ 3.13

Tính độ dài của xâu ký tự được nhập từ bàn phím.

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr ( );
    int i = 0;
    char s[20];
    char symbol;
    do
    {
        cout <<  "Hãy nhập xâu ký tự: ";
        cin >> s;
        while (s[i] != 0)
            ++i;
```

```
        cout << "Độ dài của xâu ký tự là " << i<< endl;
        cout << " Bạn có muốn làm tiếp không (c/k) " ;
        cin >> symbol;
    }while (symbol == 'c' || symbol == 'C'); }
```

Trong ví dụ, độ dài của xâu *s* được xác định thông qua vòng lặp *while*. Giá trị của biến đếm *i* sẽ được tăng lên một đơn vị khi duyệt xâu cho đến khi gặp ký tự kết thúc xâu - ký tự '\0'. Sau khi in kết quả lên màn hình, bằng vòng lặp *do - while* chương trình sẽ cho phép người sử dụng tiếp tục làm việc khi trả lời bằng ký tự *'c'* hoặc *'C'* cho câu hỏi *"Bạn có muốn làm tiếp không (c/k)"*. Trong trường hợp ngược lại chương trình sẽ kết thúc.

3.2.4. Câu lệnh break

Câu lệnh *break* cho phép chương trình thoát khỏi vòng lặp *for, while, do - while* và chuyển quyền điều khiển đến câu lệnh nằm ngay sau các vòng lặp trên. Như vậy, việc sử dụng câu lệnh này đã đưa ra một khả năng mới cho phép chương trình thoát ra khỏi vòng lặp mà không cần kiểm tra điều kiện kết thúc của vòng lặp.

Câu lệnh có dạng sau:

break;

Ví dụ 3.14

Từ bàn phím hãy nhập số nguyên bất kỳ và kiểm tra số đó có phải là nguyên tố hay không.

Ví dụ này đã được trình bày trong khi xem xét câu lệnh *for*. Ở đây ta sẽ dùng ví dụ để minh hoạ cho câu lệnh break.

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

```
#include <conio.h>
void main ( )
{
    clrscr ( );
    int n;
    cout << "Hãy nhập số nguyên n: ";
    cin >> n;
    for(int divisor =2; ; divisor ++)
        if(n % divisor == 0)
            break;
    if(divisor == n)
        cout << n << "là số nguyên tố";
    else
        cout << n << " không phải là số nguyên tố";
    getch ( );
    }
```

3.2.5. Câu lệnh continue

Câu lệnh này cho phép kết thúc việc thực hiện chu trình hiện thời của các vòng lặp *for, while* và *do - while.* Câu lệnh có cú pháp:

continue;

Câu lệnh *continue* được thực hiện qua các bước sau:

- *Bước 1.* Kết thúc chu trình hiện thời của vòng lặp. Giá trị tức thời của các biến được lưu giữ lại.
- *Bước 2.* Đối với vòng lặp while và do - while, lệnh continue sẽ chuyển đến việc kiểm tra điều kiện dừng

của vòng lặp, đối với for, lệnh sẽ chuyển đến việc thực hiện biểu thức 3.

- *Bước 3.* Tiếp tục thực hiện vòng lặp

Ví dụ 3.15

Từ bàn phím nhập số nguyên dương và kiểm tra số đó có phải là chính phương hay không.

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    clrscr ( );
    unsigned int n;
    cout << " Nhập số nguyên n: ";
    for(int x =2;  ; x++)
    {
        if( x*x < n)
            continue;
    if(x*x == n)
        cout<< n << " là số chính phương";
    else
        cout << n << " không phải là số chính phương;
    break;
    getch ( );
    }
}
```

3.2.6. Câu lệnh switch

Câu lệnh switch căn cứ vào giá trị của biểu thức nguyên để thực hiện việc rẽ nhánh chương trình. Câu lệnh có cú pháp sau:

```
switch (Biểu thức nguyên)
{
    case m1:
        Nhóm lệnh 1;
        break;
    case m2:
        Nhóm lệnh 2;
        break;
    .....
    default:
        Nhóm lệnh n;
        break;
    }
```

Câu lệnh thực hiện qua các bước sau:

- *Bước 1.* Tính giá trị của biểu thức nguyên.
- *Bước 2.* Nếu giá trị của biểu thức trùng với một giá trị nào đó trong các m_i, chương trình sẽ thực hiện tất cả các lệnh được viết sau nhãn *case* m_i. Trong trường hợp ngược lại chương trình sẽ thực hiện các lệnh được viết sau nhãn d*efault*. Trong cả hai trường hợp, khi gặp câu lệnh *break* chương trình sẽ thực hiện câu lệnh được viết sau thân *switch*.

Khi sử dụng câu lệnh *switch* cần chú ý một số điểm sau:

- Các biểu thức m_i không được có giá trị trùng nhau vì vậy số lượng các nhãn *case* không được vượt quá số giá trị mà biểu thức nguyên có thể nhận được.
- Thứ tự sắp xếp của các nhãn *case* và *default* là hoàn toàn bất kỳ. Điều đó có nghĩa là *case* được viết đầu tiên có thể không được trình biên dịch so sánh trước.
- Một vài nhãn *case* có thể có chung một nhóm lệnh, chẳng hạn:

```
....
case m1:
case m2:
    Nhóm lệnh;
    break;
```

khi đó nếu giá trị của *biểu thức nguyên* trùng với một trong các giá trị m_1 hoặc m_2 thì nhóm lệnh được viết sau *case* m_2 sẽ được thực hiện.

- Các câu lệnh *break* có thể bỏ qua. Trong trường hợp này sau khi thực hiện xong các lệnh của nhánh chương trình sẽ tiếp tục thực hiện các lệnh của nhánh tiếp theo.
- Nhãn *default* không bắt buộc phải có. Khi đó nếu giá trị của biểu thức nguyên không trùng với bất cứ giá trị nào của m_i thì sẽ không có lệnh nào của *switch* được thực hiện.

Ví dụ 3.16

r có thể là bán kính của vòng tròn, cạnh của hình vuông hay tam giác đều. Với giá trị cho trước của r và mã của hình được

nhập từ bàn phím hãy tính chu vi của các hình trên. Mỗi hình trên được mã hoá bởi:

'c' nếu là vòng tròn;

's' nếu là hình vuông;

't' nếu là tam giác đều.

```
#include <iostream.h>
#include <conio.h>
    void main ( )
    {
        clrscr ( );
        char sys; // mã của hình
        float r; //bán kính hoặc cạnh
        float x; // chu vi
        int i;
        do
        {
            cout << " Hãy đưa giá trị của cạnh hoặc bán kính: ";
            cin >> r;
            cout << " Hãy đưa mã của hình: ";
            cin >> sys;
            do
            {
            i=0;
            switch (sys)
            {
                case 'c' :
```

```
            x = 2*3.14*r;
            break;
        case 's':
            x = 4*r;
            break;
        case 't':
            x = 3*r;
            break;
        default:
            cout << "Mã sai! Hãy nhập mã khác.";
            i=1;
            cin >.sys;
            break;
    }
    }while (i ==1);
    cout << "Chu vi của hình là: " << x <<endl;
    cout << "Có làm việc tiếp không";
    cin >> sys;
}while(sys == 'c' || sys == 'C' );}
```

3.2.7. Câu lệnh goto và nhãn

Câu lệnh *goto* cho phép chuyển quyền điều khiển của chương trình đến một lệnh nào đó được viết sau nhãn không phụ thuộc vào điều kiện. Câu lệnh có dạng:

goto nhãn;

trong đó *nhãn* có cùng dạng như tên biến và có 2 dấu chấm đứng sau.

Ví dụ:

```
int i=2;
cout << "nhảy đến nhãn End";
goto End;
.....
End: cout<< "kết thúc";
```

Trong ví dụ này, sau khi thực hiện *câu lệnh cout << "nhảy đến nhãn End"; bằng câu lệnh goto* End, chương trình sẽ thực hiện tiếp câu lệnh *cout << "kết thúc"*; viết sau nhãn *End.*

Nhìn chung không nên lạm dụng việc sử dụng câu lệnh *goto* trong khi viết chương trình. Câu lệnh này chỉ nên dùng khi cần phải ngưng chương trình trong những trường hợp cần thiết

Một điều cần lưu ý là các câu *lệnh if - else, for, while, do - while* v.v. đều có thể viết lồng nhau. Ví dụ dưới đây sẽ mô tả việc sử dụng các câu lệnh *for* lồng nhau.

Ví dụ 3.17

Viết chương trình tìm và hiển thị tất cả các số nguyên tố nhỏ hơn hoặc bằng một số nguyên n cho trước được nhập từ bàn phím.

```
#include <iostream.h>
#include < conio.h>
    void main ( )
    {
        clrscr ( );
        int n,m;
        cout << "Hãy nhập số nguyên: ";
```

```
        cin >> m;
        for (int n = 2; n <= m; m++)
        {
            for(int divisor =2; n% divisor !=0; divisor ++);
            if(divisor == n)
                cout << i<< "là số nguyên tố" << endl;
        }
        getch ( );
}
```

Trong ví dụ này, thay vì việc chỉ kiểm tra một số n có phải là nguyên tố hay không, bằng vòng lặp:

```
for (int n = 2; n <= m; m++)
```

chương trình đã cho phép kiểm tra tất cả các số nguyên nhỏ hơn hoặc bằng một số m đã được nhập từ bàn phím.

Chương IV

BỘ TIỀN XỬ LÝ

Ngôn ngữ C và C++ có chứa một khả năng ít gặp đối với các ngôn ngữ bậc cao – khả năng điều khiển quá trình biên dịch. Đây là một trong các đặc tính đưa ngôn ngữ lại gần với ngôn ngữ lập trình bậc thấp như Assembler. Quá trình biên dịch trong ngôn ngữ C++ được điều khiển bởi một chương trình riêng biệt được gọi là bộ tiền xử lý. Bộ tiền xử lý đã đưa ra nhiều khả năng mới cho người lập trình. Các khả năng này bao gồm:

- Sử dụng Macro;
- Nối kết các file nguồn;
- Biên dịch có điều kiện.

Việc sử dụng các khả năng của bộ tiền xử lý đem lại nhiều lợi thế trong lúc lập trình như chương trình sẽ trở nên sáng sủa hơn và nhất là tạo nhiều thuận lợi khi cần phải đánh giá một thuật toán hoặc lời giải của một bài toán thông qua chương trình. Ngoài các đặc điểm trên, chương trình có sử dụng bộ tiền xử lý trong nhiều điều kiện khác nhau có thể dễ dàng thay đổi cho phù hợp với các yêu cầu mới.

Khi sử dụng bộ tiền xử lý, mọi thay đổi trong file nguồn được xác định bởi các lệnh đặc biệt được gọi là chỉ thị của bộ tiền xử lý. Bộ tiền xử lý có sử dụng 3 loại chỉ thị:

- Định nghĩa Macro;

- Nối kết các file nguồn;
- Biên dịch có điều kiện.

Trong phần này sẽ giới thiệu một số chỉ thị chính được dùng để định nghĩa Macro và biên dịch có điều kiện. Phần nối kết các file nguồn sử dụng chỉ thị *#include* đã được trình bày trong các phần trước và sẽ không được trình bày lại ở đây.

4.1. ĐỊNH NGHĨA MACRO

4.1.1. Macro không sử dụng tham số

Kiểu Macro này được định nghĩa thông qua cú pháp sau:

#define Tên_Macro Chuỗi

Sau khi gặp chỉ thị *#define*, bộ tiền xử lý sẽ thay thế tất cả các *Tên_Macro* được gặp bằng chuỗi được viết trong chỉ thị. Các *Chuỗi* này sẽ được sử dụng trong quá trình biên dịch do đó chúng phải bao gồm các ký tự được phép sử dụng trong ngôn ngữ và có thể là hằng số hay biểu thức. Cần lưu ý là do tất cả các *Tên_Macro* sẽ được thay thế bằng chuỗi nên mặc dù việc sử dụng Macro có làm cho chương trình ngắn gọn trong cách viết nhưng không làm giảm độ lớn của file khi biên dịch.

Ví dụ 4.1

```
#define TWO 2
#define MSG "Message"
#define FOUR TWO*TWO
#define PX printf("x = %d\n",x)
#define FMT "x = %d\n"
void main( )
{
```

```
    int x = TWO;
    PX;
    x = FOUR;
    printf(FMT,x);
    printf("TWO : MSG\n");
}
```

Trước khi biên dịch, bộ tiền xử lý sẽ phân tích file nguồn và thay thế tên của Macro bằng các chuỗi được định nghĩa trong các chỉ thị. Như vậy trước khi biên dịch, chương trình *main* sẽ có dạng:

```
void main ( )
{
    int x =2; // TWO được thay thế bởi 2
    printf("x = %d \n",x); // Thay thế PX
    x = 4; // Thay thế FOUR
    printf("x = %d\n",x); // Thay thế FMT
    printf("%s\n", "Message");// Thay thế MSG
    printf("TWO:MSG\n");
}
```

Trong khi thực hiện các chỉ thị của bộ tiền xử lý, câu lệnh cuối cùng của hàm *main( )* không chịu tác dụng của chỉ thị, nguyên nhân là do trong câu lệnh này, TWO và MSG được sử dụng như các chuỗi bình thường trong câu lệnh.

Khi sử dụng macro cần lưu ý một số điểm sau:

- Chỉ thị *#define* có thể có cấu trúc lồng nhau. Trong ví dụ trên *FOUR* đã được định nghĩa thông qua *TWO*.

- Nếu một Macro được định nghĩa nhiều hơn một lần thì chỉ thị trước sẽ có tác dụng cho đến khi gặp chỉ thị đó ở vị trí tiếp theo.
- Tác dụng của Macro có thể bị loại bỏ từ một vị trí nào đó trong chương trình nếu tại vị trí này ta sử dụng chỉ thị *#undef*. Chỉ thị có cú pháp sử dụng:

  #undef Tên_Macro

Trong ví dụ dưới đây:

```
#define GLOB 10
.....
#define GLOB 100
.....
#undef GLOB
.....
```

kể từ sau khi sử dụng chỉ thị *#undef GLOB*, tên Macro này xem như không được định nghĩa và việc sử dụng tên này ở các dòng lệnh sau đó sẽ làm phát sinh lỗi khi biên dịch.

4.1.2. Macro có sử dụng tham số

Ngoài các Macro thay thế đơn giản, bộ tiền xử lý còn cho phép sử dụng các Macro có dùng tham số với cú pháp sau:

#define Tên_Macro(Danh sách tham số) chuỗi

Trong cú pháp này cần lưu ý:

- Giữa *Tên_Macro* và dấu mở ngoặc không được tồn tại dấu cách.
- Tên của Macro chỉ có tác dụng cho định nghĩa có chứa

Macro vì vậy tên này có thể trùng với một tên khác được sử dụng trong chương trình.

Ví dụ 4.2

```
#include <iostream.h>
#include <conio.h>
#define T(x) x*3.14
void main( )
{

clrscr( );
int T =3; // Trùng tên với tên của macro
cout << T(T); // Không phát sinh lỗi.
}
```

Danh sách các tham số được viết trong định nghĩa của Macro được gọi là các tham số hình thức còn các tham số của macro được sử dụng trong chương trình được gọi là các tham số thực. Khi gặp Macro có dùng tham số trong chương trình, bộ tiền xử lý sẽ thực hiện các công việc sau:

- Ở *Chuỗi* được viết trong chỉ thị *#define*, các tham số hình thức sẽ được thay thế bởi các tham số thực.
- Tên của Macro được viết trong chương trình sẽ được thay thế bởi chuỗi vừa được xử lý trước đó.

Ví dụ 4.3

Sử dụng Macro có tham số để tính diện tích hình tròn. Đối số được sử dụng cho bán kính hình tròn là r.

```
#include <iostream.h>
#include <conio.h>
#define PI 3.141492
#define CIRCLE(s)   PI*(s)*(s)
void main ( )
{
    clrscr ( );
    float x;
    cout <<"Nhập bán kính hình tròn:\n";
    cin >> x;
    float D_tich;
    D_tich = CIRCLE(x);
    cout << "Diện tích của hình tròn là " << D_tich;
}
```

Trong chương trình trên, việc sử dụng các dấu ngoặc đơn trong *chuỗi* (PI*(s)*(s)) là cần thiết vì qua đó người lập trình có thể xác định mức độ ưu tiên khi thực hiện các phép toán được viết trong *Chuỗi*. Cách viết này là bắt buộc trong các trường hợp khi Macro được sử dụng như là một biểu thức và khi đối số của Macro lại là một chuỗi được viết như biểu thức. Chẳng hạn trong ví dụ trên, nếu viết:

```
#define CIRCLE(s) PI*s*s
```

thì khi gặp Macro với tham số là *x + 1*, bộ tiền xử lý sẽ thực hiện phép thay *thế z = CIRCLE (x+1)*bởi *z = PI*x+1*x+1*. Rõ ràng

biểu thức sau khi được thay thế không phải là công thức để tính diện tích hình tròn với bán kính *x+1*.

Trong nhiều trường hợp, Macro đã được tham số hoá có thể được dùng để thay thế cho hàm. Ví dụ:

```
#define max(a,) ((a) > (b) ?(a):(b))
```

là Macro dùng để xác định giá trị lớn nhất trong hai số. Với Macro này, việc xác định giá trị lớn nhất của hai số *x+y* và z có thể được thực hiện qua câu lệnh:

```
m = max(x+y,z);
```

Cần phải lưu ý đến một số vấn đề sau khi sử dụng Macro với tham số truyền:

- Macro được thực hiện nhanh hơn hàm bởi vì trong quá trình thực hiện không phải truyền tham số.
- Chương trình sử dụng Macro sẽ đòi hỏi nhiều bộ nhớ hơn so với sử dụng hàm (xem chương 5) do mỗi lần Macro được gọi thì tại vị trí đó bộ tiền xử lý sẽ chen thêm các dòng định nghĩa của Macro.
- Sử dụng Macro có thể làm phát sinh hiệu ứng phụ.
- Các tham số của Macro không cần xác định kiểu trong khi kiểu các tham số khi gọi hàm là không thể thay đổi.

4.2. BIÊN DỊCH CÓ ĐIỀU KIỆN

Các chỉ thị biên dịch có điều kiện cho phép chọn một đoạn chương trình có được biên dịch hay không. Cơ cấu thực hiện các chỉ thị này được biểu diễn ở dạng *if – else – endif* quen thuộc:

1. **Phần if trong chỉ thị thường có dạng:**

#ifdef Tên_Macro

Đoạn chương trình

hoặc:

#if Biểu thức hằng

Đoạn chương trình

Tên_Macro trong chỉ thị *#ifdef* và biểu thức hằng trong chỉ thị *if* là các điều kiện xác định các lệnh tiếp đó có được thực hiện hay không. Đối với chỉ thị *#ifdef*, điều kiện chỉ thoả mãn nếu trước chỉ thị này, *Tên_Macro* đã được định nghĩa bằng chỉ thị *#define*, đối với chỉ thị *#if*, điều kiện chỉ thoả mãn nếu giá trị của *Biểu thức hằng* được đánh giá khác 0.

2. **Phần else của chỉ thị có dạng**

#else

Đoạn chương trình

3. **Để kết thúc cho cấu trúc này, trong C++ dùng chỉ thị:**

#endif

Bộ tiền xử lý sẽ kiểm tra các điều kiện được viết trong các chỉ thị *#ifdef* hoặc *#if*. Nếu điều kiện thoả mãn, đoạn chương trình được ngay sau đó sẽ được kết nối vào chương trình để biên dịch, đoạn chương trình được viết sau *#else* sẽ bị bỏ qua hoặc ngược lại.

Ngoài các chỉ thị nêu trên, C++ còn sử dụng một số chỉ thị khác phục vụ cho việc biên dịch có điều kiện.

Chỉ thị *#defined* dùng để kiểm tra một *tên* nào đó đã được

định nghĩa hay chưa. Chỉ thị này hoàn hoàn có thể thay thế cho *#ifdef*. Chẳng hạn *#defined DEBUG* là hoàn hoàn tương đương với *#ifdef DEBUG*.

Chỉ thị *#elif* được xem như cách viết ngắn gọn của cấu trúc "else – if". Nó có thể được dùng để tạo nên một tập hợp các điều kiện được lồng vào nhau trong quá trình biên dịch. Ví dụ sau sẽ minh hoạ cho việc sử dụng *#elif*

```
#if defined PROC_TABLE
#define TableSize
#elif defined NAME_TABLE
#define Table_Size_Name
#endif.
```

Chương V

BIẾN CON TRỎ, BIẾN THAM CHIẾU VÀ HÀM

5.1. BIẾN CON TRỎ

Về thực chất, con trỏ là một biến mà giá trị của nó là một địa chỉ cho phép trong bộ nhớ. Với con trỏ ta có thể truy nhập đến một biến bất kỳ trong chương trình hoặc truy nhập đến các ký ức trong bộ nhớ ROM và RAM. Trong nhiều trường hợp, việc sử dụng con trỏ là phương tiện duy nhất để có thể thiết kế các chương trình hiệu quả và ngắn gọn. Có thể nói việc sử dụng biến con trỏ đã làm cho ngôn ngữ C có thể so sánh với ngôn ngữ bậc thấp Assembler khi cần làm việc với phần cứng.

5.1.1. Khai báo con trỏ

Việc khai báo con trỏ được thực hiện thông qua cú pháp sau:

Kiểu_con trỏ *Tên_con_ trỏ;

Kiểu_con trỏ: Ngoài các kiểu dữ liệu đã được sử dụng cho biến, con trỏ còn có thể có kiểu là *void*. Đây là kiểu dữ liệu rất hay được sử dụng cho hàm.

Tên_con_trỏ: Được đặt theo nguyên tắc đặt tên cho biến.

Một số ví dụ về khai báo con trỏ:

- Con trỏ kiểu int: int *x;
- Con trỏ kiểu char: char *s;
- Con trỏ kiểu float: float* a;

5.1.2. Khởi tạo con trỏ

Sau khi được khai báo, biến con trỏ chứa một giá trị bất kỳ. Đây có thể là địa chỉ của một ô nhớ bất kỳ nào đó trong bộ nhớ, vì vậy việc sử dụng con trỏ này có thể dẫn đến hậu quả đáng tiếc. Như vậy, một điều cần hết sức lưu ý là con trỏ cần phải được khởi tạo giá trị trước khi sử dụng. Con trỏ có thể được khởi tạo qua các cách sau:

- Khởi tạo giá trị NULL.

Ví dụ: int *p = NULL

Với cách khởi tạo này, con trỏ p chưa chỉ đến một địa chỉ cụ thể nào đó trong bộ nhớ. Việc sử dụng giá trị này tuy không có ý nghĩa nhưng cũng không làm phát sinh lỗi nghiêm trọng.

- Khởi tạo bằng địa chỉ của một biến

Ví dụ:

```
int x;
int *p = &x;
```

Trường hợp này người ta nói con trỏ *p* chỉ đến biến x và có giá trị bằng địa chỉ của x. Địa chỉ này của x đã được trình biên dịch cấp phát khi khai báo *x*. Cần lưu ý là kiểu của con trỏ p phải tương ứng với kiểu của x.

- Khởi tạo bằng địa chỉ cụ thể trong bộ nhớ

Ví dụ:

```
char * ch = (char*) 4000;
```

Ở đây con trỏ *ch* đã chỉ đến địa chỉ 4000 trong bộ nhớ. Do 4000 là hằng số nên khi khởi tạo phải sử dụng phép ép kiểu.

5.1.3. Các phép toán đặc biệt về con trỏ

Có 2 phép toán đặc biệt đối với con trỏ: phép toán & và phép toán *, cả hai phép toán này đều là các phép toán một ngôi.

Phép toán & dùng để xác định địa chỉ của toán hạng và có cú pháp sử dụng:

&Toán _hạng;

Ở đây toán hạng có thể là biến hoặc các thành phần của mảng.

Ví dụ:

```
int x, *px;
px = &x; // Con trỏ px có giá trị bằng địa chỉ của biến x;
int x[5]; // Mảng x gồm năm phần tử;
int *px;
px = &x[2]; // px có giá trị bằng địa chỉ của phần tử x[2].
```

Phép toán * dùng để xác định giá trị của biến mà địa chỉ của nó là giá trị của con trỏ toán hạng. Phép toán được sử dụng như sau:

****Toán hạng;***

Ví dụ:

```
int x = 5;
int *px;
px = &x; // Giá trị của con trỏ px là địa chỉ của biến x;
```

```
int y =*px; // y = x ;  *px là giá trị của biến x;
```

Các phép toán & và * là các phép toán có tác dụng loại trừ lẫn nhau. Điều đó có nghĩa là, nếu:

```
float x, y; *px  ;
px = &x; // px trỏ đến x;
```

thì sẽ suy ra:

```
&*px = px;
```

và :

```
*&x = x;
```

Bạn đọc hãy tự chứng minh hai kết luận trên.

Khi làm việc với con trỏ cần lưu ý:

- Con trỏ phải được khởi tạo trước khi sử dụng
- Kiểu của con trỏ và kiểu biến mà nó chỉ đến phải tương ứng với nhau. Nếu điều này không được đảm bảo thì khi biên dịch, chương trình sẽ có sự nhắc nhở về kiểu con trỏ và khi thực hiện có thể dẫn đến kết quả sai.
- Con trỏ là biến nên cũng có thể tham gia vào biểu thức.
- Có thể sử dụng 2 cách khai báo, ví dụ char **name* hoặc *char*name*. Cả 2 cách khai báo này là hoàn toàn tương đương, nhưng cách thứ 2 thường được sử dụng để khai báo cho các hàm nguyên mẫu, nơi mà ta chỉ chú ý đến kiểu của biến mà không cần quan tâm đến tên của nó.
- Phép toán * dùng để xác định giá trị của biến do con trỏ chỉ đến. Vì vậy, nếu px là con trỏ chỉ đến biến x thì việc sử dụng *px là hoàn toàn tương đương với việc sử dụng trực tiếp x.

5.1.4. Các phép toán số học với con trỏ

Về bản chất, con trỏ được định nghĩa như một biến đặc biệt do đó trong các phép toán số học chỉ có 2 phép toán có thể thực hiện trên con trỏ: phép cộng và phép trừ. Khi sử dụng các phép toán này trên con trỏ tuy vậy cũng cần phải để ý đến một số hạn chế.

Ví dụ:

```
int x[3], *px;
```

Nếu:

```
px = &x[0]; //px chỉ đến phần tử đầu của mảng
```

thì:

px + 1 sẽ chỉ đến phần tử a[1];

px + 2 sẽ chỉ đến phần tử a[2];

Qua ví dụ này có thể nhận thấy, khi thực hiện phép cộng và trừ trên con trỏ thì *mỗi đơn vị trong phép toán tương ứng với số byte trong bộ nhớ được phân bố cho kiểu của biến mà con trỏ chỉ đến*.

Các phép toán ++ và -- cũng được dùng cho con trỏ như các biến khác. Đây là các phép toán tăng giảm một đơn vị. Khi thực hiện trên biến con trỏ thì giá trị của con trỏ sẽ tăng hay giảm tương ứng với kích thước của đối tượng do con trỏ chỉ đến. Trong trường hợp này người ta nói con trỏ chỉ đến phần tử đứng trước hay đứng sau nó.

Ví dụ:

```
int x[3], *px;
```

Nếu:

px = &x[2];

Thì:

px ++ sẽ bằng &x[3] và px-- sẽ bằng &x[1];

Trong tất cả các ví dụ được trình bày ở trên ta đã sử dụng tính chất liên tục của các phần tử của mảng khi được lưu trữ trong bộ nhớ.

Trên con trỏ p có thể thực hiện các phép toán sau:

*p++ (*p--);// Trước tiên lấy giá trị nằm trong địa chỉ do con trỏ *p* chỉ đến sau đó giá trị của *p* tăng (hoặc giảm) một đơn vị (chỉ đến phần tử tiếp theo).

++p (--p); // Trước tiên con trỏ *p* tăng hoặc giảm đi một đơn vị (chỉ đến phần tử đứng sau hay đứng trước) sau đó lấy giá trị chứa trong địa chỉ này.

Đối với con trỏ ta cũng có thể sử dụng các phép toán += và -=. Ví dụ:

```
int x, *px;
px = &x; // Khởi tạo con trỏ
px +=1; // px = px +1;
```

Khi sử dụng các phép toán trên con trỏ cần lưu ý:

- Không thực hiện các phép toán: cộng các con trỏ với nhau, cộng hoặc trừ con trỏ với một số thực (float hay double), nhân hoặc chia con trỏ.
- Các con trỏ *p1* và *p2* có thể được so sánh với nhau hoặc trừ cho nhau nếu hai con trỏ này chỉ đến các biến mà giữa chúng có một mối quan hệ nào đó, ví dụ như các phần tử khác nhau của mảng.

Dưới đây là một chương trình ví dụ về con trỏ.

Ví dụ 5.1

Hãy nhập một số nguyên từ bàn phím và in ra các giá trị của byte thấp và byte cao của số nguyên đó.

Khác với chương trình khi minh hoạ các phép toán chia lấy phần dư và phần nguyên, trong ví dụ này yêu cầu của bài toán sẽ được giải quyết thông qua con trỏ.

```
#include <iostream.h>
void main ()
{
    int x;
    unsigned char *px;
    cout << "Nhập x: ";
    cin >> x;
    px = (char *) & x;
    cout << "Giá trị của byte thấp là: "<< int(*px) << endl;
    cout << " Giá trị của byte cao là: " <<int (*++px);
}
```

Để in giá trị từng byte của biến *x* ta đã sử dụng một con trỏ *px* kiểu *char* chỉ đến địa chỉ đầu (byte thấp của biến x). Do con trỏ *px* và *x* có kiểu khác nhau nên để khởi tạo *px* ta phải dùng phép ép kiểu. Như vậy, giá trị byte thấp của x sẽ chính là *px và giá trị byte cao sẽ chính là giá trị nằm trong ô ký ức tiếp theo (tức là *++px). Cũng cần lưu ý là để tránh in ra các ký tự tương ứng với mã ASCII được chứa trong các byte này, trong chương trình cũng đã sử dụng phép ép kiểu khi in.

Một con trỏ lại có thể chỉ đến một con trỏ khác khi khai báo.

Về nguyên tắc, phép khai báo này sẽ cho phép mở rộng khả năng làm việc gián tiếp với các biến. Nhược điểm của cách khai báo này là chương trình trở thành phức tạp hơn và do vậy cũng sẽ khó kiểm soát hơn.

Ví dụ 5.2

```
#include <iostream.h>
void main ( )
{
    int x, *px, **pp;
    x = 20;
    px = &x;
    pp = &px;
    cout << **pp; // In giá trị của x.
}
```

Một kiểu con trỏ rất hay được sử dụng trong ngôn ngữ C++ đó là con trỏ kiểu *void*. Khi khai báo con trỏ kiểu này, thông thường người sử dụng chưa xác định một kiểu cố định cho con trỏ, kiểu cụ thể sẽ được xác định trong quá trình thực hiện chương trình. Khi sử dụng con trỏ kiểu *void* cần lưu ý một điều là con trỏ này có thể nhận một kiểu bất kỳ nhưng điều ngược lại là không cho phép. Khi gán kiểu con trỏ *void* ta cần phải sử dụng phép chuyển kiểu bắt buộc để tránh gây ra lỗi khi biên dịch.

Ví dụ 5.3

```
void main( )
    {
    int *ip;
```

```
void* vp'
char *c;
int i;
int i=5;
c = vp; //sai
ip = &i;
ip = vp; //sai
vp = &i; //Ok
ip = int*(vp);//Ok}
```

5.1.5. Mảng và con trỏ

Giữa con trỏ và mảng tồn tại một mối quan hệ mật thiết và nhìn chung chúng có thể thay thế cho nhau khi sử dụng. Con trỏ chỉ đến một mảng được định nghĩa theo các nguyên tắc chung đã được chỉ ra và giữa con trỏ và mảng phải có sự tương ứng về kiểu. Sau khi được định nghĩa thì việc sử dụng tên con trỏ hoàn toàn tương đương với địa chỉ đầu của mảng và có thể dùng con trỏ để truy nhập đến các phần tử của mảng.

5.1.5.1. Mảng một chiều và con trỏ

1. Mảng một chiều

Hãy bắt đầu bằng mảng một chiều.

Một mảng nguyên một chiều gồm năm phần tử được khai báo như sau:

```
int a[5];
```

Một mảng kiểu ký tự dùng để chứa 20 ký tự (hay còn gọi là chuỗi) được khai báo:

```
char s[21];
```

Đối với mảng kiểu chuỗi cần lưu ý là phải để dành một byte để chứa ký tự kết thúc chuỗi - ký tự '\0' (NULL).

Khi khai báo mảng, các phần tử của mảng có thể được khởi tạo bằng một số cách khác nhau.

- Chỉ ra số phần tử và giá trị của từng phần tử.

Ví dụ:

```
int a[5] = {1, 2, 3, 4, ,5};
char s[6] = {'a', 'b', 'c', 'd', 'e'}; hoặc:
char s[6] = "abcde";
```

Khi chỉ ra số phần tử của mảng khi khởi tạo cần lưu ý:

- Nếu số lượng các phần tử (kích thước mảng) lớn hơn số lượng các giá trị được khởi tạo thì các phần tử không được khởi tạo sẽ có giá trị bằng 0
- Nếu kích thước mảng nhỏ hơn số lượng các giá trị được khởi tạo thì trình biên dịch sẽ thông báo lỗi.

Cũng cần lưu ý là nếu không được khởi tạo khi khai báo thì các phần tử của mảng sẽ nhận một giá trị ngẫu nhiên có sẵn trong bộ nhớ .

Ví dụ 5.4

In số ngày của các tháng trong năm

```
#include <iostream.h>
int day[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
void main ( )
{
```

```
    clrscr ( );
    for(int index =0; index < 12; index++)
        cout << " Tháng " << index +1 << " có " <<
            day[index] << " ngày ";
}
```

- Chỉ ra các giá trị cần khởi tạo cho mảng nhưng không đưa kích thước mảng.

Ví dụ:

```
int a[5] = {1, 2, 3, 4, ,5};
char s[6] = {'a', 'b', 'c', 'd', 'e'}; hoặc:
char s[6] = "abcde";
```

Trong trường hợp này, tuỳ theo số lượng các giá trị được khởi tạo trình biên dịch sẽ cấp phát vừa đủ bộ nhớ dành cho mảng.

Rõ ràng theo cách khởi tạo này, người lập trình không cần để tâm đến vấn đề phát sinh khi kích thước của mảng và số lượng giá trị khởi tạo không phù hợp với nhau. Tuy vậy trong trường hợp này có một vấn đề nhỏ cần phải giải quyết khi xử lý trên mảng, đó là việc xác định số lượng các phần tử của mảng.

Ví dụ 5.5

```
#include <iostream.h>
int day[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
void main ( )
{
    clrscr ( );
```

```
    int count = sizeof(day) /sizeof(int);
    for(int index =0; index <; index++)
        cout << " Tháng " << index +1 << " có " <<
            day[index] << " ngày ";
}
```

Trong ví dụ này, mặc dù kích thước của mảng không được đưa ra khi khởi tạo nhưng giá trị này có thể được xác định thông qua toán tử *sizeof* bằng công thức:

```
count = sizeof(day) /sizeof(int);
```

Tức là số phần tử của mảng bằng số byte được cấp phát cho cả mảng chia cho số byte được cấp phát cho một phần tử.

Ví dụ 5.6

Hãy sắp xếp các phần tử của mảng int theo thứ tự tăng dần.

```
#include <iostream.h>
#define SIZE 5
void main ( )
{
    int a[SIZE];
    int temp; // Biến tạm
    cout << "Hãy nhập các phần tử của mảng:" <<endl;
    for( int i=0; i<SIZE; i++)
    {
        cout << "a[" << i << "] =";
        cin >> a[i];
    }
    for( i= 0; i< SIZE-1; i++)
```

```
            for( int j = i+1; j < SIZE; j++)
                if(a[i] > a[j])
                {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
        cout<< "Hiển thị các phần tử của mảng theo thứ tự tăng dần"
                <<endl;
        for( i =0; i< SIZE; i++)
            cout << "a[" << i << "] =" <<a[i] << endl;
}
```

2. Mảng một chiều và con trỏ

Giữa con trỏ và mảng tồn tại một mối quan hệ mật thiết và nhìn chung chúng có thể thay thế cho nhau khi sử dụng. Trong hai cách làm việc này thì việc sử dụng con trỏ thường cho phép truy nhập đến các phần tử của mảng nhanh hơn so với phương pháp dùng chỉ số của mảng. Nhìn chung, nếu các phần tử của mảng được sử dụng theo một thứ tự tăng giảm tuần tự thì việc sử dụng con trỏ sẽ nhanh và thuận tiện hơn. Trong trường hợp ngược lại thì việc sử dụng chỉ số sẽ làm cho chương trình dễ hiểu hơn mặc dù tốc độ thực hiện của chương trình có thể giảm.

Con trỏ chỉ đến một mảng được khai báo theo các nguyên tắc đã được chỉ ra và giữa con trỏ và mảng mà nó chỉ đến phải có sự tương ứng về kiểu. Hãy xét một vài ví dụ về con trỏ chỉ đến mảng và cách sử dụng nó để truy nhập đến các phần tử của mảng.

Ví dụ

```
int a[5] = {1,2,3,4,5};
int *pa = a;  // Hoặc pa = &a[0];
```

Ở đây con trỏ *pa* đã chỉ đến phần tử đầu của mảng và vì vậy thông qua con trỏ này ta có thể truy nhập đến các phần tử của mảng. Theo các phép toán làm việc với con trỏ ta có:

```
*pa = a[0] = 1;
*(pa +1) = a[1] = 2;
.....
```

Hay tổng quát:

$*(pa+i) = a[i], \quad 0 \le i \le 4;$

Ví dụ nếu:

```
char s[5] = "abcd";
char *ps = s;
```

thì:

```
*ps = 'a'; *(ps+1) = 'b';  *(ps+2) = 'c'; *(ps+3) = 'd'; *(ps+4) = '\0';
```

Mặc dù tên con trỏ và tên của mảng đều có giá trị là địa chỉ trong bộ nhớ nhưng cần lưu ý:

- Tên con trỏ là biến do đó việc sử dụng phép toán tăng giảm trên con trỏ là hoàn toàn hợp lệ. Tức là ta có thể viết:

```
pa++ hay pa--;
```

- Tên mảng là hằng nên việc sử dụng phép toán tăng giảm trên con trỏ là không hợp lệ. Tức là không thể viết:

```
a++ hay a--;
```

Ví dụ 5.7

Sử dụng con trỏ hãy tính tổng các phần tử của mảng các số nguyên.

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    int a[10];
    cout << "Hãy nhập số phần tử của mảng:  "<< endl;
    int n;
    cin >>n;
    cout << "Hãy nhập giá trị các phần tử của mảng" << endl;
    for(int i=0; i < n ; i++)
        cin >> a[i];
    int *pa = a;
    long int Sum =0;
    for(i=0; i< n; i++)
        Sum += *(pa+i);
    cout  << "Tổng các phần tử của mảng là: " << Sum;
    getch( );
}
```

Trong ví dụ này, thay vì việc sử dụng chỉ số để truy nhập đến các phần tử của mảng, chương trình đã sử dụng con trỏ *pa* chỉ đến mảng để truy nhập và xử lý mảng.

Điểm hạn chế khi làm việc với mảng là kích thước tối đa của nó phải được xác định rõ ràng trong quá trình biên dịch (ở ví dụ

trên kích thước tối đa là 10). Để kích thước này có thể được cập nhật khi thực hiện chương trình ta có thể sử dụng con trỏ thay thế cho mảng. Khi đó, bộ nhớ được cấp phát sẽ được xác định trong quá trình thực hiện bằng toán tử *new* và con trỏ sẽ chỉ đến địa chỉ đầu của khối bộ nhớ này. Ví dụ dưới đây sẽ mô tả điều này.

Ví dụ 5.8

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
    int *a;
    cout << "Hãy nhập số phần tử của mảng:  "<< endl;
    int n;
    cin >>n;
    a = new int [n]; // Con trỏ a chỉ đến địa chỉ đầu của bộ
                     //nhớ được cấp phát gồm n phần tử
    cout << "Hãy nhập giá trị các phần tử của mảng" << endl;
    for(int i=0; i < n ; i++)
        cin >> *(a+i);
    long int Sum =0;
    for( i= 0; i< n; i++)
        Sum += *(a+i);
    cout  << "Tổng các phần tử của mảng là: " << Sum;
    delete a; // Giải phóng vùng nhớ đã cấp cho a
```

```
    getch ();
}
```

Cũng giống như C, C++ có thể sử dụng một số hàm để phân bố bộ nhớ trong thời gian thực hiện chương trình như *malloc*, *calloc* và hàm giải phóng bộ nhớ đã được cấp phát như *free*. Chẳng hạn ta có thể cấp phát bộ nhớ gồm 10 phần tử kiểu *char* và trả lại địa chỉ đầu cùng vùng đã cấp phát cho con trỏ *p* như sau:

```
char *p ;
p = (char*)malloc(10);
```

Hoặc cấp phát bộ nhớ gồm 10 phần tử kiểu *int* và trả lại địa chỉ đầu của vùng nhớ này cho con trỏ *p* bằng cách:

```
int* p;
p = (int*) malloc(10*sizeof(int));
```

Vùng nhớ đã được cấp phát có thể được giải phóng sau khi sử dụng bằng hàm *free:*

```
free(p);
```

Khi sử dụng các hàm trên cần chú ý một số điểm sau:

- Tham số được truyền cho hàm *maloc* chính là số byte của vùng nhớ cần cấp phát.
- Khi không thể cấp phát vùng nhớ, hàm trả lại giá trị NULL cho con trỏ *p*. Điều này có nghĩa là con trỏ này chưa trỏ đến bất kỳ vị trí nào trong bộ nhớ.
- Sau khi vùng nhớ đã được cấp phát, các phần tử của nó có thể được truy nhập thông qua con trỏ chỉ đến

đầu vùng nhớ này (con trỏ *p*).

- Để sử dụng các hàm *malloc, calloc* và *free,* trong chương trình cần sử dụng file tiêu đề alloc.h hoặc stdlib.h.
- Câu lệnh int* p = (int*)malloc(0) trả lại giá trị NULL cho con trỏ p. Điều này có nghĩa là con trỏ *p* chưa chỉ đến bất kỳ vị trí nào trong bộ nhớ và vì vậy việc sử dụng nó không gây ra hậu quả nghiêm trọng.

Cách sử dụng của hàm *calloc* bạn đọc có thể tham khảo trong trợ giúp của trình biên dịch.

Giống như các hàm đã nêu trên, *new* và *delete* cũng được dùng để cấp phát bộ nhớ động với một số khác biệt:

- *new và delete* chỉ được dùng cho C++.
- *new* và *delete* là các toán tử nên chúng được xem như một bộ phận sẵn có của C++. Khi sử dụng các toán tử này không cần phải kết nối bất cứ một file tiêu đề nào.
- Câu lệnh int *p = new int [0] trả lại giá trị khác 0 cho con trỏ *p*. Giá trị này của con trỏ truy vậy có thể làm phát sinh lỗi nghiêm trọng nếu được sử dụng cho các mục đích khác.

Cần lưu ý là về bản chất cả *new, delete* và *malocc* và *free* đều dùng để cấp phát bộ nhớ động cho các biến, nhưng các toán tử *new, delete* trong C++ đã đem lại cho người lập trình một sự linh hoạt và mềm dẻo cao do khả năng định nghĩa chồng các toán tử của lớp (xem chương IX).

5.1.5.2. Mảng nhiều chiều và con trỏ

1. Mảng nhiều chiều

Phần này chủ yếu đề cập đến việc khai báo và sử dụng mảng hai chiều.

Một số ví dụ về khai báo mảng hai chiều:

```
int a[2][4];
```

Đây là mảng hai chiều với kích thước 8 phần tử bao gồm 2 hàng và 4 cột. Cần lưu ý là trong bộ nhớ các phần tử sẽ được phân bố lần lượt theo hàng và chỉ số của hàng và cột luôn bắt đầu bằng giá trị 0. Như vậy thứ tự lưu trữ của các phần tử trong bộ nhớ của mảng *a* lần lượt sẽ là:

a[0][0], a[0][1], a[0][2], a[0][3], a[1][0], a[1][1], a[1][2], a[1][3].

Có thể sử dụng mảng hai chiều để khai báo cho dữ liệu kiểu chuỗi. Chẳng hạn:

```
char s[2][20];
```

Với cách khai báo này, *s* là mảng hai chiều kiểu chuỗi có thể được sử dụng để lưu trữ hai xâu với độ dài tối đa là 20 ký tự kể cả ký tự kết thúc '\0'.

Cũng như mảng một chiều, đối với mảng hai chiều ta cũng có thể sử dụng nhiều cách khởi tạo khác nhau khi khai báo.

- Chỉ ra số phần tử và giá trị của các phần tử.

Ví dụ:

```
int a[2][3] = { {1,2,3}, {4,5,6}};
```

Với cách khởi tạo này thì mảng hai chiều có thể được xem như sự lưu trữ kế tiếp của hai mảng một chiều và ta có:

```
a[0] [0] =1, a[0][1] =2, a[0][2] = 3;
```

a[1][0] = 4, a[1][1] = 5, a[1][2] = 6;

Vì các phần tử của mảng hai chiều được phân bố liên tục theo hàng nên ta cũng có thể khởi tạo mảng hai chiều theo cách đã được thực hiện đối với mảng một chiều.

Ví dụ:

int a[2][3] = { 1,2,3,4,5,6};

Theo cách khởi tạo này thì ta cũng có:

a[0] [0] =1, a[0][1] =2, a[0][2] = 3;

a[1][0] = 4, a[1][1] = 5, a[1][2] = 6;

Mặc dù có thể khởi tạo mảng hai chiều bằng hai cách trên nhưng cũng cần lưu ý sự khác biệt giữa hai cách khởi tạo này.

Ví dụ:

int a[2][3] = { {5,6},{7,8}}; //a[0][2] =0, a[1][2] =0

int a[2][3] = {5,6,7,8} // a[1][1] =0, a[1][2] =0;

- Chỉ ra các giá trị cần khởi tạo cho mảng nhưng không đưa kích thước mảng.

Khác với mảng một chiều, trong trường hợp đối với mảng hai chiều cần chỉ rõ kích thước của cột khi khởi tạo các giá trị. Cùng với số lượng các giá trị được khởi tạo, kích thước của cột sẽ cho phép trình biên dịch xác định số lượng phần tử của mảng để cấp phát bộ nhớ khi làm việc.

Ví dụ:

int a[][3] = {{1,2,3}, {4,5,6}};

Ví dụ 5.9

Tính tổng lượng mưa trong từng năm và tổng lượng mưa của cùng một tháng trong 5 năm.

```
# include <iostream.h>
# include <conio.h>
#define TWLV  12 // Số tháng trong năm
#define YRS  5   // Số năm
void main ( )
{
    static float rain[ ][TWLV] = {
    {10.2,8.1, 4.2, 2.1, 1.8, 0.2, 0.3, 1.1, 2.3, 6.1,7.4},
    {9.2, 9.8, 4.4, 3.3, 2.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 5.2},
    {6.6, 5.5, 3.8, 2.8, 1.6, 0.2, 0.0, 0.,0, 1.3, 2.6, 4.2},
    {4.3, 4.3, 4.3, 3.0, 2.0, 1.0, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6},
    {8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.2}};
    int year, month;
    float subtot;
    //Tính lượng mưa của các tháng trong từng năm
    for (year = 0; year < YRS; year++)
        {
        subtot =0;
        for(month=0,subtot =0; month < TWLV; month ++)
            {
            subtot += rain[year][month]
            cout <<" Lượng mưa trong năm thứ  "<<year +1
                 << " là " << subtot <<endl;
            }
        }
    //Tính lượng mưa của cùng một tháng trong 5 năm
```

```
for (month = 0; month < TWLV; month++)
    {
    subtot =0;
    for(year=0,subtot =0; year < YRS; year ++)
        {
        subtot += rain[year][month]
        cout <<"Lượng mưa của tháng thứ"<<month +1
             << " là " << subtot <<endl;
        }
    }
}
```

2. Mảng nhiều chiều và con trỏ

Giữa mảng nhiều chiều và con trỏ cũng có một mối liên quan với nhau. Hãy giả sử rằng ta có các khai báo sau:

```
int zippo [4][2];
int *pri;
```

Khi đó phép gán:

```
pri = zippo;
```

sẽ cho phép *pri* chỉ đến phần tử đầu của mảng zippo. Tức là:

```
pri == &zippo[0][0];
```

Sau phép gán này có thể sử dụng con trỏ *pri* để chỉ đến các phần tử khác của mảng:

```
pri == &zippo[0][0];
pri +1 == &zippo[0][1];
pri + 2 == &zippo[1][0];
```

pri + 3 == &zippo[1][1];

Một cách tổng quát, nếu zippo là mảng gồm n hàng và m cột thì:

pri + i*m + j == &a[i][j];

và:

*(pri + i*m+j) = a[i][j];

Với $0 \le i \le n$ và $0 \le j \le m$

Khi sử dụng mảng nhiều chiều cần lưu ý là phép toán lấy địa chỉ các phần tử của mảng số thực không phải bao giờ cũng đúng. Ta hãy xét chương trình nhập các số liệu cho ma trận thực từ bàn phím bằng cách sử dụng các hàm vào / ra của C.

Ví dụ 5.10

```
#include <stdio.h>
void main ( )
{
    float a[3][5];
    printf("Hãy nhập giá trị các phần tử của mảng\n");
    for(i=0; i<3; i++)
        for(j =0; j < 5; j++)
            {
            printf("a[%d][%d] =", i,j);
             scanf("%f", &a[i][j]);
            }
}
```

Đối với một số chương trình biên dịch, chương trình này sẽ phát sinh lỗi vì phép toán lấy địa chỉ của mảng hai chiều là

không được phép. Lỗi này có thể tránh được nếu để ý rằng:

a+i*m+j == &a[i][j];

khi đó câu lệnh *scanf("%f", &a[i][j]);* có thể được thay thế bởi *scanf("%d", (float*) a+i*5 +j);*

Cần lưu ý là để xác định được địa chỉ của phần tử ở hàng i và cột j chúng ta phải dùng phép chuyển kiểu bắt buộc đối với a: *(float*)a*, đây là con trỏ chỉ đến thành phần *a[0][0]* của ma trận. Bằng cách này, thành phần *a[i][j]* sẽ có địa chỉ là *(float*)a + i*m+j.*

Một trường hợp nữa cần quan tâm đối với mảng hai chiều là mảng này có thể được biểu diễn qua mảng một chiều từ các đẳng thức:

zippo[0] == &zippo[0][0];

zippo[1] == &zippo[1][0];

zippo[2] == &zippo[2][0];

zippo[0] == &zippo[3][0];

Ví dụ dưới đây sẽ minh hoạ cho trường hợp trên:

Ví dụ 5.11

```
#include <iostream.h>
#define ROW 3
#define COL 4
void main ( )
{
    int a[ROW][COL];
    cout <<"Hãy nhập các phần tử của mảng:"<<endl;
    int i,j;
```

```
    for(i=0; i < ROW; i++)
        for(j =0; j < COL; j++)
        {
            cout << "a["<< i <<"]"<<"[" <<j <<"] =";
            cin >> a[i][j];
        }
    long Sum =0;
    for(i=0; i < ROW; i++)
        for(j =0; j < COL; j++)
            Sum +=  *(a[i] +j );
    cout <<"Tổng các phần tử của mảng là: "<< Sum;
}
```

5.1.5.3. Mảng các con trỏ

Theo định nghĩa, con trỏ là biến do đó chúng cũng có thể được khai báo theo kiểu mảng. Cú pháp của việc khai báo mảng các con trỏ có dạng:

Kiểu *tên[kích thước];

Như vậy, một mảng các con trỏ thuộc kiểu *int* gồm 4 phần tử có thể được khai báo như sau:

```
int *px[4];
```

Rõ ràng các con trỏ này phải được khởi tạo trước khi chúng được sử dụng. Một trong những phương pháp khởi tạo giá trị của con trỏ mà ta đã xét là gán địa chỉ của các biến đã được khai báo cho các con trỏ này. Chẳng hạn:

```
int *px[4];
```

```
int x1, x2, x3, x4;
px[0] = &x1;
px[1] = &x2;
px[2] = &x3;
px[3] = &x4;
```

Cũng có thể khai báo một mảng các con trỏ, trong đó mỗi phần tử của mảng lại là các con trỏ chỉ đến các biến thuộc kiểu *char* theo cách sau:

```
char *py[ ];
```

Cần chú rằng, nếu không sử dụng [ ] thì *py* sẽ chỉ là con trỏ chỉ đến biến thuộc kiểu char. Trong cách khai báo này, kích thước của mảng sẽ được trình biên dịch cấp phát khi khởi tạo các giá trị của mảng. Chẳng hạn:

```
char *menu[ ] = {"Save", "Open", "Save As"};
```

là một mảng các con trỏ, trong đó các phần tử của nó là con trỏ chỉ đến một xâu ký tự tương ứng. Trong trường hợp này ta có:

```
menu[0] = "Save",
menu[1] = "Open".
menu[2] = "Save As"
```

Như vậy ts sẽ có:

```
*(menu[0] +0) = 'S', *(menu[1] +1) = 'a', v.v.
```

Trong ví dụ trên, ta cũng có thể dùng mảng hai chiều để thay thế cho mảng các con trỏ. Chẳng hạn có thể khai báo mảng các con trỏ *menu* bằng cách sau:

```
char menu[3][10] = {"Save", "Open", "Save As"};
```

Điểm khác biệt ở đây là ở chỗ số lượng các con trỏ đã được định trước (3) và mỗi con trỏ sẽ chỉ đến một xâu ký tự có kích thước cũng đã được xác định trước trong quá trình khởi tạo mảng. Mặc dù có thể sử dụng cả hai cách trên khi khai báo mảng của xâu ký tự nhưng cách khai báo mảng các con trỏ hay được sử dụng hơn do các ưu điểm sau:

- Cho phép tham chiếu đến các thành phần của mảng nhanh hơn
- Đơn giản hoá quá trình điều khiển bộ nhớ. Chẳng hạn trong trường hợp cần phải thay đổi vị trí của hai hàng nào đó thuộc mảng, ta chỉ cần hoán vị giá trị của các con trỏ tương ứng cho nhau trong khi mảng do các con trỏ chỉ đến vẫn nằm ở vị trí cũ.
- Do kích thước của mảng sẽ được xác định khi biên dịch đúng bằng kích thước cần cung cấp cho mảng nên cách khai báo này sẽ tiết kiệm được bộ nhớ. Đây cũng là một ưu điểm quan trọng so với trường hợp phải xác định rõ giá trị kích thước của mảng khi khai báo.

Ví dụ dưới đây sẽ mô tả một trường hợp cụ thể khi sử dụng mảng các con trỏ.

Ví dụ 5.12

Hãy hiển thị các phần tử của mảng int theo thứ tự tăng dần mà không làm thay đổi thứ tự vật lý các phần tử của mảng.

Ví dụ tương tự đã được trình bày trong phần mảng một chiều. Ở đó thứ tự vật lý các phần tử của mảng đã bị thay đổi sau khi sắp xếp. Ví dụ này sẽ trình bày cách sử dụng mảng các con trỏ để giải quyết yêu cầu của bài toán.

```
#include <iostream.h>
#define SIZE 5
void main ( )
{
    int a[SIZE];
    int *pa[5];
    int *temp; // Biến tạm
    cout << "Hãy nhập các phần tử của mảng:" <<endl;
    for( int i=0; i<SIZE; i++)
    {
        cout << "a[" << i << "] =";
        cin >> a[i];
        pa [i] = &a[i];
    }
    // Thay đổi vị trí các con trỏ
    for( i= 0; i< SIZE-1; i++)
        for( int j = i+1; j < SIZE; j++)
            if(*pa[i] > *pa[j])
            {
                temp = pa[i];
                pa[i] = pa[j];
                pa[j] = temp;
            }
    cout<< " In lại thứ tự vật lý của các phần tử :"<<endl;
    for( i =0; i< SIZE; i++)
        cout << "a[" << i << "] =" << a[i] << endl;
```

```
    cout << "Hiển thị các phần tử của mảng theo thứ tự tăng dần"
         <<endl;
    for( i =0; i< SIZE; i++)
         cout << "a[" << i << "] =" <<*pa[i] << endl;
}
```

Trong ví dụ ta đã dùng mảng các con trỏ *pa* gồm 5 phần tử để chỉ đến các phần tử của mảng. Đầu tiên các con trỏ *pa[0], pa[1], pa[2], pa[3], pa[4]* lần lượt chỉ đến các phần tử *a[0], a[1], a[2], a[3], a[4].* Bằng thuật toán sắp xếp, sau khi so sánh các giá trị của các phần tử do các con trỏ này chỉ đến, các con trỏ *pa[0], pa[1], pa[2], pa[3], pa[4]* sẽ lần lượt chỉ đến các phần tử của mảng theo thứ tự tăng dần về giá trị trong khi các phần tử của mảng vẫn giữ nguyên thứ tự ban đầu.

5.2. BIẾN THAM CHIẾU

Cũng như con trỏ, biến tham chiếu được dùng để truy nhập gián tiếp đến một biến khác. Như vậy, về khái niệm có thể xem con trỏ và biến tham chiếu gần giống nhau nhưng giữa chúng cũng tồn tại một số sự khác biệt. Có thể xem biến tham chiếu như một bước tiến của con trỏ và khi sử dụng nó ta cần lưu ý một số vấn đề sau:

- Biến tham chiếu cũng chứa địa chỉ của một đối tượng khác.
- Biến tham chiếu cần phải được khởi tạo khi khai báo.
- Không được cấp phát bộ nhớ động cho biến tham chiếu.
- Không dùng các phép toán tăng hoặc giảm địa chỉ đối với biến tham chiếu

Cú pháp khai báo của biến tham chiếu có dạng sau:

Kiểu& Tên biến = Giá trị;

Trong khai báo này:

Kiểu: Kiểu của biến tham chiếu, có thể là *char, int, float,* v.v

&: Biểu thị đây là biến tham chiếu

Giá trị: Giá trị khởi tạo của biến tham chiếu. Giá trị này có thể là hằng hoặc là một biến khác v.v.

Mặc dù biến tham chiếu là con trỏ, nhưng ta lại có thể sử dụng tên của biến tham chiếu để truy nhập đến giá trị của nó. Đây là điểm khác biệt giữa biến thuộc kiểu tham chiếu và biến thuộc kiểu con trỏ.

Ví dụ 5.13

```
#include <string.h>
void main()
{
    int& i=3;
    int j;
    if (i ==3) // sử dụng tên của tham chiếu như giá trị
    {
        i - -; // Giá trị của i giảm đi 1
        j =1;
    }
    cout << " Giá trị của i là: " << i;
    cout << "Giá trị của j là : " << j;
}
```

Ở ví dụ này, biến *i* đã tham chiếu đến giá trị 3. Sau khi khai báo có thể thấy việc truy nhập đến biến tham chiếu được thực

hiện hoàn toàn như một biến bình thường và phép toán tăng giảm một đơn vị trên biến tham chiếu sẽ làm tăng hoặc giảm giá trị mà biến tham chiếu chứ không phải tăng giảm đại chỉ như trong trường hợp biến con trỏ.

Hãy xét một ví dụ khác về biến tham chiếu. Trong ví dụ này cả hai biến đều tham chiếu đến cùng một địa chỉ vì vậy trên thực tế có thể xem hai biến này là như nhau.

Ví dụ 5.14

```
#include <iostream.h>
#include <conio.h>
void main()
    {
    int i=3;
    int& j =i;  // j và i đều chỉ đến cùng vị trí
    j =5;       //Giá trị của i cũng thay đổi i=5
    cout << " Giá trị của i là:  " << i <<endl;
    cout << " Giá trị của j là:  " << j <<endl;
    }
```

Khi sử dụng biến tham chiếu, tuy vậy ta không được dùng kiểu khai báo void với mục đích chung như kiểu con trỏ. Chẳng hạn việc khai báo sau là sai;

```
void& a-reference=3;
```

Cũng là sai nếu người lập trình thay đổi giá trị của biến tham chiếu kiểu *const*. Ví dụ:

```
const int& number =9; //Biến tham chiếu kiểu const
number =5; //Sai;
number =9; //Cũng sai
```

Như vậy ta có thể hình dung một biến tham chiếu giống như một bí danh của biến khác. Biến tham chiếu sẽ làm cho các hàm có thay đổi nội dung của các đối số của nó được viết một cách rõ ràng và tự nhiên hơn.

5.3. HÀM

Hàm là đơn vị cơ bản trong ngôn ngữ C++ khi thực hiện chương trình. Sau khi thực hiện, hàm có thể nhận một giá trị nào đó tương ứng với kiểu của hàm.

Việc sử dụng hàm cho phép chia chương trình thành những chương trình nhỏ (module) tương đối độc lập với nhau. Điều này sẽ có tác dụng làm cho chương trình trở thành gọn gàng, dễ hiểu và thuận tiện khi cần sửa lỗi. Ngoài ra, việc tạo ra các hàm còn cho phép sử dụng chúng trong các chương trình khác nhau mà không phải viết lại.

Phần này sẽ nghiên cứu một số vấn đề liên quan đến hàm:

- Định nghĩa hàm.
- Khai báo nguyên mẫu của hàm.
- Truyền tham số cho hàm.
- Gọi hàm.
- Hàm đệ qui

5.3.1. Định nghĩa, khai báo và gọi hàm

5.3.1.1. Định nghĩa hàm

Nói một cách ngắn gọn, việc định nghĩa hàm nhằm đưa ra một số câu lệnh để xử lý một công việc nào đó. Cấu trúc đầy đủ

của việc định nghĩa hàm có dạng sau:

```
Kiểu_hàm  Tên_hàm(Khai báo các tham số hình thức)
    {
        Khai báo các biến cục bộ;
        Các lệnh;
        return (biểu thức);
    }
```

Trong đó:

Kiểu_hàm: là kiểu của giá trị mà hàm có thể nhận được. Các kiểu mà hàm có thể nhận được bao gồm:

- char, int, float, double v.v: đây là các kiểu đã được sử dụng cho biến.
- void: khi hàm có kiểu này, hàm sẽ không nhận giá trị trả về, khi đó hàm có vai trò như thủ tục trong pascal.
- Kiểu con trỏ, cấu trúc v.v: kiểu cấu trúc sẽ được trình bày trong phần sau của giáo trình.

Tên_hàm: được đặt theo nguyên tắc đặt tên cho biến.

Các tham số hình thức: Là các thông tin được truyền cho hàm. Giá trị của các tham số này hoàn toàn phụ thuộc vào các tham số thực khi gọi hàm.

return: là câu lệnh cho phép trả lại giá trị cho hàm và kết thúc việc thực hiện hàm. Giá trị này được xác định thông qua biểu thức. Kiểu của giá trị này phải tương ứng với kiểu hàm, trong trường hợp ngược lại, trình biên dịch sẽ sử dụng phép chuyển kiểu tự động. Khi kiểu của hàm là *void* thì câu lệnh

return (không có biểu thức) có thể được dùng để kết thúc hàm hoặc có thể bỏ qua. Mặc dù có thể sử dụng nhiều câu lệnh *return* nhưng do câu lệnh *return* sẽ kết thúc việc thực hiện hàm nên chỉ có một câu lệnh *return* duy nhất được thực hiện.

Ví dụ về định nghĩa hàm:

```
int vidu( int x, int y)
{
    ..............
    return 0;
}
```

Khi các hàm được định nghĩa có cùng tên, các hàm này sẽ được C++ cho là khác nhau nếu một trong các điều kiện sau được đảm bảo:

- Khác lớp (khái niệm về lớp sẽ được trình bày ở phần tiếp của giáo trình).
- Khác số lượng tham số truyền.
- Khác kiểu của tham số truyền.

Cần lưu ý là C++ không dùng *kiểu_hàm* để phân biệt các hàm. Như vậy việc định nghĩa các hàm có cùng tên, cùng số lượng và kiểu tham số truyền (trong cùng lớp) là không được phép.

Một khái niệm liên quan đến việc định nghĩa hàm mà ta cần quan tâm đó là các cấp lưu trữ của biến. Trong C và C++, mỗi biến được đặc trưng bởi 3 thuộc tính sau:

- Kiểu của biến

- Phạm vi làm việc của biến
- Thời gian tồn tại của biến

Trong các thuộc tính này thì *phạm vi hoạt động* của biến phụ thuộc vào vị trí khi khai báo biến và phạm vi đó có thể là một khối lệnh trong hàm, một hàm hoặc một tập tin nguồn. *Thời gian tồn tại của* biến là khoảng thời gian biến còn tồn tại khi thực hiện chương trình.

Phụ thuộc vào thời gian tồn tại và phạm vi làm việc, các biến được phân bố ở từng vùng khác nhau trong bộ nhớ, hay nói cách khác là *các cấp lưu trữ* (class storage) của chúng là khác nhau. Có 4 biến từ dùng để chỉ định các cấp lưu trữ: *auto, extern, static* và *register* và một biến có thể được khai báo bằng cách sau;

Biến_từ Kiểu_biến Tên_biến;

Dưới dây xin giới thiệu một cách ngắn gọn về các biến từ này.

1. Biến từ auto

Biến từ này thường được dùng để khai báo các biến cục bộ và các tham số hình thức của hàm. Do các biến kiểu này mặc định được xem như các biến thuộc kiểu auto nên từ khoá này có thể được bỏ qua. Khi được khai báo theo kiểu auto, các biến sẽ được trình biên dịch cấp phát bộ nhớ trong ngăn xếp và giá trị của chúng là ngẫu nhiên.

Ví dụ trong định nghĩa hàm:

```
Fun(int x, int y)
{
    float m;
```

```
        int i;
        .....
        for(int i =0; i< 100; i++)
        {
            int n;
            .....
        }
    }
```

Các biến x, y, i và m, n đều là các biến cục bộ và có cấp lưu trữ auto. Trong khi các biến x,y, m và i có phạm vi hoạt động là toàn bộ thân của hàm thì biến n chỉ có phạm vi hoạt động trong thân vòng lặp *for*.

2. Biến từ extern

Cấp lưu trữ này thường được dùng cho các biến ngoài (toàn cục) và cũng là cấp lưu trữ mặc định cho các biến đó. Các biến được khai báo với cấp lưu trữ này sẽ được cấp phát bộ nhớ trong vùng dữ liệu của chương trình và sẽ tồn tại trong suốt thời gian thực hiện của chương trình.

Ví dụ:

```
int i,j; // Biến extern
void main ( )
{
    .....
}

float x,y; // Biến extern
```

```
void fun( )
{
.....
}
```

Trong ví dụ này, mặc dù đều là các biến *extern* nhưng *i, j* có phạm vi hoạt động tính từ hàm *main* trong khi các biến *x, y* chỉ có phạm vi hoạt động bắt đầu từ hàm *fun*.

3. Biến từ static

Loại biến từ này phải được viết rõ ràng khi khai báo biến và các biến với cấp lưu trữ này sẽ được cấp phát bộ nhớ trong vùng dữ liệu của chương trình. Các biến này có các đặc điểm:

- Thời gian tồn tại của biến bằng thời gian thực hiện chương trình.
- Phạm vi hoạt động của biến là trong thân hàm mà biến được khai báo.
- Các biến *static* được khởi tạo giá trị mặc định bằng 0.

Ví dụ:

```
void fun( )
{
    static int i; // Biến kiểu static
    .....
    i = i+2;
}
```

Cần lưu ý là mặc dù có thể là biến cục bộ nhưng mỗi khi thoát khỏi thân hàm, giá trị của biến vẫn được lưu trữ và sẽ

được tiếp tục sử dụng khi quyền điều khiển lại được trao cho hàm có chứa lời khai báo biến. Chẳng hạn, trong ví dụ trên khi hàm *fun* được thực hiện lần đầu thì sau khi hàm kết thúc giá trị của *i* sẽ là 2. Giá trị này sẽ được dùng lại khi quyền điều khiển lại được trao cho hàm fun và sau khi ra khỏi hàm lần 2 giá trị của *i* sẽ là 4. v.v

4. Biến từ register

Loại biến từ này chỉ được áp dụng cho các biến cục bộ hoặc các biến hình thức kiểu *int* và *char* và cũng phải được khai báo rõ ràng.

Ví dụ:

```
fun (register int x0
{
    register k;
    .....
}
```

Đối với các biến thuộc cấp lưu trữ này cần lưu ý:

- Các biến được lưu trữ trong thanh ghi thay cho việc lưu trữ trong bộ nhớ của máy. Do các phép toán được thực hiện trên thanh ghi bao giờ cũng nhanh hơn các biến được lưu trữ trong các ô ký ức nên việc sử dụng cấp lưu trữ này cho biến sẽ làm giảm đáng kể thời gian thực hiện các phép toán.
- Số lượng các biến có cấp lưu trữ register không được vượt quá số lượng các thanh ghi có thể được sử dụng ở thời điểm khai báo. Nếu không còn thanh ghi tự do, biến sẽ được mặc nhiên xem là có cấp lưu trữ auto.

5. Biến từ kiểu const

Trong C cũng như C++, việc khai báo một biến kiểu *const* sẽ làm cho biến đó chỉ được phép đọc trong chương trình. Các biến kiểu này phải được khởi tạo khi thực hiện và mọi cố gắng để làm thay đổi giá trị của biến sau đó sẽ bị chương trình bắt lỗi. Phạm vi của các giá trị *const* giữa C và C++ tuy vậy lại có sự khác nhau. Trong C, các giá trị *const* có phạm vi toàn cục, có nghĩa là chúng có thể được nhìn thấy ở các files khác trừ khi chúng được khai báo kiểu *static*. Trong C++, các biến được khai báo kiểu *const* có phạm vi hoạt động cục bộ, trong file chúng được khai báo. Chẳng hạn, việc khai báo sau sẽ làm cho biến *i* có phạm vi hoạt động cục bộ:

```
const int i=5;
```

và việc sử dụng giá trị của *i* ở các file khác sẽ bị trình liên kết báo lỗi khi thực hiện. Để cho giá trị của *i* có thể được sử dụng ở các file khác (phạm vi hoạt động của *i* là toàn cục) cần sử dụng cách khai báo sau:

```
extern const int i=9;
```

Khi đó *i* cần phải được khai báo lại ở các file có sử dụng giá trị của nó, tuy vậy ở các file này giá trị của *i* không được phép khởi tạo lại;

```
extern const int i;
```

Cũng cần lưu ý là tất cả các biến được khai báo theo kiểu const trong các hàm sẽ luôn có phạm vi hoạt động cục bộ. Việc sử dụng từ khóa *extern* cho các biến kiểu này là không cho phép và phát sinh lỗi. Chẳng hạn việc khai báo sau là sai:

```
void example()
    {
        extern const int i=5; //sai
        .....
    }
```

Như vậy một biến được khai báo kiểu *const* để có phạm vi hoạt động toàn cục khi sử dụng từ khóa *extern*, biến đó phải được khai báo trước tất cả các hàm trong file.

Khi khai báo một biến kiểu *const*, ngoài việc giá trị của biến không thể thay đổi trong quá trình thực hiện chương trình, trình biên dịch còn có thể thực hiện việc kiểm tra kiểu của biến này. Tuy vậy trong khi biên dịch, trình biên dịch sẽ không phân bố bộ nhớ cho các biến này khi không cần thiết. Trong đoạn chương trình sau:

```
const int TRUE=1;
const int FALSE=0;
while(TRUE)
{
    .....
    int a = FALSE;
}
```

các biến TRUE và FALSE sẽ được thay thế bằng giá trị khi biên dịch mà không cần phải phân bố bộ nhớ cho chúng. Để có thể phân bố bộ nhớ cho các biến kiểu const ta cần khai báo theo kiểu sau:

```
const int& number =9;
```

Theo cách khai báo này, *number* được xem như tất cả các

biến tham chiếu khác với một điều khác biệt là giá trị của nó không thể bị thay đổi khi thực hiện chương trình.

6. Biến từ volatile

Biến từ này được sử dụng để báo cho C biết rằng giá trị của một biến có thể được thay đổi bằng một cách nào đó mà không được mô tả trong chương trình (chẳng hạn có thể bị thay đổi bởi các chương trình của Dos hay Bios...).

5.3.1.2. Gọi hàm

Ngoài hàm *main* là hàm chính sẽ được gọi và thực hiện ngay sau khi khởi động chương trình, để một hàm nào đó có thể được thực hiện ta cần phải chuyển quyền điều khiển cho hàm (gọi hàm). Hàm nhận quyền điều khiển được gọi là hàm bị gọi, hàm mà từ đó quyền điều khiển được trao cho hàm khác được gọi là hàm gọi. Khi nhận giá trị trả về, việc gọi hàm có dạng sau:

Tên_biến = Tên_hàm(danh sách các tham số thực);

Trong trường hợp này giá trị của hàm sẽ được gán cho biến.

Khi hàm không nhận giá trị trả về (có kiểu *void*) câu lệnh gọi hàm có dạng:

Tên_hàm(danh sách các tham số thực);

Trong các lời gọi hàm này, có thể hiểu các tham số thực là các giá trị cụ thể nào đó được truyền cho hàm. Các giá trị này có thể là hằng, giá trị của biến, giá trị của con trỏ v.v.

5.3.1.3. Khai báo nguyên mẫu hàm

Nếu lời gọi hàm được viết trước định nghĩa thì hàm cần

được khai báo trước khi gọi. Việc khai báo này được gọi là khai báo hàm nguyên mẫu. Khi khai báo hàm nguyên mẫu, cần chỉ rõ kiểu hàm, tên hàm và kiểu của các tham số truyền. Việc khai báo hàm nguyên mẫu đặc biệt có hiệu quả khi lời gọi và định nghĩa hàm được viết ở các file khác nhau và vì vậy sẽ được sử dụng trong tài liệu. Chẳng hạn:

```
int vidu(int, int);
```

là khai báo nguyên mẫu của hàm:

```
int vidu(int x, int y)
{
.....
}
```

Hoặc:

```
void vidu(int, float, char);
```

là khai báo nguyên mẫu của hàm:

```
void vidu(int x, float y, char c)
{
.....
}
```

Trong C++ khai báo hàm nguyên mẫu của các hàm chuẩn được thực hiện trong các file tiêu đề, vì vậy trước khi sử dụng các hàm này cần kết nối các file tiêu đề vào file nguồn bằng chỉ thị #include.

5.3.2. Truyền tham số cho hàm

Có 3 cách truyền tham số cho hàm:

- Truyền theo giá trị
- Truyền theo địa chỉ
- Truyền theo tham chiếu.

Trong phần này ta sẽ lần lượt nghiên cứu các cách truyền tham số này.

5.3.2.1. Truyền theo giá trị

Đây là cách truyền tham số đã được sử dụng trong ngôn ngữ C. Theo cách truyền này thì ***các giá trị*** của các tham số thực sẽ được sao chép vào ngăn xếp. Các giá trị này sau đó sẽ được truy nhập thông qua các tham số hình thức được sử dụng trong hàm. Như vậy hàm bị gọi chỉ làm việc với các bản sao của các tham số thực, vì vậy mọi sự thay đổi giá trị của các bản sao trong chương trình bị gọi sẽ không làm thay đổi giá trị gốc của các tham số thực được truyền cho hàm.

Ví dụ 5.15

Định nghĩa hàm hoán vị giá trị của hai số nguyên và viết chương trình sử dụng hàm này.

```
#include <iostream.h>
#include <conio.h>
void hoanvi ( int, int);// Khai báo nguyên mẫu hàm
void main () .
{
```

```
    clrscr( );
    int a =7, b =8;
    hoanvi(a, b);  // Gọi hàm hoanvi
    cout << " Giá trị của a và b sau khi hoán vị là : " <<  "a
                = "<< a<< ", b = " <<b;
    getch( );
}
// Định nghĩa hàm
void hoanvi(int x, int y)
{
    int temp;
    temp =x;
    x = y;
    y = temp;
}
```

Trong ví dụ, hàm *hoanvi* được gọi trong hàm *main* với các tham số thực a và *b*. Do lời gọi hàm được viết trước khi hàm được định nghĩa nên ta phải khai báo hàm nguyên mẫu trước khi gọi.

Kết quả của chương trình sẽ là:

Giá trị của a và b sau khi hoán vị là : a =7, b =8

Như vậy có thể thấy giá trị của các tham số thực *a* và *b* không thay đổi so với giá trị trước khi gọi hàm do việc truyền tham số được thực hiện theo giá trị.

Ví dụ 5.16

Xây dựng hàm để tính luỹ thừa của một số nguyên và viết chương trình sử dụng hàm.

```
#include <iostream.h>
#include <conio.h>
int power(int, int); // Khai báo hàm nguyên mẫu
void main( )
{
clrscr( );
int x;
x = power(2,3);
cout << x<< endl;
x = power(-3,3);
cout << x << endl;
x = power(4,-2);
cout << x << endl;
x = power(5,10);
cout << x << endl;
getch( );
}
// Hàm luỹ thừa
int power(int base, int exp)
{
    int result;
```

```
    for(result =1; exp >0; exp --)
        result *= base;
    return result;
}
```

Sau khi thực hiện chương trình cho kết quả sau:

8

-27

1

761

Kết quả này có nghĩa là:

2 luỹ thừa 3 bằng 8

-3 luỹ thừa 3 bằng – 27

4 luỹ thừa –2 bằng 1

5 luỹ thừa 10 bằng 761

Trong kết quả được đưa ra thì rõ ràng các kết quả thứ 3 và thứ 4 là sai. Kết quả sai do hàm không được xây dựng để tính luỹ thừa của số âm. Kết quả 4 sai do giá trị luỹ thừa tính được vượt quá giới hạn của số *int*.

Có thể mở rộng hàm power cho nhiều trường hợp và kiểu biến khác nhau. Hàm power dưới đây được xây dựng để tính cả luỹ thừa âm của một số kiểu double.

```
double power( double base, int exp)
{
```

```
    double result;
    if(exp > 0) // luỹ thừa >0
    {
        for(result =1; exp > 0; exp--)
            result *= base;
        return (result);
    }
    else if(base !=0) // luỹ thừa  <0
    {
        for(result =1; exp < 0; exp++)
            result /= base;
        return (result);
    }
    else
    {
        cout << "Không thực hiện luỹ thừa của 0"
        return 0;
    }
}
```

Bạn đọc tự viết chương trình sử dụng hàm này như một bài tập nhỏ.

5.3.2.2. Truyền theo địa chỉ

Đây cũng là một cách truyền tham số đã được sử dụng trong ngôn ngữ C chuẩn với tham số truyền cho hàm là địa chỉ của các

biến. Với cách truyền tham số này, địa chỉ của tham số thực sẽ được sao chép vào ngăn xếp và thông qua con trỏ chỉ đến các địa chỉ này (các tham số hình thức) hàm bị gọi có thể truy nhập đến các giá trị gốc của các tham số thực và làm thay đổi giá trị của các tham số này.

Hãy xét lại hàm hoán vị giá trị của hai số nguyên đã được trình bày ở trên bằng cách truyền tham số theo địa chỉ.

Ví dụ 5.17

```
#include <iostream.h>
#include <conio.h>
void hoanvi ( int *, int *); // Khai báo nguyên mẫu hàm
void main ()
{
    clrscr( );
    int a =7, b =8;
    hoanvi(&a, &b);  // Gọi hàm và truyền theo địa chỉ
    cout << "Giá trị của a và b sau khi hoán vị là : " <<  "a
            = "<< a<< ", b = " <<b;
    getch( );
}
// Định nghĩa hàm
void hoanvi(int *x, int *y)
{
    int temp;
    temp =*x;
```

```
    *x = *y;
    *y = temp;
}
```

Trong ví dụ, địa chỉ của các tham số thực *a* và *b* được sao vào ngăn xếp và đây cũng chính là trị của các tham số hình thức (các con trỏ x và y). Như vậy ta có:

x = &a, y = &b

hay nói cách khác *x* và *y* trỏ đến hai biến *a* và *b* và do vậy **x* và **y* chính là cách truy nhập gián tiếp đến hai biến *a* và *b*. Bằng cách này rõ ràng trong hàm bị gọi ta đã làm việc trực tiếp với các tham số thực và vì vậy mọi sự thay đổi của tham số hình thức sẽ phản ánh lên tham số thực.

Kết quả của chương trình sẽ là:

Giá trị của a và b sau khi hoán vị là : a =8, b =7

Ta hãy xét thêm một số ví dụ về cách truyền các tham số cho hàm bằng địa chỉ.

Ví dụ 5.18

Xây dựng hàm để in các giá trị chứa trong bộ nhớ từ một địa chỉ nào đó. Hàm sẽ kết thúc làm việc khi gõ một phím bất kỳ.

```
#include <iostream.h>
#include <conio.h>
void dum(unsigned int );// Nguyên mẫu hàm
void main ( )
{
    clrscr( );
```

```
    unsigned long int START;
    cout << "Nhập địa chỉ bắt đầu khảo sát:";
    cin >> START;
    dum(START);
}
//Hàm hiển thị nội dung bộ nhớ bắt đầu từ địa chỉ START
void dum (unsigned int start)
{
    char far *p;
    int i;
    p = (char far*)start; // Chuyển thành con trỏ chỉ đến kiểu char
    for(i = 0 ;; i++ , p++)
    {
        if(!i%16)
            cout<<endl;
        cout.width(5); // Độ rộng của vùng hiển thị
        cout << (int)*p;
        if(kbhit( ) )
            return;
    }
}
```

Chương trình đưa ra địa chỉ của vùng nhớ đầu tiên cần khảo sát thông qua biến ***START***. Bắt đầu từ địa chỉ này chương trình sẽ hiển thị nội dung của các ô ký ức kế tiếp nhau cho đến khi gõ một phím bất kỳ từ bàn phím. Trong chương trình có sử dụng từ khoá *far* với mục đích cho phép con trỏ có thể truy nhập

đến các ô ký ức không nằm trong cùng segment với đoạn mã chương trình. Cũng với mục đích này, địa chỉ đầu START được khai báo theo kiểu *unsigned long int*. Trên mỗi dòng của màn hình sẽ in nội dung của 16 byte liên tiếp trong bộ nhớ. Trong chương trình có sử dụng hàm *kbhit( )*, hàm này dùng để kiểm tra trên bàn phím có phím nào được gõ hay không. Khi một phím được gõ hàm này trả lại giá trị khác 0 và chương trình sẽ ngừng thực hiện.

Ví dụ 5.19

Giả sử ngăn xếp có cấu trúc lưu trữ là vectơ. Hãy xây dựng các hàm loại bỏ và bổ sung phần tử vào ngăn xếp và viết chương trình sử dụng các hàm này.

Như đã biết, ngăn xếp là một kiểu danh sách tuyến tính đặc biệt mà phép bổ sung và phép loại bỏ luôn luôn được thực hiện ở một đầu gọi là đỉnh (top).

Nếu sử dụng một vectơ S gồm n phần tử để làm cấu trúc lưu trữ của ngăn xếp và nếu gọi *Top* là địa chỉ của phần tử đỉnh ngăn xếp thì *Top* sẽ có giá trị biến đổi khi ngăn xếp hoạt động. Cụ thể, khi một phần tử mới được bổ sung vào ngăn xếp thì giá trị của *Top* sẽ tăng lên một đơn vị và khi một phần tử bị loại khỏi ngăn xếp thì giá trị của *Top* sẽ giảm một đơn vị.

Dưới đây ta sẽ xét một số phương pháp để giải quyết yêu cầu của bài toán này.

- Nếu ta qui ước địa chỉ của phần tử đỉnh ngăn xếp là tương đối (như chỉ số) so với địa chỉ đáy của nó thì khi ngăn rỗng *Top = -1* (đáy của ngăn xếp có địa chỉ tương đối là 0).

Chương trình được viết như sau:

```
#include<iostream.h>
#include <conio.h>
void Push(int *S, int n, int *Top, int X); // Bổ sung phần tử
int Pop(int *S, int *Top); // Loại phần tử
void main()
{
    int Top =-1; // Đỉnh ngăn xếp
    int i;
    int Size;
    int *S;
    cout << " Nhập kích thước của ngăn xếp: ";
    cin >> Size;
    S = new int [Size];
    Push(S,Size,&Top,10); // Đưa giá trị 10 vào ngăn xếp
    Push(S,Size,&Top,20); // Đưa giá trị 20 vào ngăn xếp
    cout << Pop(S,&Top) << endl; // In giá trị 20
    cout << Pop(S,&Top); // In giá trị 10
 }
void Push(int *S, int n, int *Top, int X)
{
    if(*Top > = n -1)
    {
        cout <<"Ngăn xếp đầy" <<endl;
        return;
    }
    *Top = *Top +1;
```

```
    S[*Top] = X;
  }
int Pop(int *, int *Top)
{
if(*Top ==-1)
  {
  cout<< "Ngăn xếp rỗng" <<endl;
  return(0);
  }
 *Top = *Top -1;
  return(S[*Top +1]);
  }
```

Trong chương trình có sử dụng hàm Push để bổ sung hai giá trị 10 và 20 vào ngăn xếp. Các giá trị này sau đó lại được in ra màn hình theo thứ tự ngược lại bằng hàm Pop. Để ý rằng nếu kích thước *Size* của ngăn xếp có giá trị nhỏ hơn 2 thì chương trình sẽ có kết quả sau:

Ngăn xếp đầy

10

Ngăn xếp rỗng

0

Thông báo "*Ngăn xếp đầy*" phát sinh do số phần tử được bổ sung nhiều hơn kích thước của ngăn xếp. Thông báo "*Ngăn xếp rỗng*" phát sinh do số giá trị bị loại bỏ nhiều hơn số phần tử có trong ngăn xếp. Giá trị 0 được in sau cùng chính là giá trị trả về của hàm *Pop* khi ngăn xếp rỗng. Vì hàm *Pop* có kiểu là *int* nên chương trình này có một nhược điểm là khi một phần tử của

ngăn xếp có giá trị trùng với giá trị trả lại của hàm khi ngăn xếp rỗng (trong trường hợp này là giá trị 0) thì hàm *Pop* sẽ phát sinh thông báo "Ngăn xếp rỗng" khi thực hiện loại bỏ phần tử này.

- Nếu địa chỉ của phần tử đỉnh ngăn xếp là tuyệt đối thì khi ngăn xếp rỗng địa chỉ này sẽ bằng địa chỉ của ngăn xếp trừ đi một đơn vị.

Chương trình được viết như sau:

```
#include <iostream.h>
#include<conio.h>
#include <stdlib.h>
void main()
{
    void Push(int *,int **,int ,int );
    int *Pop(int *, int **); // lay ra phan tu
    int Size;
    clrscr();
    cout <<" Kich thuoc ngan xep";
    cin >> Size;
    int *S = new int [Size];
    int *Top = S-1;
    clrscr();
    Push(S,&Top,Size,190);
    Push(S,&Top,Size,200);
    int *r;
    while ((r = Pop(S,&Top)) != NULL)
```

```
    cout<<*r<< endl;
}
void Push(int *p,int **T,int n,int X)
{
     if(*T >= p+n-1)
     {
          cout << "Ngăn xếp đầy"<<endl;
          return;
     }

      (*T)++;
     **T = X;
}

int *Pop(int *p,int **T)
{
      if(*T <= p-1)
          {
          cout <<"Ngăn xếp rỗng";
           return (NULL);
          }
  int *tr;
  tr = *T;
  (*T) --;
  return(tr);
}
```

Chương trình có một số điểm mới sau:

Để mọi sự thay đổi của Top trong các hàm *Push* và hàm *Pop* có thể phản ánh ra tham số thực trong hàm main thì *Top* phải được truyền theo kiểu con trỏ. Tuy vậy, do *Top* lại được khai báo theo kiểu con trỏ nên tham số hình thức tương ứng với tham số truyền này phải được khai báo theo kiểu int ***Top* (con trỏ của con trỏ) trong các hàm này. Khi đó qua **T* ta sẽ truy nhập đến địa chỉ của đỉnh ngăn xếp và qua ***T* ta sẽ truy nhập đến giá trị của phần tử này.

Giá trị trả lại của hàm *Pop* là một con trỏ kiểu *int*. Giá trị này chính là địa chỉ của phần tử đỉnh ngăn xếp và thông qua địa chỉ này ta có thể truy nhập đến giá trị của phần tử đỉnh. Việc định nghĩa giá trị trả lại của hàm theo kiểu con trỏ sẽ cho phép chọn giá trị trả lại hoàn toàn khác biệt của hàm *Pop* trong trường hợp ngăn xếp rỗng (địa chỉ NULL so với các trường hợp còn lại (địa chỉ cụ thể của ô nhớ).

5.3.2.3. Truyền theo tham chiếu

Đây là một cách truyền tham số mới được đưa vào ngôn ngữ C++. Trong trường hợp này tham số được truyền cũng là địa chỉ của biến và giống như khi truyền tham số theo con trỏ, mọi sự thay đổi giá trị của tham số được truyền trong hàm bị gọi sẽ làm thay đổi giá trị gốc của chính các tham số đó.

Hãy xét lại ví dụ hoán vị giá trị của hai số để mô tả cho việc việc sử dụng biến tham chiếu như tham số truyền của hàm.

Ví dụ 5.20

```
include <iostream.h>
#include <conio.h>
void hoanvi ( int &, int &);// Khai báo nguyên mẫu hàm
void main ()
{
    clrscr( );
    int a =7, b =8;
    hoanvi(a, b);  // Gọi hàm
    cout << "Giá trị của a và b sau khi hoán vị là : " <<  "a
            = "<< a<< ", b = " <<b;
    getch( );
}
// Định nghĩa hàm
void hoanvi(int & x, int& y) // Truyền theo kiểu tham chiếu
{
    // Truy nhập đến các biến tham chiếu
    int temp;
    temp =x;
    x = y;
    y = temp;
}
```

Qua ví dụ đơn giản trên có thể nhận thấy một số đặc điểm sau khi truyền tham số kiểu biến tham chiếu cho hàm:

- Cách gọi hàm (truyền tham số thực cho hàm) giống như trường hợp truyền theo giá trị.
- Trong định nghĩa của hàm gọi, các tham số hình thức truyền cho hàm được viết theo kiểu tham chiếu.
- Việc truy nhập đến các biến tham chiếu trong hàm bị gọi được thực hiện giống như trường hợp truyền theo giá trị.

Các ví dụ khác về cách sử dụng biến tham chiếu như là các tham số truyền cho hàm bạn đọc có thể tham khảo thêm ở các phần sau.

5.3.2.4. Truyền tham số ngầm định cho hàm

Một trong các điểm mạnh của ngôn ngữ C^{++} so với C là khả năng định nghĩa các tham số truyền mặc định cho hàm. Khi gọi một hàm nếu các tham số mặc định này không truyền cho hàm thì hàm sẽ sử dụng các giá trị đã được gán trong định nghĩa. Trong trường hợp các tham số này được truyền cho hàm thì hàm sẽ nhận các giá trị mới này. Khi khai báo tham số truyền mặc định cần lưu ý:

- Các giá trị mặc định cho các tham số chỉ được đưa ra trong khi khai báo hàm nguyên mẫu và không được lặp lại trong định nghĩa hàm. Chương trình biên dịch sẽ sử dụng các thông tin trong cách khai báo hàm nguyên mẫu để tạo một lệnh gọi.
- Một hàm có thể có nhiều giá trị mặc định. Những tham số mặc định phải được khai báo sau tất cả các tham số khác.
- Khi gọi một hàm có nhiều giá trị mặc định ta chỉ có thể bỏ bớt các đối số theo thứ tự từ phải sang trái và phải bỏ liên tiếp nhau.

Các ví dụ dưới đây sẽ mô tả cho việc truyền tham số mặc định cho hàm.

```
void def_par(int i=100); // Khai báo tham số mặc định
void main()
{
    def_par() // Gọi def_par với tham số i =100;
    def_par(2000) // Gọi der_par với tham số i=2000
}
// Định nghĩa def_par()
voide der_par(int i)
{
    cout <<"Giá trị của i "<< i;
}
```

Nếu ta khai báo một hàm nguyên mẫu theo cách sau:

```
void def_par(int a=1,int b,int c=3,int d =9)
```

thì trình biên dịch sẽ báo lỗi vì các tham số mặc định được khai báo trước các tham số khác (a khai báo trước b). Cách khai báo đúng của hàm nguyên mẫu này là:

```
void def_par(int a,int b=1,int c=3,int d =9)
```

Các cách gọi hàm def_par sau là sai:

```
der_par(3,10,12); //Sai vì các đối số bị bỏ không liên tiếp nhau.
```

Còn các cách gọi sau là đúng:

```
def_par(2,4,5,6); //Đủ tham số
der_par(12,3); //Nhận các giá trị ngầm định của c và d
```

Khi truyền khai báo tham số mặc định cho hàm, một điều

cần lưu ý là tham số này có thể là một hằng, một biến toàn cục hoặc thậm chí là một hàm đã được định nghĩa.

5.3.2.5. Truyền tham số kiểu const cho hàm

Khi một biến được truyền cho hàm với từ khóa *const* thì hàm bị gọi sẽ không được phép thay đổi giá trị của biến này. Do đặc điểm này nếu một biến được truyền cho hàm theo kiểu giá trị thì việc sử dụng từ khóa *const* là không cần thiết, vì hàm bị gọi trong trường hợp này cũng không thể thay đổi giá trị gốc của biến. Chẳng hạn việc sử dụng *const* trong khi khai báo *void some_function(const int)* có thể xem như vô ích. Việc truyền tham số tới từ khóa này chỉ có tác dụng khi ta truyền các biến cho hàm theo kiểu con trỏ hoặc tham chiếu. Trong các trường hợp này ta muốn sử dụng các điểm mạnh của việc truyền biến bằng con trỏ hoặc tham chiếu nhưng lại không muốn giá trị của nó có thể bị hàm bị gọi làm thay đổi. Hãy xem xét ví dụ sau:

```
void New Value(const char*); // Khai báo NewValue
void main()
    {
    char *p="12345";
    NewValue(p);
    }
// Định nghĩa NewValue
void NewValue(const char* p)
    {
    cout<< "p=%s" << p;
    p ="abcd";//sai }
```

5.3.3 Hàm đệ qui

Đệ qui là một nguyên tắc làm việc đem lại hiệu quả cao khi giải các bài toán phức tạp trong nhiều lĩnh vực khác nhau. Nói một cách chung nhất thì một đối tượng được gọi là đệ qui nếu nó chứa hoặc có thể xác định thông qua chính bản thân mình. Ngôn ngữ C++ có chứa hai cấu trúc qua đó có thể cho phép thực hiện các thuật toán đệ qui: hàm đệ qui và cấu trúc đệ qui. Cấu trúc đệ qui sẽ được nghiên cứu trong chương VI.

Một hàm đệ qui trong C++ có thể được định nghĩa nếu nó có chứa lời gọi đến chính bản thân mình. Chẳng hạn hàm *fun* dưới đây là một hàm đệ qui:

```
int fun(int a, int b)
{
    int i;
    .....
    i = fun(a-1, b-1);
    ......
}
```

Việc sử dụng hàm đệ qui không đòi hỏi thêm ở người lập trình một kiến thức hoặc một nỗ lực gì mới. Điều cốt yếu để xây dựng được hàm kiểu đệ qui là phải định nghĩa được nhiệm vụ tương ứng dưới dạng đệ qui. Để có thể sử dụng hàm đệ qui một cách dễ dàng hơn, dưới đây sẽ trình bày cách thực hiện hàm đệ qui trong trình biên dịch C và C++. Trong các trình biên dịch này, hàm đệ qui bao gồm các phép gọi liên tiếp đến bản thân nó. Theo một nguyên tắc chung nhất, khi gọi hàm, trình biên dịch sẽ thực hiện các bước sau:

- Lưu lại trong ngăn xếp địa chỉ sẽ quay lại thực hiện khi kết thúc hàm bị gọi (*gọi tắt là địa chỉ trở về*).
- Truyền các tham số của hàm gọi vào ngăn xếp.
- Truyền các tham số cục bộ của hàm bị gọi vào ngăn xếp.

Ở đây, ta sẽ không đi sâu vào cơ cấu tryền các thông tin này vào ngăn xếp. Phần tiếp theo sẽ trình bày một số ví dụ về hàm đệ qui.

Ví dụ 5.21

Xây dựng hàm đệ qui tính giai thừa của số n.

Định nghĩa theo kiểu đệ qui, giai thừa của số *n* được xác định như sau:

0! =1;

Nếu n ≥ 1 thì n! = n*(n-1)!.

Hàm đệ qui tính n! giai thừa được viết như sau:

```
long int fact(int n)
{
    if(n ==0)
        return 1;
    return(n*fact(n-1));
}
```

Bạn đọc có thể sử dụng hàm này để viết chương trình tính giai thừa của một số nguyên nào đó.

Ví dụ trên sử dụng phương pháp đệ qui trực tiếp, ngoài phương pháp này trong C còn cho phép sử dụng phép nội suy

gián tiếp. Phương pháp này có thể được minh hoạ qua ví dụ sau:

```
int fun(char a, char b)
{
    ..................
    posr(x, y, z);
    ..................
}

void posr(int a, int b, int c)
{
    ..................
    fun(a,b);
    ..................
}
```

Ví dụ 5.22

Từ bàn phím hãy đưa một xâu ký tự (đến 39 ký tự) vào mảng str. Hãy in xâu ký tự lên màn hình theo thứ tự ngược lại với khi nhập.

```
#include <iostream.h>
void prec(char*);
void main( )
{
    char str[40];
    cout << "Hãy nhập chuỗi:";
    cin >>str;
```

```
    prec(str);
}
// Hàm đệ qui xử lý chuỗi
void prec(char* s)
{
    if(*s != NULL)
    {
        prec(++s);
        cout << *(--s);
    }
}
```

5.3.4. Hàm main

Cũng như các chương trình viết bằng ngôn ngữ C, các chương trình viết bằng C++ cũng cần một hàm chính gọi là hàm *main( )*. Khác với hàm *main* trong C không được định nghĩa kiểu, hàm *main* trong C++ sẽ ngầm định nhận kiểu *int*. Khi chương trình bắt đầu được thực hiện, hàm *main* được gọi bởi một chương trình khởi động đặc biệt (*start up code*) có sẵn trong C++. "Start up code" trong C++ có 4 cách gọi. Cách gọi nào được sử dụng khi khởi động chương trình sẽ phụ thuộc vào tùy chọn trong Options/Application:

Dos standard.
Dos overlays.
Windows Application.
Windows Dll.

Các File "Start up code" tương ứng với các tùy chọn này là *C0.asm, C0.asm, C0w.asm, Cod.asm*. Như vậy có thể nhìn thấy

rằng cả 2 loại chương trình Dos Standard và Dos Overlays đều sử dụng chung Start up code có tên là *C0.asm*.

Giá trị được trả lại bởi hàm *main()* sẽ được Dos sử dụng để kiểm tra lỗi khi thực hiện chương trình. Nếu giá trị này bằng 0 thì hàm thực hiện không có lỗi, trong khi các giá trị khác 0 đều được C++ sử dụng trong trường hợp có lỗi khi thực hiện chương trình. Hầu hết các chương trình ứng dụng đều sử dụng các giá trị nhỏ hơn 4 hoặc 5, các giá trị lớn hơn 5 thường được dùng cho các lỗi phức tạp hơn.

Hàm chính *main* có sử dụng một số tham số truyền ngầm định. Sự hiện diện của các tham số này nhằm tạo điều kiện thuận lợi khi thiết lập giao diện với chương trình, nhất là trong trường hợp cần phải truyền thông tin cho chương trình trước khi nó được thực hiện,

Trong tất cả các ví dụ được sử dụng cho đến nay, các tham số này không được sử dụng. Mặc dù hàm *main* có thể sử dụng hoặc không sử dụng các tham số này, mỗi khi hàm được gọi trong ngăn xếp luôn được cấp phát bộ nhớ dành riêng cho các tham số đó. Một hàm *main* có sử dụng tham số truyền được khai báo như sau:

```
int main(int arg,char**argv, char** env)
    {
    .....
    }
```

Các tham số này được truyền cho hàm *main* từ Start up code. Đến lượt mình Start up code lại nhận các giá trị này từ hệ điều hành DOS. Trong các tham số này thì:

argc: Số lượng các tham số được chỉ ra trên dòng lệnh.

argv: Mảng các con trỏ của các tham số. Đây chính là các hằng kiểu chuỗi được người sử dụng viết trên dòng lệnh khi thực hiện chương trình. Các hằng này phải được viết trên dòng lệnh sau tên của chương trình và phải được viết cách nhau.

env: Mảng các con trỏ có chứa các thiết lập môi trường của Dos.

Ví dụ sau sẽ mô tả cho việc sử dụng các tham số được truyền cho hàm số *main*. Giả sử *example.exe* thuộc như mục *c:\borlandc* được thực hiện trên dòng lệnh với các tham số *parm1, parm2, parm3, parm4*. Tức là trên dòng lệnh có gõ lệnh sau:

example parm1 parm2 parm3 parm4 ↵

trong trường hợp này *main* sẽ được gọi với các tham số:

argc=5;

trong mảng argv có chứa các giá trị sau:

argv[0]: "C:\borlandc\example.exe" nếu sử dụng Dos 3.x

argv[0]: " " - Nếu sử dụng các phiên bản của Dos sau 3.x.

argv[1]: "Parm1"

argv[2]: "Parm2"

argv[3]: "Parm3"

argv[4]: "Parm4"

argv[5]: 0

mảng *env* có chứa các giá trị:

env[0]: "c:\command.com"

env[1]: "prompt = pg"

env[2]: "path=c:\dos;c:\borlandc;c:\windows"

env[3]: 0

Dưới đây là một ví dụ về cách sử dụng các tham số truyền cho hàm main.

Ví dụ 5.21

Hãy đọc dữ liệu từ file văn bản input, chuyển các ký tự thành chữ thường và sau đó ghi vào file output.

Trong chương trình, các file input và output sẽ được sử dụng như tham số truyền của hàm main. Giả sử chương trình sau khi liên kết có tên là convert.exe, khi đó chương trình sẽ được thực hiện khi gõ convert intput output trên dòng lệnh.

```
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>
void main(int argc, char *argv[ ])
{
    if(argc !=3)
    {
    cout <<"Dòng lệnh chưa đủ các tham số";
    exit(0);
    }
    // Mở file input.txt để đọc
    ifstream InputFile(argv[1]);
    // Mở file output để ghi
    ofstream OutputFile(argv[2]);
```

```
    // Các thao tác kiểm tra khi mở file
    if(!InputFile)
    {
        cout <<"Không mở được file input.txt" <<endl;
        return;
    }
    if(!OutputFile)
    {
        cout <<"Không mở được file output.txt" <<endl;
        return;
    }
    // Sao chép nội dung của file sang bộ đệm

    while(InputFile && OutputFile)
    {
        char buffer[80];
        InputFile.getline(buffer, sizeof(buffer));
        for(int i = 0; buffer[i] != '\0'; i++)
            buffer[i] = toupper(buffer[i]);
        OutputFile << buffer <<"\n";
    }

}
```

Chương VI

CÁC KIỂU DỮ LIỆU PHỨC TẠP

Chương này sẽ đề cập đến một số kiểu dữ liệu phức tạp như *enum, struct, union, typedef,* v.v.

6.1. KIỂU DỮ LIỆU TYPEDEF

Ngôn ngữ C và C++ đưa ra một cấu trúc rất thuận lợi cho phép người sử dụng định nghĩa một tên riêng đối với các biến thuộc các kiểu dữ liệu khác nhau. Việc định nghĩa này được thực hiện qua cú pháp:

typedef Kiểu_cũ Kiểu_mới

trong đó:

Kiểu_cũ là các kiểu dữ liệu chuẩn trong ngôn ngữ như int, float, char v.v. Ngoài các kiểu dữ liệu này, *Kiểu _cũ* có thể là tên của kiểu đã được định nghĩa trước đó qua *typedef*.

Kiểu_mới là tên mới được dùng để thay thế cho tên các kiểu dữ liệu được chỉ ra bởi tên cũ. Tên mới phải được đặt tuân theo các nguyên tắc như khi đặt tên cho biến (thông thường sử dụng chữ in hoa để tránh nhầm lẫn). Tên mới này ngoài ra còn không được trùng với tên của các biến có cùng phạm vi hoạt động.

Sau khi đã được định nghĩa, tên mới hoàn toàn có thể được sử dụng để thay thế cho kiểu dữ liệu được chỉ ra bởi *Kiểu_cũ*. Ví

dụ, sau khi định nghĩa:

typedef int PRIM

tên mới *PRIM* có thể được sử dụng thay cho *int*. Tức là khai báo:

PRIM x, y;

sẽ tương đương với:

int x, y;

typedef có thể được sử dụng với các kiểu dữ liệu phức tạp như mảng, con trỏ, cấu trúc. Các ví dụ dưới đây sẽ minh hoạ cho cách định nghĩa này:

- typedef float EXP[100]

Sau định nghĩa này EXP có thể dùng để khai báo các mảng thuộc kiểu *float* với kích thước là 100. Chẳng hạn khai báo *EXP a, b;* sẽ hoàn toàn tương đương với *float a[100], b[100];*

- typedef char*WORD

Sau định nghĩa này, *WORD* có thể dùng để khai báo cho các con trỏ thuộc kiểu *char*. Như vậy khai báo *char *p, *q*; có thể được thay thế bởi *WORD p, q*;.

- typedef char* FUN( );

Sau định nghĩa này, *FUN* có thể dùng để khai báo cho các hàm với giá trị trả lại là con trỏ kiểu char. Chẳng hạn khai báo *char *arc ( )* có thể được thay thế bởi *FUN arc*;

Các biến được khai báo qua kiểu dữ liệu *typedef* được sử dụng trong chương trình giống như các biến thuộc kiểu tương ứng mà chúng biểu diễn. Tuy vậy cũng cần chú ý rằng, *typedef* không định nghĩa một kiểu dữ liệu mới mà chỉ xác định tên mới cho các kiểu dữ liệu đã tồn tại.

Việc sử dụng *typedef* có những ưu điểm sau:

- Chương trình được viết rõ ràng hơn do có thể sử dụng tên mới của kiểu dữ liệu nhằm phản ánh ý nghĩa của chúng.
- Cho phép khai báo các biến ở dạng ngắn gọn hơn.

6.2. DỮ LIỆU THUỘC KIỂU ENUM

Đây là một kiểu dữ liệu mà các biến được khai báo từ chúng chỉ có thể nhận giá trị là các hằng số nguyên trong tập hợp các số nguyên đã được xác định trước. Các biến thuộc kiểu dữ liệu này đặc biệt phù hợp với các trường hợp được đặc trưng bởi nhiều lựa chọn khác nhau. Một ví dụ tiêu biểu đó là các biến thuộc kiểu Logic với 2 giá trị có thể nhận được là TRUE (giá trị là 1) và FALSE (giá trị bằng 0). Trong các trình biên dịch của C và C++, kiểu dữ liệu này sẽ được khai báo dưới dạng sau:

enum Tên_kiểu {danh sách tên các giá trị};

trong đó:

enum là từ khoá bắt buộc dùng để định nghĩa kiểu dữ liệu.

Tên_kiểu: Tên kiểu dữ liệu với các giá trị có thể đếm được. Tên này được đặt theo nguyên tắc đặt tên cho biến.

Danh sách tên các giá trị: Tên các giá trị mà các biến thuộc kiểu dữ liệu này có thể nhận được, các tên này phải được viết cách nhau bởi dấu phẩy (,).

Ví dụ:

```
enum Boolean {FALSE, TRUE};
```

Khi làm việc trình biên dịch sẽ gắn tên trong danh sách với

các giá trị là hằng số nguyên theo thứ tự tăng dần. Phần tử đầu tiên trong danh sách luôn có giá trị ngầm định bằng 0. Như vậy, trong ví dụ trên FALSE sẽ có giá trị bằng 0 và TRUE có giá trị bằng 1. Các giá trị trong danh sách có thể thay đổi bằng cách gán trực tiếp các giá trị cần sử dụng cho các tên trong danh sách. Chẳng hạn bằng cách định nghĩa:

enum Counter{Start, First, Second, Tenth =10, Eleventh};

Start, First, Second sẽ lần lượt có các giá trị 0, 1, 2, thay vì giá trị ngầm định là 3, *Tenth* sẽ có giá trị mới là 10 và tiếp đó giá trị của *Eleventh* sẽ là 11.

Sau khi định nghĩa một kiểu dữ liệu mới qua từ khoá enum, ta có thể khai báo các biến thuộc kiểu dữ liệu này. Việc khai báo biến cổ thể được thực hiện đồng thời với việc định nghĩa kiểu dữ liệu:

enum Boolean {FALSE, TRUE}x,y;

hoặc sau khi định nghĩa kiểu dữ liệu bằng cách:

enum Boolean x,y;

Không phụ thuộc vào cách khai báo, các biến thuộc kiểu dữ liệu *enum* chỉ có thể nhận được các giá trị đã được xác định khi định nghĩa kiểu dữ liệu này.

6.3. DỮ LIỆU CẤU TRÚC

Thông thường, khi thiết kế một chương trình, tất cả các dữ liệu có quan hệ logic với nhau thường được nhóm lại trong một thực thể. Các dữ liệu trong nhóm này được phân bố liên tục trong bộ nhớ và điều đó cho phép đơn giản hoá việc xử lý các loại dữ liệu này. Một ví dụ tiêu biểu cho việc nhóm các dữ liệu đó là

việc sử dụng mảng. Sử dụng mảng cho phép truy nhập nhanh và dễ dàng đến các phần tử của mảng thông qua tên mảng và chỉ số. Tuy vậy mảng vẫn có nhược điểm của nó: tất cả các phần tử của mảng bắt buộc phải là các dữ liệu thuộc cùng một kiểu.

Trên thực tế, không ít trường hợp ta cần nhóm các dữ liệu với nhiều kiểu khác nhau nhưng lại có quan hệ logic với nhau. Chẳng hạn, thông tin về con người như tên, địa chỉ, tuổi tác, giới tính, v.v. Để thực hiện được mục đích này, ngôn ngữ C++ đưa ra một kiểu dữ liệu đặc biệt gọi là *cấu trúc*. Việc sử dụng cấu trúc cho phép người lập trình nhóm các dữ liệu đơn giản với các kiểu khác nhau vào cùng một nhóm với một tên duy nhất. Các dữ liệu này thường được gọi là các *phần tử* hay các *trường* của cấu trúc.

Có nhiều cách khai báo một cấu trúc, trong đó cách hay sử dụng nhất có dạng:

```
struct Tên_cấu _trúc
{
    Khai báo các phần tử của cấu trúc;
};
```

Trong cách khai báo này, tên cấu trúc phải được đặt tuân theo các nguyên tắc đặt tên cho biến. Dữ liệu được khai báo trong thân của cấu trúc được gọi là các phần tử của cấu trúc. Các phần tử này có thể là một kiểu dữ liệu bất kỳ được dùng trong C++.

Sử dụng cấu trúc ta có thể định nghĩa một kiểu dữ liệu chứa các thông tin về con người như sau:

```
struct nhan_su
{
```

```
    char hoten[20];
    int tuoi;
    float can_nang;
};
```

Hoặc một cấu trúc chứa các thông tin về thời gian:

```
struct time
{
    int hour;
    int minute;
    float second;
};
```

Cần phải chú ý rằng, với việc định nghĩa một cấu trúc ta chưa xác định một biến cụ thể nào mà chỉ xác định một bảng mẫu, bảng mẫu này sau đó có thể được sử dụng cho nhiều biến khác nhau. Chính vì nguyên nhân này mà khi định nghĩa cấu trúc, trình biên dịch sẽ không cung cấp bộ nhớ. Bộ nhớ chỉ được cấp phát khi một biến thuộc kiểu cấu trúc được khai báo.

Để khai báo một biến thuộc kiểu dữ liệu cấu trúc nào đó, có thể sử dụng một trong các cách sau:

- Khai báo các biến ngay trong khi định nghĩa cấu trúc. Ví dụ:

```
struct time
{
    int hour;
    int minute;
```

```
        float second;
    }u, v, t;
```

Theo cách khai báo này, *u, v, t* là các biến thuộc kiểu cấu trúc *time*.

Khai báo sau khi định nghĩa cấu trúc. Ví dụ struct time u, v, t.

Khi các biến được khai báo cùng với định nghĩa cấu trúc thì tên của cấu trúc có thể được bỏ qua, ví dụ như trong trường hợp sau:

```
struct
{
    int hour;
    int minute;
    float second;
}u, v, t;
```

So với cách khai báo trước thì tên của cấu trúc *time* đã bị loại bỏ. Tuy vậy, cũng cần lưu ý rằng đối với các kiểu cấu trúc loại này, khi cần khai báo một biến nào đó thuộc kiểu cấu trúc ta phải định nghĩa lại cấu trúc. Điều này sẽ làm cho chương trình trở thành phức tạp và dài dòng không cần thiết.

6.3.1. Các phép toán trên cấu trúc

Các phép xử lý trên cấu trúc thường hay được sử dụng bao gồm:

- Lấy địa chỉ của cấu trúc.
- Truy nhập đến các phần tử của cấu trúc.
- Khởi tạo cấu trúc.

6.3.1.1. Lấy địa chỉ cấu trúc

Khi một biến được khai báo theo kiểu cấu trúc thì địa chỉ của biến có thể được xác định như mọi biến khác bằng phép toán &. Để minh hoạ cho việc sử dụng phép toán & trên cấu trúc ta hãy xét lại cấu trúc *time* làm ví dụ:

```
struct time
{
    int hour;
    int minute;
    float second;
}u, v, t;
struct time *p;
.....
p = &u;
```

Trước khi gán địa chỉ biến *u* thuộc kiểu cấu trúc *time* cho con trỏ *p*, *p* phải được khai báo theo kiểu con trỏ chỉ đến cấu trúc thuộc kiểu *time*.

6.3.1.2. Truy nhập đến các phần tử của cấu trúc.

Các phần tử của cấu trúc có thể được truy nhập bằng hai cách:

* Sử dụng tác tử • (dấu chấm) qua cú pháp:

Biến_cấu_trúc• Tên phần tử

Ví dụ:

```
struct time
{
```

```
        int hour;
        int minute;
        float second;
    }u, v, t;
    int k, n;
.....
// truy nhập đến hour và minute của biến u;
    k = u.hour;
    t = u.minute;
```

* Truy nhập thông qua con trỏ bằng tác tử -> Con_trỏ_cấu_trúc -> Tên phần tử;

Ví dụ:

```
struct time
{
    int hour;
    int minute;
    float second;
};
struct time *p;
struct time x;
p = &x;
int k, n;
k = u->hour;
t = u->minute;
```

6.3.1.3. Khởi tạo các biến cấu trúc

Các biến thuộc kiểu cấu trúc có thể được khởi tạo như tất cả các biến thuộc các kiểu dữ liệu khác. Chẳng hạn có thể sử dụng

phép khởi tạo sau:

```
static struct time u = {5, 4, 25.36};
```

Trong phép khởi tạo này, cùng với việc khai báo biến *u* thuộc kiểu cấu trúc *time*, các phần tử của biến cũng được khởi tạo giá trị. Cụ thể:

```
u.hour = 5;
u.minute = 4;
u.second = 25.36.
```

Phần dưới đây sẽ trình bày một số ví dụ về kiểu dữ liệu cấu trúc.

Ví dụ 6.1. *Xây dựng chương trình xử lý ngăn xếp*

Ví dụ về ngăn xếp đã được trình bày trong chương V. Ví dụ này sẽ sử dụng dữ liệu kiểu cấu trúc để giải quyết yêu cầu của bài toán.

Ở đây cấu trúc dữ liệu của ngăn xếp sẽ được định nghĩa với ba trường:

- Trường top: Số nguyên chỉ đỉnh ngăn xếp.

- Trường size: Kích thước ngăn xếp, sử dụng kích thước ngầm định gồm 20 phần tử.

- Trường node: Mảng một chiều, mỗi phần tử của mảng là một nút của ngăn xếp

```
struct stack
{

    int top;
    int size;
```

```
    int* nodes;
};
```

- Các hàm xử lý trên ngăn xếp bao gồm:
- Khởi tạo ngăn xếp: InitStack
- Kiểm tra ngăn xếp rỗng: Empty;
- Bổ sung phần tử vào ngăn xếp: Push.
- Loại bỏ phần tử khỏi ngăn xếp: Pop.

Dưới đây sẽ trình bày lần lượt các hàm vừa nêu.

```
// Hàm khởi tạo ngăn xếp
void InjtStack(struct stack & p, int Size)
{
    p.size = Size;
    p.top = -1;
    p.node = new int[size];
}
// Kiểm tra ngăn xếp rỗng
int Empty ( struct stack & p)
{
    return (p.top ==-1);
}
// Bổ sung phần tử vào ngăn xếp – xử lý ngăn xếp đầy
void push (struct stack & p, int x)
{
    if(p.top == p.size –1)
    {
```

```
        cout <<" Ngăn xếp đầy"<<endl;
        return;
        }
        p.nodes[++(p.top)] =x;
}
// Loại bỏ phần tử khỏi ngăn xếp
int pop (struct stack & p)
{
        if(Empty(p))
                cout<<"Ngăn xếp rỗng"<<endl;
        else
                return(p.nodes[p.top--]);
}
```

Hàm *main* được viết như sau:

```
#include <iostream.h>
void push (struct stack & p, int x);
int pop (struct stack & p);
void InitStack(struct stack & p, int Size = 20);
int Empty ( struct stack & p);
void main ( )
{
        struct stack S;
        InitStack(S);
        // Bổ sung phần tử vào ngăn xếp
        push(S,10);
        push(S,20);
        // Loại bỏ phần tử khỏi ngăn xếp
```

```
    while( !Empty(S))
cout<< pop(S) <<endl;
}
```

Ví dụ 6.2

Sử dụng cấu trúc dữ liệu kiểu cấu trúc, viết chương trình chuyển cơ số.

Chương trình dưới đây sẽ cho phép đổi một số ở hệ thập phân sang một cơ số khác (cơ số 2, 3, 4, ...). Thông tin đầu vào của chương trình gồm số thập phân và cơ số cần đổi. Việc chuyển đổi được thực hiện bằng cách lần lượt chia số thập phân cho cơ số và đẩy số dư vào ngăn xếp, lấy thương của phép chia trên tiếp tục chia cho cơ số và lại đẩy số dư vào ngăn xếp v.v. Quá trình dừng lại cho đến khi thương số bằng 0. Số cần chuyển đổi sẽ được hiển thị bằng cách lần lượt in các phần tử của ngăn xếp ra màn hình.

```
#include <iostream.h>
void push (struct stack & p, int x);
int pop (struct stack & p);
void InitStack(struct stack & p, int Size = 20);
int Empty (struct stack & p);
struct stack
{

    int top;
    int size;
    int* nodes;
};
```

```
void main ( )
{
    struct stack S;
    InitStack(S);
    // Bổ sung phần tử vào ngăn xếp
    cout << "Nhập số thập phân cần đổi: "<<endl;
    int so;
    cin >> so;
    cout << "Nhập cơ số cần đổi: "<<endl;
    int coso;
    cin >> coso;
    int sodu;
    while(so !=0)
    {
        sodu = so%coso;
        push(S, sodu);
        so = so/coso;
    }
    cout << "Số đã đổi là" <<endl;
    while (!Empty(S))
        cout << pop(S);
    }
// Hàm khởi tạo ngăn xếp
void InitStack(struct stack & p, int Size)
{
```

```
    p.size = Size;
    p.top = -1;
}
// Kiểm tra ngăn xếp rỗng
int Empty (struct stack & p)
{
    return (p.top ==-1);
}
// Bổ sung phần tử vào ngăn xếp – xử lý ngăn xếp đầy
void push (struct stack & p, int x)
{
    if(p.top == p.size –1)
    {
    cout <<" Ngăn xếp đầy"<<endl;
    return;
    }
    p.nodes[++(p.top)] =x;
}
// Loại bỏ phần tử khỏi ngăn xếp
int pop (struct stack & p)
{
    if(Empty(p))
        cout<<"Ngăn xếp rỗng"<<endl;
    else
        return(p.nodes[p.top--]);
}
```

6.3.1.4. Sử dụng cấu trúc theo kiểu đệ qui

Ngoài các kiểu dữ liệu bình thường, các thành phần của cấu trúc cũng có thể được khai báo theo kiểu con trỏ. Hãy xét việc định nghĩa một cấu trúc sau:

```
struct point
{
    int k;
    int *p;
    char *q[5];
};
```

và biến *pr* được khai báo theo kiểu *point*.

Trong cấu trúc này, việc truy nhập đến các thành phần thuộc kiểu con trỏ của *pr* hoàn toàn tuân theo các nguyên tắc chung- *pr.p, pr.q[1].* Đây chính là các con trỏ chỉ đến các thành phần của cấu trúc và do đó chúng có thể được sử dụng trong tất cả các phép toán có sử dụng địa chỉ. Chẳng hạn, biến mà *p* chỉ đến trong trường hợp này sẽ là **pr.p*. Lưu ý là ở đây ta không cần phải sử dụng dạng **(pr.p)* vì phép toán • có mức ưu tiên cao hơn phép toán *.

Kiểu dữ liệu của con trỏ được sử dụng trong thành phần của cấu trúc có thể là bất kỳ kể cả kiểu cấu trúc. Mặc dù ngôn ngữ C không cho phép thành phần của một cấu trúc lại là cấu trúc cùng kiểu với nó nhưng nếu thành phần này được khai báo theo kiểu con trỏ thì điều này lại hoàn toàn được phép. Trong những trường hợp này, cấu trúc đã được định nghĩa theo kiểu đệ qui.

Ví dụ về cách định nghĩa đệ qui của cấu trúc:

```
struct topol
{
    int k;
    struct topol *next;
};
```

Việc sử dụng tính đệ qui đối với cấu trúc cho phép thiết lập việc xử lý dữ liệu theo kiểu danh sách tuyến tính và cấu trúc cây. Danh sách tuyến tính là tập các dữ liệu, trong đó mỗi phần tử của dữ liệu trỏ đến dữ liệu tiếp theo. Ưu điểm của danh sách kiểu tuyến tính được thể hiện rất rõ trong những bài toán tìm kiếm, sắp xếp. Trong các bài toán thuộc kiểu này thông thường cần giải quyết các vấn đề sau:

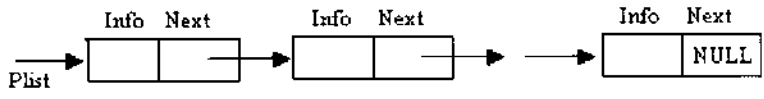
- Tập hợp các dữ liệu phải được sắp xếp theo một kiểu nhất định.

- Dữ liệu mới được bổ sung vào danh sách phải được đặt tại vị trí thích hợp phù hợp với yêu cầu.

Ví dụ sau sẽ minh hoạ cho việc sử dụng tính đệ qui của cấu trúc trong việc tổ chức dữ liệu theo kiểu danh sách tuyến tính.

Ví dụ 6.3. *Xây dựng chương trình quản lý danh sách sinh viên*

Trong chương trình, danh sách sinh viên được quản lý theo kiểu liên kết đơn. Về thực chất, danh sách liên kết đơn bao gồm nhiều nút và các nút là có thứ tự. Mỗi nút có trường Info chứa nội dung thật sự của nó và trường Next chứa con trỏ chỉ đến nút tiếp theo trong danh sách liên kết. Thứ tự của các nút được thể hiện qua trường liên kết Next: con trỏ đầu (được gọi là Plist) chỉ đến nút đầu tiên trong danh sách, nút đầu chỉ đến nút thứ 2, ..., nút cuối cùng của danh sách là nút có trường Next với giá trị NULL.



Đối với danh sách liên kết đơn ta có:

- Khi khởi động con trỏ đầu danh sách được gán giá trị NULL (Chưa có phần tử).
- Khi danh sách rỗng, không thể thực hiện thao tác loại bỏ phần tử khỏi danh sách.
- Không có trường hợp danh sách đầy (điều này hoàn toàn phụ thuộc vào bộ nhớ tự do của máy tính).
- Việc truy nhập đến các phần tử của danh sách phải được thực hiện thông qua con trỏ Plist chỉ đến phần tử đầu, các phần tử tiếp theo sẽ được truy nhập thông qua trường Next của phần tử đứng trước nó.

Như vậy, mỗi nút của danh sách liên kết đơn có thể được xem như một biến của cấu trúc với hai thành phần dữ liệu Info và Next:

```
struct Node
{
    Sinhvien Info;
    Node * Next;
};
```

Trong đó Sinhvien là cấu trúc chứa các thông tin về các sinh viên. Ở đây để đơn giản, ta qui ước các thông tin cần quản lý của mỗi sinh viên bao gồm mã và tên:

```
struct Sinhvien
{
```

```
    int masv;
    char ten[15];
};
```

Để đơn giản cho việc quản lý danh sách liên kết, trong chương trình sẽ sử dụng dữ liệu kiểu con trỏ chỉ đến nút thông qua việc khai báo:

```
typedef struct Node*NODEPTR;
```

khi đó thay cho việc khai báo *struct Node *Plist* ta có thể viết ngắn gọn hơn *NODEPTR Plist*.

Các phép xử lý trên danh sách liên kết đơn bao gồm:

- Khởi tạo danh sách liên kết:

```
void Initialize(NODEPTR *plist)
{
    *plist = NULL;
}
```

Hàm này có tác dụng khởi tạo con trỏ Plist với giá trị NULL. Để có thể thực hiện việc thay đổi giá trị của tham số thực, tham số này phải được truyền theo kiểu con trỏ.

- Kiểm tra danh sách rỗng:

```
int Empty( NODEPTR *plist)
{
    return(*plist ==NULL) ?1:0);
}
```

- Cấp phát một biến động làm một nút cho danh sách

```
NODEPTR GetNode( )
{
```

```
        NODEPTR p;
        p = new Node[1];
        return p;
    }
```

- Giải phóng biến động sử dụng cho nút đã cấp phát trước đó

```
void FreeNode(NODEPTR p)
{
    delete p;
}
```

- Xác định con trỏ của nút i trong danh sách liên kết

```
NODEPTR NodePointer(NODEPTR *plist, int i)
{
    NODEPTRp;
    int position;
    p = *plist;
    position =0;
    while(p !=NULL && position <i)
    {
        p = p->Next;
        position ++;
    }
return(p);
}
```

- Xác định vị trí của nút p trong danh sách liên kết

```
int Position( NODEPTR *plist, NODEPTR p)
```

```
{
    int pos;
    NODEPTR q;
    q = *plist;
    pos =0;
    while(q !=0 && q != p)
    {
        q = q->Next;
        pos++;
    }
    if(q == NULL)
        return -1;
    return pos;
}
```

- Xác định nút trước của nút p trong danh sách

```
NODEPTR PreNode (NODEPTR *plist, NODEPTR p)
{
    NODEPTR q;
    if(p == *plist)
        return NULL;
    q = *plist;
    while(q !=NULL && q->Next != p)
        q = q->Next;
    return q;
}
```

- Thêm nút có nội dung x vào đầu danh sách

```
void Push(NODEPTR *plist, Sinhvien x)
{
    NODEPTR p;
    p = GetNode();
    p->Info = x;
    p->Next = *plist;
    *plist = p;
}
```

- Thêm một nút mới sau nút p

```
void InsAfter( NODEPTR p, Sinhvien x)
{
    NODEPTR q;
    if(p == NULL)
        cout << " Không thêm vào danh sách được\n";
    else
    {
        q = GetNode( );
        q->Info = x;
        q->Next = p->Next;
        p->Next = q;
    }
}
```

- Xoá và trả lại nút đầu danh sách

```
Sinhvien FirstDel( NODEPTR *plist)
{
    NODEPTR p;
```

```
    Sinhvien x;
    if(Empty(plist))
     cout <<"Không có Sinh viên trong danh sách\n";
    else
    {
        p = *plist;
        x = p->Info;
        *plist = p->Next;
        delete p;
        return(x);
    }
}
```

- Xoá và trả lại nút sau p

```
Sinhvien AfterDel(NODEPTR p)
{
    NODEPTR q;
    Sinhvien x;
    if( (p== NULL ) || (p->Next == NULL))
        cout << "Không xoá được\n";
    else
    {
        p = p->Next;
        x = q->Info;
        p->Next = q->Next;
        delete q;
        return (x);
```

```
    }
}
```

- Duyệt danh sách

```
void Traverse(NODEPTR *plist)
{
    NODEPTR p;
    int stt =0;
    p = *plist;
    if(p == NULL)
      cout<<endl << "Không có SV trong danh sách\n";
    while(p != NULL)
    {
    cout <<" ";
    cout.width(5);
    cout << stt;
    cout.width(8);
    cout << p->Info.masv;
    cout.width(12);
    cout << p->Info.ten <<endl;
    p = p->Next;
    }
}
```

- Tìm kiếm theo mã

```
NODEPTR Search(NODEPTR *plist, int x)
{
    NODEPTR p;
```

```
    p = *plist;
    while(p->Info.masv != x && p != NULL)
        p = p->Next;
    return (p);
}
```

- **Sắp xếp theo phương pháp lựa chọn**

```
void SelectionSort(NODEPTR *plist)
{
    NODEPTR  p, q, pmin;
    Sinhvien min;
    for( p = *plist; p->Next !=NULL; p = p->Next)
    {
        min = p->Info;
        pmin = p;
        for(q=p->Next; q !=NULL; q = q->Next)
        if(min.masv > q->Info.masv);
        {
            min = q->Info;
            pmin = q;
        }
    pmin->Info = p->Info;
    p->Info = min;
    }
}
```

- **Xoá tất cả các nút trong danh sách**

```
void ClearList(NODEPTR *plist)
```

```
    {
        NODEPTR p, q;
        q = NULL;
        p = *plist;
        while(p != NULL)
        {
            q = p;
            p = p->Next;
            delete q;
        }
        *plist = NULL;
    }
```

Chương trình chính main sử dụng các hàm xử lý trên danh sách liên kết được viết như sau:

```
void main( )
{
    NODEPTR Plist;
    Sinhvien sv;
    NODEPTR p;
    int vitri, chucnang;
    clrscr( );
    // Khởi động danh sách liên kết
    Initialize(&Plist);
    do
    {
      cout<<"CHUONG TRINH QUAN LY SINH VIEN";
```

```
cout<<"\n\n" << "Cac chuc nang cua chuong trinh\n";
  cout<<"1: Xem danh sach sinh vien \n";
  cout<<"2: Them sinh vien vao danh sach\n";
  cout<<"3: Xoa sinh vien trong danh sach\n";
  cout <<"4: Hieu chinh sinh vien\n";
  cout<<"5: Sap xep sinh vien theo MASV\n";
  cout<<"6: Tim kiem sinh vien theo MASV\n";
  cout<<"7: Xoa toan bo danh sach\n";
  cout<<"0: ket thuc chuong trinh\n";
  cout<<"Chon chuc nang: ";
  cin >>chucnang;
  switch(chucnang)
  {
   case 1:
    cout<< "\n"<<"Danh sach sinh vien: ";
   cout<<"\n"<<"    STT    MASV              TEN\n";
   Traverse(&Plist);
   break;
  case 2:
   cout <<"\n Vitri them ( 0, 1, ....):  ";
   cin >> vitri;
   cout <<"Ma so sinh vien:  ";
   cin >>sv.masv;
   cout <<"ten sinh vien:  ";
   cin>>sv.ten;
   if(vitri ==0)
```

```
            Push(&Plist,sv);
        else
        {
            p = NodePointer(&Plist, vitri-1);
            if(p == NULL)
                cout << "Vi tri khong hop le";
            else
                InsAfter(p,sv);
        }
        break;
    case 3:
        cout <<"Vi tri xoa (0,1,2 ....):  ";
        cin >> vitri;
        if(vitri ==0)
            FirstDel(&Plist);
        else
        {
            p = NodePointer(&Plist, vitri -1);
            AfterDel(p);
        }
        break;
    case 4:
        cout <<"Vi tri hieu chinh (0, 1,2 ....):  ";
        cin >> vitri;
        p = NodePointer(&Plist, vitri);
        if(p ==NULL)
```

```
            cout<< " Vi tri khong phu hop:  ";
        else
        {
         cout <<" STT: "<<vitri<< " MASV:  "
         << p->Info.masv <<"  TEN: " << p->Info.ten;
        cout <<" Ma so sinh vien moi:  ";
        cin >>sv.masv;
        cout << "ten sinh vien moi: ";
        cin >> sv.ten;
        p->Info = sv;
        }
        break;
    case 5:
        cout <<" Ban co chac khong? (c/k):  ";
        char c = getche( );
        if(c =='c' || c == 'C')
              SelectionSort(&Plist);
        break;
     case 6:
        cout << " Ma so sinh vien can tim:  ";
        cin >> sv.masv;
        p = Search(&Plist, sv.masv);
        if(p == NULL)
              cout <<" Sinh vien voi ma so " << sv.masv
                   <<" Khong co trong danh sach";
        else
```

```
            cout <<"Tim thay o vi tri "<< Position(&Plist, p);
            break;
            case 7:
             cout<<" Ban co chac khong? (c/k)  ";
     c = getche( );
    if(c == 'c' || c =='C')
            ClearList(&Plist);
    break;
    }
  }while(chucnang !=0);
  ClearList(&Plist);
}
```

Chương trình này chỉ nhằm minh hoạ cho việc sử dụng cấu trúc đệ qui của struct, việc thiết kế chương trình còn rất nhiều điều cần bổ sung và hiệu chỉnh và có thể xem như bài tập cho bạn đọc.

6.4. DỮ LIỆU KIỂU HỢP

Hợp là một kiểu biến trong C++ dùng để lưu trữ các đối tượng có kiểu và kích thước khác nhau tại cùng một vị trí trong bộ nhớ ở những thời điểm khác nhau. Cú pháp định nghĩa hợp cũng giống như kiểu cấu trúc và có dạng sau:

```
union Tên_hợp
{
Khai báo các phần tử;
};
```

Sự giống nhau giữa hợp và cấu trúc không chỉ thể hiện sự giống nhau về cú pháp định nghĩa. Sự giống nhau này bao gồm cả các phép toán trên cấu trúc và hợp. Như vậy, tất cả các quy tắc, định nghĩa và cách làm việc trên cấu trúc đều đúng với hợp. Giữa hợp và cấu trúc, tuy vậy vẫn tồn tại một sự khác nhau cơ bản: đó là việc cấp phát và sử dụng bộ nhớ dành cho một biến khi khai báo. Đối với cấu trúc, trình biên dịch sẽ cấp phát bộ nhớ cho từng thành phần của nó khi biến được khai báo, trong khi đó đối với hợp, mặc dù có thể có nhiều thành phần dữ liệu có kiểu khác nhau nhưng trình biên dịch chỉ cấp phát một vùng nhớ duy nhất có kích thước bằng kích thước lớn nhất trong số các phần tử và tất cả các phần tử đều dùng chung vùng nhớ này. Có thể minh hoạ điều này qua ví dụ dưới đây:

```
struct SET
{
    int k;
    float f;
    char c;
} TEST;
```

Theo phép khai báo trên, TEST là một biến thuộc kiểu hợp có tên là SET. Khi cấp phát bộ nhớ cho biến TEST, trình biên dịch sẽ sử dụng kích thước bộ nhớ bằng kích thước lớn nhất trong số các thành phần của SET. Trong trường hợp này, vùng nhớ được cấp phát sẽ bao gồm 4 byte bằng kích thước của biến *f*.

Một điểm khác nhau cơ bản nữa giữa hợp và cấu trúc là

không thể khởi tạo các phần tử của hợp khi khai báo biến.

Dưới đây sẽ mô tả một ví dụ về cách sử dụng hợp.

Ví dụ 6.4

Một số nguyên kiểu int được lưu trữ trong bộ nhớ với kích thước là 2 byte (byte thấp và byte cao). Hãy hiển thị lên màn hình giá trị riêng biệt của từng byte.

Ví dụ này đã được trình bày trong các phần trước thông qua việc sử dụng phép toán lấy phần dư và phần nguyên của phép chia cũng như việc sử dụng con trỏ. Dưới đây sẽ trình bày phương pháp sử dụng *hợp*.

```
#include <iostream.h>
#include <conio.h>
union exampl
{
    int k;
    unsigned  char ch[2];
};
void main( )
{
    clrscr( );
    exampl cnvt;
    cout <<" Hãy nhập số nguyên:";
    cin >> cnvt.k;
    // Chú ý sử dụng phép chuyển kiểu
```

```
    cout << "Byte thấp là: " << (unsigned)cnvt.ch[0];
    cout << "\n Byte cao là:  " <<(unsigned)cnvt.ch[1];
}
```

Trong ví dụ, cả hai biến *k* và mảng *ch* đều dùng chung một vùng nhớ gồm 2 byte. Sau khi nhập giá trị cho *k*, vùng nhớ này sẽ chứa giá trị được đưa từ bàn phím, do mảng *ch* gồm 2 byte cũng sử dụng vùng nhớ này nên có thể truy nhập đến từng byte thông qua các phần tử của mảng *ch*.

6.5. CẤU TRÚC BIT HAY VÙNG BIT

Một trong các ưu điểm của ngôn ngữ C++ là khả năng cho phép xử lý từng bit riêng biệt trên một byte. Phương pháp mà C++ sử dụng để truy nhập đến các bit riêng biệt là dựa vào cấu trúc. Vùng bit là một kiểu dữ liệu cho phép nối liền giữa mức cao của chương trình và mức thấp của phần cứng của máy.

Việc đưa ra kiểu dữ liệu này dựa trên 3 nguyên nhân cơ bản thường phát sinh trên thực tế:

- Điều kiện dùng để điều khiển việc rẽ nhánh trong các chương trình phức tạp thường dựa trên các khoá thuộc kiểu *int*. Trong rất nhiều trường hợp, các khoá này chỉ nhận một trong hai giá trị 0 hay 1. Như vậy mặc dù chỉ cần một bit để lưu trữ giá trị này, chương trình vẫn phải dành ít nhất là 1 byte cho mục đích này. Để tiết kiệm bộ nhớ dành cho các biến kiểu này ta có thể dùng vùng bit thay cho kiểu int.

- Trong nhiều trường hợp khi làm việc với các thiết bị vào/ra có thể cần làm việc với từng bit riêng biệt của các thiết bị.
- Một vài kiểu dữ liệu, ngoài giá trị chính của mình còn có thể đi kèm theo một số tham số khác. Khi làm việc với các dữ liệu kiểu này, để thuận tiện thường dùng một từ của máy để chứa tất cả các tham số đó của dữ liệu. Chẳng hạn có thể dùng 8 bit để chứa mã của một ký tự và một số bit dành cho màu nền, một số bit dành cho độ sáng v.v.

Vùng bit được định nghĩa như các thành phần của cấu trúc thông qua cú pháp sau:

unsigned int Tên_vùng Kích_thước;

trong đó *Kích_thước* sẽ xác định số bit của vùng tương ứng. Chẳng hạn trong khai báo:

```
struct field
{
    unsigned int f1:1;
    unsigned int f2:3;
    int k;
}x,y;
```

các biến *x, y* được khai báo như các biến cấu trúc thuộc kiểu field. Cấu trúc *field* gồm 3 thành phần: vùng bit *f1* gồm 1 bit; *f2* gồm 3 bit và một biến *k* thuộc kiểu *int*.

Khi một cấu trúc có chứa các vùng bit, các vùng bit này sẽ được phân bố trên cùng một từ nếu có thể, theo thứ tự định nghĩa chúng. Trong trường hợp nếu từ đang sử dụng không còn chỗ dành cho chúng, các vùng bit này sẽ được phân bố ngay ở từ tiếp sau đó. Tuy vậy, nếu 2 vùng bit trong cùng một cấu trúc được định nghĩa cách nhau bởi một biến khác thì các vùng bit này sẽ được phân bố trên hai từ khác nhau. Chẳng hạn trong cấu trúc:

```
struct prim
{
    unsigned int field1: 1;
    int k;
    unsigned int field2: 3;
}x;
```

các vùng bit *field1* và *field2* sẽ được phân bố trên hai từ khác nhau.

Việc truy nhập đến dữ liệu kiểu vùng bit và sử dụng chúng cũng được thực hiện tương tự như các thành phần của cấu trúc. Trong ví dụ trên, các vùng bit của biến *x* có thể được truy nhập qua cách sau:

```
x.field1 =0;
x.field2 =1;
```

Khi một vùng bit được gán giá trị vượt quá kích thước mà nó được khai báo, trong vùng bit chỉ chứa giá trị nằm trong các

bit thấp mà giá trị được gán biểu diễn dưới dạng nhị phân. Cũng cần lưu ý là khi tham gia vào biểu thức, giá trị của vùng bit sẽ tự động được chuyển về kiểu int.

Khi sử dụng vùng bit cần lưu ý:

- Không được phép sử dụng con trỏ đối với vùng bit.
- Các phần tử của mảng không được định nghĩa kiểu vùng bit.

Phần II

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Chương VII

LỚP VÀ ĐỐI TƯỢNG

7.1. ĐỊNH NGHĨA LỚP VÀ KHAI BÁO ĐỐI TƯỢNG

Một trong những điểm đặc biệt của ngôn ngữ C^{++} là khả năng *đóng gói* (Encapsulation) dữ liệu và các hàm xử lý các dữ liệu đó vào một lớp đơn. Với đặc tính này, C^{++} đã thay đổi phương pháp làm việc với dữ liệu (*data*) và chương trình (*Code*) khác hẳn với ngôn ngữ C. Trong C^{++} , một lớp đơn (được định nghĩa như *struct, union* hoặc *class*) bao gồm các hàm thành phần và dữ liệu có liên quan. Có thể xem dữ liệu của lớp (*Data member*) bao gồm các thuộc tính chung mà tất cả các đối tượng thuộc lớp đều có, trong khi các hàm thành phần *(Member Function)* - còn được gọi là phương thức (*Method*) sẽ thể hiện các tác động có thể được thực hiện trên dữ liệu của các đối tượng thuộc lớp. Như vậy có thể xem lớp là một khái niệm trừu tượng

và là một bản mẫu mô tả các thông tin cấu trúc dữ liệu (các thông tin này được phản ánh qua các thành phần dữ liệu được khai báo trong lớp) và các thao tác cụ thể (các hàm thành phần trong lớp) có thể thực hiện được trên các dữ liệu đó. Các hàm và dữ liệu này được gọi là giao diện sử dụng (*user interface*) của lớp. User Interface có thể được xem như cầu nối giữa đối tượng và thế giới bên ngoài. Tuy vậy, khi sử dụng lớp có thể không cần biết đến những gì xảy ra bên trong và vì vậy có thể xem lớp như một hộp đen chỉ có đầu vào và đầu ra. Khi khai báo lớp, tên của lớp được đặt theo nguyên tắc như đặt tên cho biến trong C và được viết sau một trong các từ khoá *class, struct, union*.

Hãy xét một số ví dụ về cách định nghĩa lớp:

- Lớp Circle (vòng tròn): Lớp Circle cần phải mô tả các đặc tính chung nhất mà mọi vòng tròn đều có như toạ độ tâm, bán kính vòng tròn cũng như các tác động có thể thực hiện trên các vòng tròn như hiển thị, xoá và di chuyển.

Từ cách phân tích trên có thể đưa ra cách khai báo sau cho lớp Circle :

```
class Circle
{
    // Mẫu các thông tin dữ liệu của các vòng tròn
    int CenterX, CenterY; // Toạ độ tâm
    int Radius; // Bán kính
    // Các tác động có thể thực hiện trên các vòng tròn
    public:
        void Show( ); // Hiển thị vòng tròn
```

```
    void Hide( ); // ẩn vòng tròn
    void MoveTo(int NewCenterX, int NewCenterY);
}; // Lưu ý dấu chấm phẩy ở đây
```

- Lớp String (Lớp chuỗi): Dữ liệu của lớp này mô tả các đặc tính chung của các xâu ký tự được sử dụng trong ngôn ngữ như con trỏ chỉ đến xâu và độ dài của xâu. Ngoài các dữ liệu này, trong định nghĩa của lớp còn chứa các hàm thành phần mô tả các tác động có thể thực hiện trên xâu ký tự như hiển thị, chuyển xâu thành chữ viết hoa, lấy n ký tự bên trái của xâu v.v.

Như vậy lớp String có thể được khai báo như sau:

```
class String
{
// Mẫu các thông tin dữ liệu
    char *char_ptr; // Con trỏ chỉ đến xâu;
    int length;     // Độ dài của xâu
// Các tác động
public:
    void Show( ); // Hiển thị xâu
    String Upper( ); // Chuyển thành chữ viết hoa
    String Left(int n); //Lấy n ký tự bên trái của xâu.
};
```

Lớp có thể được xem như là một kiểu dữ liệu mới do người sử dụng định nghĩa vì vậy có thể khai báo các biến từ kiểu dữ liệu này. Các biến này được gọi là đối tượng (*object*) hoặc thể hiện (*instance*) của lớp đó. Việc khai báo đối tượng được thực

hiện như trong trường hợp khai báo các biến thuộc kiểu dữ liệu chuẩn. Chẳng hạn có thể khai báo các đối tượng thuộc lớp Circle như sau:

Circle C1, C2;

Hoặc các đối tượng thuộc kiểu chuỗi :

String S1, S2;

Ngoài cách khai báo này có thể sử dụng các cách khai báo khác:

Circle C[10]; //Mảng 10 đối tượng thuộc lớp Circle

Circle *C; //Con trỏ chỉ đến đối tượng thuộc lớp Circle;

Cần nhắc là khi khai báo các biến thuộc kiểu một lớp, từ khóa *class* không cần viết lặp lại như trong trường hợp sử dụng cấu trúc trong C.

Sau khi được khai báo, mỗi đối tượng của lớp sẽ có các dữ liệu riêng của mình. Các dữ liệu và các hàm thành phần của một đối tượng được truy nhập qua cách sau:

Tên_đối_tượng.Dữ_liệu_thành_phần;

Ví dụ: C1. X, hay S1.length;

hoặc:

Tên_đối_tượng.Hàm_thành_phần;

Ví dụ: C1. Show( ); // Hiển thị vòng tròn

hay:

S1.Upper( ); // Chuyển S1 thành chữ hoa.

Điểm khác nhau cơ bản giữa lớp (class) trong C++ và cấu

trúc (struct) trong C đó là khả năng truy cập đến các thành phần của chúng và đây cũng chính là mấu chốt của tính đóng gói trong C. Ngoài việc cho phép việc khai báo dữ liệu và các hàm thành phần tác động lên các dữ liệu đó ở trong cùng một lớp, trong khi thiết kế và phát triển C++, các nhà thiết kế còn đưa ra nhiều khả năng mới của một lớp trong C++ so với cấu trúc trong C, đó là khả năng và mức độ truy nhập đến các thành phần dữ liệu của một lớp. Trong khi các thành phần cấu trúc trong C được truy nhập tự do bởi các hàm có cùng phạm vi hoạt động với chúng thì với C++ , các hàm thành phần và dữ liệu của một lớp chỉ có thể được truy nhập thông qua các đối tượng của lớp này và ngoài ra người lập trình cũng có thể kiểm soát mức độ truy nhập đến các thành phần dữ liệu của các lớp một cách rất chặt chẽ. Việc kiểm soát này được thực hiện thông qua các từ khóa *public, private* và *protected* khi khai báo các thành phần của một lớp.

7.2. THÀNH PHẦN DỮ LIỆU (DATA MEMBER)

7.2.1. Dữ liệu thành phần kiểu private

Khi các thành phần dữ liệu của một lớp được khai báo theo kiểu *private* thì chỉ có các hàm thành phần của chính lớp đó hoặc các lớp bạn của nó có thể truy nhập đến các thành phần này (các lớp bạn của một lớp xem phần 7.4.7).

Hãy xem lại cách định nghĩa lớp Circle và ghi đoạn mã dưới đây vào file tiêu đề có tên là *circle.h*:

```
class Circle
{
private:
    int CenterX, CenterY;
    int Radius;
public:
    void Show( ) { }
    void Hide( ) { }
    void MoveTo(int NewCenterX, int New CenterY) { }
};
```

File tiêu đề này sẽ được sử dụng trong ví dụ 7.1.

Ví dụ 7.1

```
#include <iostream.h>
#include "circle.h"
void main()
    {
    Circle C1;  // Khai báo đối tượng C1 kiểu Circle
    cout << C1.Radius; // Hiển thị bán kính của C1
    }
```

Ví dụ trên tuy vậy sẽ phát sinh lỗi khi biên dịch do Radius được khai báo theo kiểu *private* và việc truy nhập đến các thành phần này ở bên ngoài lớp Circle (trong hàm *main*) là không được phép.

7.2.2. Dữ liệu thành phần kiểu public

Với từ khóa *public*, C++ cho phép người sử dụng truy nhập đến các thành phần của lớp ở bên ngoài lớp đó. Điều này có thể thấy rõ nếu trong file cirle.h từ khoá *private* được thay thế bởi từ khoá public khi khai báo các dữ liệu thành phần.

Ví dụ 7.2

```
#include <iostream.h>
#include "circle.h" // Đã thay từ khoá private bằng public
void main( )
{
    Circle C1;  // Khai báo đối tượng C1 kiểu Circle
    C1.Radius = 2; // Khởi tạo bán kính của C1.
    C2.Radius = 3; // Khởi tạo bán kính của C2.
    cout << "R =" << C1.Radius << endl; //Hiển thị bán kính của C1
    cout << "Toa do X =" << C1. CenterX  << endl;
    cout << "Toa do Y =" << C1. CenterY  << endl;
}
```

Mặc dù việc khai báo dữ liệu theo kiểu *public* sẽ cho phép chúng được truy nhập bên ngoài lớp nhưng trên thực tế vì các dữ liệu này là các dữ liệu riêng nên thường được khai báo theo kiểu *private* để tránh việc thay đổi giá trị của chúng bên ngoài lớp. Trong các trường hợp này giá trị của các biến với mức truy nhập

private có thể được truy nhập từ bên ngoài thông qua các hàm thành phần của lớp (xem phần 7.4)

7.2.3. Dữ liệu thành phần kiểu protected

Các thành phần kiểu *protected* chỉ được phép truy nhập đến bởi các hàm trong các lớp dẫn xuất từ một lớp cơ sở. Khái niệm này sẽ được đề cập đến ở chương VIII.

Lưu ý:

- Trong C++ , cả ba từ khóa *struct, union, class* đều có thể được sử dụng để định nghĩa một lớp với một số khác biệt sau:
- Các thành phần của các lớp được định nghĩa bởi từ khóa *struct* sẽ có mức độ truy nhập là *public* theo mặc định (từ khoá này có thể được bỏ qua khi khai báo). Tuy nhiên mức độ truy nhập này có thể bị thay đổi theo ý của người sử dụng.
- Các thành phần của các lớp được định nghĩa bởi *union* đều có mức độ truy nhập là *public*. Mức độ truy nhập này là cố định, không thể bị thay đổi.
- Các lớp được định nghĩa bởi từ khoá *class* có các thành phần với mức độ truy nhập mặc định là *private*. Mức độ truy nhập này tuy vậy lại có thể bị thay đổi theo ý của người sử dụng.

7.2.4. Dữ liệu thành phần kiểu class (Các lớp lồng nhau)

Cũng giống như cấu trúc, một lớp có thể được khai báo lồng trong một lớp khác. Chẳng hạn trong ví dụ sau, lớp *Inner* được

khai báo lồng bên trong lớp *Outer*.

```
class Outer {
    public:
        int x;
        class Inner {
    public:
        int value;
    };
};
```

Khi lớp *Inner* được khai báo bên trong lớp *Outer*, các thành phần của lớp này có thể được truy nhập theo cách sau:

```
void main()
    {
    Outer::Inner object;
    // Khai báo một đối tượng thuộc kiểu Inner
    int v = object.value;
    // Truy nhập đến thành phần value của object.
    }
```

7.2.5. Dữ liệu thành phần kiểu static

Khi một thành phần dữ liệu được khai báo theo kiểu *static* thì tất cả các thể hiện của lớp đó đều được phép dùng chung thành phần dữ liệu này. Điều này xem ra cũng giống như trong ngôn ngữ C. Dữ liệu theo kiểu *static* được phân bố ở một vùng nhớ cố định trong quá trình liên kết cũng giống như các biến được khai báo theo kiểu toàn cục (*Global*). Việc truy nhập đến các biến static cần phải được thực hiện thông qua toán tử phạm vi (::) cùng với tên của lớp. Cần lưu ý một số điều sau khi sử

dụng dữ liệu thành phần kiểu *static:*

- Nếu một đối tượng được khai báo theo kiểu *static* thì tất cả dữ liệu của nó cũng thuộc kiểu *static.*
- Khi một đối tượng với thành phần dữ liệu kiểu *static* được tạo thì thành phần *static* này sẽ không được phân bố bộ nhớ vì thành phần này là duy nhất cho tất cả các đối tượng của lớp. Như vậy việc khai báo và khởi tạo dữ liệu thành phần kiểu *static* được tiến hành giống như các biến tổng thể.

Để mô tả cho việc sử dụng dữ liệu thành phần kiểu static ta hãy thực hiện một thay đổi nhỏ trong file tiêu đề circle.h bằng cách khai báo thêm biến Count trong lớp Circle.

```
class Circle
{
private:
    int CenterX, CenterY;
    int Radius;
public:
    static int Count; // Thêm biến thuộc kiểu static
public:
    void Show( ) { }
    void Hide( ) { }
    void MoveTo(int NewCenterX, int NewCenterY) { }
};
```

Ví dụ 7.3

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

```
int Circle:: Count;
void main( )
{
    Circle C1;  // Khai báo đối tượng C1 kiểu Circle
    Circle C2;
    C1.Count = 2;
    C2.Count =3;
    cout<< " C1.Count = "<<C1.Count <<endl; //Hiển thị giá trị Count
    của C1
    cout << "C2.Count =" <<C2.Count << endl; // Hiển thị giá trị Count
    của C2
}
```

Trong ví dụ trên, nếu bỏ qua khai báo *int Circle: : Count;* thì chương trình có thể được biên dịch nhưng trong quá trình liên kết có thể phát sinh lỗi. Sau khi thực hiện cũng cần lưu ý một điều là cả hai biểu thức *C1.Count* và *C2.Count* đều có giá trị bằng 3 vì thực chất hai biến *Count* này chỉ là một.

Như vậy khi một thành phần dữ liệu kiểu *static* được khai báo thì trình liên kết sẽ phân bố bộ nhớ cho thành phần này chỉ một lần và chỉ khi thành phần dữ liệu này được định nghĩa. Như vậy cấp lưu trữ của *Count* trong ví dụ trên luôn là *static* mặc dầu hai đối tượng *C1* và *C2* được khai báo theo kiểu auto.

Khi định nghĩa một thành phần kiểu *static* ta cũng có thể khởi tạo nó. Chẳng hạn, giá trị của biến *Count* có thể được khởi tạo bằng lệnh sau:

```
int Circle::Count =3; // Khởi tạo Count =3
```

Vì dữ liệu kiểu *static* được dùng chung cho tất cả các đối

của một lớp khác. Cách khai báo này có thể được minh hoạ qua ví dụ sau:

```
class Point {
    int CenterX, CenterY;
        // Các hàm thành phần được định nghĩa ở đây.
};
class Circle {
        int Radius;
        Point P1;
            // Các hàm thành phần được định nghĩa ở đây.
};
```

Việc sử dụng dữ liệu thành phần kiểu này sẽ được xem xét kỹ hơn trong phần các hàm thành phần của lớp.

7.2.7. Một vài điểm cần lưu ý khi khai báo và sử dụng class

Tính đóng gói (*encapsulation*) cho phép đóng gói dữ liệu và các hàm thao tác trên các dữ liệu đó trong cùng một lớp. Các dữ liệu thành phần có thể được khai báo bởi các từ khóa *private, public* hoặc *protected.*

Không được phép khởi tạo các dữ liệu thành phần bên trong thân của một lớp. Một lớp không phải là một đối tượng (*object*) và do đó không được cấp phát bộ nhớ cho đến khi một đối tượng của nó được thể hiện. Khi một đối tượng thuộc một lớp nào đó được khởi tạo thì ta mới có thể truy nhập đến các thành phần dữ liệu của đối tượng (trừ phi thành phần dữ liệu được khai báo theo kiểu *static*).

7.3. HÀM THÀNH PHẦN (FUNCTION MEMBER)

Như đã đề cập đến ở phần trước, một lớp của C++ có thể chứa các hàm cũng như các dữ liệu thành phần. Các hàm thành phần này sẽ cho phép truy nhập và xử lý các dữ liệu của lớp đó. Các hàm thành phần là các hàm được khai báo trong phần định nghĩa của một lớp và luôn gắn chặt với chính lớp đó (Các hàm thành phần còn có tên gọi là phương thức trong các ngôn ngữ lập trình hướng đối tượng khác như *Pascal, Visual Basic, SmallTalk* v.v.).

Có hai phương pháp để thiết lập một hàm thành phần của một lớp:

- Định nghĩa hàm ở bên trong lớp.
- Khai báo hàm ở bên trong lớp và định nghĩa hàm ở bên ngoài lớp.

Hãy bắt đầu nghiên cứu hàm thành phần của lớp thông qua lớp Circle. Để thực hiện điều này định nghĩa của lớp Circle trong file tiêu đề circle.h được sửa đổi như dưới đây:

```
class Circle
{
private:
    int CenterX, CenterY;
    int Radius;
public:
    static int Count;
public:
    void Set(int, int, int); // Khởi tạo các tham số của vòng tròn
```

```
    int GetRadius( )
    {
        return Radius;
    }
    int GetY( ) ; // Nhận giá trị của CenterY;
    {
        return CenterY;
    }
    int GetX( ) // Nhận giá trị của CenterX
    {
        return CenterX;
    }
    void Show( ) { }; // Hiển thị vòng tròn
    void Hide( ) { }; // ẩn vòng tròn
    void MoveTo(int NewCenterX, int NewCenterY){ };
};
```

Theo phương pháp thứ nhất, các hàm thành phần *GetRadius(), GetY(), GetX(), Show( ), Hide( ) MoveTo, SetX( ), SetY( ), SetRadius()* của lớp Circle được khai báo và định nghĩa ở bên trong lớp *Circle*. Trong trường hợp này, các hàm được khai báo theo kiểu *inline*. Trong C++, các hàm *inline* sẽ được gọi một cách hiệu quả hơn so với các hàm khác. Tuy vậy không phải hàm nào cũng đều có thể khai báo theo kiểu *inline*. Thông thường *inline* là các hàm ngắn gọn được định nghĩa và khai báo bên trong thân của một lớp, trong trường hợp này không cần dùng từ khóa đặc biệt nào khác. Các hàm thành phần được định nghĩa bên ngoài thân của lớp vẫn có thể thuộc kiểu *inline* nếu ta sử dụng từ khóa *inline* trước tên hàm khi định nghĩa. Tuy vậy cũng cần lưu ý rằng sự phức tạp của các hàm *inline* luôn được trình

biên dịch kiểm soát một cách chặt chẽ. Không phải mọi hàm được khai báo theo kiểu *inline* đều được trình biên dịch chấp nhận. Nếu trình biên dịch phát hiện rằng hàm đó có kích thước lớn hoặc quá phức tạp thì hàm sẽ được xem như bình thường mà không được xem như kiểu *inline*. Các hàm thuộc kiểu *inline* được C++ xem như *macro* khi biên dịch: ở bất cứ nơi nào trong chương trình có lời gọi hàm, trình biên dịch sẽ thay thế tên hàm bởi các câu lệnh bên trong thân của chính hàm đó. Điều này sẽ tránh được các phí tổn về thời gian so với trường hợp phải gọi hàm, tuy vậy kích thước của chương trình thay vào đó sẽ bị tăng lên.

Phương pháp thứ hai dùng để thiết lập hàm thành phần là khai báo nguyên mẫu của các hàm này bên trong thân của lớp và định nghĩa hàm bên ngoài của lớp đó. Trong trường hợp này khi định nghĩa hàm, ngoài việc phải chỉ rõ kiểu của hàm, tên hàm cũng như các tham số truyền còn phải chỉ rõ tên của lớp chứa hàm này.

Việc định nghĩa hàm thành phần bên ngoài một lớp được thực hiện qua cú pháp sau:

Kiểu-hàm Tên_Lớp :: Tên_hàm(...)

Trong phép định nghĩa này, *::* được gọi là toán tử phạm vi.

Như vậy hàm *Set* được khai báo bên trong lớp *Circle* có thể được định nghĩa bên ngoài theo cách sau:

```
void Circle:: Set(int X, int Y, int R) //Viết trong cùng file circle.h
        {
        CenterX = X;
        CenterY = Y;
        Radius = R;
        }
```

Hàm này dùng để khởi tạo các giá trị ban đầu của đối tượng.

Qua cách khai báo trên của lớp có thể thấy, mặc dù các dữ liệu của nó được khai báo bằng mức truy nhập *private* nhưng các biến này vẫn có thể được truy nhập thông qua các hàm thành phần của lớp ở bên ngoài lớp này. Như vậy các dữ liệu này vẫn có thể được truy nhập đến mà không phải làm việc trực tiếp với chúng ở bên ngoài lớp. Hàm *main( )* được viết như sau:

Ví dụ 7.4

```
#include <iostream.h>
#include "circle.h"
int Circle:: Count;
void main( )
{
    Circle C1;  // Khai báo đối tượng C1 kiểu Circle
    C1.Set(120, 140, 40);
    // Hiển thị các giá trị dữ liệu của C1 sau khi thiết lập các giá trị
    cout<< " C1.CenterX = "<< C1.GetX( ) <<endl;
    cout<< " C1.CenterY = "<< C1.GetY( ) <<endl;
    cout << "C1Radius =" << C1.GetRadius << endl;
}
```

7.3.1. Các loại hàm thành phần đặc biệt

Trong số các hàm thành phần của một lớp, có hai loại hàm đặc biệt là *constructor* và *destructor*. Mặc dù đây là các hàm không bắt buộc phải có nhưng việc sử dụng các hàm này mang lại rất nhiều tiện lợi. Một đặc điểm khác biệt của các hàm này so với các hàm khác là chúng tự động được gọi bởi trình biên dịch

theo từng sự kiện đang xảy ra và do đó có thể giải phóng cho người lập trình khỏi những thao tác nhiều khi gây ra những khó chịu không cần thiết.

7.3.1.1. Constructor

Constructor là một hàm thành phần dùng để khởi tạo một đối tượng cụ thể từ một lớp. Khi *constructor* không được định nghĩa thì trình biên dịch sử dụng một *constructor* ngầm định (trong tất cả các ví dụ trên ta đã sử dụng constructor này). Với *constructor* ngầm định, tất cả các dữ liệu thành phần của một đối tượng sẽ được khởi tạo giá trị ban đầu là 0. Một lớp có thể chỉ có constructor ngầm định nhưng cũng có thể có một vài *constructor* khi cần thiết. Điều quan trọng cần lưu ý là một khi *constructor* đã được định nghĩa, C++ đảm bảo rằng *constructor* sẽ được gọi và thực hiện trước các hàm khác của lớp.

Constructor có một số đặc điểm sau:

- Có tên trùng với tên lớp;
- Không có kiểu giá trị trả lại kể cả kiểu void.
- Có thể phân loại các *Constructor* theo các nhiệm vụ mà chúng thực hiện:
- Constructor mặc định. Đây là constructor dùng để khởi tạo đối tượng với các giá trị ngầm định của các dữ liệu thành phần.
- Constructor với tham số truyền - Dùng để khởi tạo đối tượng với các giá trị được truyền cho các dữ liệu thành phần.
- Constructor sao chép - Dùng để khởi tạo một đối tượng

mới và sao chép dữ liệu cho nó từ một đối tượng khác đã tồn tại.

*** Constructor mặc định**

Khi một *constructor* được khai báo không sử dụng các tham số truyền, *constructor* đó được gọi là mặc định. Như vậy theo cách khai báo này thì mỗi lớp chỉ có một *constructor* mặc định duy nhất. Thậm chí nếu không có một *constructor* nào được khai báo thì C++ sẽ tự động tạo ra *constructor* mặc định. Với *constructor* mặc định, tất cả các thành phần dữ liệu của đối tượng sẽ được khởi tạo giá trị 0. Tuy vậy kiểu *constructor* này rất ít được sử dụng trong thực tế bởi vì với các giá trị khởi tạo bằng 0 của các dữ liệu thành phần thường chưa đủ để cho các đối tượng làm việc.

Ví dụ 7.5

```
class Circle
{
private:
    int CenterX, CenterY;
    int Radius;
public:
    static int Count;
    int GetRadius( )
    {
        return Radius;
    }
    int GetY( ) ;
    {
```

```
        return CenterY;
    }
    int GetX( )
    {
        return CenterY;
    }
    void Show( ) { }; // Hiển thị vòng tròn
    void Hide( ) { }; // ẩn vòng tròn
    void MoveTo(int NewCenterX, int NewCenterY){ };
};
int Circle:: Count = 0;
void main( )
{
    Circle C1;  // Khai báo đối tượng C1 kiểu Circle
    cout<< " C1.CenterX = "<< C1. GetX()<<endl;
    cout<< " C1.CenterY = "<< C1. GetY()<<endl;
    cout<< " C1.Radius = "<< C1. GetRadius()<<endl;
}
```

Trong ví dụ này, không có *contructor* nào được định nghĩa trong lớp nên khi khởi tạo đối tượng C1, trình biên dịch đã sử dụng *constructor* ngầm định vì vậy tất cả các dữ liệu của C1 đều có giá trị bằng 0.

*** Constructor với tham số**

Nhiệm vụ cơ bản của *constructor* của một lớp là khởi tạo các đối tượng trước khi sử dụng chúng. Điều này đòi hỏi các *constructor* phải được truyền tham số. Khi được gọi *constructor* sẽ khởi tạo giá trị cho các biến, phân bố bộ nhớ động cho các biến, gọi hàm số v.v. Trong ví dụ về lớp Circle, có thể thấy nếu

sử dụng *constructor* kiểu này thì việc xây dựng hàm *set* để thiết lập các giá trị cho các biến là không còn cần thiết nữa.

Dưới đây hãy xem xét một lớp chứa dữ liệu và *constructor* dùng để khởi tạo các đối tượng:

Nội dung của file tiêu đề circle.h được sửa lại như sau:

```
class Circle
{
private:
    int CenterX, CenterY;
    int Radius;
public:
    static int Count;
public:
Circle(int x, int y, double r); // Constructor khởi tạo giá trị
    int GetRadius( )
    {
        return Radius;
    }
    int GetX( )
    {
        return CenterX;
    }
    int GetY( )
    {
        return CenterY;
    }
    int GetX( );
```

```
    void Show( ) { }; // Hiển thị vòng tròn
    void Hide( ) { }; // ẩn vòng tròn
    void MoveTo(int NewCenterX, int NewCenterY){ };
};
Circle:: Circle(int x, int y,double r) // Constructor khởi tạo giá trị
{
    Radius = r;
    CenterX = x;
    CenterY = y;
}
```

Ví dụ 7.6

```
#include <iostream.h>
#include "circle.h"
int Circle:: Count = 0;
void main( )
{
    // Khai báo đối tượng C1 kiểu Circle
    Circle C1(120, 150, 20);
    cout<< " C1.CenterX = "<< C1. GetX()<<endl;
    cout<< " C1.CenterY = "<< C1. GetY()<<endl;
    cout<< " C1.Radius = "<< C1. GetRadius()<<endl;
}
```

Trong định nghĩa của lớp có dùng constructor với 3 tham số vì vậy khi một đối tượng của *Circle* được khởi tạo, nó phải có ba đối số để cung cấp tham số cho *constructor*. Khi thực hiện, chương trình sẽ in giá trị của tọa độ tâm và bán kính của vòng tròn C1 lên màn hình.

Có hai điểm cần lưu ý:

- Khi đã định nghĩa một *constructor* của lớp thì C++ sẽ không phát sinh *constructor* mặc định cho lớp đó. Như vậy việc khởi tạo một đối tượng với các dữ liệu thành phần có giá trị mặc định bằng 0 (chẳng hạn Circle C1) trong hàm main sẽ bị báo lỗi. Để có thể thực hiện được điều này thì cần phải định nghĩa lại constructor mặc định một cách rõ ràng. Đối với lớp Circle thì constructor này được định nghĩa như sau:

```
Circle:: Circle( ); // Constructor khởi tạo giá trị
{
    Radius = r;
    CenterX = x;
    CenterY = y;
}
```

- Một *constructor* với tất cả các tham số truyền ngầm định bằng 0 cũng được xem như *constructor* mặc định vì nó có thể được gọi mà không cần đến tham số và điều này sẽ làm phát sinh lỗi khi khởi tạo đối tượng với tham số ngầm định. Chẳng hạn trong cùng lớp Circle không thể tồn tại hai constructor:

```
Circle:: Circle( ); // Constructor khởi tạo giá trị
    {
    Radius = r;
    CenterX = x;
    CenterY = y;
    }
```

và:

```
Circle:: Circle(int x = 0, int y =0 ,double r =0)
```

```
{
    Radius = r;
    CenterX = x;
    CenterY = y;
}
```

vì điều này sẽ dẫn đến việc không rõ ràng khi khởi tạo đối tượng theo kiểu Circle C1.

Dưới đây sẽ đưa ra một ví dụ tương đối hoàn chỉnh về lớp Circle.

Lớp vòng tròn Circle có các hàm thành phần sau:

- Hàm Show - hiển thị vòng tròn;
- Hàm Hide - xoá vòng tròn
- Hàm MoveTo - di chuyển vòng tròn đến toạ độ cho trước.
- Hàm Expand dùng để tặng bán kính vòng tròn;
- Hàm Contract dùng để giảm bán kính vòng tròn.
- Hàm Drag dùng để di chuyển vòng tròn khi ta sử dụng các phím di chuyển ↑, ↓, →, ←.

Ngoài các hàm thành phần này của lớp, trong chương trình còn sử dụng một hàm toàn cục *GetDelta*, hàm này có hai tham số truyền theo kiểu tham chiếu int& DeltaX và int& DeltaY. Các tham số này sẽ có giá trị bằng 1 nếu người sử dụng gõ các phím ↑, → và có giá trị bằng -1 nếu gõ các phím ↓, ←. Ngoài ra hàm sẽ nhận giá trị bằng 1 khi người sử dụng gõ một trong các phím này và có giá trị bằng 0 khi gõ phím ↵ (Enter). Giá trị bằng 0 của hàm sẽ kết thúc chương trình, trong khi với giá trị bằng 1 của hàm GetDelta, hàm Drag sẽ thay đổi toạ độ tâm của vòng

tròn và di chuyển nó đến toạ độ tâm mới này.

Ví dụ 7.7

```
// File circle.h
enum Boolean {false,true};
Boolean GetDelta(int& DeltaX, int& DeltaY) // Hàm toàn cục
{
    char KeyChar;
    Bollean Quit;
    DeltaX =0;
    DeltaY =0;
    do
    {
        KeyChar = getch();
        if(KeyChar = = 13) //Gõ Enter
            return(false);
        if(KeyChar ==0)
        {
            Quit = true;
            KeyChar = getch(); // đọc phím mở rộng
        switch(KeyChar)
        {
            case 72:DeltaY = -1; break;
            case 80:DeltaY = 1; break;
            case 75:DeltaX = -1; break;
            case 77:DeltaX = 1; break;
            default:Quit = false;
```

```
            }
        } while(!Quit);
}

class Cirle
{
        protected:
            int Radius;
            int CenterX, CenterY;
        public:
        Circle(int , int , int );
        void Show();
        void Hide();
        void MoveTo(int , int );
        void Expand( int);
        void Contract( int )
         void Drag(int );
};
```

Các hàm thành phần của các lớp được định nghĩa trong File circle.cpp

```
// File circle.cpp
#include "circle.h"
#include <graphics.h>
#include <conio.h>
// Các hàm thành phần của lớp Circle
```

```
Circle::Circle(int InitX, int InitY, int InitRadius)
{
      CenterX = InitX;
      CenterY = InitY;
      Radius = InitRadius;
}
void Circle::Show()
{
      Visible = true;
      circle(X,Y,Radius);
}
void Circle::Hide()
{
      unsigned int TempColor;
      TempColor = getcolor();
      setcolor(getbkcolor());
      circle(X,Y,Radius);
      setcolor(Tempcolor);
}
void Circle::Expand(int ExpandBy)
{
      Hide();
      Radius +=ExpandBy;
      if(Radius <0)
      {
            Radius =0;
            Show();
```

```
    }
void Circle::Contract(int ContractBy)
{
    Expand(-ContractBy);
}
void Circle::MoveTo(int NewX, int NewY)
{
    Hide();
    X = NewX;
    Y = NewY;
    Show();
}
void Circle::Drag(int DragBy)
{
    int DeltaX, int DeltaY;
    int FigureX, FigureY;
    Show();
    FigureX = GetX();
    FigureY = GetY();
    while(GetDelta(DeltaqX, DeltaY))
    {
        FigureX += (DeltaX*DragBy);
        FigureY += (DeltaYTDragBy);
        MoveTo(FigureX, FigureY);
    }
}
```

```
#include "circle.h"
#include<graphics.h>
#include<conio.h>
int main()
    int graphdriver = DETECT;
    int graphmode;
    initgraph(&graphdriver,&graphmode,"c:\\borland\\bgi")
    Circle ACircle(151,82,50);
    // Gõ các phím →,←,↑,↓ để di chuyển các đối tượng
    // Gõ Esc thoát khỏi chương trình
    ACircle.Drag(5);
    ACircle.Hide( );
    ACircle.Show( );
    ACircle.Drag(10)
    closegraph( );
    return 0;
}
```

*** Constructor sao chép (Copy Constructor)**

Constructor sao chép được sử dụng khi cần khởi tạo một đối tượng mới và sao chép dữ liệu từ một đối tượng đã tồn tại sang đối tượng này. Constructor chỉ có một đối số, đó là tham chiếu đến một đối tượng đã tồn tại của cùng một lớp.

Ví dụ 7.8

```
Counter
{
        long in Count;
```

```
    public:
        // Các Constructor
        Counter(Counter&); // Constructor sao chép
};
```

Định nghĩa Constructor sao chép

```
Counter::Counter(Counter& Reference)
{
    Count = Reference.Count
}
```

Constructor sao chép có thể sử dụng qua ví dụ sau:

```
void main()
{
    Counter Object(5); // Sử dụng Constructor
    Counter Object1 = Object; //Sử dụng Copy Constructor
    }
```

Trong hàm *main()*, dấu = được sử dụng trong lệnh *Counter Object1 = Object* có tác dụng chỉ cho trình biên dịch hãy sử dụng *constructor* sao chép giữa hai đối tượng *Object* và *Object1*. Constructor sao chép có ý nghĩa rất quan trọng bởi vì nếu không có *constructor* kiểu này thì trình biên dịch sẽ không có khả năng thực hiện các thao tác sao chép cho một đối tượng mới từ một đối tượng đã có sẵn.

Một lớp có thể có nhiều *constructor* với điều kiện các *constructor* này phải có số lượng hoặc kiểu của các tham số khác nhau. Việc cho phép định nghĩa nhiều constructor sẽ rất thuận tiện khi cần tạo ra các đối tượng với các mục đích sử dụng khác nhau và điều này rất hay được sử dụng trong kỹ thuật lập trình. Để minh hoạ cho điều này, hãy xem xét một định nghĩa tương

đối đơn giản của lớp String - lớp làm việc với ký tự kiểu chuỗi. Trong ví dụ có sử dụng cả constructor ngầm định, với tham số truyền và constructor sao chép.

```
#include <iostream.h>
#include <string.n>
class String
{
    char* char_ptr; // Con trỏ chỉ đến xâu ký tự
    int length;     // Độ dài của xâu
public:
    String(char*text);    // (1) Constructor khởi tạo với tham số truyền
    String(int size =80); // (2) Constructor khởi tạo chuỗi rỗng với độ dài
                             ngầm định
     String(String& Other_String); //  (3) Constructor sao chép
     ~String() { delete char_ptr;) // Destructor sẽ được giới thiệu ở
                                      phần sau
     void Show( );

};
String::String(char* text)
{
    length = strlen(text);
    char_ptr = new char [length+1];
    strcpy(char_ptr, text);
}
String::String(int size)
{
```

đối đơn giản của lớp String - lớp làm việc với ký tự kiểu chuỗi. Trong ví dụ có sử dụng cả constructor ngầm định, với tham số truyền và constructor sao chép.

```
#include <iostream.h>
#include <string.n>
class String
{
    char* char_ptr; // Con trỏ chỉ đến xâu ký tự
    int length;     // Độ dài của xâu
public:
    String(char*text);    // (1) Constructor khởi tạo với tham số truyền
    String(int size =80); // (2) Constructor khởi tạo chuỗi rỗng với độ dài
                             ngầm định
     String(String& Other_String); //  (3) Constructor sao chép
     ~String() { delete char_ptr;) // Destructor sẽ được giới thiệu ở
                                      phần sau
     void Show( );

};
String::String(char* text)
{
    length = strlen(text);
    char_ptr = new char [length+1];
    strcpy(char_ptr, text);
}
String::String(int size)
{
```

```
        length = size;
        char_ptr = new char [length+1];
        *char_ptr = ' ';
    }
    String::String(String& Other_String)
    {
        length = Other_String.length;
        char_ptr = new char[length+1];
        strcpy(char_prt, Other_String.char_ptr);
    }
        void String::Show(void)
        {
        cout << char_ptr << endl;
        }
```

Lớp String có thể được sử dụng khi làm việc với dữ liệu kiểu chuỗi qua ví dụ dưới đây:

```
    void main()
    {
        String AString("Dương Tử Cường"); //Sử dụng constructor (1)
        AString.Show(); // Hiển thị đối tượng
        String BString; // Sử dụng constructor (2)
        CString.Show();
        String CString = AString; // Sử dụng constructor (3)
        AString.Show();
    }
```

*** Private constructor**

Thông thường *constructor* của một lớp thường được khai báo ở phần *public* của lớp đó. Tuy vậy trong một vài trường hợp, *constructor* cũng có thể được khai báo ở phần *private*. Khi đó người lập trình sẽ không được phép tạo một đối tượng từ lớp đó ngoại trừ trong các trường hợp khi các điều kiện sau đây được đảm bảo trước khi đối tượng được tạo:

- Thành phần tĩnh của lớp gọi *constructor*.
- Lớp bạn của lớp đó gọi *constructor*.
- Một đối tượng đã tồn tại của lớp có hàm thành phần cho phép tạo ra một đối tượng mới qua việc gọi *constructor*.

Như vậy từ một lớp với *private constructor* ta sẽ không được phép tạo ra các đối tượng thuộc kiểu *static, global* hoặc *auto*. Ví dụ dưới đây sẽ minh hoạ cho việc sử dụng sai constructor.

Ví dụ 7.9

```
class Counter{
 counter(); //constructor thuộc kiểu private
 };
// Tạo đối tượng chung
    Counter Object; //Sai
// Tạo đối tượng kiểu static
    static Counter counter; //sai
// Tạo đối tượng kiểu Auto
    void main()
    {
        Counter object; //sai
    }
Counter
```

```
{
        long count;
    public:
        Counter(long i=0){ count =i} ;
        Counter(long);
};
```

7.3.1.2. Destructor

Trong khi *constructor* của một lớp được dùng để khởi tạo đối tượng của lớp đó thì *destructor* lại được sử dụng để thực hiện một loạt tác động khi đối tượng được loại bỏ. Khác với *constructor*, *destructor* sẽ tự động được gọi khi một đối tượng bị loại bỏ. Như vậy có thể thấy thông thường *destructor* thực hiện các công việc ngược lại của *constructor*. Chẳng hạn khi làm việc với File, trong khi *constructor* chứa các lệnh để mở File thì *destructor* chứa các lệnh đóng File, khi làm việc với các biến thuộc kiểu con trỏ, *constructor* l có tác dụng cấp phát bộ nhớ trong khi destructor lại có vai trò giải phóng bộ nhớ đã được cấp phát cho các biến này khi các đối tượng bị loại bỏ.

Khi sử dụng *destructor*, cần lưu ý một số điểm dưới đây:

- Mỗi lớp chỉ có một *destructor* duy nhất.
- Tên của *destructor* giống như tên của lớp và được bắt đầu bởi ký tự ~.
- Không được phép truyền tham số cho *destructor*.
- Destructor không có kiểu trả về, thậm chí là kiểu void;
- Destructor có thể được khai báo *virtual* trong khi

constructor không thể là *virtual* (Xem Virtual function ở chương IX).

*** Public destructor**

Destructor của một lớp thông thường được khai báo trong vùng *public* của lớp để cho phép tất cả các đối tượng của lớp có thể truy nhập đến nó. Để minh hoạ cho việc sử dụng Desstructor hãy xét định nghĩa tương ứng trong lớp String. Destructor này sẽ có tác dụng giải phóng phần bộ nhớ đã được cấp phát cho con trỏ char* char_ptr và được viết như sau:

```
~String()
{
        delete char_ptr;
}
```

Destructor này sẽ tự động được gọi khi đối tượng thuộc lớp String bị loại bỏ. Trong ví dụ về lớp String, tất cả các đối tượng của lớp này chỉ bị loại bỏ khi chương trình tương ứng kết thúc.

Một đối tượng của một lớp có thể bị loại bỏ mà không cần đợi đến khi chương trình kết thúc. Để làm được điều này các đối tượng của lớp cần phải được khai báo theo kiểu con trỏ và khi cần thiết có thể xoá bỏ các đối tượng này bằng câu lệnh delete.

Đối với lớp String, hàm *main* trong trường hợp này có thể được viết như sau:

```
void main()
{
String *AString = new String("Dương Tử Cường"); // Sử dụng constructor (1)
```

```
    AString->Show();   // Hiển thị đối tượng
String *BString = new String;  // Sử dụng constructor (2)
    BString->Show();
    delete AString; // Xoá đối tượng AString
    delete BString; // Xóa đối tượng BString.
   }
```

*** Private destructor**

Mặc dù *destructor* thường được khai báo theo kiểu public nhưng C++ vẫn cho phép người lập trình khai báo *destructor* trong vùng *private* của một lớp. Khi *destructor* của một lớp được khai báo theo kiểu *pivate* thì không thể có bất cứ đối tượng nào thuộc kiểu *auto, static* hoặc toàn cục của lớp được thể hiện bởi vì khi đó các đối tượng này không thể bị loại bỏ khi kết thúc công việc. Thông thường khi *destructor* được khai báo theo kiểu *private* thì các đối tượng của lớp sẽ được cấp phát bộ nhớ động khi thực hiện chương trình. Trong C++, bất cứ một đối tượng nào được cấp phát bộ nhớ động đều cần được giải phóng khỏi bộ nhớ này khi thoát khỏi chương trình về Dos. Tuy vậy các đối tượng kiểu này lại không thể bị xóa nếu sử dụng toán tử *delete*. Để có thể xóa đối tượng, có thể thực hiện bằng một trong những cách sau:

- Khai báo một hàm thành phần kiểu public dùng để xóa đối tượng.
- Khai báo một lớp bạn (friend class - xem phần 3.4.7) để xóa đối tượng.

Dưới đây là một ví dụ về private destructor:

Ví dụ 7.10

```
class PrivateExample
```

```
{
    private:
        int* value;
        ~PrivateExample() { delete value} ;
    public:
        PrivateExample(){ value = new int[10];}
        void Delete() { delete this;}
};
static PrivateExample object;
    // Không thể tạo đối tượng kiểu static.
PrivateExample object2;
    // Không thể tạo đối tượng toàn cục;
void main()
{
    PrivateExample object3;
    // Không thể tạo đối tượng kiểu auto.
static PrivateExample object4; // Không được.
// Có thể tạo đối tượng theo cách sau:
PrivateExample* object5 = new PrivateExample;
delete object5; // Sai - Không thể xóa đối tượng bằng delete;
object->Delete(); // thực hiện tốt;
}
```

7.3.2. Toán tử gán

Cũng như *constructor* mặc định, toán tử gán sẽ được tự động phát sinh nếu ta không định nghĩa một toán tử gán đặc biệt cho lớp. Toán tử gán mặc định sẽ sao chép đối tượng gốc vào đối tượng đích theo từng bit một. Tuy vậy thường thì toán tử gán

mặc định cũng không có thể đáp ứng cho một lớp phức tạp. Trong các trường hợp này thông thường người sử dụng phải tạo ra các toán tử gán mới.

Có thể phân biệt toán tử gán và constructor sao chép bằng ví dụ dưới đây qua việc sử dụng lớp String.

String Dest;

String Src("Ngon ngu C");

Dest = Src; // Sử dụng toán tử gán cho đối tượng Dest đã tồn tại

String Dest1 = Src; //Sử dụng constructor sao chép cho đối tượng mới Dest1

Qua ví dụ đơn giản trên có thể thấy cả *constructor* sao chép và toán tử gán đều sao chép các thành phần của một đối tượng sang đối tượng khác. Tuy vậy điều khác biệt ở đây là ở chỗ với *constructor* sao chép ta có thể tạo ra một đối tượng mới trong khi toán tử gán chỉ thay đổi giá trị của một đối tượng đã có.

7.3.3. Sử dụng đối tượng như một thành phần dữ liệu

Như đã trình bày trong phần 7.3.7, thành phần dữ liệu (*data member*) của một lớp có thể là một đối tượng của một lớp khác. Đây là một trường hợp hay bắt gặp trong kỹ thuật lập trình vì vậy sẽ được xem xét kỹ hơn.

Ví dụ 7.11

```
class First
{
    public:
        int value;
        First(){ value =0;}
        int Gatvalue() { return value;}
```

```
};
class Second
{
        int id;
    public:
        First object;
        Second() { id=0;}
        int Getname() {return id;}
};
void main()
{
    First One;
    Second Two;
}
```

Trong ví dụ này, khi khai báo đối tượng *One* thuộc lớp *Fisrt*, *constructor* của lớp này sẽ được gọi. Điều cần lưu ý là *constructor* của lớp này cũng sẽ được gọi khi đối tượng *Two* của lớp *Second* được khai báo do đối tượng *object* của lớp *First* được khai báo trong *Second*. Khi *Two* được tạo, *constructor* của lớp này sẽ bị gọi, tuy vậy trước khi thực hiện các lệnh trong thân của *constructor* này, *costructor* của lớp *First* sẽ được thực hiện. Thứ tự thực hiện này luôn được trình biên dịch đảm bảo và cho phép đối tượng *Two* được khởi tạo sau đối tượng *First*.

Ví dụ dưới đây sẽ mô tả cho trường hợp khi một *constructor* của một lớp cần sử dụng một *constructor* của một lớp khác với các tham số truyền.

Ví dụ 7.12

```
class First
{
public:
      int value;
      First() { value=0;}
      First(int) { value =i;}
      int Getvalue() { return value;}
};
class Second
{
          int id;
      public:
          First object;
          Second() { id =0;}
          Second(int v):object(v){ id=v;}
          int Getname() { return id;}
};
void main()
{
       First One;
       Second Two;
       Second Thee(100);
}
```

Khi đối tượng *Three* của lớp *Second* được tạo trong hàm *main*, đầu tiên trình biên dịch sẽ gọi *constructor* của lớp *First* với giá trị khởi tạo của biến khác 0. Sau khi *constructor* của lớp *First* được gọi, các tham số của đối tượng *Three* sẽ được khởi tạo bằng chính *constructor* của lớp *Second*. Điều này được thực hiện bởi lệnh: *Second(int v): object(v){ id=v; }*

Chú ý rằng khi sử dụng phương pháp này, dấu : phải tồn tại giữa *constructor* của các lớp. Điều này sẽ đảm bảo rằng đối tượng của lớp *First* sẽ được khởi tạo trước khi thân của đối tượng thuộc lớp *Second* được thực hiện. Bằng cách này cũng có thể khởi tạo nhiều thành phần dữ liệu kiểu lớp. Chẳng hạn:

```
Biglass::Biglass(int w, char* title): class1(3),class2
                ("hello",title,5), class3(w)
{
....
....
}
```

Một điều cần lưu ý là khi thành phần dữ liệu của một lớp được khai báo theo kiểu con trỏ chỉ đến đối tượng thuộc một lớp khác, thì *constructor* của đối tượng thành phần này sẽ không tự động được gọi khi đối tượng của lớp chứa nó được tạo. Ví dụ sau sẽ minh hoạ cho điều này:

Ví dụ 7.13

```
class Box
{
    public:
        Box(int x,int y,int width,int height){ }
        void display();
};
class PushButton
{
        int state;
        Box* Ouline;
```

```
    public:
        PushBtton(int px,int py){ }
        void display();
};
void main()
{
    pushButton Object;// không gọi Constructor của Box:
}
```

Để minh hoạ một cách cụ thể cho việc sử dụng đối tượng của một lớp như dữ liệu thành phần của lớp khác, hãy xét một ví dụ đơn giản về lớp Line với hai dữ liệu thành phần là các đối tượng của lớp Point. Các đối tượng này đặc trưng cho hai điểm đầu và cuối của các đoạn thẳng thuộc lớp Line.

Ví dụ 7.14

```
#include <graphics.h>
class Point
{
    int x, y;
public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }
    Point (Point& P)
    {
        x = P.x;
        y = P.y;
```

```
    }
    int GetX( )
    {
        return x;
    }
    int GetY( )
    {
        return y;
    }

};
class Line
{
    Point Start;
    Point End;
public:
        Line(Point S, Point E): Start(S), End(E)
        {

        }

        void Show( )
        {
            line(S.GetX(), S.GetY(), E.GetX(), E.GetY());
        }
void main( )
{
    int gdriver = DETECT, gmode ;
    initgraph(&gdriver, &gmode," ");
```

```
    Line L(Point(50, 50), Point (100,100));
    Line. Show();
}
```

Trong hàm *main( )* ta đã khởi tạo một đoạn thẳng với toạ độ điểm đầu là (50,50), toạ độ điểm cuối là (100,100) và bằng cách gọi hàm *Show* đoạn thẳng này sẽ được hiển thị lên màn hình. Bạn đọc có thể tự xây dựng thêm các hàm thành phần khác cho lớp Line.

7.3.4. Sử dụng con trỏ this

Hãy xét định nghĩa của một lớp trong ví dụ dưới đây:

```
class Example{
    public:
        int value =2; // Khởi tạo giá trị sai
        Example(int v);
}
    // Định nghĩa Constructor
Example::Example(int v)
{
    value =v;
}
};
```

Trong định nghĩa này của lớp, biến value đã được khởi tạo với giá trị bằng 2. Điều này là không được phép vì lớp là một khái niệm trừu tượng và khi định nghĩa nó ta chỉ mới đưa ra các đặc tính chung của tất cả các đối tượng của lớp. Việc khởi tạo giá trị cho các dữ liệu thành phần chỉ đúng nếu ta đang đề cập đến một đối tượng cụ thể của lớp.

Hãy định nghĩa lại lớp Example và quan sát đoạn mã của construtor.

```
class Example
{
    public:
        int value;
        Example(int v) { value =v;}
};
void main()
{
    Examlpe Object(2);
    // Khởi tạo giá trị value =2 cho Object1
    Example Object(3);
    // Khởi tạo giá trị value =3 cho Object3
}
```

Trong đoạn mã của constructor, giá trị của biến *value* đã được khởi tạo bằng giá trị của tham số hình thức *v*. Điều này có vẻ mâu thuẫn với kết luận vừa nêu ra ở trên vì khi đó *value* sẽ được khởi tạo khi chưa có một đối tượng nào của lớp được thể hiện. Tuy vậy điều này lại hoàn toàn được phép. Trong trường hợp này, C++ đã sử dụng một con trỏ *this* chỉ đến một đối tượng ẩn của một lớp và qua con trỏ này C++ sẽ cho phép các hàm thành phần của đối tượng ẩn này truy nhập đến các biến thành phần của lớp đó. Khi một đối tượng của lớp được tạo ra thì đối tượng này sẽ thay thế cho đối tượng ẩn và sử dụng các hàm thành phần để truy nhập đến các dữ liệu thành phần của chính đối tượng này.

Sử dụng con trỏ *this*, constructor *Example* có thể được viết

lại một cách rõ ràng như sau:

```
Example::Example(int v)
{
    this-> value=v;
}
```

Trong hàm main khi đối tượng Object được khởi tạo bằng dòng lệnh Examlpe Object(2); thì constructor của lớp này sẽ tự động được gọi. Khi đó con trỏ *this* sẽ được thay thế bởi con trỏ chỉ đến đối tượng Object và giá trị value được khởi tạo trong constructor chính là giá trị của đối tượng này. Điều này cũng hoàn toàn đúng đối với việc truy cập đến các dữ liệu thành phần trong các hàm thành phần khác của lớp.

Trong kỹ thuật lập trình có thể nhận thấy con thỏ *this* ít khi được sử dụng một cách rõ ràng, ngoại trừ các trường hợp cần trỏ lại tham chiếu đến đối tượng đang làm việc (xem chương IX - Định nghĩa chồng các hàm & toán tử).

7.3.5. Từ khóa friend (bạn)

Như đã đề cập đến ở trên, khả năng truy nhập đến các thành phần dữ liệu của một lớp được chia làm nhiều mức khác nhau. Mức độ truy nhập này phụ thuộc vào từ khóa được dùng khi khai báo các thành phần dữ liệu. Theo cách khai báo này, các thành phần dữ liệu kiểu *private* chỉ được phép truy nhập thông qua các hàm thành phần của chính lớp đó. Tuy vậy trong một vài trường hợp, các loại dữ liệu kiểu *private* cũng cần được truy nhập từ bên ngoài lớp chứa nó. Để thực hiện điều này, ngôn ngữ C^{++} đưa ra một từ khóa mới - *friend*, từ khóa này được viết bên trong một lớp khi khai báo và qua đó chỉ ra một hàm bên ngoài lớp đó (hoặc thậm chí một lớp) có khả năng không hạn chế

trong việc truy nhập đến tất cả các loại dữ liệu thành phần của lớp này. Các hàm hoặc các lớp này được gọi là *friend function* (hàm bạn) hoặc *friend class* (lớp bạn). Trước hết hãy xét một ví dụ về một lớp bạn - friend:

```
class Node
{
    friend class ObjectList; // ObjectList là lớp bạn của Node
    int value;
    Node* predecessor;
    Node* successor'
public:
    void Value(int) { value =i;}
};
class ObjectList
{
private:
    Node* head;
    Node* tail;
    Node* current;
public:
    void InsertNode(Node*){ } ;
    void DeleteNode(Node*){ } ;
    int CurrentObject(Node* node) { return node->value;}
} ;
```

Qua ví dụ này có thể nhận thấy lớp *ObjectList* là một lớp bạn của lớp *Node*. Điều này được thực hiện thông qua việc sử dụng từ khóa *friend* trong khi khai báo lớp *Node*. Do là lớp bạn của *Node*, *ObjectList* có thể truy nhập đến các dữ liệu thành

phần thuộc kiểu private của lớp này một cách không hạn chế.

Từ khóa *friend* có thể được sử dụng cho hàm. Khi một hàm được khai báo với từ khóa *friend* bên trong một lớp, hàm đó cũng có thể truy nhập đến tất cả các dữ liệu của lớp mà không quan tâm đến mức độ truy nhập của chúng. Hãy xét ví dụ sau:

```
class Node
{
    friend in GetObject(Node*);
    // GetObject là hàm bạn của Node
    int Value;
    Node* predecessor:
public:
        void Value(int i) { value =i;}
};
    // Định nghĩa hàm bạn
    GetObject(Node* n)
    {
        return->value;
        // Truy nhập đến dữ liệu kiểu private của Node;
    }
```

Khi sử dụng từ khóa *friend* cho một hàm, nếu hàm đó lại là một hàm thành phần của một lớp khác, phải dùng toán tử phạm vi khi khai báo *friend* để chỉ rõ tên của lớp chứa hàm. Chẳng hạn trong ví dụ trên, nếu *GetObject* là hàm thành phần của lớp *GetObject*, cần phải sử dụng việc khai báo sau bên trong lớp *Node:*

friend int *GetList*::*GetObject*(Node*);

Khi sử dụng từ khóa *friend*, phải lưu ý đến một số vấn đề sau:

- Quan hệ *friend* không có tính kế thừa. Điều đó có nghĩa là nếu lớp *A* là lớp bạn của *B* và lớp *C* được dẫn xuất từ *A* thì không có nghĩa là *C* là bạn của *B*.
- Quan hệ *friend* không có tính bắc cầu. Như vậy, nếu lớp *A* là bạn của lớp *B* và lớp *B* lại là bạn của lớp *C* thì điều khẳng định *A* là bạn của *C* chưa chắc là đúng. Điều này cũng giống như trên thực tế: bạn của bạn mình chưa chắc đã là bạn của mình.

7.3.6. Hàm thành phần kiểu static

Việc khai báo một thành phần kiểu *static* rất giống với việc khai báo một thành phần dữ liệu kiểu *static*. Hàm thành phần kiểu *static* có một số khác biệt so với các hàm thành phần bình thường:

- Các hàm thành phần kiểu *static* có thể được sử dụng thông qua tên của lớp mà không cần tham chiếu đến một đối tượng cụ thể nào của lớp. Lý do là các hàm kiểu này là duy nhất và được dùng chúng cho tất cả các đối tượng của lớp đó.
- Các hàm thành phần kiểu *static* không có con trỏ *this*, do vậy chúng không thể truy nhập đến các dữ liệu thành phần trừ phi chúng được truyền con trỏ *this* một cách rõ ràng.
- Các hàm thành phần kiểu *static* trong C++ có cấp lưu trữ *external* trong quá trình liên kết trong khi cấp lưu trữ

của các hàm *static* trong C là *internal*. Khi một hàm được khai báo là *static* thì tất cả các biến của nó cũng thuộc loại *static*. Thông thường các hàm kiểu *static* được dùng để xử lý các công việc chung cho tất cả các đối tượng của một lớp.

Dưới đây là một số ví dụ về việc sử dụng hàm thành phần kiểu static.

Ví dụ 7.15

```
class Simple{
    private:
        int v1,v2,v3;
    public:
        static void sum();
}
void Simple::sum()
{
    v1 = v2+v3;
}
void main()
{
    Simple s1;
    s1.sum();
    Simple::sum();
}
```

Trong hàm *main*, *sum* được gọi theo hai cách: qua đối tượng *s1* của lớp *Simple* và qua chính tên của lớp này. Cả hai cách gọi này đều cho phép trong C++. Tuy vậy, trong chương trình, hàm

sum l[illegible] được định nghĩa sai. Nguyên nhân là do hàm *sum()* được khai bao theo kiểu *static* do đó việc sử dụng con trỏ *this* để truy nhập đến các biến của lớp theo cách *v1 = v2+v3;* trong thân hàm là không hợp lệ.

Để *sum* có thể truy nhập đến các biến *v1,v2* hoặc *v3* chúng ta có thể sử dụng hai cách sau:

- Khai báo v1,v2,v3 là các biến toàn cục.
- Truyền con trỏ *this* cho hàm một cách rõ ràng, trong trường hợp này *sum* được định nghĩa lại như sau:

```
void Simple::sum(Simple* s)
{
    s->v1 = s->+s->v3;
}
```

khi đó trong thân của lớp, *sum* sẽ được khai báo theo cách:

```
class Simple
{
    int v1,v2,v3;
public:
    static void sum(Simple*);
};
```

và việc gọi *sum* trong hàm *main* được gọi qua cách sau:

```
void main()
{
    Simple s1;
    s1.sum(&s1);
}
```

7.3.7. Hàm thành phần với con trỏ this kiểu const

Các hàm thành phần được khai báo với từ khóa *const* là một kiểu hàm mới trong C++. Từ *const* trong các trường hợp này không chỉ đến giá trị được trả lại từ hàm (kiểu của hàm) mà được sử dụng cho con trỏ *this* trong thân hàm.

Việc khai báo hàm thành phần với từ *const* có thể được thực hiện theo cách sau:

```
class Example
{
        int value;
    public:
        int GetValue()const;
        int RedValue() const { return value;}
};
int Example::Getvalue() const
{
    return value;}
```

Từ khóa *const* được viết khi khai báo hàm phải được viết lại khi định nghĩa hàm. Nếu điều này không được thực hiện thì trình biên dịch có thể nhầm lẫn rằng ta đang định nghĩa một hàm khác. Trong ví dụ trên, trình biên dịch có thể đưa ra thông báo lỗi sau:

"GetValue() is not a member of Example"

Việc khai báo một hàm với từ khóa *const* có mục đích gì? Bằng cách khai báo này ta sẽ làm việc với con trỏ *this* được truyền đến hàm thành phần. Thông thường C++ truyền con trỏ ẩn *this* cho các hàm thành phần không phải *static*. Các hàm thành phần đó về sau có thể sử dụng con trỏ *this* để truy nhập

đến các dữ liệu thành phần.

Có thể quay lại minh họa cho việc sử dụng con trỏ *this* qua ví dụ sau:

```
class Example
{
    int value;
public:
    GetValue(int x)
    {
        value =x; // Dùng con trỏ this để gán giá trị
    }
};
```

Khi khai báo lớp *Example* thì chưa có một đối tượng cụ thể nào thuộc kiểu *Example* được khởi tạo. Tuy vậy, trong hàm *GetValue()* ta vẫn có thể gán giá trị cho biến *value* của một đối tượng chưa được khởi tạo qua con trỏ *this*.

Trên thực tế, các hàm thành phần thuộc lớp *Example* (các hàm này không được khai báo bởi từ khóa *const*) có sử dụng khai báo ngầm định sau cho *this:*

```
Example *const this;
```

Khai báo trên đưa ra con trỏ *const this* tham chiếu đến các đối tượng không thuộc kiểu *const* thuộc lớp *Example* và có thể dùng con trỏ *this* này để truy nhập đến các dữ liệu thành phần.

Như vậy một hàm thành phần bình thường *normal* của lớp *Example* sẽ được mô tả bằng cách sau:

```
class Example
{
```

```
public:
        int normal(Example *const this);
};
```

Thực ra, đây chỉ là mô tả cho việc sử dụng con trỏ *this* của trình biên dịch. Mọi sự khai báo một cách rõ ràng theo kiểu trên trong chương trình sẽ phát sinh lỗi khi biên dịch.

Việc sử dụng từ khóa *const* trước thân của các hàm thành phần như trong các trường hợp của các hàm số *GetValue()* và *ReadValue()* sẽ cho phép ta thay đổi kiểu của con trỏ *this* được truyền cho các hàm thành phần. Khi đó trình biên dịch sẽ sử dụng việc khai báo sau:

```
cons Example *const this;
```

Như vậy trong các trường hợp này, con trỏ *this* đã được định nghĩa để tham chiếu đến các đối tượng thuộc kiểu *const* của lớp *Example*. Kết quả của việc khai báo này là tất cả các hàm thành phần đó không được phép thực hiện bất cứ một sự thay đổi nào trên các thành phần dữ liệu của lớp. Khả năng này trên thực tế hay được sử dụng khi các hàm trong lớp cơ sở được định nghĩa chồng trong các lớp dẫn xuất. Trong các trường hợp này, việc khai báo các hàm thành phần với từ khóa *const* sẽ không cho phép bất cứ một hàm nào đó được định nghĩa chồng có khả năng thay đổi các dữ liệu thành phần của lớp cơ sở. Các thư viện lớp của *Borland* C++ rất hay sử dụng việc khai báo này. Một điểm cần lưu ý là việc khai báo các hàm *static* với từ khóa *const* sẽ không có ý nghĩa. Ví dụ:

```
class Example
{
public:
```

```
    int value;
        static int GetValue() const { return value} ;
};
```

Nguyên nhân của vấn đề trên là các hàm thành phần thuộc kiểu *static* không được truyền con trỏ ẩn *this* khi được gọi, trong khi mục đích của việc sử dụng từ khóa *const* trong các trường hợp này là thay đổi kiểu của con trỏ *this*. Tuy vậy Borland C^{++} cũng không than phiền về cú pháp trên, thậm chí cũng không đưa ra bất cứ thông báo nào khi biên dịch.

7.3.8. Con trỏ chỉ đến hàm thành phần

Con trỏ chỉ đến hàm là một khái niệm đã được sử dụng tương đối rộng rãi trong ngôn ngữ C truyền thống. Sử dụng con trỏ chỉ đến hàm, ta có thể gọi hàm một cách gián tiếp hoặc sử dụng hàm như tham số truyền của các hàm khác.

7.3.8.1. Con trỏ chỉ đến hàm không có tham số truyền

Ví dụ sau sẽ mô tả cho việc sử dụng con trỏ chỉ đến hàm không có tham số truyền:

Ví dụ 7.16

```
class Example
{
        long value;
        int time;
    public:
        long get_value() { return value;}
        long get_time() { return time+ value;}
```

```
void main()
{
    long (Example::*fp)()= &Example::get_value;
    Example e;
    // Gọi Example::get_value() qua con trỏ chỉ đến hàm
    long v = (e.*fp)();
    // Gọi Example::get_time() qua con trỏ chỉ đến hàm
    fp = &Example::get_time;
    long t = (e.*fp)();
}
```

7.3.8.2. Con trỏ chỉ đến hàm với tham số truyền

Ví dụ 7.17

```
#include<string.h>
class Example
{
      long value;
      char name[10];
public:
      long set_value(long);
      char* set_name(char*);
};
long Example::set_value(long v)
{
      value = v;
      return value;
}
char* Example::set_name(char* string)
{
      if(strlen(string)<10)
```

```
        stcpy(name,string);
    return name;
}
void main()
{
    // Định nghĩa con trỏ chỉ đến hàm có tham số
    long (Example::*fp)(long) = &Example::set_value;
    Example e;
    // Gọi Example::set_value() qua con trỏ chỉ đến hàm
    long new_value =5;
    long v=e.*fp)(new_value);
    // Gọi Example::set_name() qua con trỏ chỉ đến hàm
    char* (Example::*fp1)(char*)=&Example::set_name;
    char* id = "Borland C++";
    char* new_name = (e.*fp1)(id);
    new_name = (e.*fp1)("Borland");
}
```

Con trỏ chỉ đến một hàm có thể được gọi thông qua con trỏ chỉ đến đối tượng của lớp có chứa hàm đó. Trong trường hợp này ta cần sử dụng toán tử ->.

Ví dụ 7.18

```
void main()
{
    // Khai báo đối tượng
    Example example;
    // Khai báo con trỏ chỉ đến đối tượng
```

```
Example* e = &example;
// Sử dụng con trỏ chỉ đến đối tượng
long (Example::*fp)(long) = &Example::set_value;
long v = (e->*fp)(103);
```

7.3.8.3. Con trỏ chỉ đến hàm thành phần kiểu static

Trong khi việc gọi các hàm không phải kiểu *static* luôn phải gắn với một đối tượng cụ thể thông qua việc sử dụng giá trị của đối tượng, tham chiếu hoặc con trỏ chỉ đến đối tượng, các hàm được khai báo với kiểu *static* không thể gọi thông qua tham chiếu hoặc con trỏ chỉ đến đối tượng.

Ví dụ 7.19

```
class StaticExample
{
    public:
        static int foo();
        static int woo();
};
    // Định nghĩa foo()
int StaticExample::foo()
{
    return value;
}
    // Định nghĩa woo()
int StaticExample::woo()
{
return 3;
}
```

```
void main()
{
    int (*fp)() = &StaticExample::foo;
    (*fp)();
    fp = StaticExample::woo;
    (*fp)();
}
```

Trong phần này, tuy vậy chỉ đề cập đến việc khai báo và sử dụng con trỏ chỉ đến các hàm thành phần trong hàm chính (*main*). Việc sử dụng con trỏ chỉ đến các hàm trên thực tế lại có ý nghĩa rất quan trọng khi sử dụng con trỏ này như là tham số truyền cho một hàm khác. Qua cách truyền này, người sử dụng có thể chọn các hàm khác nhau để thực hiện chúng tùy theo từng điều kiện được đưa ra. Để có thể sử dụng con trỏ chỉ đến hàm bạn đọc có thể xem lại các tài liệu viết về ngôn ngữ lập trình C.

7.3.9. Mảng và lớp

Ngoài các kiểu mảng được sử dụng rong ANSI C, C++ còn sử dụng một số kiểu mảng mới phát sinh từ việc sử dụng lớp. Các kiểu mảng này bao gồm:

- Mảng các đối tượng của lớp
- Mảng các con trỏ chỉ đến các đối tượng của lớp
- Mảng các thành phần dữ liệu của đối tượng
- Mảng các con trỏ của các thành phần dữ liệu của lớp
- Mảng các con trỏ của các hàm thành phần của lớp
- Mảng các con trỏ chỉ đến các dữ liệu thành phần kiểu *static*

Dưới đây sẽ lần lượt nghiên cứu một cách ngắn gọn các kiểu mảng này

7.3.9.1. Mảng các đối tượng của lớp

Cũng như các kiểu dữ liệu chuẩn, một khi ta có khả năng khai báo đối tượng của một lớp thì ta cũng có thể khai báo mảng các đối tượng. Việc khai báo này được thực hiện giống như trường hợp mảng của cấu trúc.

Dưới đây sẽ mô tả ví dụ về việc sử dụng mảng của các đối tượng trong chương trình.

Ví dụ 7.20

```
#include <stdio.h>
class Value {
    int value;
public:
    Value( ){ value =0;}
    Value (int v) { value =v;}
    int GetValue( ) {return value;}
    void SetValue(int v) { value =v;}
};
Value bills [10]; // Mảng các đối tượng của lớp Value
Value coins[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
void main( )
{
    for(int i =0; i <10; i++)
```

```
        printf("\n bills [%d] = %d", i, bills[i].GetValue( ) );
    for(i =0; i <10; i++)
        printf("\n coins [%d] = %d", i, coins[i].GetValue( ) );
}
```

Đối với mỗi thành phần của mảng đã được tạo ra, trình biên dịch sẽ gọi constructor tương ứng của lớp. Nếu constructor như vậy không được tìm thấy, lỗi sẽ được phát sinh.

Trong chương trình trên, mảng bills không được khởi tạo giá trị và điều này sẽ buộc trình biên dịch phải gọi *constructor* ngầm định của lớp *Value* đối với mỗi thành phần của mảng. Mảng *coins* được khởi tạo khi định nghĩa và do vậy trình biên dịch sẽ gọi constructor

```
Value::Value(int v)
```

đối với mỗi thành phần của mảng này. Việc truy cập đến các dữ liệu và hàm thành phần của mảng cũng được thực hiện giống như trường hợp cấu trúc:

```
printf("\n bills [%d] = %d", i, bills[i].GetValue( ) );
```

7.3.9.2. Mảng các con trỏ chỉ đến các đối tượng của lớp

Mặc dù các mảng kiểu này có rắc rối hơn một chút so với mảng các đối tượng của lớp, nhưng cú pháp khai báo của chúng cũng tương đối đơn giản. Hãy xét ví dụ đối với lớp Value đã mô tả ở trên.

```
#include <stdio.h>
// Định nghĩa lại lớp Value ở đây
Value* bills[3] = {
```

```
        new Value (0),
        new Value (1),
        new Value (2)
};
Value* coins [3];
void main( )
{
    for(int i =0; i < 3; i++)
        printf("\n bills [%d] = %d", i, bills[i].GetValue( ) );
    for( i = 0; i < 3; i++) {
        // Khởi tạo các con trỏ
        coins[i] = new Value;
        // Sử dụng các phần tử của mảng
        printf("\n coins [%d] = %d, i, coins[i].GetValue( ) );
            delete coins[i];
    }
}
```

Trong chương trình hãy chú ý đến cách mảng *bills* được khởi tạo trực tiếp qua các đối tượng được phân bố bộ nhớ động. Trong trường hợp này trình biên dịch sẽ tự động gọi constructor

Value::Value(int)

đối với mỗi thành phần của bills. Mảng *coins* không được khởi tạo khi khai báo do vậy không có constructor nào được gọi đối với mảng này. Trong hàm main ( ) cũng mô tả cách sử dụng con trỏ để truy nhập đến các hàm thành phần.

7.3.9.3 Mảng các thành phần dữ liệu của đối tượng

Mảng các thành phần dữ liệu của đối tượng cũng giống như các mảng truyền thống trong ANSI C. Hãy xét một ví dụ dưới đây:

Ví dụ 7.21

```
class SomeData{
public:
        int data;
        SomeData( ) {data = 0;}
        SomeData( int d) { data = d;}
};
SomeData d1(15), d2;
int Stuff [ 3] = {
    d1.data, d2.data, 12
};
```

Mảng *Stuff* chỉ đơn giản là mảng các số nguyên. Trong chương trình này, hai đối tượng của lớp *SomeData* đã được khai báo và khởi tạo và sau đó được sử dụng trong việc khởi tạo mảng *Stuff* với các giá trị nguyên. Sau khi khởi tạo, mảng *Stuff* có chứa các giá trị {15, 0, 20} và có thể được điều khiển như mảng các số nguyên.

7.3.9.4 Mảng các con trỏ chỉ đến các thành phần dữ liệu của lớp

Loại mảng này cũng có lợi trên thực tế bởi vì nó đưa ra khả

năng tham chiếu một cách đầy đủ đến các biến trong lớp. Ví dụ dưới đây sẽ mô tả cách sử dụng các con trỏ kiểu này

Ví dụ 7.22

```
class X
{
public:
    int value;
    int total;
    int count;
    X(int i) { value = i; total = count =0;}
    X( ) {value = total = count =0;}
};
// Định nghĩa và khởi tạo mảng các con trỏ
// chỉ đến các thành phần dữ liệu của lớp
int X::*variable [ ] = {
    &X:: value,
    &X:: total,
    &X:: count,
};
// Tạo một vài đối tượng toàn cục của lớp
X a(1), b(304);
// Sử dụng con trỏ chỉ đến thành phần dữ liệu của lớp;
void InCrementMember ( X* object, int X:: *Member)
{
```

```
    object->*Member =3;
}
void main( )
{
    // Sử dụng mảng các con trỏ của các đối tượng toàn cục
    InCrementMember(&a, variable [0] );
    cout << a.value <<'\n';
    cout << a.total <<'\n';
    cout<<a.count;
    InCrementMember(&b, variable [1] );
    // Tạo biến tự động
    X* c = new X(-13);
    // Sử dụng mảng các con trỏ với đối tượng tự động
    InCrementMember(c, variable [2]);
}
```

Cần lưu ý là việc phát sinh và khởi tạo mảng các con trỏ chỉ đến các thành phần dữ liệu của lớp phải được thực hiện bên ngoài thân của lớp. Trong trường hợp này *variable [0] = &X::value, variable [1] = &X::total, variable [2] = &X::count* và kết quả sau khi thực hiện chương trình sẽ là

```
3
0
0
```

trong đó 3 là giá trị của *value*, 0 là giá trị của total và *count* của đối tượng *a*.

7.3.9.5 Mảng các con trỏ chỉ đến các hàm thành phần của lớp

Về bản chất tự nhiên con trỏ chỉ đến hàm thành phần của lớp cũng giống như con trỏ chỉ đến thành phần dữ liệu của lớp. Ví dụ dưới đây sẽ mô tả cách sử dụng mảng của loại con trỏ này.

Ví dụ 7.23

```
#include <iostream.h>
class X{
public:
    int value;
    int total;
    int cout;
    void setValue(int v) {value = v;}
    void SetTotal(int t) {total = t;}
    void SetCount(int c) {count =c;}
};
// Định nghĩa và khởi tạo mảng các con trỏ
// chỉ đến hàm thành phần của lớp
typedef void (X::*FP) (int);
FP function [ ]= {
    &X::SetValue,
    &X::SetTotal,
    &X::SetCount
};
```

```
// Tạo đối tượng toàn cục
X a;
// Sử dụng con trỏ chỉ đến hàm thành phần của lớp
void Setmember(void (X::*foo)(int), int v)
{
    (a.*foo)(v);
}
void main( )
{
    // Sử dụng mảng các con trỏ với đối tượng toàn cục
    SetMember(&X::Setvalue,5);
    cout<< a.value<<endl;
    SetMember(&X::SetTotal, -13);
    cout<<a.total<<endl;
    int start = 10;
    SetMember(&X::SetCount, start);
    cout <<a.count;
}
```

Mảng các con trỏ chỉ đến các hàm thành phần của lớp có thể được khai báo mà không cần sử dụng *typedef* bằng cách sau:

```
// Định nghĩa mảng không sử dụng typedef
void (X::*table[ ])(int) = {
    &X::SetValue,
    &X::SetTotal,
```

```
    &X::SetCount
};
```

Trong ví dụ này hàm *SetMember* được sử dụng để gọi một cách gián tiếp đến các hàm thành phần khác nhau của lớp *X*. Tham số thứ nhất của hàm là một số nguyên biểu diễn độ dịch (offset) của hàm trong bảng các con trỏ hàm trong lớp *X*. Thông qua số nguyên này (địa chỉ), hàm *SetMember* sẽ gọi chính xác đến các hàm thành phần của lớp X. Trong hàm *main* thông qua hàm *SetMember* các hàm *SetValue*, *SetTotal* và *SetCount* của lớp sẽ lần được được đối tượng *a* gọi đến để thiết lập giá trị cho các dữ liệu thành phần của nó là *value, total* và *count*. Kết quả hiển thị sau khi thực hiện chương trình sẽ là:

```
5
-13
10
```

tương ứng với các giá trị value, total và count của đối tượng a.

7.3.9.6. Mảng các con trỏ của dữ liệu thành phần kiểu static

Các dữ liệu thành phần kiểu static của lớp cũng giống như các biến static của ANSI C chuẩn. Điều này cũng tương tự đối với con trỏ chỉ đến dữ liệu thành phần kiểu static. Ví dụ dưới đây sẽ mô tả cho điều khẳng định này.

Ví dụ 7.24

```
class C{
public:
```

```
    static int count;
};
class T{
public:
    static int value;
};
class U{
public:
    static int total;
};
// Định nghĩa các biến tĩnh sau khi khai báo chúng
int S::count =1;
int T::value = 2;
int U::total = 3;
int *static_integer [ ] = {
    &S::count,
    &T::value,
    &U::total
};

void SetVariable(int* d©t, int value)
{
    *data = value;
}

void main( )
```

```
{
    SetVariable(&S::count, 18);
    cout << "S::count = " <<S::count<<endl;
    SetVariable(static_integer [0], 54);
    cout << "S::count  = " <<S::count<<endl;
    SetVariable(static_integer [1], 0);
    cout << "T::value  = " <<T::count<<endl;
    SetVariable(static_integer [2], 13);
    cout << "U::total  = " <<U::total;
}
```

Hàm *SetVariable* được gọi với tham số là con trỏ chỉ đến thành phần dữ liệu static. Tuy vậy, như được mô tả trong chương trình, hàm này lại được định nghĩa để truy nhập đến con trỏ kiểu *int* đơn giản và vì vậy ta cũng có thể sử dụng nó để thay đổi bất cứ một biến nào có kiểu *int* được định nghĩa bên ngoài lớp. Kết quả thực hiện của chương trình này là:

```
S::count  = 18
S::count  = 54
T::value  = 0
U::Total  = 13
```

7.3.10. Lớp mẫu (class template)

Trong các phần trình bày trên đây, có thể nhận thấy rằng khi thiết lập một lớp nào đó, thông thường lập trình viên phải chỉ rõ kiểu các thành phần dữ liệu của lớp và do đó các hàm thành phần của lớp thường được thiết kế để làm việc với một số loại dữ liệu đã được xác định. Tuy vậy, trên thực tế có nhiều công việc được tiến hành trên các loại dữ liệu khác nhau nhưng

lại được xử lý theo những phương pháp giống nhau, chẳng hạn các hàm xử lý các công việc trên các danh sách liên kết (linked list) với các loại dữ liệu khác nhau như số nguyên, cấu trúc hoặc các loại dữ liệu khác đều có các loại dữ liệu khác nhau. Khi xây dựng chương trình để xử lý cùng một công việc trên các loại dữ liệu khác nhau, thay vào việc phải định nghĩa một số lớp khác nhau chứa các hàm riêng dành cho từng loại dữ liệu C++ cho phép khai báo một lớp duy nhất - *class template* với các loại dữ liệu chưa được xác định về kiểu (*generic type*). Lớp này sẽ được sử dụng trong quá trình biên dịch để làm việc với các loại dữ liệu cụ thể. Dưới đây hãy xét một ví dụ về việc khai báo một class template.

7.3.10.1. Class template với một biến thuộc kiểu chung

Ví dụ 7.25

```
template<class T> class List
{
    piblic:
        List();
        void add(T&);
        void remove(T&);
        void detach(T& t) {remove (t);}
        ~List()
};
// Định nghĩa các hàm thành phần.
template<clas T> List ::List()
{
    //.....
}
```

```
template<class T> void List    ::add(T&)
{
    //.....
template<class T> void List<T>::remove(T&)
{
    //.....
}
tempate<class T> List ::~List()
{
    //.....
}
```

Trong trường hợp các hàm của lớp chỉ xử lý trên một biến duy nhất, một *class template* khi khai báo luôn được bắt đầu bằng từ khóa *template* và được tiếp theo bằng *<class T>*. Ở đây *T* là loại dữ liệu được khai báo nhưng chưa xác định kiểu và sẽ được thay bằng các biến cụ thể khi khởi tạo đối tượng. Nếu biến *T* này bị bỏ qua khi khai báo *class template*, trình biên dịch Borland C++ cũng không than phiền về điều này ngoài việc có thể thông báo một nhắc nhở nào đó (warning message). Một điều cần lưu ý cần phải phân biệt hai khái niệm *class template* và *template class*. Thuật ngữ *class template* được dùng khi khai báo lớp trong khi thuật ngữ *template class* lại được dùng khi khởi tạo một đối tượng với một kiểu dữ liệu cụ thể. Việc tạo một *template class* của lớp *List* có thể được mô tả thông qua ví dụ sau:

```
void main()
{
```

```
    List<long> phone_numbers;
    List<char*> club_nembers;
    static long number = 123456;
    phone_numbers.add(number);
    static char* name = "Ví dụ";
    club_member.add(name);
}
```

Dễ dàng nhận thấy là trong hàm *main* có hai đối tượng của lớp với các kiểu dữ liệu khác nhau. *Phone_numbers* cho phép làm việc với dữ liệu thuộc loại *Long* trong khi *club_meimber* lại cho ta làm việc với dữ liệu thuộc kiểu char*.

Class template có thể được khai báo với nhiều tham số truyền. Điều này được mô tả thông qua ví dụ 7.27.

Ví dụ 7.26

```
#include<stdio.h>
template <class T, int i, char* cp> class Myclass
{
    T* object;
    char* title;
        int size, length;
    public:
        Myclass();
        Myclass(nit);
        int foo(char*)l
    };
    // Định nghĩa các hàm thành phần.
     template <class T, int i, char* cp>
```

```
Myclass<T,i,cp>::Myclass()
{
    // Sử dụng các biến được khai báo trong class template.
    title = cp;
    size = i;
    object = new T();
}
template <class T, int i, char* cp>
Myclass<T,i,cp>::Myclass(int x)
{
    length = i;
    title = cp;
    size = x;
    object = new T();
}
template <class T,int i, char* string>
Myclass<T,i,cp>::Foo(char* cp)
{
    return atoi(cp);
}
```

Class template *Myclass* được khai báo với 3 đối số thuộc kiểu *class T, int* và *char**. Cần lưu ý đây là các đối số của lớp chứ không phải là đối số của các hàm thành phần của lớp. Các đối số này chỉ được phép sử dụng bên trong các hàm thành phần của *class template*.

Class template *Myclass* có thể được sử dụng trong ví dụ dưới đây:

```
class List{ } ;
void main()
{
    Myclass<in,5,"Who"> x(2);
    Myclass<float,100,"What"> y(3);
    Myclass<List,100,"Where"> z();
}
```

7.3.10.2. Class template với nhiều biến thuộc kiểu chung (generic type)

Trong các ví dụ trên có thể nhận thấy việc khai báo *class template* chỉ được giới hạn với một biến thuộc kiểu *generic*. Trên thực tế, số lượng các biến này không hạn chế. Việc khai báo nhiều đối số thuộc kiểu *generic* không có gì phức tạp, vì vậy xem vấn đề này như một bài tập nhỏ dành cho bạn đọc.

7.3.11. Hàm mẫu (Function Template)

Khái niệm về *template* không chỉ được giới hạn bởi lớp mà còn có thể được sử dụng cho hàm. Việc sử dụng *function template* sẽ cho phép xây dựng nên các hàm có thể làm việc với các kiểu đối số truyền khác nhau tùy theo từng trường hợp cụ thể. Để tìm hiểu về khái niệm *function* chúng ta hãy xét ví dụ về việc tìm giá trị lớn nhất của hai số. Trong ngôn ngữ C, vấn đề này được giải quyết bằng hai phương pháp sử dụng *macro* và hàm.

Sử dụng *macro*, việc tìm giá trị lớn nhất của hai số có thể được thực hiện thông qua dòng lệnh : #define max(a,b) ((x>y) ? x: y). Việc sử dụng *macro* tuy đơn giản nhưng có một hạn chế là không cho phép trình biên dịch kiểm tra kiểu của các tham số. Điều này có thể gây ra phiền phức nếu sử dụng *macro* để tìm giá trị

lớn nhất của các kiểu dữ liệu không thích hợp (chẳng hạn kiểu *char*).

Việc tìm giá trị lớn nhất của hai số có thể được thực hiện thông qua việc định nghĩa một hàm:

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

Việc sử dụng hàm tuy vậy lại phát sinh một điều hạn chế là chỉ được phép làm việc với các kiểu dữ liệu đã được định nghĩa. Chẳng hạn không thể sử dụng hàm *int max(...)* để tìm giá trị lớn nhất của hai số thuộc kiểu *double*. Để thực hiện được điều này cần phải định nghĩa một hàm khác:

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

Ngoài việc tìm giá trị lớn nhất của hai số thuộc kiểu *double*, hàm *double max(...)* còn có thể sử dụng để tìm giá trị lớn nhất của hai số integer - các số có thể chuyển kiểu lên dạng *double*. Tuy vậy hàm này lại không cho phép làm việc với các loại dữ liệu khác như con trỏ, cấu trúc và đối tượng của lớp.

Để giải quyết các vấn đề này, C++ đưa ra một khái niệm mới: *function template*. Sử dụng *function template*, hàm *max* có thể được khai báo như sau:

```
template<class T> T max(T a, T b)
{
    return a>b? a:b;
}
```

Việc khai báo trong trường hợp này cũng được bắt đầu với từ khóa *template*, một từ khóa mới được sử dụng trong C++. Sau từ khóa *template*, với *<class T>* ta đã khai báo một biến chưa xác định kiểu (generic type), mà kiểu của nó sẽ được xác định trong quá trình biên dịch. Hàm *main* có sử dụng hai đối số *a* và *b* với kiểu *T* đã được khai báo trước và giá trị trả lại của hàm cũng là kiểu *T*. Như vậy với *template function*, hàm *max(...)* bây giờ đã có thể được sử dụng để tính giá trị lớn nhất của hai đại lượng cùng kiểu, các đại lượng này có thể là cấu trúc, con trỏ, các số thuộc dạng *integer*, thuộc loại *double* v.v... Function template đã được định nghĩa ở trên có thể được sử dụng ở ví dụ sau:

Ví dụ 7.27

```
template<class whatever>
whatever max(whatever z, whatever y)
{
    return x>y ? x:y;
    }
void main()
{
    cout << "Max(3,5) là"<< max(3,5) << endl;
    cout << "Max('3','5') là"<< max('3','5') << endl;
}
```

Khi một hàm kiểu *template* được gọi, trình biên dịch sẽ sinh ra một hàm cụ thể để làm việc với các loại dữ liệu đã được xác

định khi truyền tham số cho hàm *template*. Trong trường hợp này không có một phép chuyển kiểu nào cần phải thực hiện. Tuy vậy khi truyền đối số, nếu đối số thứ nhất thuộc kiểu *integer* thì đối số thứ hai cũng phải thuộc kiểu *integer*, nếu đối số thứ nhất thuộc kiểu *double* thì đối số thứ hai cũng phải thuộc kiểu *double*.

Việc sử dụng *template function* sẽ tránh cho người lập trình phải viết nhiều hàm khác nhau, tuy chúng cùng thực hiện một công việc giống nhau nhưng lại trên các loại dữ liệu có kiểu khác nhau. Cũng như các hàm khác, *function template* có thể được định nghĩa chồng:

Ví dụ 7.28

```
#include <stdlib.h>
int max(char* a, char* b)
{
    int x,y;
    x = atoi(a);
    y = atoi(b);
    return x>y ? x:y;
}
```

Trong trường hợp này, khi hàm *max(char*, char*)* được gọi, trình biên dịch sẽ không sử dụng *template function* vì một hàm tương ứng đã được định nghĩa.

Chương VIII

TÍNH THỪA KẾ

Một trong những đặc điểm của các vật thể sống là khả năng có thể sinh ra con cái với những đặc điểm giống tổ tiên của chúng. Tạo hóa từ hàng ngàn tỷ năm đã xây dựng nên cuộc sống như chúng ta đã biết với sự truyền bá các đặc điểm từ thế hệ này qua các thế hệ khác, đây chính là kết quả của sự kế thừa. Không có bất cứ sự sống nào, dù đơn giản hay phức tạp mà không có tính thừa kế.

Nếu trong cuộc sống sự kế thừa đem lại nhiều bất ngờ thú vị thì trong việc thiết kế các phần mềm nó cũng đưa lại rất nhiều lợi ích như việc sử dụng các chương trình đã có sẵn cho các công việc thiết kế mới nhằm làm giảm bớt mức độ phức tạp của các chương trình được viết ra. Ý tưởng này được xuất hiện trước những năm 1960, 70 và đã đem lại sự phát triển mạnh mẽ trong kỹ thuật lập trình. Tính thừa kế có thể nói là một trong các đặc điểm riêng của C^{++} và đem lại nhiều khả năng lớn đối với khái niệm lớp. Trong C^{++} thuật ngữ thừa kế chỉ được áp dụng cho lớp và các khái niệm có liên quan. Các biến không thể thừa kế từ các biến khác và hàm cũng không thể thừa kế từ các hàm khác.

Tính thừa kế cho phép ta có thể tiếp tục hoặc mở rộng với một khả năng không hạn chế đối với một lớp nào đó. Xuất phát

từ một lớp đơn giản, có thể nhận được những lớp tuy đơn giản nhưng lại có thể bao hàm các sự kiện và dữ liệu ở mức độ phức tạp cao hơn và vẫn cho phép dễ dàng sửa đổi và kiểm tra lỗi (Debug).

Sự đột phá chính trong các chương trình viết bằng C++ là ở việc phát triển các lớp để giải quyết các vấn đề được đặt ra. Các lớp này được xây dựng và phát triển từ những lớp cơ sở đơn giản thông qua tính thừa kế. Mỗi một lớp thừa kế một lớp trước đó sẽ có thể sử dụng tất cả các đặc tính của lớp này cộng thêm các đặc tính mới riêng của nó. Một chương trình có thể có hàng trăm lớp, tuy nhiên tất cả các lớp này đều bắt nguồn từ một số lớp cơ sở nào đó. C++ khác với các ngôn ngữ lập trình khác bởi đặc tính không cho phép thừa kế đơn mà còn cho phép thừa kế bội. Với đặc tính này một lớp có thể được thừa kế từ một hoặc nhiều lớp khác trong cùng một thời điểm và do đó có thể thừa kế tất cả các đặc điểm của tất cả các lớp này.

Để bắt đầu nghiên cứu về tính thừa kế, hãy xét ví dụ sau:

```
class Box{
    public:
        int width,height;
        void SetWidth(int w) { width = w;}
        void SetHeight(int h){ height = h;}
    };
class ColorBox:public Box{
    public:
        int Color;
        void SetColor(int c) (int c) { color = c;}
    };
```

Trong C++, lớp *Box* được gọi là lớp cơ sở (*Base Class*) của lớp *ColorBox*.

Lớp này đến lượt mình lại được gọi là lớp dẫn xuất (*Derived Class*) của *Box*. Các lớp cơ sở nhiều lúc cũng được gọi là các lớp cha. Ở ví dụ trên, trong lớp *ColorBox* chỉ định nghĩa một hàm *SetColor*, tuy vậy lớp này vẫn có thể sử dụng hai hàm *SetWidth* và *SetHeight* của lớp *Box* do tính thừa kế. Hai lớp trên có thể được sử dụng trong chương trình chính sau:

Ví dụ 8.1

```
ColorBox cb;
void main()
{
    // Sử dụng các hàm thành phần trong ColorBox;
    cb.SetColor(5); // Không thừa kế.
    cb.SetWidth(3); // Sử dụng hàm thừa kế.
    cb.SetHeight(50); // Sử dụng hàm thừa kế.
}
```

Chú ý rằng các hàm dẫn xuất được sử dụng giống hệt như các hàm bình thường khác. Lớp *ColorBox* thậm chí không đề cập đến vấn đề các hàm *SetWidth* và *SetHeight* là các hàm thừa kế khi gọi. Sự giống nhau này là một điều đặc biệt trong C++. Với đặc điểm này của lớp, khi gọi một hàm ta sẽ không cần quan tâm đến hàm đó có phải là hàm thừa kế hay không vì cả hai cách gọi hàm đều giống nhau.

8.1. THỪA KẾ ĐƠN (SINGLE INHERITANCE)

Một lớp có thể thừa kế các đặc điểm từ nhiều lớp khác nhau.

Trường hợp đơn giản nhất sử dụng tính thừa kế là thừa kế đơn (thừa kế từ một lớp duy nhất). Hầu hết các quy tắc của thừa kế đơn đều có thể sử dụng cho thừa kế bội (*Multiple Inheritance*).

Khi xây dựng một lớp mới trong chương trình, cần phải xem xét có nên sử dụng một lớp nào đó đã tồn tại làm lớp cơ sở của lớp mới hay không. Thông thường cần phải chọn những lớp có chứa nhiều đặc tính (dữ liệu hoặc hàm) mà lớp mới cần phải sử dụng. Các lớp này là những lớp thích hợp nhất bởi vì lớp mới có thể thừa hưởng những đặc tính cần thiết và loại bỏ những đặc tính không cần sử dụng thông qua việc định nghĩa chồng các hàm đó.

Cũng như trong cuộc sống, không phải mọi thứ trong C++ đều có thể được truyền qua tính thừa kế. Cụ thể các trường hợp sau đây không cho phép sử dụng tính thừa kế:

- Constructor.
- Destructor.
- Các toán tử mới do người sử dụng định nghĩa (New user operators).
- Các toán tử gán do người sử dụng định nghĩa (New Assignment User operators).
- Quan hệ Friend.

Các lớp dẫn xuất tự động gọi *constructor* của lớp cơ sở khi các lớp dẫn xuất đó được thể hiện. Tuy vậy sau khi các lớp dẫn xuất được thiết lập thì *constructor* của lớp cơ sở cũng thực hiện xong vai trò của mình. Như vậy có thể nói rằng lớp dẫn xuất thừa kế *constructor* của lớp cơ sở những việc gọi các *constructor* này không được thực hiện rõ ràng trong lớp dẫn xuất như các hàm thừa kế khác. Hãy xét đoạn chương trình sau:

```
class Parent{
    int value;
public:
    Parent() { value =0;}
    Parent(int v){ value =v;}
};
class Child:public Parent{
        int Total;
    public:
        Child(int t) { Total =t;}
        void SetTotal(int t);
};
void Child::SetTotal(int t)
{
    Parent::Parent(i);
    // Sai vì không thể thừa kế Constructor của lớp cơ sở
    Total =t;
}
```

Cũng như *constructor*, việc gọi rõ ràng một *destructor* là không được phép trong chương trình. Destructor sẽ được tự động gọi cho đối tượng kết thúc công việc và ra khỏi phạm vi của nó.

Quan hệ *Friend* không có tính kế thừa. Điều này cũng giống như trong thực tế: bạn của bố mẹ không nhất thiết phải là bạn của con v.v.

8.1.1. Các mức độ truy nhập đến lớp cơ sở (Access Levels To Base Class)

Khi một lớp được dẫn xuất từ một lớp cơ sở, tất cả các kiểu

dữ liệu cũng như hàm số được khai báo *public* trong lớp cơ sở tự động trở thành *private* trong lớp dẫn xuất. Ví dụ:

```
class First
{
private;
        int x;
public:
        int y;
public:
        void Setvalue(int v.int u)
        {
        x =v;
        y = u;
        }
};
class Second:First
{
    int total;
public:
    void SetTotal(int t) { total =t;}
};
```

Trong đoạn chương trình trên biến *y* và hàm *SetValue* của lớp cơ sở sẽ trở thành các thành phần kiểu *private* của lớp *Second*. Quyền truy nhập *(access modifier)* của lớp dẫn xuất đến các phần tử trong lớp cơ sở tuy vậy có thể dễ dàng được thay đổi nếu ta chỉ rõ các mức độ truy nhập đó. Khi đó lớp dẫn xuất có thể được khai báo theo cách sau:

```
class Second:access modifier First
```

```
{
.....
};
```

trong đó *access modifier* có thể là *public* hoặc *private*.

Như vậy *access modifier* được sử dụng để thay đổi quyền hạn truy nhập của lớp dẫn xuất đến các thành phần mà nó được kế thừa. Mức độ truy nhập này được chỉ rõ trong bảng 8.1. tùy thuộc vào *access modifier* được sử dụng.

Khi cần viết một lớp dẫn xuất từ một lớp khác đã tồn tại, người lập trình cần phải hiểu rõ quan hệ giữa lớp cơ sở và lớp dẫn xuất. Mức độ truy nhập của các lớp dẫn xuất đến các thành phần của lớp cơ sở cần phải được lưu ý khi sử dụng tính thừa kế. Các mức độ truy nhập đến các thành phần của lớp cơ sở có thể hoàn toàn khác nhau trong lớp dẫn xuất và lớp cơ sở. Một lớp có thể được dẫn xuất từ một lớp cơ sở với hai mức độ truy nhập khác nhau: *public* và *private*, trong đó *private* là mức truy nhập ngầm định. Với mức độ truy nhập *private*, tất cả các thành phần trong lớp cơ sở thuộc kiểu *public* và *protected* đều được chuyển thành *private* trong lớp dẫn xuất, trong khi các thành phần thuộc kiểu *private* trong lớp cơ sở vẫn giữ nguyên mức độ truy nhập của nó và do đó lớp dẫn xuất sẽ không có quyền truy nhập đến chúng. Với mức độ truy nhập là *public*, lớp dẫn xuất vẫn không có quyền truy nhập đến tất cả các thành phần khác thuộc kiểu *public* và *protected* vẫn giữ nguyên mức độ truy nhập của chúng trong lớp dẫn xuất.

Bảng 8.1

Mức truy nhập trong lớp cơ sở	Access Modifier	Quyền truy nhập đến các thành phần trong lớp cơ sở
public	public	public
private	public	không truy nhập được
protected	public	protected
public	private	private
private	private	không truy nhập được
protected	private	private

Tóm lại:

- Khi các thành phần của lớp cơ sở được khai báo theo kiểu *private*, các thành phần này chỉ được phép truy nhập đến ngay bên trong lớp đó.
- Khi các thành phần của lớp cơ sở được khai báo theo kiểu *protected*, các thành phần này chỉ được phép truy nhập đến ngay bên trong lớp cơ sở và các lớp khác dẫn xuất từ nó.
- Khi các thành phần của lớp cơ sở được khai báo theo kiểu *public* các thành phần này sẽ được phép truy nhập đến ngay bên trong lớp cơ sở và các lớp dẫn xuất cũng như ở bên ngoài các lớp đó.

Có thể xem xét cách sử dụng mức độ truy nhập thông qua một số ví dụ sau:

```
class Second: public First
{
    .....
}
```

Theo cách khai báo này thì các thành phần *public* trong *First* sẽ trở thành *public* trong *Second*, các thành phần *protecd* vẫn giữ nguyên mức độ truy nhập trong khi các thành phần thuộc kiểu *private* trong *First* thì *Second* không được phép truy nhập đến. Một ví dụ khác:

```
class Second: private First
{
    //.....
}
```

Ở đây các thành phần *public* và *protected* trong *First* sẽ trở thành *private* trong *Second*, trong khi tất cả các thành phần thuộc kiểu *private* trong *First* thì *Second* không được phép truy nhập đến.

8.1.2. Gọi Constructor của lớp cơ sở

Khi một lớp dẫn xuất được thể hiện, *constructor* của lớp cơ sở cũng phải được thực hiện. Thông thường *constructor* mặc định của lớp cơ sở sẽ được gọi một cách tự động. Trong trường hợp cần phải thực hiện *constructor* của lớp cơ sở với các tham số truyền ta phải dùng phép khai báo đặc biệt. Ví dụ dưới đây sẽ minh họa cho vấn đề này:

Ví dụ 8.2

```
class First{
        int a,b,c;
    public:
        First(int x, int y, int z){ a=x;b=y;c=z;}

int GetA( )
```

```
{
    return a;
}
int GetB( )
{
    return b;
}
int GetC( )
{
    return c;
}

};
class Second:public First{
    int value;
public:
    Second(int d):First(d,d+1,d+5) { value = d;}
    Second(int d, int e);
    int GetValue( ){return value;}
};
// Định nghĩa Constructor Second(int d, int e)
Second::Second(in d, int e): First(d,e,13);
{
    value = d+e;
}
// Hàm chính trong chương trình
#include<iostream.h>
void main()
{
```

```
First F(1,3,5);
cout<< "Các thành phần dữ liệu của F"
cout<<"\n a="<< F.GetA( )<< "b ="<< F.Get( )<< "c="<< F.GetC( );
Second S(5);
cout<< "Các thành phần dữ liệu của lớp S";
 cout<< "\n a="<< S.GetA( )<< "b=" <<S.GetB( )<< "c="
        <<S.GetC( )<< "value=" << S.Value( );
Second S1(1,4);
cout<< " Các thành phần dữ liệu của lớp S1";
 cout<< "\n a="<< S1.GetA( )<< "b=" <<S1.GetB( )<< "c="
        <<S1.GetC( )<< "value=" << S1.Value( );
```

Sau khi hàm *main* được thực hiện thì trên màn hình sẽ xuất hiện kết quả sau:

Các thành phần dữ liệu của F

a=1 b=3 c=5

Các thành phần dữ liệu của S

a=5 b=6 c=10 value=5

Các thành phần dữ liệu của S1

a=1 b=4 c=13 value=5

Trong ví dụ này, *constructor Second: :Second(int)* được khai báo theo kiểu *inline* bởi vì việc định nghĩa hàm được thực hiện ngay bên trong lớp. Trong cả hai trường hợp sử dụng *inline* hoặc *non-inline constructor* thì các *constructor* của lớp cơ sở đều được truyền tham số và thực hiện trước khi *constructor* của lớp thừa kế được thực hiện.

Hãy xét ví dụ khác mô tả về sự dẫn xuất của một lớp từ một lớp khác bằng ví dụ về xây dựng một chương trình đồ họa trong C^{++}. Trước tiên ta hãy đề cập đến một lớp đơn giản nhất của tất

cả các đối tượng trong kỹ thuật đồ họa - lớp *Location*. Lớp này định nghĩa vị trí của một điểm trên màn hình.

Thêm vào đó, nếu để ý có thể nhận thấy rằng một điểm trên màn hình sẽ được đặc trưng bởi hai thông tin: vị trí của điểm (thông qua lớp Location) và cách mô tả điểm đó trên màn hình. Việc mô tả điểm trên màn hình luôn được gắn liền với trạng thái của điểm đó - nhìn thấy hoặc không và nếu nhìn thấy thì màu gì). Trong hai thông tin này thì vị trí của điểm là cơ sở nhất - sẽ không có điểm nào có thể được hiển thị trên màn hình nếu không xác định được vị trí của điểm đó. Bởi vì tất cả các đối tượng đồ họa đều được xây dựng từ các điểm do đó chúng ta cần xây dựng lớp *Point* (xây dựng một điểm trên màn hình) dùng chung cho tất cả các đối tượng đó. Về phần mình, tất cả các điểm đều phải được xác định từ một vị trí trên màn hình do đó chúng cần được dẫn xuất từ lớp *Location*. Lớp này sẽ chứa thông tin về các tọa độ X, Y của một điểm trên màn hình. Khai báo của hai lớp *Point* và *Location* dưới đây được chứa trong File tiêu đề point.h.

```
// Point.h có chứa hai lớp:
// Lớp Location mô tả tọa độ X, Y của một điểm trên màn hình
// Lớp Point mô tả trạng thái của điểm trên màn hình: tối-sáng
    enum Boolean {false, true};
    class Location {
        protected:
            int X;
            int Y;
        public:
            Location(int InitX, int InitY);
```

```
        int GetX();
        int GetY();
    };
class Point: public Location{
    protected:
        Boolean Visible;
    public:
        Point(int InitX, int InitY);
        void Show();
        void Hide();
        Boolean IsVisible();
        void MoveTo(int NewX, int NewY);
};
```

Trong ví dụ này *Location* là lớp cơ sở và *Point* là lớp dẫn xuất từ lớp đó. Hai lớp này được đóng gói trong một File để tiện cho việc sử dụng chúng trong khi phát triển chương trình. Lớp *Point* được dẫn xuất từ *Location* với mức độ truy nhập public vì vậy *Point* có thể truy nhập đến các thành phần *X, Y* kiểu *protected* trong lớp *Location*. Ngoài việc thừa kế các hàm thành phần *GetX()* và *GEtY()* trong *Location, Point* còn có thêm các hàm thành phần khác;

Hàm *Show()* - Hiển thị một điểm trên màn hình.

Hàm *Hide()* - Ẩn điểm trên màn hình.

Hàm *IsVisble()* - Trả lại trạng thái của điểm trên màn hình.

Hàm *MoveTo()* - Di chuyển điểm trên màn hình.

Để mô tả một điểm trên màn hình, tuy vậy cần phải đưa thêm

một đặc tính nữa của điểm - đó là điểm được phát sáng hay không. Đặc tính này được mô tả thông qua dữ liệu thành phần *Visible* kiểu *protected* trong lớp *Point*. Biến này sẽ có giá trị bằng 1 khi điểm được phát sáng và bằng 0 trong trường hợp ngược lại. Như vậy *Visible* có thể được định nghĩa thuận tiện nhất thông qua dữ liệu kiểu *enum:*

enum Boolean {false, true};

Dưới đây là định nghĩa của các hàm thành phần của cả hai lớp. Tất cả các định nghĩa này được thực hiện trong File Point1.Cpp.

```
// Point1.cpp chứa định nghĩa của hai lớp Location và Point
#include "point.h"
#include <graphics.h>
// Các hàm thành phần của lớp Location
Location::Location(int InitX, int InitY)
{
    X = InitX;
    Y = IntY;
}
Location::GetX(void)
{
    return X;
}
Location::GetY(void)
{
    return Y;
}
// Các hàm thành phần của lớp Point
```

```
Point::Point(int InitX, init InitY):Location(InitX, InitY)
{
    Visible = false; // Không hiển thị điểm khi khởi tạo
}
void Point::Show(void)
{
    Visible = true;
    putpixel(X,Y,getcolor( ));
    // Hiển thị điểm, sử dụng màu ngầm định;
}
void Poin::Hide(void)
{
    Visible = false;
    putpixel(X,Y,getbkcolor( ));
}
Boolean Point::IsVisible(void)
{
    return Visible;
}
void Point::MoveTo(int NewX, int NewY)
{
    Hide( ); // Xóa điểm trên màn hình
    Y = NewX;
    X = NewY;
    Show ( ); // Hiển thị điểm ở vị trí mới
}
```

Dưới đây là chương trình sử dụng hai lớp trên để vẽ các điểm trên màn hình. Chương trình này được lưu trong file PIXEL.CPP.

Ví dụ 8.3

```
// PIXEL.CPP được biên dịch cùng POINT.CPP
#include <graphics.h>
#include <conio.h>
#include "point.h"
int main()
{
    // Khởi tạo đồ họa
    int graphdriver = DETECT;
    int graphmode;
    initgraph(&graphdriver,&graphmode."C:\\BorlandC\\Bgi ");
    Point APoint(100,500); // Khởi tạo điểm X,Y ở tọa độ 100, 50
    APoint.Show( ); // Hiển thị điểm
    getch( ); // Chờ gõ ký tự từ bàn phím
    APoint.Hide( );
    getch ( );
    closegraph( );
    return 0;
}
```

Có thể sử dụng lớp *Location* và *Point* như các lớp cơ sở để xây dựng lên các đối tượng đồ họa khác như đường tròn, hình vuông v.v. Đoạn chương trình dưới đây có sử dụng lớp Circle, lớp này được phát triển từ hai lớp *Location, Point* và dùng để vẽ đường tròn trên màn hình. Lớp *Circle* có chứa các dữ liệu và các hàm thành phần sau:

Radius: Bán kính đường tròn;

Show(): Hiển thị đường tròn;

Expand(): Tăng bán kính;

Contract(): Giảm bán kính;

MoveTo(): Di chuyển đường tròn.

Ví dụ 8.4

```
// CIRCLE.CPP - File chứa khai báo, định nghĩa và
// sử dụng lớp Circle
// Link cùng với file Point1.obj
#include<graphics.h>
#include "point.h"
#include<conio.h>
class Circle: Point
// Thừa kế từ Point và Location với mức truy nhập private
{
    int Radius;
public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show(void);
    void Hide();
    void Expand(int ExpandBy);
    void Contract(int ContractBy);
    void MoveTo(int NewX, int NewY);
};
// Định nghĩa các hàm thành phần của lớp Circle
Circle::Circle(int InitX, int InitY, int InitRadius):
            Point(InitX, intY)
```

```
{
    Radius = InitRadius;
}
void Circle::Show(void)
{
    Visible = true;
    circle(X,Y,Radius);
}
void Circle::Hide(void)
{
    unsigned int TemColor;
    // Biến tạm để ghi lại màu hiện thời
    Temcolor = getcolor( ); // Lưu màu hiện thời.
    setcolor(getbkcolor( ));
    // Thiết lập màu vẽ giống màu nền.
    Visible = false;
    circle(X,Y,Radius);
    // Xóa đường tròn - vẽ bằng màu nền.
    setcolor(TemColor); // Khôi phục lại màu vẽ cũ.
}
void Circle::Expand(int ExpandBy)
{
    Hide( ); // Xóa đường tròn cũ
    Radius +=ExpandBy; // Tăng bán kính
    if(Radius < 0) // Không sử dụng bán kính < 0
        Radius =0;
    Show( ); // Vẽ đường tròn mới.
```

```
}
void Circle::Contract(int ContractBy);
{
    Expand(-ContractBy);
}
void Circle::MoveTo(int NewX, int NewY)
{
    Hide(); // Xóa đường tròn cũ
    X = NewX; // Thiết lập vị trí mới
    Y = NewY;
    Show( ); // Vẽ đường tròn mới
}
int main( )
{
    // Khởi tạo đồ họa
    int graphdriver = DETECT;
    int graphmode;
    initgraph(&graphdriver,&graphmode,
        "C:\\BORLANDC\\BGI");
    Circle MyCircle(100,200,50);
    MyCircle.Show( );
    getch( );
    MyCircle.MoveTo(200,250);
    getch( );
    MyCircle.Expand(50);
    getch( );
    MyCircle.Contract(75);
```

```
}
void Circle::Contract(int ContractBy);
{
    Expand(-ContractBy);
}
void Circle::MoveTo(int NewX, int NewY)
{
    Hide(); // Xóa đường tròn cũ
    X = NewX; // Thiết lập vị trí mới
    Y = NewY;
    Show( ); // Vẽ đường tròn mới
}
int main( )
{
    // Khởi tạo đồ họa
    int graphdriver = DETECT;
    int graphmode;
    initgraph(&graphdriver,&graphmode,
        "C:\\BORLANDC\\BGI");
    Circle MyCircle(100,200,50);
    MyCircle.Show( );
    getch( );
    MyCircle.MoveTo(200,250);
    getch( );
    MyCircle.Expand(50);
    getch( );
    MyCircle.Contract(75);
```

```
    getch( );
    closegraph( );
    return 0; }
```

Như đã đề cập đến ở trên, khi một đối tượng của một lớp dẫn xuất được thể hiện thì việc truyền tham số cho *constructor* của lớp cơ sở phải được thực hiện. Trong một vài trường hợp, tuy vậy khi đó một vài thành phần dữ liệu của *constructor* cơ sở có thể chưa được khởi tạo. Trong trường hợp này C++ sẽ giải quyết vấn đề đó ra sao. Chúng ta hãy xét ví dụ dưới đây:

Ví dụ 8.5

```
#include <stdio.h>
class Next{
    public:
        char* String;
        Next(char* cp) {String = cp;}
        };
    class Last: public Next
    {
        int value;
            char name[30];
        public:
            Last(int d);
    } ;
// Định nghĩa Constructor của lớp Last
Last::Last(int d):Next((char*)0)
{
    sprintf(name,"%d",d);
```

```
value = d;
// Khởi tạo biến của cơ sở
   String = name;}
```

Qua ví dụ trên ta nhận thấy, constructor *Last: Last(int)* đang chờ đợi để truyền con trỏ kiểu ký tự cho lớp cơ sở. Tuy vậy con trỏ đó lúc này còn chưa được khởi tạo, do vậy *Last: Last(int)* bắt buộc phải truyền con trỏ *Null* để hợp thức hóa việc gọi *constructor* của lớp cơ sở bên trong thân của lớp dẫn xuất. Việc khởi tạo con trỏ được tiến hành bên trong thân *constructor* của lớp dẫn xuất và qua đó sẽ khởi tạo lớp cơ sở.

Phương pháp trên không thể thực hiện được trong các trường hợp sau:

- Tham số đang chờ đợi lại được dùng để khởi tạo các thành phần khác hoặc được dùng như các tham số truyền của lớp khác.
- Tham số đang chờ đợi được khởi tạo được khai báo theo kiểu private trong lớp cơ sở (chỉ có thể trì hoãn việc khởi tạo các thành phần kiểu *public* hoặc *protected*).

8.1.3. Lớp hạt giống (Seed class)

Các lớp trong C++ được thiết kế để thực hiện một mục đích nào đó và thông thường từ các lớp cơ sở đến các lớp dẫn xuất, mức độ phức tạp của chúng ngày càng tăng. Một trong các mục đích của việc dẫn xuất một lớp từ một lớp khác là để tránh việc trùng lặp khi viết chương trình. Khi có nhiều lớp khác nhau được dẫn xuất từ một lớp khác thì lớp cơ sở này thường rất đơn giản và nó thường chứa các đặc điểm chung của tất cả các lớp dẫn xuất. Các lớp mà mục đích được xây dựng chỉ nhằm chia sẻ các đoạn mã cho các lớp dẫn xuất được

gọi là các lớp hạt giống (*seed class*). Cần lưu ý là mặc dù các lớp này không cần phải thiết kế như một lớp trừu tượng (xem chương X) nhưng việc sử dụng để khởi tạo các đối tượng cũng hầu như không đem lại lợi ích gì trong khi xây dựng chương trình. Hãy xét đoạn chương trình sau:

```
class BookSelf
{
        int color;
        int width, height;
        int shelves;
    public:
        BookSelf(int, int, int, int);
        int GetColor( ) { return color;}
        int GerWidth ( ) { return width;}
        int GetHeight( ) { return shelver;}
};
class Desk
{
        int color;
        int width, height;
        int drawers;
        int material;
    public:
        Desk(int, int, int, int);
            int GetColor( ) { return color;}
            int GetWidth( ) { return width;}
            int GetHeight( ) { return shelves;}
            int GetDrawers( ) { return drawers;}
            int GetMaterial( ) { return material;}
```

```
    };
BookSelf::Bookself(int c, int w, int h, int s)
{
    color =c;
    width = w;
    height = h;
    shelves = s;
}

Desk::Desk(int w, int h, int c, int d, int m)
{
    width = w;
    height = h;
    color = c;
    drawers = d;
    material = m
}
```

Đoạn chương trình trên có hai lớp: *BookShelf* và *Desk*. Hai lớp này có rất nhiều đặc điểm chung nhưng vẫn tồn tại các điểm khác nhau giữa chúng. Constructor của *BookSelf* có bốn đối số trong khi *constructor* của *Desk* lại có năm đối số. Thêm vào đó thứ tự xuất hiện của các đối số trong hai *constructor* này cũng có những điểm khác nhau. Để tránh viết lại các điểm được xem là giống nhau giữa hai lớp, ta có thể xây dựng một *seed class* - lớp Furniture.

Ví dụ 8.6

```
// Lớp hạt giống - Seed class
class Furniture
{
        int color;
```

```
        int width, height;
    public:
        Furniture(int, int, int);
        int GetColor() { return color;}
        int GetWidth() { return width;}
        int GetHeight() { return shelves;}
    };
    Furniture::Furniture(int c, int w, int h)
    {
        color = c;
        width = w;
        height = h;
    }
    class BookSelf: public Furniture
    {
        int shelves;
    public:
        BookSelf(int, int, int, int);
        int GetShelves() { return shelves;}
    };
    class Desk: public Furniture
    {
        int drawers;
        int material;
    public:
        Desk( int, int, int, int, int);
        int GetDrawers() { return drawers;}
        int GetMaterial() { return material;}
    };
```

```
BookSelf::BookSelf( int c, int w, int h, int s) : Furniture(c, w, h)
{
    shelver = s;
}
Desk::Desk(int w, int h, int c, int d, int m) : Furniture(c, w, h)
{
    drawers = d;
    material = m'
}
void main()
{
    // Sử dụng hai lớp dẫn xuất
    Desk d(1, 2, 3, 4, 5);
    BookShelf b(10, 11, 12, 13);
}
```

8.1.4. Chuyển kiểu giữa các lớp

Xét hai lớp sau:

```
class A{..... } ;
class B:public A
{
.....
};
```

Bởi vì lớp *B* là lớp dẫn xuất từ *A* do đó B được thừa kế các thành phần của *A* mà mức độ truy nhập cho phép. Như vậy ta có thể xem *B* là một class ở cấp cao hơn *A* và do đó ta có thể thực hiện việc chuyển kiểu từ lớp *B* sang lớp *A*. Tuy vậy điều ngược lại là không được phép.

Ví dụ 8.7

```
void main()
{
    A a;
    B b;
    A* ap;
    B* bp;
    // Cho phép thực hiện việc chuyển kiểu từ b sang a
    a = b;
    ap = bp; // cho phép.
    // Không cho phép thực hiện việc chuyển kiểu từ a sang b
    b = a;
    bp = ap; // Không cho phép;
}
```

Ví dụ sau minh họa cho việc chuyển kiểu giữa các lớp qua hàm:

Ví dụ 8.8

```
void foo(A* object){ }
 void main()
 {
    A* ap;
    B* bp;
    foo(ap);
    foo(bp);
}
```

Trong ví dụ này, khi hàm *foo(bp)* được gọi, trình biên dịch sẽ thực hiện việc chuyển *bp* từ kiểu *B* sang kiểu *A* và sau đó truyền tham số cho hàm *foo(A* object)*.

8.1.5. Sử dụng phạm vi trong lớp

Theo nguyên tắc sử dụng hàm chồng, trong C++ các dữ liệu và hàm thành phần thuộc các lớp khác nhau có thể được đặt cùng tên (thậm chí có cùng đối số hoặc kiểu của các đối số trong trường hợp hàm thành phần). Như vậy trong một kiểu dẫn xuất có thể tồn tại các dữ liệu và hàm thành phần trùng với các dữ liệu và hàm trong lớp cơ sở (trùng tên, trùng đối số, trùng kiểu của các đối số). Trong trường hợp này để có thể gọi chính xác một biến hoặc hàm thuộc lớp cơ sở từ một đối tượng thuộc lớp dẫn xuất C++ sử dụng toán tử phạm vi (Scope Resolution).

Ví dụ 8.9

```
class A{
    public:
        int foo(){ return 1;}
    };
    class B:public A{
    public:
        int foo(return 2;}
    };
    void main()
    {
        A a;
        B b;
        int x = a.foo();
        int y = b.foo();
    }
```

Một số người lập trình thường nhầm lẫn và không xác định được giá trị của y khi thực hiện hàm *b.foo()*. Lớp *B* có 2 hàm *foo()*. Một hàm được thừa kế từ lớp *A* và một hàm là của chính

lớp *B*. Trong trường hợp trên *x* có giá trị là 1 và *y* có giá trị là 2. Trình biên dịch đã gọi chính xác các hàm thậm chí các hàm đó trùng tên nhau nhờ sử dụng quy tắc về phạm vi. Theo quy tắc này thì nếu tên của các biến hoặc hàm trong lớp cơ sở được khai báo lại trong lớp dẫn xuất thì tên trong lớp dẫn xuất sẽ che khuất tên tương ứng trong lớp cơ sở. Điều này cũng giống như khi sử dụng phạm vi của các biến trong từng vùng v.v.

Tuy vậy, trong C++ cũng tồn tại nhiều điều khác biệt. Ta có thể bắt buộc trình biên dịch "nhìn" ra ngoài phạm vi hiện thời và truy nhập đến các tên đã bị che khuất trong phạm vi đó bằng cách sử dụng các toán tử phạm vi (*Scope Resolution Operator*). Các biến và hàm được gọi trong bị che khuất có thể được gọi theo cách sau:

<Tên lớp::<Tên biến hoặc hàm>;

Đoạn mã sau sẽ minh họa cho việc sử dụng toán tử phạm vi:

```
class A{
    public:
    int foo(){ return 1:]
};
class B : public A{
    public:
    int foo(return 2;}
    int f(){ return A::foo();}
    // Sử dụng toán tử phạm vi để gọi foo() thuộc A
};
```

Trong trường hợp này, khi gọi hàm *B: f()* thì hàm *foo()* của lớp *A* sẽ được gọi và thực hiện. Việc sử dụng toán tử phạm vi không những chỉ được thực hiện bên trong một hàm thuộc một

8.1.5. Sử dụng phạm vi trong lớp

Theo nguyên tắc sử dụng hàm chồng, trong C++ các dữ liệu và hàm thành phần thuộc các lớp khác nhau có thể được đặt cùng tên (thậm chí có cùng đối số hoặc kiểu của các đối số trong trường hợp hàm thành phần). Như vậy trong một kiểu dẫn xuất có thể tồn tại các dữ liệu và hàm thành phần trùng với các dữ liệu và hàm trong lớp cơ sở (trùng tên, trùng đối số, trùng kiểu của các đối số). Trong trường hợp này để có thể gọi chính xác một biến hoặc hàm thuộc lớp cơ sở từ một đối tượng thuộc lớp dẫn xuất C++ sử dụng toán tử phạm vi (Scope Resolution).

Ví dụ 8.9

```
class A{
    public:
        int foo(){ return 1;}
    };
    class B:public A{
    public:
        int foo(return 2;}
    };
    void main()
    {
        A a;
        B b;
        int x = a.foo();
        int y = b.foo();
    }
```

Một số người lập trình thường nhầm lẫn và không xác định được giá trị của y khi thực hiện hàm *b.foo()*. Lớp *B* có 2 hàm *foo()*. Một hàm được thừa kế từ lớp *A* và một hàm là của chính

lớp *B*. Trong trường hợp trên *x* có giá trị là 1 và *y* có giá trị là 2. Trình biên dịch đã gọi chính xác các hàm thậm chí các hàm đó trùng tên nhau nhờ sử dụng quy tắc về phạm vi. Theo quy tắc này thì nếu tên của các biến hoặc hàm trong lớp cơ sở được khai báo lại trong lớp dẫn xuất thì tên trong lớp dẫn xuất sẽ che khuất tên tương ứng trong lớp cơ sở. Điều này cũng giống như khi sử dụng phạm vi của các biến trong từng vùng v.v.

Tuy vậy, trong C++ cũng tồn tại nhiều điều khác biệt. Ta có thể bắt buộc trình biên dịch "nhìn" ra ngoài phạm vi hiện thời và truy nhập đến các tên đã bị che khuất trong phạm vi đó bằng cách sử dụng các toán tử phạm vi (*Scope Resolution Operator*). Các biến và hàm được gọi trong bị che khuất có thể được gọi theo cách sau:

<Tên lớp::<Tên biến hoặc hàm>;

Đoạn mã sau sẽ minh họa cho việc sử dụng toán tử phạm vi:

```
class A{
    public:
    int foo(){ return 1:]
};
class B : public A{
    public:
    int foo(return 2;}
    int f(){ return A::foo();}
    // Sử dụng toán tử phạm vi để gọi foo() thuộc A
};
```

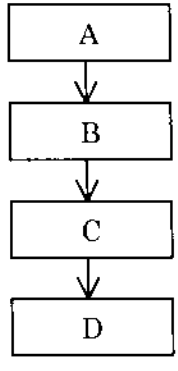
Trong trường hợp này, khi gọi hàm *B: f()* thì hàm *foo()* của lớp *A* sẽ được gọi và thực hiện. Việc sử dụng toán tử phạm vi không những chỉ được thực hiện bên trong một hàm thuộc một

lớp mà còn có thể được thực hiện trong thời gian gọi hàm. Hãy xét ví dụ dưới đây:

Ví dụ 8.10

```
void main()
{
    B b;
    B* b1 = new B;
    int x = b.A::foo();
    // Sử dụng toán tử phạm vi để gọi foo() thuộc A
    int y = b.foo();
    int z = b1->A::foo();
    // Sử dụng toán tử phạm vi để gọi foo() thuộc A
}
```

Trong trường hợp chương trình có sử dụng nhiều mức độ thừa kế khác nhau, toán tử phạm vi vẫn cho phép truy nhập đến thành phần của bất cứ lớp cơ sở nào. Hãy xét cấu trúc thừa kế sau và chương trình minh họa cho cấu trúc này:



Hình 8.1

```
class A{
    public:
        int foo(){ return 1;}
};
class B:public A{
    public:
        int foo(){ return 2;}
};
class C:public B{
    public:
        int foo(){ return 3;}
};
class D:public C{
    public:
        int foo(){ return 4;}
        int f1(){ rerturn A::foo();}
    // Cho phép sử dụng toán tử phạm vi
        int f2(){ rerturn B::foo();}
    // Cho phép sử dụng toán tử phạm vi
        int f3(){ rerturn C::foo();}
    // Cho phép sử dụng toán tử phạm vi
        int f4(){ rerturn D::foo();}
    // Cho phép sử dụng toán tử phạm vi
};
```

Cùng với toán tử phạm vi ::, ta có thể sử dụng toán tử "."

Ví dụ 8.11

```
class A{
    public:
```

```
        int value;
};
class B:public A{
{
public:
    int count
};
void main()
{
    B b;
    int i = b.count;
    int j = b.B::count;
    int k = b.value;
    int l = b.A::value;
}
```

8.1.6. Mở rộng các hàm thừa kế

Như ta đã biết, một trong những nguyên nhân chính dẫn đến việc dẫn xuất một lớp từ một lớp cơ sở là do lớp cơ sở có chứa nhiều hàm hoặc dữ liệu mà lớp dẫn xuất cần sử dụng đến. Tuy vậy, để có thể đáp ứng được các yêu cầu của lớp dẫn xuất các hàm này còn cần phải phát triển thêm. Việc viết lại toàn bộ các hàm đó trong lớp dẫn xuất sẽ làm phí tổn nhiều thời gian. Để tránh vấn đề này, C++ cho phép sử dụng lại các đoạn mã của hàm được viết trong lớp cơ sở qua việc mở rộng chúng trong lớp dẫn xuất. Điều này được thực hiện thông qua việc định nghĩa lại các hàm trong lớp cơ sở ở lớp dẫn xuất. Hãy xem xét đoạn chương trình dùng để thiết kế một hình tượng (*Icon*) cho một nút bấm ấn - thả (*push - button*).

```
class Box {
    int left, top;
```

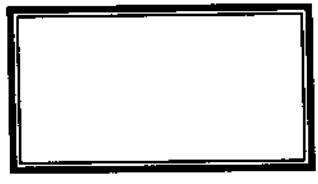
```
    int width, hight;
public:
    Box(int l, int t, int  w, int h )
    {
        // Khởi tạo các dữ liệu thành phần của lớp
        left = l; top = t;
        width  = w; hight = h;

    }
    void Display()
    {
        rectangle(left, top, left+ width, top + hight);
    };
class PushButton {
    int state;
    Box* Outline;
    Box* Button;
public:
    PushButton(int px, int py);
void Display( );
~PushButton( )
{
    delete Outline ;
    delete Button;
}
};
PushButton::PushButton(int x, int y)
{
    const int OUTLINE_WIDTH = 36;
```

```
    const int OUTLINE_HEIGHT = 20;
    const int BUTTON_WIDTH = 32;
    const int BUTTON_HEIGHT = 16;
   Outline=new Box(x,y,OUTLINE_WIDTH,OUTLINE_HEIGHT);
    int bx = x+ (OUTLINE_WIDTH - BUTTON_WIDTH)/2;
    int by = y+(OUTLINE_HEIGHT - BUTTON_HEIGHT)/2;
  Button=new Box(bx,by,BUTTON_WIDTH,BUTTON_HEIGHT)
}

void PushButton::Display( )
 {
     Outline->Display( );
     Button->Display( );
     }
```

Lớp *PushButton* cần hiển thị một nút bấm. Nút bấm đơn giản chỉ bao gồm hai hình chữ nhật được lồng vào nhau như hình 8.2:



Hình 8.2

Trong lớp *PushButton* cần phải xây dựng hàm để thực hiện yêu cầu trên. Hàm này cần phải vẽ hai hình chữ nhật có kích thước khác nhau. Thay vì phải viết lại các công việc này, trong lớp *PushButton* có định nghĩa lại hàm *Display* trong lớp cơ sở, hàm này có cùng tên với hàm đã được định nghĩa trong lớp cơ sở.

Lớp PushButton có thể được dùng trong ví dụ sau:

Ví dụ 8.12

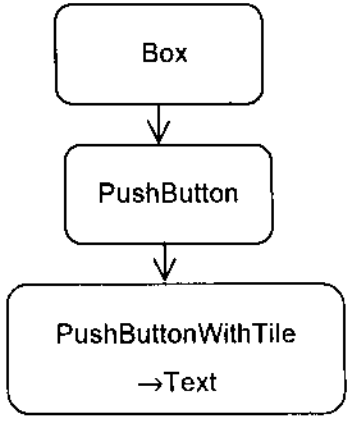
```
int main( )
{
    int gdrv = DETECT, gmode, Err;
    initgraph(&gdrv, &gmode,"C:\\borlandc\\bgi");
    Err = graphresult( );
    if(Err != grOk)
    {
        cout<< "graphics error" <<endl ;
        grapherrormsg( );
        cout << " press  any key to halt" ;
        getch( );
        exit(1);
    }
    PushButton Pb(30,40);
    Pb.Display( );
    getch();
    closegraph( );
    return 0;
    }
```

Giả sử rằng cùng với *PushButton* (nút bấm), chương trình cần phải hiển thị tiêu đề của *PushButton* (tiêu đề phản ánh tác dụng của nút này). Như vậy cần sử dụng lớp *PushButton* cùng với một hàm khác dùng để hiển thị chuỗi văn bản trong nút. Điều này sẽ được giải quyết nếu ta tạo một lớp dẫn xuất từ lớp *PushButton* và mở rộng hàm *Display* cho lớp đó. Lớp dẫn xuất mới được gọi là *PushButtonWithTile* và sẽ sử dụng hàm *PushButton::Display()* để hiển thị hình tượng và sau đó sẽ viết dòng tiêu đề lên trên hình tượng đó. Có thể biểu diễn sự thừa kế

của các lớp thông qua cấu trúc cây ở hình 8.3. Trong cấu trúc cây này, *Text* là một lớp mới không dẫn xuất từ *PushButton* và dùng để xuất một xâu ký tự tại điểm có tọa độ (x,y).

Sau đây là đoạn mã của các lớp:

```
class Text{
    int x, y;
    char *str;
    int len;
public:
    Text(int x1,int y1, char* string)
    {
        x = x1;
        y = y1;
        len = strlen(string);
        str = new char[len+1];
        strcpy(str,string);
    }
```



Hình 8.3

```
    void Display( )
    {
        outextxy(x,y,str);
    }
};
class PushButtonWithTile;public PushButton{
        Text *Title;
public:
        PushButtonWithTile(int x, int y, char* title);
        void Display();
        ~PushButtonWithTile() { delete Title}
    };
PushButtonWithTile::PushButtonWithTile(int x, int y,
                char* legend):
            PushButton(x,y)
{
    Title = new Text(x,y,legend);
}

void PushButtonWithTile::Display()
{
    PushButton::Display( );
    Title->Display();
}
```

Hàm main trong ví dụ này được viết lại như sau:

```
int main( )
 {
```

```
int gdrv = DETECT, gmode, Err;
initgraph(&gdrv, &gmode,"C:\\borlandc\\bgi");
Err = graphresult( );
if(Err != grOk)
{
cout<< "graphics error" <<endl ;
grapherrormsg( );
cout << "press any key to halt" ;
getch( );
exit(1);
}
PushButtonWithTile Pb(30,40,"OK");
Pb.Display( );
getch();
closegraph( );
return 0;
}
```

Trong ví dụ này, hàm được mở rộng của *PushButton* là *Display()* trong lớp mới *PushButtonWithTile*. Hàm này không chỉ vẽ nút bấm (*PushButton)* mà còn hiển thị dòng mô tả tác dụng của *PushButton* này. Trong lớp *PushButtonWithTile*, hàm *Display()* của lớp này sử dụng toán tử phạm vi để truy nhập đến các hàm *Dislay()* có cùng tên trong các lớp cơ sở *PushButton* và *Text*.

8.1.7. Thu hẹp tác động của các hàm thừa kế

Các lớp cơ sở thường chứa các hàm hoặc dữ liệu gần giống với các hàm và dữ liệu mà lớp dẫn xuất có thể sử dụng. Gọi là

gần giống vì trong nhiều trường hợp các hàm trong lớp cơ sở có thể chưa đạt được yêu cầu của lớp dẫn xuất. Ở trên chúng ta đã xét trường hợp khi cần mở rộng các tác động của các hàm này trong lớp dẫn xuất. Trong trường hợp các hàm trong lớp cơ sở có chứa các tác động không cần thiết trong lớp dẫn xuất, các tác động này cần phải được loại bỏ. Trong C++, các lớp dẫn xuất cũng có thể hạn chế các tác động của lớp cơ sở trên lớp của mình. Để minh họa cho điều này, hãy quay lại ví dụ trên với hình dạng của *PushButton* mới không có hình chữ nhật ở bên trong.

Để đạt được mục đích này, rõ ràng *PushButton: :Display()* cần phải được thay đổi để loại bỏ tác động không cần thiết - vẽ thêm một hình chữ nhật ở bên trong. Có thể thực hiện điều này nếu sử dụng một lớp mới dẫn xuất từ *PushButton* - lớp *SimplePushButton*. Trong trường hợp này, các lớp được định nghĩa lại như sau:

```
class PushButton{
protected:
        int state;
        Box* outline;
        Box* button;
public:
        PushButton(int x, int y);
        void Display();
        ~PushButton() { }
};
class SimplePushButton:public PushButton{
    public:
        Simple PushButton(int x, int y);
```

```
        void Display();
        ~SimplePushButton() { }
    };
SimplePushButton:: SimplePushButton(int x, int y):
                            PushButton(x,y)
    {
    }
void SimplePushButton::Display()
    {
    outline->Display();
    }
```

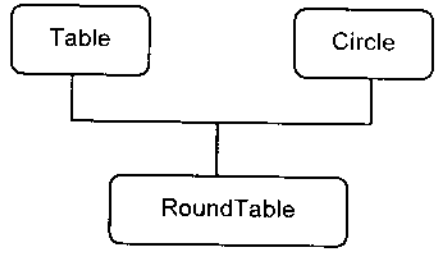
Điều cần lưu ý ở đây là hàm *Display* của lớp *SimplePushButton*. Hàm này không chứa bất cứ một đoạn mã nào mới so với hàm *Display* của *PushButton*. Tuy vậy với một số thay đổi nhỏ trong lớp *SimplePushButton*, hàm sẽ ngăn không có các đối tượng của lớp *SimplePushButton* hiển thị hình chữ nhật bên trong. Điều cần nói ở đây là để đạt được mục đích mới này, người lập trình không cần thực hiện bất cứ một sự thay đổi nào trong lớp cơ sở. Đây là một điểm rất quan trọng bởi vì các lớp cơ sở thường chỉ có thể bị thay đổi bởi chính người cung cấp các lớp đó. Nguyên nhân ở đây là khi được cung cấp cho người sử dụng, các lớp này thường đã được biên dịch dưới dạng file Obj.

8.2. THỪA KẾ BỘI (MULTIPLE INHERITANCE)

Trong cuộc sống, sự kết hợp giữa cha và mẹ sẽ tạo cho con cái thêm nhiều đặc tính phong phú: đó là khả năng thích nghi hơn với điều kiện của cuộc sống và khả năng chọn lọc những đặc tính tốt của cha và mẹ. Các lớp dẫn xuất trong C++ không chỉ bị hạn chế bởi một lớp cơ sở duy nhất mà còn có thể thừa kế nhiều

lớp cơ sở khác nhau. Khi đó lớp dẫn xuất sẽ thừa kế tất cả các đặc tính mà mỗi lớp cơ sở có được. Sự thừa kế này được gọi là thừa kế bội và mặc dù có thể dẫn đến sự phức tạp trong ngôn ngữ cũng như đối với trình biên dịch đặc tính này cũng mang lại rất nhiều điều thuận lợi trong khi thiết kế chương trình.

Hãy xét một lớp *RoundTable*, lớp này không chỉ có các đặc tính của lớp *Table* mà còn phải có các đặc tính của lớp *Circle* chứa các đặc tính của hình dạng bao quanh. Sự thừa kế này có thể được mô tả qua cấu trúc cây sau:



Hình 8.4

Dưới đây là chương trình minh họa cho cấu trúc cây trên:

Ví dụ 8.13

```
#include<stdio.h>
class Circle{
        float Radius;
    public:
        Cirle(float r) { Radius =r;}
        float Area( ) { return Radius*Radius*3.14;}
};
class Table{
    {
```

```
        float Height;
    public:
        Table(float h) { Height= h;}
        float Height( ) { return Height;)
};
class RoundTable:public Table,public Circle{
        int Color;
    public:
        RoundTable(float h, float r, int c);
        int Color( ){ return Color;}
};
RoundTable::RoundTable(float h, float r, int c):
                        Cirle(r),Table(h)
    {
    Color =c;
    }
voi main()
{
    RoundTable table(15.0,3.0,5);
    printf("\n Các đặc tính của bảng là :");
    printf("\n Chiều rộng =%f", table.Height());
    printf("\n Diện tích =%f", table.Area());
    printf("\n Màu =%d", table.Color());
}
```

Trong ví dụ trên, lớp *RoundTable* đã trở thành rất đơn giản do thừa hưởng rất nhiều đặc tính từ các lớp cơ sở. Khi khai báo một lớp dẫn xuất nào đó mang tính thừa kế bội, cần khai báo các lớp cơ sở của nó. Sau khi được khai báo, các lớp cơ sở này có thể được truy nhập trực tiếp trong lớp dẫn xuất của chúng. Sự truy

nhập này hoàn toàn tương tự giống như trong trường hợp thừa kế đơn.

Hàm *main()* trong ví dụ trên gọi 3 hàm thành phần *RoundTable*::Height(), *RoundTable::Area()* và *RoundTable:Color* mà không cần phân biệt chúng có phải là hàm thừa kế hay không. Phương pháp truy nhập này cũng được sử dụng trong trường hợp thừa kế đơn.

8.2.1. Thực hiện Constructor của các lớp cơ sở

Giống như trường hợp thừa kế đơn, các *constructor* của các lớp cơ sở trong trường hợp thừa kế bội cần phải được gọi trước *constructor* của lớp dẫn xuất. Thứ tự gọi các *constructor* của các lớp này sẽ phụ thuộc vào thứ tự khai báo của chúng trong lớp dẫn xuất. Hãy xét lại lớp *RoundTable:*

```
class RoundTable:public Table, public Circle {
        int Color;
    public;
        RoundTable(float h, floar r, int c);
        int Color() { return Color;}
};
```

Các lớp cơ sở được khai báo trong lớp dẫn xuất có thứ tự lần lượt là *Table, Circle*. Như vậy khi thể hiện một đối tượng thuộc lớp *RoundTable*, các constructor sẽ lần lượt được gọi theo thứ tự sau:

Table::Table(float);

Circle::Circle(float);

RoundTable::RoundTable(float);

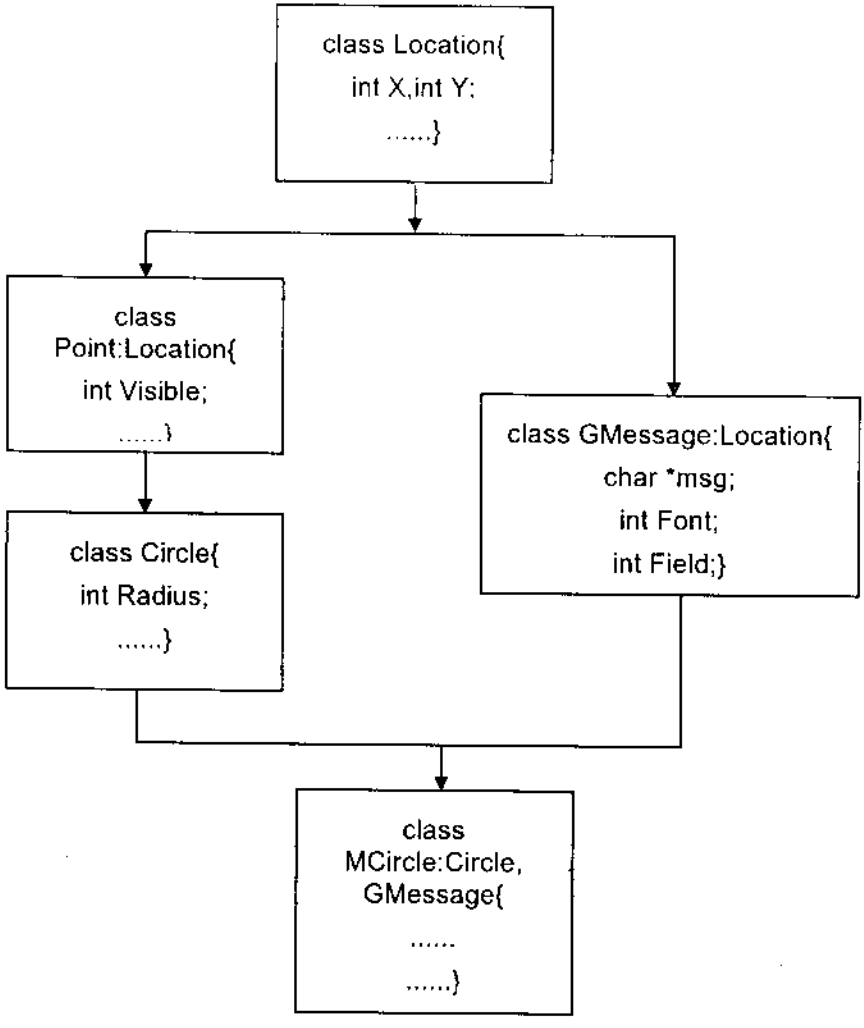
Để các *constructor* của các lớp cơ sở có thể được gọi trước khi

constructor của lớp dẫn xuất được thực hiện, các tham số của các lớp cơ sở này cần được truyền trước khi định nghĩa *constructor* của lớp *RoundTable*. Điều này được thực hiện bằng cách sau:

```
RoundTable::RoundTable(float h, float r, int c):Circle( ),Table(h)
        {
        // Khởi tạo các thành phần của RoundTable;
        }
```

Trên thực tế, một điều cần lưu ý là thứ tự gọi các *constructor* của các lớp cơ sở sẽ được xác định theo thứ tự khai báo của chúng trong lớp dẫn xuất chứ không phải theo thứ tự khi định nghĩa các lớp này. Chẳng hạn trong ví dụ trên, mặc dù lớp *Circle* được định nghĩa trước *Table* nhưng *constructor* của lớp này lại được gọi sau *constructor* của lớp *Table* do *Table* được khai báo trước *Circle* khi khai báo lớp dẫn xuất.

Hãy tiếp tục xét lớp vòng tròn trong ví dụ 7.7 để mô tả đặc tính thừa kế bội của lớp. Chương trình được phát triển tiếp để có thể vẽ một đường tròn cùng với một dòng văn bản đi kèm theo đường tròn đó. Một ý tưởng có thể được thực hiện một cách dễ dàng nhất đó là khai báo thêm một dữ liệu thành phần thuộc kiểu chuỗi trong lớp *Circle* và sau đó phát triển hàm *Circle::Show* để nó có thể vẽ đường tròn cùng với chuỗi ký tự kèm theo. Tuy vậy với ý tưởng này, người lập trình đòi hỏi phải viết lại lớp *Circle*, một điều mà trên thực tế không phải bất cứ lúc nào cũng có thể thực hiện được nếu không được cung cấp mã nguồn của *Cirle*. Thêm vào đó chuỗi ký tự và đường tròn là



Hình 8.5

hai đối tượng hoàn toàn khác nhau. Khi nghĩ về một chuỗi ký tự, thông thường người ta liên tưởng ngay đến fonts, kích thước chữ và một số thuộc tính khác. Tất cả các thuộc tính này hầu như không có bất cứ một sự liên hệ nào đối với đường tròn.

Như vậy, để có thể vẽ được đường tròn cùng với chuỗi ký tự đi kèm theo nó mà vẫn đảm bảo được tính độc lập của hai đối tượng này, ngoài lớp *Circle* ta cần xây dựng thêm một lớp *GMessage*. Lớp này sẽ dùng để hiển thị một xâu ký tự bắt đầu tại một điểm nào đó trên màn hình. Dựa trên dặc tính tương ứng bội trong C++ ta sẽ xây dựng lớp *MCircle*. *MCircle* sẽ thừa kế cả hai lớp *Circle* và *GMessage* và dùng để vẽ đường tròn cùng với dòng văn bản đi kèm theo nó. Dưới đây là chương trình minh hoạ cho ví dụ này:

Ví dụ 8.14

```
// MIRCIRCLE - Minh họa cho tính thừa kế bội.
#include <graphics.h>
#include "point.h"
#include <string.h>
#include <conio.h>
// Chương trình cần được liên kết cùng point1.obj và
// graphics.lib
class Circle: public Point // Thừa kế từ Point và Location
// với mức truy nhập private
    {
    protected:
        int Radius;
    public:
        Circle(int InitX, int InitY, int InitRadius);
        void Show(void);
        void Hide();
        void Expand(int ExpandBy);
        void Contract(int ContractBy);
```

```
            void MoveTo(int NewX, int NewY);
        };
    class GMessage: public Location
    {
        char* msg;
        int Font;
        int Field;
    public:
        GMessage(int msgX, int msgY,nit MsgFont,int
            FieldSize,char *text);
        void Show(void);
    };
    class MCircle:Circle, GMessage
    {
    public:
MCircle(int mcircX, mcirc Y,nit mcircRadius,int Font, char* msg);
        void Show(void);
    };
    // Định nghĩa các hàm thành phần của lớp Circle
    Circle:Circle(int InitX, int InitY, int InitRadius);
    Point(InitX, intY)
    {
        Raidus = InitRadius;
    }
    void Circle::Show(void)
    {
        Visible = true;
        circle(X,Y,Radius);
    }
```

```
void Circle:Hide(void)
{
    unsigned int TemColor;
    // Biến tạm để ghi lại màu hiện thời
    Temcolor = getcolor();
    // Lưu màn hiện thời.
    setcolor(getbkcolor());
    // Thiết lập màu vẽ giống màu nền.
    Visble = false;
    circle(X,Y,Radius);
    // Xóa đường tròn - vẽ bằng màu nền.
    setcolor(TemColor); // Khôi phục lại màu vẽ cũ.
}
void Circle:Expand(int ExpandBy)
{
    Hide(); // Xóa đường tròn cũ
    Radius +=ExpandBy; // Tăng bán kính
    if(Radius <0) // Không sử dụng bán kính <0
        Radius = 0;
    Show(); // Vẽ đường tròn mới.
}
void Circle:Contract(int ContractBy);
{
    Expand(-ContractBy);
}
void Circle::MoveTo(int NewX, nit NewY)
{
    Hide(); // Xóa đường tròn cũ
    X = NewX; // Thiết lập vị trí mới
```

```
        Y = NewY;
        Show(); // Vẽ đường tròn mới }
        // Định nghĩa các hàm thành phần của lớp GMessage
    GMessage::GMessage(int msgX, int msgY,int MsgFont,
        int FieldSize, char *text):Location(msgX,msgY)
        {
            Font = MsgFont;
            // Font chuẩn được thiết kế trong graphics.h
            Field = FieldSize; // Độ dài của vùng chứa xâu ký tự
            msg = text;              // Xâu ký tự
        }
    void GMessage::Show(void)
    {
        int size = Field/(8*strlen(msg)); // số điểm cho một ký tự
        settextjustify(CENTER_TEXT,CENTER_TEXT);
        // Dóng vào giữa đường tròn
        settextstyle(Font,SORIZ_DIR,size);
        // Thay đổi tỷ lệ nếu size > 1
        outtextxy(X,Y,msg);
        // Hiển thị Text
    }
    // Định nghĩa các hàm thành phần của MCircle
MCircle: MCircle(int mcircX, mcircY,int mcircRadius,int Font,
            char *msg):Circle(mcircX, mcircY, mcircRadius)
    {
    }
    void MCircle::Show(void)
    {
        Circle:Show(); // Sử dụng toán tử phạm vi để gọi
```

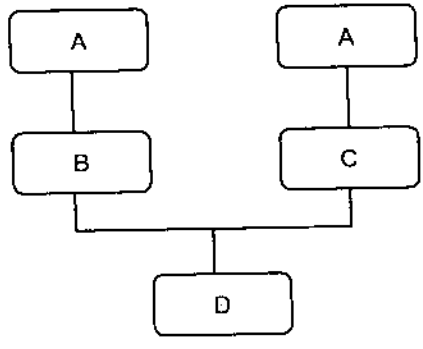
```
    GMessage::Show(); // Sử dụng toán tử phạm vi để gọi
}
int main()
{
    // Khởi tạo đồ họa
    int graphdriver = DETECT;
    int graphmode;
    initgraph(&graphdriver,&graphmode,
    "C:\\BORLANDE\\BGI");
    MCircle Small(250,100,25,SANS_SERIF_FONT,"you");
    Small.Show();
    Mcircle Medium(250,150,100, TRIPLEX_FONT,"World");
    Medium.Show()
    MCircle Large(250,250,225,GOTHIC_FONT,"Universe");
    Large.Show();
    getch();
    closegraph();
    return 0;
}
```

8.2.2. Sử dụng các lớp cơ sở ảo (Virtual Base Class)

Các lớp cơ sở ảo chỉ được sử dụng trong ngữ cảnh của việc thừa kế bội. Hãy xét cấu trúc cây dưới đây (hình 8.6). Trong cấu trúc cây này, lớp B và lớp C đều được dẫn xuất từ lớp A trong khi lớp D lại được dẫn xuất từ hai lớp B và C. Như vậy lớp D có sử dụng lớp A như một lớp cơ sở. Vấn đề nảy sinh ở đây là có hai lớp cơ sở A riêng biệt hoàn toàn giống nhau được sử dụng như là các lớp cơ sở của lớp D, mỗi lớp cơ sở này đều có dữ liệu riêng của chính mình. Trên thực tế điều này cũng giống như trường hợp có hai người không những sinh đôi mà còn có cùng tên.

Vấn đề này được xử lý như thế nào trong C++. Hãy xét đoạn chương trình sau:

```
class A{
    public:
        int Value;
    };
class B:public A
{
};
class C:public A
{
};
class D:public B, public C {
    public:
        int GetValue() { return Value;}
};
```



Hình 8.6

Trong hàm *GetValue* của *D*, lệnh truy nhập đến thành phần *Value* là không rõ ràng. Ở trường hợp này, trình biên dịch C++ sẽ thông báo lỗi:

"Field "Value" is ambiguous in "D" in function D:: GetValue()"

Lỗi này phát sinh do trình biên dịch không có khả năng phát hiện bản sao nào của *Value* đang được tham chiếu đến. Để giải quyết vấn đề này, chương trình cần phải sử dụng toán tử phạm vi (*scope resolution Operator*) cho hàm *D::GetValue():*

int GetValue() { return C::Value:}

Các hàm ở bên ngoài lớp *D* cũng có thể truy nhập đến từng thành phần *Value* cụ thể trong các lớp nếu ta sử dụng toán tử phạm vi như trong đoạn chương trình dưới đây:

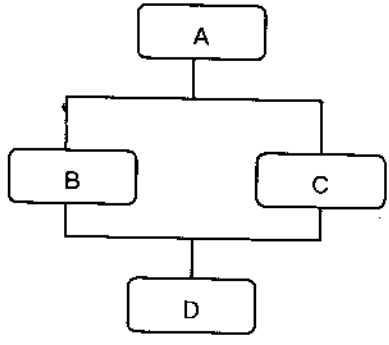
Ví dụ 8.15

```
void main()
{
    D d;
    int v = d.B::value;
    D* object = new D;
    int w = object->C::value;
}
```

Việc tồn tại nhiều bản sao của lớp cơ sở khi thừa kế như trong trường hợp trên không những có thể gây ra nhầm lẫn mà còn có thể làm lãng phí bộ nhớ một cách không cần thiết. Để giải quyết vấn đề này, C++ dùng từ khóa *virtual* khi khai báo các lớp cơ sở. Với cách khai báo này, trình biên dịch chỉ được phép đưa ra một bản sao duy nhất của lớp cơ sở khi lớp này được khai báo trong lớp dẫn xuất. Như vậy lớp dẫn xuất *D* có thể được định nghĩa lại như sau:

```
class B: public virtual A { } ;
class C:public virtual A{ } ;
```

```
class D:public B, public C
    {
    public:
        int Getvalue( ) { return Value;}
    };
```



Hình 8.7

Trong cách khai báo này, các từ khóa *public* và *virtual* có thể thay đổi vị trí cho nhau, điều này không làm ảnh hưởng đến ý nghĩa của các từ khóa đó. Qua việc khai báo trên, hàm *GetValue()* có thể được gọi trong *D* theo cách bình thường mà không cần phải sử dụng toán tử phạm vi. Như vậy sau khi sử dụng từ khóa *virtual* khi khai báo các lớp *B* và *C*, cấu trúc cây của sự thừa kế sẽ có dạng như hình 8.7.

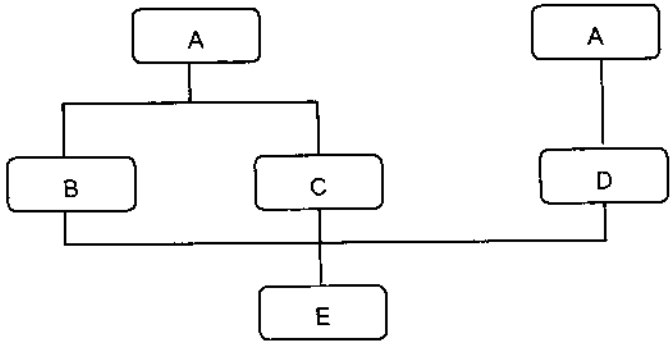
8.2.3. Sử dụng các lớp cơ sở virtual và Non-Virtual cùng nhau

Trong phần này ta sẽ gọi các lớp cơ sở không được khai báo với từ khóa virtual là các lớp *Non-Virtual*. Một lớp trong C++ có thể được dẫn xuất từ cả hai loại lớp cơ sở: *virtual* và *Non-Virtual*. Điều này có thể được mô tả qua ví dụ sau:

```
class A{ };
```

```
class B: public virtual A{ };
class C: public virtual A{ };
class D: public A{ };
class E: public B, public C, public D{ };
```

Các lớp này thể được mô tả qua cấu trúc cây ở hình 8.8



Hình 8.8

Theo định nghĩa của các lớp thì lớp *D* được dẫn xuất từ hai bản sao của lớp *A*. Tuy vậy, lớp A lại được định nghĩa như một lớp cơ sở ảo trong hai lớp *B* và *C*. Điều này không làm phát sinh lỗi trong C++ nhưng có thể gây ra sự nhầm lẫn cho người sử dụng. Trong ví dụ trên, khi một đối tượng của lớp *E* được khởi tạo thì các *constructors* của các lớp cơ sở sẽ được gọi theo thứ tự *A, B, C, D*. Thứ tự gọi các *constructor* của các lớp cơ sở được xác định theo nguyên tắc sau:

- Đầu tiên các *constructor* của các lớp cơ sở ảo sẽ được gọi theo thứ tự xuất hiện của chúng trong khi khai báo lớp dẫn xuất.
- Sau khi *constructor* của tất cả các lớp cơ sở ảo được gọi, *constructor* của các lớp cơ sở còn lại (Non-Virtual) sẽ

được gọi. Thứ tự thực hiện của các *constructor* này cũng được quyết định bởi thứ tự khai báo của các lớp đó trong lớp dẫn xuất.

8.2.4. Sử dụng phép chuyển kiểu

Phép chuyển kiểu từ một con trỏ chỉ đến lớp dẫn xuất đến một con trỏ chỉ đến lớp cơ sở có thể được thực hiện trong cả hai trường hợp thừa kế đơn và thừa kế bội. Tuy vậy, đối với trường hợp thừa kế bội, phép chuyển kiểu này có thể gặp khó khăn do sự không rõ ràng khi thừa kế. Cần lưu ý là trong cả hai trường hợp phép chuyển kiểu ngược lại là không được phép.

Trong trường hợp thừa kế đơn, có thể xét ví dụ sau:

Ví dụ 8.16

```
class A{ };
class B:public A{ } ;
void main()
{
    B* b1 = new B; // Khai báo bình thường
    B* b2 = new A; // Chuyển kiểu từ A sang B - Sai
    A* a1 = new B; // Chuyển kiểu từ B sang A - Đúng.
}
```

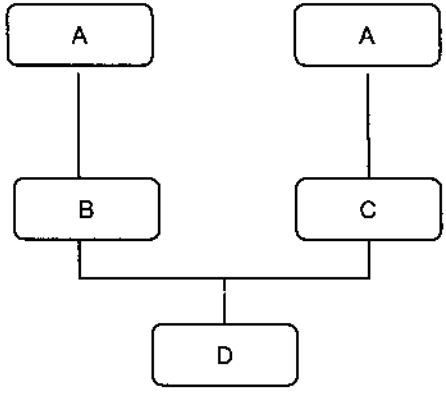
Trong ví dụ trên, khi thực hiện phép gán, trình biên dịch sẽ thực hiện việc chuyển kiểu của toán hạng bên phải phép gán sang kiểu của toán hạng bên trái.

Đối với sự thừa kế bội thì việc chuyển kiểu của con trỏ hoặc tham chiếu không phải lúc nào cũng thực hiện được. Để minh hoạ cho điều này, hãy xét đoạn chương trình sau:

Ví dụ 8.17

```
class A{
    public:
        int value;
};
class B: public A{ };
class C: public A{ };
class D: public B, public C { };
void main()
{
    A* a1 = new D; // Không thực hiện được điều này
    A* a2 = new B;
    A* a3 = new C;
}
```

Sự thừa kế này có thể được mô tả qua cấu trúc cây sau:



Hình 8.9

Lệnh A* a1 = new *D*; sẽ phát sinh lỗi khi biên dịch do khi gặp toán lệnh này trình biên dịch sẽ phát hiện rằng *D* sử dụng hai phiên bản của *A* và nó sẽ không có khả năng chọn phiên bản

nào để thực hiện việc chuyển kiểu. Để có thể thực hiện việc chuyển kiểu, câu lệnh này cần đổi lại như sau:

```
A* a1 = (C*) new D;
```

Với câu lệnh này, khi thực hiện việc chuyển kiểu trình biên dịch sẽ có thể nhận biết và sử dụng phiên bản của lớp cơ sở *A* trong lớp *C*.

Phép chuyển kiểu trên cũng có thể được thực hiện nếu A được khai báo theo kiểu *virtual* trong các lớp dẫn xuất *B* và *C*. Khi đó *D* sẽ sử dụng một phiên bản duy nhất của *A* khi thừa kế hai lớp *B* và *C*. Điều này sẽ không gây cho trình biên dịch sự nhầm lẫn khi thực hiện việc chuyển kiểu. Các lớp *B* và *C* khi đó được khai báo lại như sau:

```
class B: public virtual A{ } ;
class C: public virtual A{ } ;
```

Khi một hàm nào đó trong lớp dẫn xuất được gọi, trình biên dịch sẽ tìm hàm tương ứng trong các lớp cơ sở hoặc dẫn xuất để thực hiện. Việc tìm kiếm này được tiến hành ở thời điểm biên dịch chứ không phải ở thời điểm thực hiện chương trình và tuân theo các nguyên tắc đã được xác định trước:

- Nếu hàm không được tìm thấy trong lớp mà nó được gọi, trình biên dịch sẽ bắt đầu tìm kiếm trên cấu trúc cây.
- Trong cấu trúc cây này việc tìm kiếm lại được tiến hành theo quy tắc vùng). Quy tắc này chỉ ra rằng nếu 2 lớp đều có chứa hàm cần tìm và nếu một lớp được dẫn xuất từ một lớp khác thì vùng cần chọn sẽ là lớp dẫn xuất, tức là hàm được gọi là hàm nằm trong lớp dẫn xuất.

Hãy xét một ví dụ sau:

Ví dụ 8.18

```
class A{
    public:
        int foo() { return 1;}
};
class B: public virtual A{
    public:
        int foo(){ return 2;}
    };
class C: virtual public A, public B
{
}
void main()
{
    C c;
    int i = c.foo(); // Gọi hàm B::foo()
}
```

Khi thực hiện toán lệnh *int i = c.foo()*; trong hàm *main*, trình biên dịch tìm thấy trong hai lớp *A* và *B* đều có chứa hàm *foo* cần tìm. Tuy vậy do lớp *B* được dẫn xuất từ lớp *A* do đó trình biên dịch sẽ gọi *B::foo()* theo quy tắc vùng đã chỉ ra ở trên.

Chương IX

ĐỊNH NGHĨA CHỒNG CÁC HÀM VÀ TOÁN TỬ

9.1. HÀM ĐƯỢC ĐỊNH NGHĨA CHỒNG

Trong khi ngôn ngữ C không cho phép người lập trình sử dụng cùng một tên cho hai hàm trong cùng một chương trình thì trong C++ việc định nghĩa này lại hoàn toàn hợp lệ nếu các hàm này khác nhau về số lượng hoặc kiểu của các đối số được truyền cho hàm. Các hàm như vậy được gọi là các hàm được định nghĩa chồng. Cần chú ý rằng kiểu của các hàm không được sử dụng để phân biệt các hàm số đó. Như vậy hai hàm số *void Display()* và *long Display()* là hoàn toàn giống nhau và do đó sẽ phát sinh lỗi khi biên dịch. Các hàm thành phần của lớp và các hàm không thành phần đều có thể được định nghĩa chồng.

9.1.1. Các phép chuyển kiểu của người sử dụng

Cũng như ANSI C, C++ sử dụng một số quy tắc chuyển kiểu chặt chẽ đối với các kiểu dữ liệu chuẩn. Chẳng hạn kiểu *char* có thể được chuyển lên kiểu *int* và *int* lại có thể chuyển thành kiểu *float* hoặc *double* tùy theo từng trường hợp. Đối với dữ liệu mới do người sử dụng định nghĩa (dữ liệu kiểu lớp) thì các phép chuyển kiểu này phải được định nghĩa thông qua việc định nghĩa chồng các *constructor* hoặc qua các hàm chuyển kiểu đặc biệt.

9.1.1.1. Constructor được định nghĩa chồng (Overloaded Constructor

Phép chuyển kiểu sử dụng *Overloaded Constructor* là phép chuyển kiểu hay được sử dụng nhất trong C++.

Hãy xét ví dụ sau:

Ví dụ 9.1

```
class Example
{
    int value ;
public:
    Example( ) { value =0;}
    int GetValue( )
        {
        return value;
        }

    };
void foo(Example& e)
{
    int x = e.Getvalue( ) +300;
    cout << x;
}

    void main( )
    {
    foo(100);
    }
```

Sau khi biên dịch, trình biên dịch sẽ thông báo lỗi *"could not find match for 'Example :: Example(int)' "*; Điều này có nghĩa

là không tìm thấy constructor tương ứng *'Example :: Example(int).*

Hãy chú ý đến định nghĩa của hàm *foo* và lời gọi hàm *foo(100).* Hàm này nhận tham số là một đối tượng thuộc lớp *Example* và do vậy bất cứ mọi phép gọi hàm nào đối với *foo* với tham số thực không phải là đối tượng thuộc lớp *Example* đều phát sinh lỗi khi biên dịch bởi vì trình biên dịch sẽ không thể chuyển tham số đó thành đối tượng thuộc *Example.*

Lỗi trên có thể được khắc phục nếu ta định nghĩa thêm một constructor với tham số truyền

```
Example(int v)
{
value =v;
}
```

trong thân lớp Example. Constructor này sẽ được sử dụng khi cần thiết thực hiện việc chuyển kiểu dữ liệu từ *int* sang đối tượng của lớp Example. Một cách chi tiết hơn, trong chương trình khi gặp lời gọi hàm *foo(100)* trình biên dịch sẽ thực hiện việc chuyển giá trị 100 từ kiểu *int* sang kiểu đối tượng của lớp *Example.* Để làm điều này trình biên dịch sẽ gọi constructor với tham số truyền kiểu *int* (giá trị của *v =100*) và thực hiện câu lệnh *value = 100.* Như vậy sau phép chuyển kiểu, số nguyên 100 đã được chuyển thành đối tượng của lớp *Example* với dữ liệu thành phần *value* bằng 100. Điều cuối cùng chương trình cần thực hiện là các câu lệnh trong thân của *foo*: cộng giá trị *value* của đối tượng này với giá trị 300 và hiển thị lên màn hình.

Cần lưu ý là mặc dù chương trình đã được thực hiện tốt nhưng để tránh việc đưa ra thông báo khi biên dịch ta có thể thực hiện việc ép kiểu bằng cách sử dụng lời gọi hàm

foo((Example)100); hay *foo(Example(100))*; thay cho foo(100);

Như vậy để có thể định nghĩa một qui tắc chuyển kiểu từ dữ liệu kiểu *T* nào đó thành kiểu của lớp ta cần phải định nghĩa một *constructor* của lớp với đối số duy nhất thuộc kiểu *T*. Trình biên dịch sẽ sử dụng *constructor* này bất cứ lúc nào khi cần chuyển từ kiểu *T* sang kiểu đối tượng của lớp.

9.1.1.2 Chuyển kiểu giữa các lớp

Cũng tương tự như trên, trình biên dịch có thể thực phép chuyển kiểu từ lớp này sang lớp khác nếu người sử dụng định nghĩa một qui tắc chuyển tương ứng.

Ví dụ 9.2

```
class Example {
public:
    int word:
    Example( ) { word =0;}
};
class Counter {
    int value;
public:
    Counter( ) { value =0;}
    Counter(Example);
};
//định nghĩa qui tắc chuyển kiểu
Counter::Counter(Example e) {value = e.word;}
void foo(Counter c){ ....}
void main()
```

```
{
Example e;
foo(e); // Gọi qui tắc chuyển kiểu từ Example to Counter
}
```

Khi hàm *main* gọi *foo(e)*, do đối số của hàm *foo()* phải thuộc kiểu *Counter*, trình biên dịch sẽ sử dụng *constructor Counter::Counter(Example e)* sẽ tương đương với phép gọi hàm *foo(Counter(e));*

9.1.1.3. Sử dụng phép chuyển kiểu đặc biệt

Phần trên đã xét việc chuyển kiểu từ một đối tượng nào đó (chẳng hạn *int, double* hoặc lớp) sang kiểu của một lớp. Việc chuyển kiểu ngược lại từ đối tượng thuộc kiểu lớp sang các kiểu khác (chẳng hạn *char, int, double, char* v.v.* hoặc lớp) cũng có thể được thực hiện với một số giới hạn nào đó.

Để thực hiện phép chuyển kiểu, cần phải định nghĩa một phép chuyển kiểu bắt buộc (*typecasting operator*). Phép chuyển kiểu bắt buộc là một loại toán tử đặc biệt mà không cần định nghĩa kiểu của giá trị trả về. Trình biên dịch sẽ tự động sử dụng kiểu của toán tử như kiểu giá trị được trả về. Tuy vậy, cần lưu ý không được sử dụng các hàm chuyển kiểu dạng này để thực hiện phép chuyển kiểu từ *lớp* sang *lớp*, từ *lớp* vào *enumeration* và các kiểu định nghĩa của người sử dụng có sử dụng chỉ thị *typedef*.

Hãy xét một ví dụ chuyển kiểu từ lớp sang kiểu char* trong ví dụ đơn giản về lớp chuỗi - lớp String:

Ví dụ 9.3

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

```
#include < string.h>
class String
{
    char * char_ptr;
    unsigned int Len;
public:
    // Constructor
    String(int size =80 );
    String(String& str);
    String (const char* text);
    // Các phương thức trả về giá trị
    unsigned int Length( );
    // Phương thức hiển thị đối tượng
    void Show( );
    // Phép chuyển kiểu từ đối tượng sang kiểu char*
    operator char* ( );
    //Destructor
    ~String( ){delete char_ptr;}
};
// Phương thức khởi tạo một đối tượng với dữ liệu char* //rỗng
String::String(int size)
{
    Len = size;
    char_ptr = new char[Len+1];
    *char_ptr = '\0';
}
// Constructor khởi tạo đối tượng với một chuỗi cho trước
```

```
String::String(const char* text)
{
    Len = strlen(text);
    char_ptr  = new char[Len +1];
    strcpy(char_ptr,text);
}
// Constructor sao chép
String::String(String& Other_String)
{
    Len = Other_String.Len;
    char_ptr = new char[Len+1];
    strcpy(char_ptr, Other_String.char_ptr);
}
// phương thức trả về độ dài của đối tượng
unsigned int String::Length( )
{
    return Len;
}

// Phép chuyển kiểu từ String sang char *
String::operator char* ( )
{
     return char_ptr;
}
void String::Show( )
{
     cout << char_ptr <<endl;
```

```
}

void main()
{
// Sử dụng Constructor với tham số truyền
String S("Ngon ngu C");
S.Show( );
// Sử dụng constructor sao chép
String S1 = S;
// Hiển thị nội dung của đối tượng
S1.Show( );
// Sử dụng phép chuyển kiểu gán cho biến
char * s = (char *) S;
cout << s;
// Sử dụng phép chuyển kiểu
cout <<(char*)S;
}
```

Như vậy việc chuyển kiểu một đối tượng String sang kiểu char* được thực hiện bằng phương thức operator char* ( ); Các phép chuyển kiểu từ đối tượng sang *int, double, char* cũng được làm tương tự.

9.2. CÁC TOÁN TỬ ĐƯỢC ĐỊNH NGHĨA CHỒNG (OVERLOADING OPERATOR)

Như ta đã biết, ngoài các kiểu dữ liệu chuẩn như *char, int, double* v.v , C++ còn cho phép người sử dụng định nghĩa thêm các loại dữ liệu khác thông qua các lớp. Tuy vậy các toán tử như +, -,

*, /, +=, -=, ++ - - hoặc *new* và *delete* lại không thể được thực hiện trên các kiểu dữ liệu mới này. Chẳng hạn, không thể sử dụng các toán tử này để nhân hoặc cộng các đối tượng thuộc kiểu số phức với nhau, cũng như không thể cộng hai đối tượng kiểu chuỗi v.v. Để có thể sử dụng các toán tử này trên các kiểu dữ liệu mới, C++ đưa ra thêm một khả năng rất tiện lợi cho người lập trình- khả năng định nghĩa chồng các toán tử.

Phần này sẽ nghiên cứu việc xây dựng các toán tử chồng cho các lớp. Gọi là các toán tử chồng bởi vì trong C++ các toán tử này cũng đã được định nghĩa cho các kiểu dữ liệu thường dùng như *int, char, double* v.v.

9.2.1. Sử dụng toán tử như hàm thành phần hoặc hàm bạn

Các toán tử có thể được định nghĩa cho các lớp theo hai cách: như các *hàm thành phần (member function)* hoặc như các *hàm bạn (friend function).*

9.2.1.1. Các toán tử một ngôi (Unary Operator)

Giả sử W là một đối tượng thuộc một lớp nào đó, @ là một toán tử đã được định nghĩa dành cho các đối tượng của lớp. Khi đó biểu thức @W được gọi là phép toán một ngôi thực hiện trên đối tượng W. Để thực hiện được phép toán này trong thân lớp cần phải chứa khai báo của toán tử @ theo cách sau:

Kiểu_trả_ về operaror@ ( );

Việc định nghĩa toán tử này có thể được thực hiện theo kiểu *inline* hoặc không.

Nếu toán tử này được thực hiện theo kiểu toán tử bạn thì việc khai báo này phải được viết như sau:

friend kiểu_trả_về operator @(Kiểu của đối tượng W);

trong đó kiểu của đối tượng chính là tên lớp của *W*.

Ví dụ 9.4

Xây dựng lớp số phức Complex với toán tử thành phần phủ định (!). Phép phủ định là phép lấy giá trị đổi dấu của dữ liệu

```
#include <iostream.h>
class Complex
{
    double real;
    double Imag;
public:
    Complex( )   //Constructor ngầm định
    {
        real =0;
        imag =0;
    }
    Complex (double, double); // Constructor với tham số truyền
    Complex(Complex &); // Constructor sao chép
    Complex operator !( );
    double GetReal( )          // Lấy phần thực
    {
        return real;
    }
    double GetImag( )          // Lấy phần ảo
    {
        return imag;
```

```
    }
};
Complex::Complex(double r, double i)
{
    real = r;
    imag = i;
}
Complex::Complex(Complex &c)
{
    real = c.real;
    imag = c.imag;
}
Complex Complex:: operator !( )
{
    return Complex(-real, -imag);
}
void main( )
{
    Complex C1(12.5, 13,5);
    Complex C2 = !C1; // Sử dụng constructor sao chép
     cout<< "C2.real = "<<C2.GetReal( )<< " C2.Imag = "  <<
    C2.GetImag( );
}
```

Kết quả thực hiện của chương trình sẽ là:

```
C2.real = -12.5  C2.imag = -13.5
```

Để thực hiện được phép toán một ngôi !, trong ví dụ cần phải định nghĩa toán tử này. Trong phép định nghĩa, kiểu trả về của

toán tử sau khi thực hiện là *Complex*, sau khi thực hiện toán tử trả về một đối tượng có các thành phần dữ liệu đổi dấu với dữ liệu của đối tượng mà toán tử thực hiện trên nó. Điều này được thực hiện bằng dòng lệnh *return Complex(-real, -imag);*

Thay vì toán tử thành phần, toán tử bạn có thể được khai báo trong thân lớp bằng cách sau:

```
friend Complex operator @ (Complex );
```

Ở ngoài thân lớp toán tử này được định nghĩa:

```
Complex operator @(Complex c)
{
    return Complex(-c.real, -c.image);
}
```

Cần lưu ý là khi thực hiện phép toán *!C1* nếu ta sử dụng toán tử thành phần, *C1* sẽ là đối tượng gọi phép toán (thay thế cho đối tượng ẩn) trong trường hợp sử dụng toán tử bạn thì *C1* chính là đối tượng sử dụng như tham số truyền của lời gọi toán tử bạn này.

9.2.1.2. Các toán tử hai ngôi (Binary Operator)

Giả xử *X* và *Y* là hai đối tượng của một lớp và @ là toán tử hai ngôi. Để toán tử này có thể được thực hiện trên hai đối tượng thì trong thân lớp phải chứa định nghĩa chồng của toán tử này.

Cũng như toán tử một ngôi, việc định nghĩa toán tử hai ngôi có thể được thực hiện bằng toán tử thành phần hoặc bằng toán tử bạn.

Toán tử thành phần hai ngôi được khai báo và định nghĩa theo cách sau:

```
Kiểu_trả_về  TênLớp::operator @ (Tên lớp Y1)
{
.....
}
```

Toán tử bạn hai ngôi được khai báo và định nghĩa theo cách sau:

```
friend Kiểu_trả_về Tên lớp (Tên lớp X1, Tên lớp X2)
{
.....
}
```

Ví dụ 9.5

Xây dựng lớp Complex với định nghĩa chồng toán tử hai ngôi - toán tử cộng (+)

Để có thể thực hiện phép toán + trên hai đối tượng của lớp Complex, trong lớp Complex đã được định nghĩa ở ví dụ trên cần phải thêm định nghĩa chồng của toán tử hai ngôi này.

Sử dụng toán tử thành phần:

```
Complex Complex:: operator + (Complex C )
{
    return Complex(real + C.real, imag,+C.imag);
}
```

Khi thực hiện biểu thức *C1 + C2*, C++ sẽ gọi toán tử thành phần đã được định nghĩa trên. Trong khi thực hiện, *C1* là đối tượng gọi toán tử thành phần và sẽ thay thế cho đối tượng ẩn và *C2* chính là tham số thực sẽ được truyền cho *C*.

Sử dụng toán tử bạn:

```
friend Complex operator +( Complex C1, Complex C2)
{
return Complex(C1.real + C2.real, C1.imag + C2.imag);
}
```

Nếu sử dụng toán tử bạn hai ngôi khi thực hiện biểu thức *X1 + X2*, trong đó *X1* và *X2* là hai đối tượng của lớp Complex, C++ sẽ gọi toán tử bạn đã được định nghĩa trên khi đó *X1* và *X2* sẽ là các tham số được truyền tương ứng cho *C1* và *C2*.

Hàm main khi đó có thể được viết lại như sau:

```
void main ( )
{
    Complex C1(12.5, 20.5);
    Complex C2(25.5, 14.5);
    Complex C3 = !C1;
    cout << "C3.real = "<<C3.GetReal( )<< " C3.Imag = "<<
    C3.GetImag() <<endl;
    Complex C4 = C1 + C2;
    cout<< "C4.real = "<<C4.GetReal( )<< " C4.Imag = " C4.GetImag( );
}
```

Kết quả thực hiện của chương trình sẽ là:

```
C3.real = -12.5     C3.imag  = -20.5
C4.real = 38        C4.imag = 35
```

Bằng cách tương tự, bạn đọc có thể định nghĩa thêm các

phép toán trừ –, *, /, +=, –= v.v. cho lớp Complex.

Trong ví dụ 9.5, ta đã định nghĩa các toán tử thực hiện trên các dối tượng của cùng một lớp, như vậy các toán tử này không thể sử dụng cho các toán hạng thuộc các kiểu khác. Có thể sử dụng hai cách để thực hiện các phép toán này:

- Sử dụng phép chuyển kiểu

Để thực hiện biểu thức *C1 + 2*, với C1 là đối tượng thuộc lớp *Complex*, thì trước tiên số 2 (kiểu *int*) cần được chuyển về kiểu Complex và sau đó sử dụng phép + hai số phức để thực hiện biểu thức. Việc chuyển kiểu từ *int* sang *Complex* đòi hỏi phải định nghĩa thêm một constructor với tham số truyền duy nhất kiểu *int* trong thân lớp:

```
Complex( int v)
{
    real = v;
    imag = 0;
}
```

Sử dụng constructor này, sau khi chuyển kiểu số nguyên hai sẽ được chuyển về đối tượng Complex có phần thực *real* = 2 và phần ảo *imag* =0;

Sử dụng constructor chuyển kiểu, biểu thức C1 + 2 cần được viết lại rõ ràng như sau:

```
C1 + (Complex)2;
```

- Sử dụng phép định nghĩa chồng

Thay vì sử dụng phép chuyển kiểu ta có thể thực hiện phép cộng một số phức với một số nguyên bằng cách định nghĩa chồng

phép toán này bằng toán tử thành phần:

```
Complex Complex::operator + (int v)
{
    return Complex(real + v, imag);
}
```

Hay bằng toán tử bạn:

```
friend Complex operator +( Complex C, int v)
{
    return Complex(C.real + v, C.imag);
}
```

Bằng các toán tử này, phép cộng một số phức C với một số nguyên v sẽ cho một số phức có phần thực $real = C.real + v$ và phần ảo $imag = C.imag$.

Sử dụng hoặc phép chuyển kiểu hoặc định nghĩa chồng toán tử cộng số phức với số nguyên, hàm *main* khi đó có thể được viết như sau:

```
void main ( )
{
    Complex C1(12.5, 20.5);
    Complex C3 ; // Sử dụng constructor ngầm định
    C3 = C1 + 2; // Sử dụng định nghĩa chồng toán tử cộng
    // Nếu sử dụng phép chuyển kiểu: C3 = C1 +(Complex)2
```

```
cout<< "C3.real = "<<C3.GetReal( )<< " C3.Imag= <<C3.GetImag( )
}
```

Kết quả thực hiện của chương trình là:

C3.real = 14.5

C3.imag = 20.5

Bạn đọc có thể tự định nghĩa chồng phép toán trừ, nhân chia số phức với số *int* v.v.

Có thể chỉ ra sự khác biệt trong việc sử dụng toán tử thành phần và toán tử bạn.

Như đã đề cập, việc sử dụng một toán tử thành phần có thể được xem như một cách gọi hàm thành phần trong một lớp. Chẳng hạn trong ví dụ trên, khi toán tử thành phần + của lớp Complex đã được định nghĩa thì biểu thức *C1 + C2* có thể được xem như kết quả của phép gọi hàm thành phần *C1.operator +(C2)*. Trong phép gọi này *C1* là đối tượng đang gọi hàm thành phần và là đối tượng ẩn và việc chuyển kiểu không thể thực hiện được đối với đối tượng này. Như vậy khi sử dụng toán tử thành phần thì việc thực hiện biểu thức 2 + C1 sẽ không thể thực hiện được.

Cách giải quyết vấn đề này là sử dụng toán tử bạn hai ngôi. Với toán tử này, C++ sẽ cho phép việc chuyển đổi với cả hai toán hạng của phép toán. Với điểm khác biệt này các toán tử bạn trở thành vô cùng tiện lợi và rất hữu ích khi lập trình. Như vậy nếu sử dụng phép chuyển kiểu và toán tử bạn hai ngôi thì biểu thức 2 + C1 sẽ không còn phát sinh lỗi.

9.2.2. Toán tử gán

Cũng giống như *constructor* và *destructor*, toán tử gán là

một trong các toán tử cơ bản luôn gắn liền với một lớp, bởi vậy toán tử này chỉ được phép khai báo bằng hàm thành phần mà không được khai báo bằng lớp bạn. Trước hết hãy phân biệt rõ hai khái niệm: *constructor* sao chép và toán tử gán. Cả toán tử gán và *constructor* sao chép đều nhằm sao chép giá trị của một đối tượng đã tồn tại sang một đối tượng khác. Tuy vậy, điểm khác nhau giữa *constructor* sao chép và toán tử gán là ở chỗ: *constructor* sao chép luôn gắn liền với việc khởi tạo một đối tượng và do vậy chỉ được thực hiện một lần cho một biến, trong khi toán tử gán lại được sử dụng để gán giá trị cho một đối tượng đã được khởi tạo và có thể sử dụng nhiều lần cho đối tượng đó.

Cần phân biệt giữa hai cách định nghĩa toán tử gán sau:

```
Complex& Complex::operator =(Complex C )
{
    real = C.real;
    imag = C.imag;
    return *this
}

void Complex::operator =(Complex C )
{
    real = C.real;
    imag = C.imag;
}
```

Theo phép định nghĩa thứ nhất thì khi thực hiện phép gán C1 = C2, các thành phần dữ liệu của C2 sẽ được sao chép cho C1 và kết quả của phép gán sẽ tham chiếu đến đối tượng C1 do sự thực hiện của câu lệnh return *this (C1 chính là đối tượng ẩn).

Như vậy nếu thực hiện biểu thức (C1 = C2) + C3 thì sau khi dữ liệu của C2 được gán cho C1, biểu thức (C1 = C2) sẽ tham chiếu đến C1 và sau đó đối tượng C1 sẽ được cộng với C3.

Với phép định nghĩa thứ hai, khi thực hiện phép gán *C1 = C2*, các thành phần dữ liệu của C2 cũng sẽ được sao chép cho C1. Sự khác biệt ở đây là phép toán không trả lại kết quả và do vậy biểu thức *(C1 = C2) + C3* không thể thực hiện được.

Khi sử dụng phép gán cần chú ý các điểm sau đây:

- Trong trường hợp toán tử gán không được định nghĩa khi khai báo một lớp thì khi cần thực hiện việc gán giá trị, trình biên dịch sẽ tự động sử dụng phép gán ngầm định bằng cách sao chép từng thành phần đối tượng.
- Các lớp dẫn xuất không kế thừa toán tử gán của lớp cơ sở.

Vấn đề đặt ra ở đây là tại sao chúng ta lại không sử dụng phép gán ngầm định cho các đối tượng của một lớp? Câu trả lời là trong nhiều trường hợp đối với các lớp phức tạp, việc sử dụng toán tử gán ngầm định không thể đáp ứng được yêu cầu đặt ra của bài toán theo ý của người sử dụng.

9.2.3. Sử dụng toán tử gọi hàm - operator()

Trong ngôn ngữ C, toán tử *()* được sử dụng khi cần gọi một hàm nào đó. Ta đã rất quen biết với cách gọi hàm này, chẳng hạn *getch()* là hàm cho phép nhận một ký tự từ bàn phím, *sqrt(double m)* là hàm trả lại căn bậc hai của đối số *m*. Ngoài công việc này, hầu như khi lập trình bằng ngôn ngữ C, toán tử *()* không được dùng cho một mục đích nào khác.

Trong ngôn ngữ C++, toán tử gọi hàm *()* cũng có thể được định nghĩa chồng. Tuy vậy khác với toán tử gán, toán tử gọi hàm

() không phải là toán tử ngầm định cho một lớp và trên thực tế toán tử này chỉ được định nghĩa chồng trong những trường hợp thật cần thiết. Khi đã được định nghĩa chồng cho một lớp, toán tử gọi hàm *()* có thể được sử dụng qua cách sau: *a(x,y), b(x,y,z)*, trong đó *a* và *b* là các đối tượng của một lớp *x,y,z* là các biến thuộc một kiểu bất kỳ. Cách sử dụng này giống như gọi phần tử mảng của đối tượng. Tất nhiên, toán tử gọi hàm thực hiện điều gì thì còn phụ thuộc vào các lệnh được sử dụng khi định nghĩa chồng toán tử đó.

Cũng như các toán tử được định nghĩa chồng, toán tử gọi hàm được định nghĩa trong lớp qua hàm thành phần với cách sau:

Kiểu_trả_về operator () (danh sách các đối số)

Khi định nghĩa chồng toán tử gọi hàm cần lưu ý:

- Không sử dụng hàm thành phần kiểu static để khai báo cho toán tử gọi hàm.
- Không sử dụng hàm bạn để khai báo cho toán tử gọi hàm.

Thông thường toán tử gọi hàm được định nghĩa chồng để làm việc với mảng. Hãy xét ví dụ đơn giản sau:

Ví dụ 9.6

```
#include <iostream.h>
class Matrix
{
    int maxRows;
    int maxCols;
```

```
    double *pData;
public:
    Matrix(int MaxRows =1, int MaxCols =1)
    {
        maxCols = MaxCols;
        maxRows = MaxRows;
        pData = new double [maxRows * maxCols];
    }

    ~Matrix( )
    {
        delete [ ] pData;
    }
    void  ReSize( int NewRows, int NewCols)
    {
        delete [ ] pData;
        maxCols = NewCols;
        maxRows = NewRows;
        pData = new double [maxRows* maxCols];
    {

    double& operator ( ) (int  row, int col)
    {
        return pData[col + maxCols * row];
    }
    int getRows( )
    {
```

```
        return maxRows;
    }

        int getCols( )
    {
        return maxCols;
    }
};
void main()
{
Matrix M(2,2);
for(int i = 0; i<2; i++)
    for(int j = 0;j <2; j ++)
        M(i,j) = i*2 + j;
for(i=0;i < 2; i++)
    for(int j=0;j < 2; j++)
        cout<< M(i,j)<<endl;
}
```

Trong ví dụ này hàm *main* chỉ làm một việc đơn giản là khởi tạo một đối tượng ***M*** của ***Matrix*** với hai hàng hai cột. Bằng việc sử dụng toán tử thành phần ( ), các phần tử của mảng này được khởi tạo giá trị và sau đó hiển thị lên màn hình.

Khi sử dụng toán tử gọi hàm, có thể nhận thấy cách sử dụng của toán tử này hoàn toàn giống với việc sử dụng *constructor* để khởi tạo đối tượng nếu *constructor* tương ứng được định nghĩa. Tuy vậy, mặc dầu việc khai báo đối tượng và

việc sử dụng toán tử gọi hàm có sự giống nhau, trình biên dịch vẫn có khả năng phân biệt hai trường hợp đó khi thực hiện.

9.2.4. Định nghĩa chồng toán tử [ ]

Ngoài việc sử dụng khi cần làm việc với mảng như trong ngôn ngữ C, tổ hợp ký tự *[]* còn có thể được định nghĩa chồng trong C++. Trong C++ toán tử *[]* được gọi là toán tử chỉ số (*subscripting Operator*). Khi sử dụng toán tử này cần lưu ý:

- Toán tử *[]* là toán tử có hai toán hạng và do đó chỉ được sử dụng với một đối số duy nhất. Như vậy việc sử dụng toán tử trong biểu thức *a = b[20]* là được phép trong khi biểu thức *a = b[20][20]* là sai. Trong trường hợp thứ hai có thể dùng toán tử gọi hàm *a = b(20,20)* để thay thế cho *a = b[20][20]*.
- Không sử dụng hàm thành phần kiểu *static* để khai báo cho toán tử *[]*.
- Không sử dụng hàm bạn để khai báo cho toán tử *[]*.

Dưới đây chúng ta hãy xét một ví dụ mô tả cách định nghĩa chồng toán tử [ ].

Ví dụ 9.7

```
class Vector
{
public:
    Vector(int MaxSize)
    {
        pData = new double[MaxSize];
    }
```

```
    ~Vector( )
    {delete [ ]pData;}
    void ReSize(int NewSize)
    {
        delete [ ] pData;
        pData = new double [maxSize = NewSize];
    }
    double & operator [ ] (int index)
        { return pData[index];}
    double & operator ( ) (int index)
        { return pData[index]; }
    int getSize( )
        { return maxSize;}
    int checkIndex (int index)
        {return (index >= 0 && index <maxSize)? 0:1;}
protected:
    int maxSize;
    double *pData;
};
```

Bạn đọc có thể tự viết hàm *main* để sử dụng lớp *vector* như một bài tập nhỏ.

9.2.5. Định nghĩa chồng toán tử tăng giảm một đơn vị

Trong ngôn ngữ C, ta đã làm quen với các toán tử tăng (++) và giảm (- -) giá trị của biến một đơn vị. Ngắn gọn có thể mô tả cách sử dụng của các toán tử này thông qua ví dụ sau:

```
a =++b;                                    (1)
```

```
a = --b;                    (2)
a = b++;                    (3)
a = b--;                    (4);
```

Trong dòng lệnh (1) và (2) của toán tử này sẽ có tác dụng trước khi biểu thức được thực hiện. Cụ thể ở dòng lệnh (1) hoặc (2) đầu tiên giá trị của *b* tăng lên hoặc giảm đi một đơn vị, sau đó giá trị này của *b* mới được gán cho *a*. Trong các trường hợp này các toán tử ++ và - - được gọi là tiền tố (*prefix*). Trong khi ở ví dụ (3) hoặc (4) giá trị của *b* cho *a*. Trong các trường hợp này, các toán tử ++ và - - được gọi là hậu tố (*postfix*).

Ngoài các toán tử tăng giảm một đơn vị vừa có thể sử dụng như toán tử hậu tố hoặc tiền tố, tất cả các toán tử với một toán hạng trong C đều là toán tử tiền tố. Chẳng hạn với toán tử phủ định khi lập trình, người ta có thể sử dụng *a = !b* chứ không thể sử dụng *a = b!*.

Trong C++, toán tử ++ và - - đều có thể được định nghĩa chồng. Tuy vậy cần lưu ý là chỉ từ phiên bản 3.x, Borland C++ mới đưa thêm khả năng định nghĩa chồng cho các toán tử này khi sử dụng chúng ở dạng hậu tố. Để đưa ra khả năng định nghĩa các toán tử này ở cả hai dạng, C++ sử dụng đối số của các hàm toán tử chồng để phân biệt. Khi định nghĩa chồng các toán tử tăng hoặc giảm một đơn vị cần lưu ý:

- Các hàm toán tử chồng không có đối số khi khai báo được dành cho các toán tử tiền tố (prefix).
- Các hàm toán tử chồng có một đối số kiểu *int* khi khai báo được dành cho các toán tử hậu tố.
- Khi định nghĩa các hàm toán tử, đối số của hàm chỉ có

tác dụng để trình biên dịch phân biệt các hàm toán tử tiền tố và hậu tố mà không được sử dụng trong quá trình thực hiện chương trình. Hãy xét ví dụ sau:

Ví dụ 9.8

```
#include <iostream.h>
class X{
      public:
          int value;
          X(){ value =0;}
          X(int i){ value =i;}
      X& operator ++ ();
      X operator ++ (int);
      X& operator+(X);
      X operator = (X&);
      } ;
X & X::operator ++()
{
      value +=1;
      return *this;
}
X X::operator++(int)
{
       value +=1;
       return X(value);
}
X& X::operator+(X m)
{
```

```
        value+=m.value;
        return *this;
    }
    X X::operator=(X& m)
    {
        value = m.value;
        return X(value);
    }
    void main()
    {
        X x1(23), x2(34),x3;
        x3 = ++x1+1;// (1)
        cout << "Gia tri cua x3.value la "<< x3.value<endl;
        x3 = x2++ +1;//(2)
        cout << "Gia tri cua x3.value la "<< x3.value<< endl;
        x3 = x2+1;
        cout << "Gia tri cua x3.value la "<< x3.value<< endl;
        }
```

Khi thực hiện, chương trình sẽ hiện kết quả sau lên màn hình:

Gia tri cua x3.value là 25

Gia tri cua x3.value là 36

Gia tri cua x3.value là 36

Như vậy có thể nhận thấy, mặc dù trong ví dụ 9.8 toán tử ++ được định nghĩa như toán tử tiền tố và cả như toán tử hậu tố nhưng tác dụng của chúng hoàn toàn giống nhau và giống như toán tử tiền tố. Ở trong cả hai dòng lệnh (1) và (2) đầu tiên

x1.value và *x2.value* đều tăng lên một đơn vị sau đó giá trị này được cộng thêm một và được gán cho *x3.value*. Tuy vậy, ta muốn sử dụng các toán tử hậu tố đúng như ý nghĩa của chúng, tức là trong dòng lệnh (2) đầu tiên giá trị của *x2.value +1* được gán cho *x3* sau đó giá trị của *x2.value* mới tăng lên một đơn vị. Để thực hiện được điều này cần phải thay đổi lại cách định nghĩa của toán tử hậu tố ++. Ví dụ 9.9 sẽ minh họa cho điều này.

Ví dụ 9.9

```
#include <iostream.h>
class X{
      public:
          int value;
          X(){ value =0;}
          X(int i){ value =i;}
          X & operator ++ ();
          X operator ++ (int);
          X& operator+(X);
      X operator = X(&);
      } ;
X& X::operator ++();
      {
      value +=1;
      return *this;
      }
X X::operator++(int)
      {
      X m;
```

```
        m.value = value;
        value +=1;
        return m;
        }
    X& X::operator+operator + (X m)
        {
        value+=m.value;
        return *this;
        }
    X X::operator=(X& m)
        {
        value = m.value;
        return X(value);}
        void main()
        {
    X x1(23), x2(34),x3;
    x3 = ++x1+1;     // (1)
    cout << "Gia tri cua x3.value la "<< x3.value<< endl;
    x3 = x2++ +1;//(2)
    cout << "Gia tri cua x3.value la "<< x3.value<< endl;
    x3 = x2+1;
    cout << "Gia tri cua x3.value la "<< x3.value<< endil;
    }
```

Kết quả thực hiện của chương trình trong trường hợp này sẽ là:

Gia tri cua x3.value la 25

Gia tri cua x3.value là 35

Gia tri cua x3.value là 36

Có thể xét thêm một ví dụ về việc định nghĩa chồng toán tử tăng giảm một đơn vị đối với lớp String. Khi được sử dụng, về thực chất các toán tử này sẽ có tác dụng đối với con trỏ *char* char_ptr* của lớp.

```
// Định nghĩa thêm các toán tử tăng một đơn vị
String String::operator ++( )
{

return String(++char_ptr);
}

String String::operator ++(int)
{

String temp = *this;
char_ptr++;
return temp;
}
```

Để sử dụng các toán tử được định nghĩa chồng này, trong thân của lớp cần thêm các khai báo *String operator ++( )* và *String operator ++ (int);* Các toán tử tăng một đơn vị tiền tố và hậu tố này có thể được mô tả trong hàm *main* như sau:

```
void main()
{
    String S("Ngon ngu C");
    S.Show( );          // Hiển thị "Ngon ngu C"
```

```
// Con trỏ s = char_ptr và sau đó char_ptr tăng một đơn vị
char* s = (char*) S++;
cout<<s<<endl;  // Hiển thị "Ngon ngu C"
// Con trỏ char_ptr tăng một đơn vị  và sau đó s1 = char_ptr
char *s1 = (char*)++S;
    cout <<s1<<endl; // Hiển thị ngon ngu C
    S.Show();              // Hiển thị ngon ngu C
}
```

9.2.6. Định nghĩa chồng toán tử new và delete

Các toán tử *new* và *delete* là hai toán tử chuẩn của C++ và được dùng để cấp phát bộ nhớ động trong quá trình thực hiện chương trình (xem chương I). Chẳng hạn có thể cấp phát bộ nhớ cho 10 phần tử kiểu *int* bằng câu lệnh:

```
int* p = new int [10];
```

hoặc cấp phát bộ nhớ cho một phần tử thuộc kiểu của lớp M:

```
M* m = new M;
```

Trong hai trường hợp này con trỏ *p* và *m* sẽ chỉ đến địa chỉ đầu tiên của vùng được cấp phát và qua con trỏ này có thể truy nhập đến các phần tử tương ứng.

Ngoài việc sử dụng với mục đích chung, các toán tử *new* và *delete* có thể được định nghĩa chồng cho các lớp khác nhau. Khi đã được định nghĩa chồng trong một lớp và được gọi bởi các đối tượng của lớp đó, các toán tử này sẽ được gọi thay cho các toán tử *new* và *delete* mặc định. Điều này sẽ cho phép định nghĩa các qui trình cấp phát bộ nhớ khác nhau cho một lớp. Tuy vậy, việc cấp phát bộ nhớ động là vấn đề rất phức tạp và đòi hỏi nhiều kỹ xảo vì vậy cần hết sức thận trọng khi sử dụng.

Một trong các mục đích của toán tử *new* và *delete* khi cần định nghĩa chồng cho một lớp là các lớp này cần được cấp phát một vùng nhớ riêng để làm việc. Chẳng hạn khi cần làm việc với các đối tượng thuộc kiểu *bitmap* (kiểu ảnh), thông thường chúng ta cần cấp phát bộ nhớ trên vùng nhớ *Video* hoặc vùng nhớ *Ram* để chứa ảnh. Khai báo một lớp *Bitmap*, ta có thể dùng toán tử *new* để trình biên dịch cấp phát bộ nhớ cần thiết chứa ảnh và định nghĩa các hàm thành phần riêng của lớp để tác động lên hình ảnh đó. Hãy xét ví dụ sau:

Ví dụ 9.10

```
#include <stddef.h>
const int BYTES_PER_BITMAP = 100*100;
class BitMap{
      unsigned char* bitmap;
public:
      BitMap();
      void* operator new(zise_t);
      void operator delete(void*);
};
BitMap::BitMap()
{
   bitmap = NULL;
}
void* BitMap::operator new(size_t s)
{
   return new unsigned char* [BYTES_PER_BITMAP];
}
void BitMap::operator delete(void* a)
```

```
{
    delete a;
}
void main()
{
    BitMap* image = new BitMap();
    delete image;
}
```

Để phân bố bộ nhớ cho các hình ảnh, chương trình đã dùng bộ nhớ *Ram* của máy tính. Lớp ***BitMap*** có định nghĩa hai toán tử thành phần cho riêng. Toán tử *new* dùng để cấp phát bộ nhớ gồm *100*100* bytes. Trong định nghĩa chồng của *new*, tham số sẽ được tự động cung cấp cho hàm bởi trình biên dịch trong quá trình gọi. Tham số này được khai báo theo kiểu *size_t* và là kích thước của *unsigned int* (được tính theo đơn vị của *unsigned char)*. Tham số được sử dụng để tạo ra sự tương thích của chương trình trên các loại máy PC khác nhau. Đối với trình biên dịch, một điều cần lưu ý là toán tử *new* được xem là toán tử thành phần kiểu *static* (thậm chí cả khi không được khai báo rõ ràng). Điều này sẽ cho phép toán tử luôn được thực hiện trước khi thực hiện *constructor*. Toán tử *delete* tuy vậy lại không được xem như kiểu *static* do đó luôn được thực hiện sau *constructor*

Ví dụ này thể hiện một nhược điểm: kích thước của ảnh phải được xác định trong quá trình biên dịch. Điều này sẽ gây khó khăn khi cần làm việc với các hình ảnh có kích thước khác nhau. Nhược điểm trên của chương trình sẽ được loại bỏ trong ví dụ dưới đây:

Ví dụ 9.11

```
#include <stddef.h>
#include <graphics.h>
class BitMap{
    int width,hight;
    unsigned char* bitmap;
public:
    BitMap(int, int);
    void* operator new(size_t, int, int);
    void operator delete(void*);
    } ;
BitMap::BitMap(int x, int y)
    {
    width = x;
    hight = y;
    }
void* BitMap:operator new(size_t s, int x, int y)
    {
    return new unsigned char* [x*y];
    }
void BitMap::operator delete(void* a)
    {
    delete a;
    }
void main()
{
    int graphdriver = DETECT, mode;
```

```
    initgraph(&graphdriver,&mode,"C:\\Borland\\bgi");
    BitMap* image = new(100,200 BitMap(100,200);
    outtextxy(1,1,"abcdshdsjhdjahj");
    getimage(1,1, 100,100,image);
    putimage(101,101,image,COPY_PUT);
    delete image;
}
```

Trong chương trình này có một điều rất thú vị, đó là khi sử dụng toán tử *new* để cấp phát bộ nhớ cho đối tượng thuộc kiểu *BitMap*. Toán tử *new* trong trường hợp này đã phân bố bộ nhớ gồm *100*200* cho hình ảnh, đây cũng chính là giá trị *width* và *height* của lớp *BitMap*. Giá trị này mặc dù đã được truyền cho *new* nhưng không thể dùng để khởi tạo các dữ liệu thành phần của lớp do *new* là toán tử thành phần thuộc kiểu *static*. Để thực hiện được điều này cần phải truyền lại các tham số đó cho *constructor* khi khởi tạo một đối tượng thuộc lớp *BitMap*.

Ví dụ này chỉ nhằm minh họa cho việc định nghĩa chồng các toán tử *new* và *delete*. Các kỹ thuật xử lý ảnh, không phải là mục đích của giáo trình vì vậy các hàm thành phần của lớp dùng để thực hiện các thao tác xử lý trên ảnh không được đề cập đến.

9.2.7. Toán tử nhập xuất đối với dữ liệu của người sử dụng

Các dòng nhập xuất (stream) trong C++ có sử dụng hai toán tử đã được định nghĩa chồng để thực hiện thao tác nhập và xuất dữ liệu trên *stream*. Hai toán tử này >> (nhập dữ liệu) và << (xuất dữ liệu) không những chỉ áp dụng cho các loại dữ liệu chuẩn trong C++ (xem chương II) mà còn có thể phục vụ cho các thao tác nhập và xuất dữ liệu do người sử dụng định nghĩa. Để minh họa cho điều này, hãy xét ví dụ 9.12.

Ví dụ 9.12

```
#include <iostream.h>
class Point
{
    float x, y, z;
public:
    Point(float i, float j, float k)
    {
        x =i;
        y = j;
        z =k;
    }
        friend ostream& operator << (ostream& os, Point& p);
        friend istream& operator >> (istream& os, Point& p);
};
// Định nghĩa chồng toán tử <<
ostream& operator << (ostream& os, Point& p)
{
    return os << '('
        << p.x << " "
        << p.y << " "
        << p.z << ")"
}
// Định nghĩa chồng toán tử >
ostream& operator >> (istream& is, Point& p)
{
    return is >> p.x>>p.y>> p.z;
```

```
}
void main()
{
    Point p(2, 3, 5);
    cout << "\n Tọa độ của điểm p là "<< p;
    // Tọa độ mới của điểm
    cout << "\n Tọa độ mới ";
    cin >> p;
    // Hiển thị tọa độ mới
    cout << "\n Tọa độ của p là "<< p;
}
```

Khi định nghĩa chồng các toán tử << và toán tử >>, cần chú ý rằng các hàm toán tử tương ứng phải trả lại tham chiếu kiểu *ostream* và *istream*. Kiểu của giá trị trả về này sẽ cho phép ta sử dụng các toán tử này liên tiếp nhau trên một dòng lệnh như trong trường hợp chưa được định nghĩa chồng. Tuy vậy chương trình trên còn có một nhược điểm là không kiểm tra được kiểu dữ liệu khi nhập, do đó khi nhập dữ liệu, tọa độ của điểm có thể là một kiểu dữ liệu bất kỳ, kể cả số thập phân và ký tự. Chương trình dưới đây sẽ loại bỏ nhược điểm đã được nêu ở trên.

Ví dụ 9.13

```
#include <ctype.h>
#include <iostream.h>
class Point
{
    float x, y, z;
public:
    Point(float i, float j, float k)
```

```
    {
        x =i;
        y = j;
        z = k;
    }
    friend ostream& operator << (ostrea& os, Point& p);
    friend ostream& operator >> (ostrea& os, Point& p);
};
// Định nghĩa chồng toán tử <<
ostream& operator << (ostream& os, Point& p)
{
    return os << "("
              << p.x << " "
              << p.y << " "
              << p.z << ")"
}
void SkipNonDigits(istream& is)
{
    char c;
    while(1)
{
    is >> c;        // Nhập ký tự từ bàn phím
    if (isdigit (c) ) // Kiểm tra xem có phải là số không
    {
        is.putback(c);
        return;
    }
}
}
// Định nghĩa chồng toán tử >>
```

```
istream& operator >> (istream& is, Point& p)
{
    SkipNonDigits(is);
    is>> p.x;
    SkipNonDigits(is);
    is>> p.y;
    SkipNonDigits(is);
    is >> p.z;
    return is;
}
void main()
{
    Point p(2, 3, 5);
    cout << "\ n  Tọa độ của điểm p là" << p;
    // Tọa độ mới của điểm
    cout << "\n Tọa độ mới ?";
    cin >> p;
    // Hiển thị tọa độ mới
    cout << "\n Tọa độ của p là " << p;
}
```

Để có thể kiểm tra kiểu của dữ liệu cần nhập, chương trình có sử dụng thêm hàm SkipNonDigits(...). Đầu tiên hàm xuất dữ liệu từ dòng nhập vào ký tự c, nếu ký tự này thuộc kiểu số, hàm putback sẽ nhập ký tự này trở lại dòng nhập để phục vụ cho việc xuất dữ liệu cho tọa độ của các điểm. Nếu ký tự không thuộc kiểu số, hàm sẽ đòi hỏi phải nhập lại dữ liệu.

Bạn đọc có thể tự định nghĩa chồng các toán tử nhập, xuất đối với Complex và String.

9.2.8. Các toán tử không được định nghĩa chồng

Các toán tử sau đây không được phép định nghĩa chồng trong C++:

- Toán tử . - toán tử chọn thành phần của cấu trúc.
- Toán tử . và * - các toán tử tham chiếu đến thành phần của con trỏ.
- Toán tử phạm vi ::.
- Toán tử điều kiện ?:.

Để kết thúc cho phần này, xin giới thiệu với các bạn một số ví dụ. Các ví dụ này sẽ minh họa một cách rõ ràng hơn việc sử dụng các toán tử chồng trong các chương trình của C++.

Ví dụ 9.14

Xây dựng lớp Complex để làm việc với số phức

```
#include <stdio.h>
class Complex
{
        double real;
        double imag;
public:
    Complex(void); // Constructor ngầm định
    Complex(cost Complex&); // Constructor sao chép.
    Complex(const double& r, const double& i);
// Constructor khởi tạo
// Phương thức đặt lại hàm xử lý ngoại lệ
void Error_Complex();
// Các phương thức lấy giá trị thực và ảo.
```

```
    friend double real(const Complex& c);
    friend double imag(const Complex& c);
    // Phương thức gán
    Complex& operator = (const Complex& c);
    // Các phương thức tiện ích
    friend double abs(const Complex& c); // Lấy modul của số phức
    // Phương thức của toán tử với một toán hạng
    Complex operator !();
    // Các phương thức tính toán.
friend Complex operator+(const Complex& c1, const Comlex& c2);
friend Complex operator-(const Complex& c1, const Comlex& c2);
friend Complex operator*(const Complex& c1, const Comlex& c2);
friend Complex operator /(const Complex& c1, const Comlex& c2);
    // Toán tử hai toán hạng, sử dụng hàm thành phần
    Complex operator + = (const Complex& c);
    Complex operator - = (const Complex& c);
    Complex operator * = (const Complex& c);
    Complex operator / = (const Complex& c);
    // Phương thức hiển thị
    Print ( ) const;
    } ;
```

Khi xây dựng lớp *Complex*, trong ví dụ chỉ đưa ra một số phương thức nhằm thực hiện một số phép toán cơ bản trên số phức. Để xây dựng một lớp có đầy đủ các thao tác trên số phức, bạn có thể tự xây dựng thêm một số hàm bạn hoặc các phương thức, chẳng hạn các hàm bạn với mục đích so sánh hai số phức và thực hiện các phép toán lượng giác trên số phức v.v..

Dưới đây là định nghĩa các *constructor* và các hàm của lớp:

```
// Constructor
Complex::Complex(void)
{
    Real =0;
    Imag =0;
}
//Constructor sao chép
Complex::Complex(const Complex& c)
{
    Real = c.Real;
    Imag = c.Imag;
}
// Constructor khởi tạo
Complex::Complex(const double & r, const double& i)
{
    Real =r;
    Imag = i;
}
// Toán tử gán
Complex& Complex::operator = (const Complex& c)
{
    Real = c.Real;
    Imag = c.Imag;
    return *this;
}
// Hàm xử lý ngoại lệ
extern void set_new_handler(void(*)());
void Error_Complex()
{
```

```
        cout << endl;
        cout<< " Lỗi trong đối tượng Complex: Chia cho Zero"<< endl;
        exit(1);
    }
    // Phương thức lấy giá trị thực
    double real(const Complex &c)
    {
        return c.Real;
    }
    // Phương thức lấy giá trị ảo
    double real(cons Complex &c)
    {
        return c.Imag;
    }
    // Phương thức lấy modul của số phức.
    double abs(cons Complex &c)
    {
        double result = sqrt(c.Real*c.Real +c.Imag*c.Imag);
    }
    // Toán tử một toán hạng !
    Complex Complex::operator !()
    {
        Real = -Real
        Image = -Imag;
        return *this;
    }
    // Toán tử cộng hai số phức
    Complex Complex::operator + (const Complex & c1. const Complex & c2)
    {
```

```
        Complex result;
        result.Real = c1.Real+c2.Real;
        result.Imag = c1.Imag+ c2.Imag;
        return result;
        }
    // Toán tử trừ hai số phức
Complex Complex::operator-(const Complex&c1 const Complex&c2)
{
    Complex result;
    result.Real = c1.Real-c2.Real;
    result.Image = c1.Imag - c2.Imag;
    return result;
}
    // Toán tử nhân hai số phức
    Complex Complex::operator * (const Complex & c1, const
                        Complex & c2)
    {
        Complex result;
        result.Real = (c1.Real*c2.Real)-(c1.Imag*c2.Imag);
        result.Image = (c1.Real*c2.Imag) +(c1.Imag*c2.Real);
        return result;
    }
    // Toán tử chia hai số phức
Complex Complex::operator/(const Complex&c1,constComplex&c2)
    {
        Complex result;
        double d = (c2.Real*c2.Real) + (c2.Imag*c2.Imag);
        if(d !=0.0)
        {
```

```
        result.Real = (c1.Real*c2.Real+c1.Imag*c2.Imag)/d;
        result.Imag = (c1.Imag*c2.Real-c1.Real*c2.Imag)/d;
    }
    else
        ::set_new_handler(Error_Complex);
    return result;
    }
    // Toán tử +=
    Complex Complex::operator += (const Complex& c)
    {
        Real +=c.Real;
        Imag +=c.Imag;
        return *this;
    }
    // Toán tử -=
    Complex Complex::operator -= (const Complex& c)
    {
        Real -=c.Imag;
        return *this;
    }
    // Toán tử *=
    Complex Complex::operator +=(const Complex& c)
    {
        double OldReal = Real;
        Real = (Real.c.Real)-(Imag*c.Imag);
        Imag = (OldReal*c.Imag) + (Imag*c.Real);
        return *this;
        }
    // Toán tử /=
```

```
Complex Complex::operator /= (const Complex& c)
    {
    double d = (c.Real*c.Real) + (c.Imag*c.Imag);
    if(d !=0.0)
        {
        double OldReal = Real;
     Real = (Real*c.Real+Imag*c.Imag)/d;
     Imag = (Imag*c2Real-OldReal*c.Imag)/d;
     }
 else
     ::set_new_handler(Error_Complex);
 return *this;
     }
 // Phương thức hiển thị
 void Complex..Print() const
     {
        cout<< "So phuc la:<< endl
        cout << Real<< "+" << Imag;
     }
```

Chương X

TÍNH TƯƠNG ỨNG BỘI

10.1. SỰ RÀNG BUỘC SỚM VÀ MUỘN

Trong khi C++, khái niệm về tính tương ứng bội (*Polymorphic Function*) có một ý nghĩa hết sức quan trọng bởi nó làm tăng hiệu quả của chương trình được viết ra. Để có thể tìm hiểu chi tiết về vấn đề này trước tiên ta hãy đi sâu về hai khái niệm thường được dùng trong kỹ thuật lập trình - sự ràng buộc sớm (*Early Binding*) và sự ràng buộc muộn (*Late Binding*).

Khi làm việc với các hàm, hầu hết các ngôn ngữ (như *C* hoặc *Pascal* v.v.) đều sử dụng một quá trình làm việc quen biết gọi là *Early Binding* (sự ràng buộc sớm). Theo quá trình này, tên của một hàm cụ thể nào đó đều gắn liền với một địa chỉ tương ứng trong bộ nhớ (địa chỉ này được xác định trong quá trình biên dịch). Trong quá trình liên kết, các từ này (tên hàm) khi gặp sẽ được thay thế bởi các địa chỉ tương ứng và khi một hàm nào đó được gọi, chương trình sẽ chuyển quyền điều khiển đến địa chỉ tương ứng đã được xác định trước đó và điều cần chú ý là chỉ có duy nhất một hàm sẽ được gọi.

Việc sử dụng *Early Binding* đòi hỏi người lập trình cần phải dự đoán trước hàm nào của đối tượng nào sẽ được gọi đến trong

những trường hợp cụ thể và như vậy mặc dù có một số ưu điểm như tốc độ thực hiện nhanh do chỉ tiêu phí thời gian khi gọi hàm, khi truyền tham số, xóa Stack, trong khi các công việc như liên kết, tính toán địa chỉ của hàm đã được thực hiện trước đó trong quá trình biên dịch, phương pháp này vẫn có nhiều hạn chế và tỏ ra thiếu hiệu quả trong C++.

Late Binding sẽ cho phép các hàm được che khuất trong quá trình biên dịch và việc một hàm nào đó được gọi sẽ được quyết định trong khi thực hiện. Sử dụng phương pháp này, một hàm nào đó sẽ được gọi luôn gắn liền với đối tượng chứa nó và điều này sẽ do chương trình quyết định khi thực hiện và do vậy khác với Early Binding: không chỉ có duy nhất một hàm sẽ được thực hiện thông qua một phép gọi hàm mà có thể có nhiều hàm khác nhau sẽ được thực hiện tùy theo từng điều kiện cụ thể. Đây là một cách làm việc mới với hàm trong một số ngôn ngữ lập trình hướng đối tượng như SmallTalk, C++ v.v với độ mềm dẻo cao khi lập trình.

10.2. HÀM ẢO (VIRTUAL FUNCTION)

Trong C++, tính ràng buộc muộn cho một hàm sẽ được thực hiện nếu hàm đó được khai báo bằng từ khóa *virtual* và điều này chỉ có ý nghĩa đối với những đối tượng nằm trong cấu trúc cây. Trong trường hợp ngược lại, việc khai báo virtual cho hàm mặc dù không phát sinh lỗi nhưng sẽ không có tác dụng và chỉ làm phí tổn thời gian khi hàm được gọi và thực hiện. Để tìm hiểu về điều này, trước tiên hãy xét một số ví dụ đơn giản.

Ví dụ 10.1

```
#include <stdio.h>
class A
{
    public:
        virtual void Display()
        {
            puts("\n Class A");}
};
class B :public A
{
    public:
    virtual void Display(){ puts("\n Class B");}
};
void Show(A* a)
{
    a->Display();
}
void main()
{
    A* a = new A;
    B* b = new B;
    a->Display();
    b->Display();
    Show(a); // Gọi hàm A::Display()
    Show(b); // Gọi hàm B::Display()
}
```

Đặc tính *Polymorphic* của hàm thành phần *Display()* sẽ

không thể hiện rõ nếu chỉ xem xét hàm chính *main()*. Đặc tính này được thể hiện trong hàm *Show()* và cho phép chương trình có thể gọi một hàm *Display()* tương ứng của *lớp A* hoặc *B* để thực hiện tùy theo từng trường hợp. Trong ví dụ trên, nếu hàm *Display()* không được khai báo virtual thì *Show(a)* sẽ gọi *A::Display()* và *Show(b)* cũng sẽ gọi *A::Display()* vì tham số truyền *B** sẽ được chuyển kiểu thành *A** trước khi gọi hàm *Display()*.

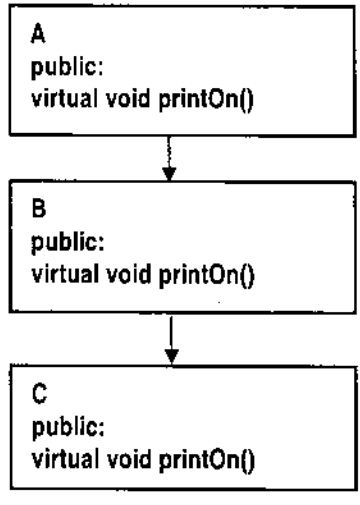
Cần lưu ý rằng chỉ có các hàm thành phần mới được khai báo theo kiểu *virtual* và một khi một hàm nào đó được khai báo theo kiểu *virtual* thì hàm đó không được khai báo lại trong các lớp dẫn xuất với cùng các đặc tính (số lượng đối số và kiểu các đối số) nhưng khác kiểu. Khi điều này được đảm bảo thì hàm sẽ tự động trở thành *virtual*, không phụ thuộc là hàm có được khai báo bằng từ khóa *virtual* trong lớp dẫn xuất hay không.

10.2.1. Các hàm được định nghĩa lại (Overriding Function)

Trong ví dụ trên, hàm ảo *B: :Display()* được chọn theo kiểu động trong thời điểm thực hiện hàm *::Show(...)*. Hàm *B::Display()* bởi vậy được gọi là *virtual override* của hàm *A::Display()*. Một hàm được khai báo trong một lớp dẫn xuất sẽ là *virtual override* của một hàm trong lớp cơ sở chỉ khi chúng có cùng số lượng và kiểu các đối số. Trong trường hợp ngược lại, các hàm này được xem như là những hàm hoàn toàn khác nhau và việc khai báo *virtual* cho chúng hoàn toàn không mang lại bất cứ một hiệu quả nào.

10.2.2. Hàm rỗng (Null Function)

Các lớp dẫn xuất (ví dụ như lớp B) không nhất thiết phải sử dụng một hàm được khai báo theo kiểu *virtual* trong lớp cơ sở (ví dụ lớp A). Tuy vậy, nếu một lớp khác lại được dẫn xuất từ các lớp dẫn xuất trên (chẳng hạn C lại dẫn xuất từ B) thì để lớp C có thể sử dụng hàm *virtual* được khai báo trong A, các hàm đó cũng cần phải được khai báo trong các lớp trung gian (lớp B). Để thực hiện được điều này, trong các lớp đó cần phải định nghĩa *Null virtual function*. Hình 10.1 mô tả một cấu trúc trong đó cần phải sử dụng *Null Function*.



Hình 10.1

Ví dụ 10.2

```
#include<stdio.h>
class A{
    public:
    virtual void prinOn(){ puts("\n Class A");}
```

```
    };
class B:public A
{
    public: virtual void printOn(){ }
};
class C:public B
{
    public:
    virtual void printOn(){ puts("\n Class C");}
};
void Show(A* a)
{
    a->printOn;
}
void main()
{
    a* a = new A;
    B* b = new B;
    C* c = new C;
    Show(a);
    Show(b);
    Show(c);
}
```

Hàm ảo *B::printOn()* được khai báo theo kiểu *Null* vì lớp B không đòi hỏi phải sử dụng hàm này. Việc khai báo theo kiểu *Null* bắt buộc phải có trong lớp B để cho phép các lớp dẫn xuất từ B có thể sử dụng các hàm *Virtual* trong A qua cách gọi *:Show(c).*

10.2.3. Các lớp trừu tượng (Abstract Classes)

Các lớp xuất hiện gần hoặc ngay tại đỉnh của một cấu trúc cây của quá trình thừa kế thường có một hoặc nhiều *Null Virtual Functions*. Các hàm này làm cho giao diện của người sử dụng khi làm việc với cấu trúc trở thành phù hợp và mềm dẻo hơn. Các giao diện này cho phép quá trình *Late Binding* được sử dụng một cách rộng rãi và làm giảm bớt gánh nặng của người lập trình trong quá trình xây dựng chương trình. Các lớp ở đỉnh của cấu trúc cây thường có tất cả các hàm được dùng chung của tất cả các lớp, mặc dù vậy tính tương ứng bội của C++ sẽ cho phép phân biệt các hàm cụ thể được gọi thông qua cơ cấu hàm ảo.

Tuy vậy, nếu một lớp chỉ bao gồm các *Null Virtual Function* thì việc thể hiện các đối tượng của lớp này sẽ không có ý nghĩa khi lập trình bởi vì ngoài việc phân bố bộ nhớ một cách không cần thiết, các đối tượng của lớp đó sẽ không thực hiện bất cứ điều gì, mặc dù trình biên dịch có thể cho phép thực hiện điều đó mà không thông báo lỗi.

Để tránh việc khởi tạo các đối tượng của các lớp có dạng trên, các lớp đó cần được khai báo theo kiểu trừu tượng (*abstract*). Khi một lớp được khai báo theo kiểu *abstract* thì mọi cố gắng thể hiện đối tượng của lớp đó đều bị trình biên dịch bắt lỗi. Một lớp được trở thành *abstract* cần phải có ít nhất một *Null Virtual Function* và trong số các *Null Virtual Function* đó phải có ít nhất một hàm khai báo theo kiểu *Pure virtual function* (hàm thuần ảo) theo cách sau:

```
virtual void printOn() = 0;
```

(Cần phân biệt sự khác nhau giữa việc khai báo trên với việc khai báo Null Function : virtual void printOn() { }).

Ví dụ 10.3

```
class A
{
    public;
        virtual void printOn() =0; // Pure Function
};
void main()
{
    A *a;
    a->printOn(); // Chương trình sẽ báo lỗi
}
```

Khi thực hiện chương trình này, việc gọi hàm *a->printOn()* trong hàm chính *main* sẽ bị trình biên dịch bắt lỗi và có thể phát sinh thông báo sau: *Abord program termination.*

Trên thực tế, việc gọi một hàm được khai báo theo kiểu *pure virtual* là tương đương với việc sử dụng một con trỏ chưa được khởi tạo để gọi gián tiếp hàm và do đó sẽ không được phép. Trong trường hợp này. Borland C^{++} sẽ tự tham chiếu đến một con trỏ chỉ đến hàm xử lý lỗi để thông báo lỗi xẩy ra.

Các lớp trừu tượng được xây dựng để phục vụ cho các lớp có tính kế thừa và các lớp này thường xuất hiện ở gần hoặc ngay tại đỉnh của cấu trúc cây. Cần chú ý rằng có thể tồn tại nhiều *Abstract classes* trong cùng một cấu trúc cây. Ví dụ dưới đây sẽ minh họa cho việc sử dụng các lớp trừu tượng.

Ví dụ 10.4

```
#include<stdio.h>
class A
{
```

```
public:
        virtual void printOn() =0; // Pure virtual function
};
class B:public A{
{
public:
    virtual void printOn() { puts("\n Class B");
};
class C:public B{
{
public:
    virtual void printC :() { puts("\n Class C");
};
void Show(A* a)
{
    a->printOn();
}
void main()
{
    B* b = new B;
    C* c = new C;
    Show(b); // sử dụng B::printOn()
    Show(c); // sử dụng C::printOn()
}
```

Lớp A được sử dụng như một lớp cơ sở cho tất cả các lớp khác. Hàm *::Show(A*)* nào được gọi sẽ được xác định cụ thể trong khi thực hiện chương trình dựa trên kiểu của đối tượng được truyền cho hàm.

Việc dẫn xuất một lớp từ một lớp trừu tượng không có nghĩa là lớp đó có thể được thể hiện. Nếu một lớp được dẫn xuất từ một lớp khác mà bản thân lớp cơ sở lại có chứa các *pure virtual function* thì lớp dẫn xuất đó cũng được xem là trừu tượng. Bằng cách này C++ đòi hỏi phải định nghĩa chồng tất cả các *pure virtual function* trong lớp dẫn xuất. Nếu điều này không đảm bảo thì có thể phát sinh ra lỗi khi biên dịch bởi vì một khi ta không có khả năng định nghĩa chồng các *pure virtual function* thì cơ cấu sử dụng của các hàm *virtual* cũng không thể truyền cho các lớp dẫn xuất. Nếu một lớp dẫn xuất từ một lớp trừu tượng mà lại là *Abstract* thì một trong các hàm *virtual* của lớp đó phải là *pure virtual.*

Như trên đã trình bày, nếu trong lớp dẫn xuất không có định nghĩa chồng cho các hàm *pure virtual* của lớp cơ sở thì điều đó có thể phát sinh ra lỗi. Ví dụ sau sẽ minh họa cho kết luận trên:

Ví dụ 10.5

```
class A{
public:
    virtual void f() =0;
    };
class B:public A
{
    int value;
    public:
    void g(int i) { value=i;}
};
```

Nếu *pure virtual funtion* được sử dụng như một hàm *virtual* bình thường thì lớp *B* có thể thừa kế hàm *A::f()* vì trong B không có định nghĩa chồng của hàm đó. Giả sử rằng C là một lớp mới được dẫn xuất từ *B*. Nếu trong C có khai báo *virtual function f()* thì hàm này sẽ không được phép truy nhập qua con trỏ *A** hoặc *B** vì trong dây chuyền của *virtual function* đã có một mắt xích bị cắt đứt tại lớp B. Việc gọi *C::f()* trong trường hợp này sẽ dẫn đến việc thực hiện hàm *pure virtual function* A::f() mà kết quả sẽ phát sinh ra lỗi. Chính vì nguyên nhân này, *pure virtual function* cần được khai báo chồng trong các lớp dẫn xuất và thậm chí phải khai báo *pure* một lần nữa nếu cần.

Có thể nói đặc tính tương ứng bội của C++ là rất quan trọng nhưng cũng tương đối khó hiểu đối với những người bắt đầu với C++. Để có thể nắm chắc hơn khái niệm này, ta hãy quay lại với chương trình xây dựng các đối tượng đồ họa đã được thiết lập trong các chương trước

Trong chương trình này, mỗi lớp trong cấu trúc cây của sự thừa kế nhằm mục đích xây dựng một đối tượng đồ họa khác nhau trên màn hình: điểm hoặc đường tròn. Với các lớp này, có thể vẽ các đường tròn hoặc các điểm trên màn hình đồ họa tại các vị trí khác nhau. Để có thể vẽ các đối tượng khác, chẳng hạn như hình vuông, cung tròn v.v ta có thể xây dựng thêm các lớp với các hàm thành phần riêng biệt trong các lớp đó để chúng có thể tự hiển thị lên màn hình.

Khi xây dựng lên các hàm thành phần (*Show*) nhằm mục đích hiển thị các đối tượng của từng lớp, thuật toán để xây dựng lên hàm thành phần này là hoàn toàn khác nhau. Chẳng hạn để hiển thị một điểm thì cần xác định vị trí (tọa độ) cũng như màu

của điểm đó, trong khi để vẽ lên một đường tròn thì ngoài tọa độ của tâm đường tròn thì còn cần thông tin về bán kính của đường tròn v.v.

Khi xây dựng các hàm thành phần *Show* của từng lớp, trình biên dịch đã dùng phương pháp ràng buộc sớm để phân biệt các hàm thành phần này của các đối tượng khác nhau. Sử dụng sự ràng buộc sớm, hàm thành phần *Show* của đối tượng nào được tham chiếu đến sẽ phụ thuộc vào các yếu tố sau:

- Sự khác biệt về các đặc tính của các tham số truyền: số lượng hoặc kiểu của các đối số - chẳng hạn *Show(int, char)* và *Show(char*, float)* là hoàn toàn khác nhau.
- Sử dụng toán tử phạm vi - chẳng hạn *Circle::Show()* và *Apoint::Show(),* :: *Show()* là hoàn toàn khác nhau.
- Sử dụng phép gọi hàm từ một đối tượng của lớp - chẳng hạn *ACircle. Show* gọi *Circle::Show* trong khi *APoint.Show* sẽ gọi *Point::Show*. Cũng vậy nếu ta gọi hàm thành phần từ con trỏ chỉ đến đối tượng của lớp - chẳng hạn: *APoint_pointer->Show* sẽ gọi *Point::Show*.

Một đặc điểm của hầu hết các thư viện đã được xây dựng sẵn (nhất là các công cụ đồ họa) là các hàm thành phần của các lớp đã được biên dịch dưới dạng file OBJ hoặc file thư viện LIB. Sử dụng sự ràng buộc sớm trong tất cả các trường hợp này sẽ gây không ít khó khăn khi cần xây dựng thêm một lớp mới hoặc thậm chí khi cần phát triển một lớp sẵn có. Chẳng hạn, để di chuyển một đối tượng đồ họa - điểm hoặc đường tròn, trong chương trình trên có thể sử dụng hai hàm thành phần riêng biệt cho từng lớp.

```
void Point::MoveTo(int NewX, int NewY)
{
    Boolean vis = Visible;
    if(vis) Hide();
    X = NewX; Y = NewY;
    if(vis) Show();
}
void Circle::MoveTo(int NewX, int NewY)
{
    Boolean vis = Visible;
    if(vis) Hide();
    X = NewX; Y = NewY;
    if(vis) Show();
}
```

Mặc dù hai hàm thành phần *MoveTo* trong hai lớp *Point* và *Circle* là hoàn toàn giống nhau (số lượng và kiểu của các đối số, thậm chí các câu lệnh trong chúng cũng giống nhau) nhưng trong chương trình vẫn phải xây dựng hai hàm khác nhau cho từng lớp. Việc viết lại đoạn mã của *MoveTo* của lớp *Point* trong lớp *Circle* tuy có vẻ đơn giản nhưng trong thực tế lại có một số nhược điểm:

- Không sử dụng được tính thừa kế của lập trình hướng đối tượng và do đó làm cho chương trình trở nên có kích thước lớn không cần thiết.
- Nếu *Point* được cung cấp dưới dạng file OBJ hoặc dưới dạng File LIB thì việc viết lại *MoveTo* trong *Circle* sẽ trở thành khó khăn nếu *MoveTo* được xây dựng từ một thuật toán phức tạp.

Vấn đề nảy sinh ở đây là tại sao không thừa kế hàm *MoveTo* của *Point* trong lớp *Circle* như đã thừa kế hai hàm thành phần khác của *Point* như *GetX* và *GetY*? Nguyên nhân là mặc dù hàm *MoveTo* trong hai lớp này là hoàn toàn giống nhau nhưng các hàm *Hide()* và *Show()* được gọi trong *MoveTo* của *Circle::MoveTo* lại khác với *Hide()* và *Show()* được gọi trong *MoveTo* của *Point::MoveTo*. Các hàm này chỉ có tên là giống nhau. Việc thừa kế *MoveTo* của lớp *Point* trong *Circle* sẽ dẫn đến việc gọi các hàm *Show()* và *Hide()* của lớp *Point* khi hàm thành phần *MoveTo* của lớp *Circle* được gọi. Điều này xảy ra do trình biên dịch đã sử dụng sự ràng buộc sớm khi biên dịch, do hai hàm *Show()* và *Hide()* của *Point* luôn được gắn liền với các đối tượng gọi các hàm này. Sự ràng buộc sớm sẽ đưa đến kết quả sai của chương trình vì như trên đã đề cập, các hàm *Hide()* và *Show()* trong hai lớp được thiết kế hoàn toàn khác nhau.

Để giải quyết có hiệu quả vấn đề phức tạp trên, hai hàm *Show()* và *Hide()* trong hai lớp *Point* và *Circle* được khai báo theo kiểu *Virtual*. Khi đó trong quá trình biên dịch *Point*, hai hàm này sẽ không gắn với đối tượng của lớp *Point* (hay nói cách khác là hai hàm này bị che khuất trong quá trình biên dịch). Lớp *Point* khi đó có thể được biên dịch và liên kết (*Link*) dưới dạng OBJ hoặc LIB để cung cấp cho người sử dụng. Khi xây dựng lớp *Circle* dẫn xuất từ *Point* và thừa kế hàm *MoveTo* của lớp này thì khi một đối tượng của lớp *Point* hoặc *Circle* gọi hàm *MoveTo*, hàm *MoveTo* sẽ tự động tham chiếu đến các hàm *Show()* và *Hide()* của chính các lớp đó. Cần lưu ý là trong trường hợp này trình biên dịch đã sử dụng sự ràng buộc muộn khi gọi hàm *Show()* và *Hide()*. Khi đó việc quyết định một hàm nào đó được gọi sẽ được xác định trong quá trình thực hiện chương trình chứ không phải trong quá trình biên dịch.

Hai hàm *Show* và *Hide* trong hai lớp *Point* và *Circle* được khai báo lại như sau:

virtual void Show(void);

virtual void Hide(void);

Để nghiên cứu một cách chi tiết hơn về cách sử dụng hàm ảo, ta sẽ sử dụng việc khai báo các hàm *Show* và *Hide* như các hàm ảo để xây dựng một hàm chung - hàm *Drag* dùng để kéo một đối tượng thuộc lớp *Point, Circle* hoặc một đối tượng đồ họa bất kỳ được dẫn xuất từ lớp *Point* đi một khoảng cách nào đó khi gõ các phím →, ←, ↑, ↓. Xem lại ví dụ trước, có thể thấy để một đối tượng bất kỳ được dẫn xuất từ lớp *Point* có thể sử dụng hàm *Drag*, hàm này cần được truyền đối số là một tham chiếu đến lớp *Point*. Khi được gọi, và được truyền đối số là một đối tượng *APoint* từ lớp *Point* hoặc *Acircle* trình biên dịch sẽ tự động thực hiện việc chuyển kiểu các đối tượng.

Ngoài đối số là đối tượng của một lớp, hàm *Drag* còn được truyền một đối số *DragBy* - khoảng cách sẽ di chuyển đối tượng. Như vậy hàm *Drag* có thể được định nghĩa như sau:

```
void Drag(Point& AnyFigure, int DragBy)
{
  int DeltaX, Delta Y;
  // Xác định hướng di chuyển của đối tượng
  int FigureX, FigureY;
  AnyFigure.Show(); // Hiển thị đối tượng cần di chuyển
  FigureX = AnyFigure.GetX();
  FigureY = AnyFigure.GetY();
  // Vòng lặp để kéo đối tượng
```

```
while(GetDelta(DeltaX,DeltaY)
{
// Tọa độ mới theo trục X
FigureX +=DeltaX*DragBy;
// Tọa độ mới theo trục y
FigureY +=DeltaY*DragBy;
// Dịch chuyển đối tượng
AnyFigure.MoveTo(FigureX,FigureY);
 }
}
```

Hàm *Drag* có sử dụng một hàm khác chưa được nhắc đến - hàm *GetDelta*. Hàm *GetDelta* làm một nhiệm vụ duy nhất là xác định hướng dịch chuyển của đối tượng khi người sử dụng gõ các phím →, ←, ↑, ↓. Các giá trị của *DeltaX* và *DeltaY* sẽ xác định hướng dịch chuyển và bao gồm:

DeltaX = 1 nếu phím → được gõ;

DeltaX = -1 nếu phím ← được gõ;

DeltaY = 1 nếu phím ↑ được gõ;

DeltaY = -1 nếu phím ↓ được gõ.

Phân tích hàm *Drag* ta nhận thấy hàm được truyền đối số là đối tượng của một lớp dẫn xuất từ lớp *Point*. Trình biên dịch sẽ dùng đối số này để xác định đúng đối tượng tác động lên các hàm thành phần như *GetX()*, *Get()*, *MoveTo(...)* và dịch chuyển đối tượng đối (chú ý cách gọi các hàm thành phần trên trong hàm *Drag*). Tuy vậy, cũng cần lưu ý các hàm thành phần *GetX()*, *GetY()* và *MoveTo* là các hàm được thừa kế trong các tất cả các lớp được dẫn xuất từ lớp Point do đó chúng luôn được gắn liền

với đối tượng đang gọi chúng. Điều cần lưu ý ở đây là phải xác định đúng các hàm *Show(), Hide()* của đối tượng đang được dịch chuyển trong hàm *MoveTo* bởi vì các hàm này có nội dung hoàn toàn khác nhau trong các lớp dẫn xuất từ lớp *Point*. Vấn đề này tuy vậy lại được giải quyết khá dễ dàng nếu các hàm *Show()* và *Hide()* được khai báo theo kiểu *virtual*. Khi đó việc xác định hàm *Show* và *Hide* của đối tượng nào được gọi sẽ được thực hiện thông qua phép ràng buộc muộn và từ đối tượng gọi các hàm này. Như vậy có thể nhận thấy nếu hàm *Drag* được khai báo trong lớp *Point* và các lớp dẫn xuất thừa kế hàm này thì tất cả các vấn đề trên sẽ được giải quyết một cách trọn vẹn. Khi một đối tượng của các lớp dẫn xuất từ lớp *Point* gọi hàm thành phần này, thì do khả năng ràng buộc muộn, chương trình sẽ gọi đúng các hàm thành phần của các đối tượng đó.

Một vấn đề nữa cần đặt ra ở đây là có nên khai báo hàm *Drag* trong lớp *Point* và *virtual* không? Khi khai báo một hàm là *virtual*, thông thường người thiết kế chương trình đưa ra một khả năng nhằm mở rộng và phát triển các hàm đó trong các lớp dẫn xuất. Nếu khả năng này không đòi hỏi thì việc khai báo các hàm này theo kiểu *virtual* là không cần thiết vì các hàm *virtual* thường cần bộ nhớ lớn hơn cũng như số lần truy nhập đến bộ nhớ cũng nhiều hơn so với các hàm tương đương không được khai báo bằng từ khóa *virtual*. Việc khai báo hàm này là *virtual* hay không hoàn toàn phụ thuộc vào mục đích và khả năng phát triển của hàm đó trong các lớp dẫn xuất. Trong chương trình dưới đây, hàm *Drag* sẽ được khai báo theo kiểu *virtual*.

```
// File figuresh
enum Boolean {false,true};
```

```
    class Location
        {
        protected:
            int X;
            int Y;
    public:
        Location(int InitX, int InitY){ X = InitX; Y = InitY);}
        int GetX(return X;}
        int GetY(return Y;}
    };
class Point: public Location
    {
    protected:
        Boolean Visible;
        public:
        Point(int InitX, int InitY);
        virtual void Show();
        virtual void Hide();
        Boolean IsVisible(return Visible;}
        void MoveTo(int NewX, int NewY);
        virtual void Drag(int DragBy);
    };
class Cirle:public Point
    {
    protected:
        int Radius;
    public:
        Circle(int InitX, int InitY, int InitRadius);
        void Show();
```

```
        void Hide();
        void Expand(int ExpandBy);
        void Contract(int ContractBy);
    };
    // Khai báo prototype của hàm GetDelta - Hàm dùng chung
    Boolean GetDelta(int& DeltaX, int& DeltaY);
```

Các hàm thành phần của các lớp được định nghĩa trong File figure.cpp

```
    // File figure.cpp
    #include "figure.h"
    #include <graphics.h>
    #include <conio.h>
    // Các hàm thành phần của lớp Point
    Point::Point(int InitX, int InitY):Location(InitX, InitY)
    {
        Visible = false;
    }
    void Point::Show()
    {
        Visible = true;
        putpixel(X,Y,getcolor());
    }
    void Point::Hide()
    {
        Visible = false;
        putpixe(X,Y,getbkcolor());
    }

    void Point::MoveTo(int NewX, int NewY)
```

```
{
    Hide();
    X = NewX;
    Y = NewY;
    Show();
    }

Boolean GetDelta(int& DeltaX, int& DeltaY)
    {
    char KeyChar;
    Boolean Quit;
    DeltaX =0;
    DeltaY =0;
    do
        {
        KeyChar = getch();
        if(KeyChar = = 13) //Gõ Enter
            return(false);
        if(KeyChar ==0)
            {
            Quit = true;
            KeyChar = getch(); // đọc phím mở rộng
    switch(KeyChar)
        {
        case 72:DeltaY = -1; break;
        case 80:DeltaY = 1; break;
        case 75:DeltaX = -1; break;
        case 77:DeltaX = 1; break;
        default:Quit = false;
```

```
            }
        } while(!Quit);
    }
void Point::Drag(int DragBy)
{
    int DeltaX, int DeltaY;
    int FigureX, FigureY;
    Show();
    FigureX = GetX();
    FigureY = GetY();
    while(GetDelta(DeltaX, DeltaY))
    {
        FigureX += (DeltaX*DragBy);
        FigureY += (DeltaY*DragBy);
        MoveTo(FigureX, FigureY)
    }
}
// Định nghĩa các hàm thành phần của Circle
Circle::Circle(int InitX, int InitY, int InitRadius):
        Point(InitX, InitY)
    {
        Radius = InitRadius;
    }
    void Circle::Show()
    {
        Visible = true;
        circle(X,Y,Radius);
    }
    void Circle::Hide()
```

```
    {
    unsigned int TempColor;
    TempColor = getcolor();
    setcolor(getbkcolor());
    Visible = false;
    circle(X,Y,Radius);
    setcolor(Tempcolor);
}
void Circle::Expand(int ExpandBy)
    {
    Hide();
    Radius +=ExpandBy;
    if(Radius <0)
        Radius =0;
    Show();
    }
void Circle::Contract(int ContractBy)
    {
        Expand(-ContractBy);
}
```

File figure.cpp - khai báo và định nghĩa thêm lớp Arc (vẽ cung tròn) dẫn xuất từ Circle và mô tả sử dụng của các lớp qua hàm main()

// Liên kết với Figures.obj và graphics.lib

Ví dụ 10.6

```
#include "figures.h"
#include<graphics.h>
```

```
#include<conio.h>
class Arc:public Circle
{
        int StartAngle;
        int EndAngle;
    public:
        Arc(int InitX, int InitY, int InitRadius, int InitStartAngle, int InitEndAngle);
        Circle(InitX, InitY,InitRadius)
        {
            StartAngle = InitStartAngle;
            EndAngle = InitEndAngle;
        }
        void Show();
        void Hide();
    };
// Định nghĩa các hàm thành phần của Arc
void Arc::Show()
{
    Visible true;
    arc(X,Y,StartAngle,EndAngle,Radius);
}
void Arc::Hide()
{
    int TempColor;
    TempColor = getcolor();
    setcolor(getbkcolor());
    Visible = false;
    arc(X,Y,StartAngle,EndAngle,Radius);
```

```
    setcolor(Tempcolor);
}
int main()
{
    int graphdriver = DETECT;
    int graphmode;
    initgraph(&graphdriver,&graphmode,"c:\\borland\\bgi")
    Circle ACircle(151,82,50);
    Arc AnArc(151,82,25,0,190);
    // Gõ các phím →,←,↑,↓ để di chuyển các đối tượng
    // Gõ Esc thoát khỏi chương trình
    AnArc.Drag(5);
    AnArc.Hide();
    AnArc.Drag(10);
    closegraph();
    return 0;
}
```

10.2.4. Hàm bạn ảo (Virtual friend function)

Các hàm thành phần của một lớp có thể được khai báo theo kiểu *virtual* để các lớp dẫn xuất từ lớp này có thể sử dụng hàm để phát triển và thực hiện các công việc riêng của chính lớp dẫn xuất đó. Tuy vậy vấn đề đặt ra là có thể khai báo một hàm bạn (friend function) của một lớp theo kiểu *virtual* được không? Câu trả lời là có thể được, nhưng khi đó hàm bạn phải là một hàm thành phần của một lớp khác. Trong tất cả các trường hợp khác, việc khai báo một hàm bạn là *virtual* sẽ phát sinh lỗi trong chương trình. Ví dụ dưới đây sẽ minh họa cho việc sử dụng một hàm *virtual friend function*.

Ví dụ 10.7

```
#include <stdio.h>
class C; // Khai báo prototype của lớp C;
class A
{
      public:
           int a;
           A(int i) { a=i;}
           virtual void printOn(C&);
};
class B:public A
{
public:
      int b;
      B(int i, int j) : A(i) { b=j;}
      virtual void printOn(C&);
};
class C
{
      friend void A::printOn(C&);
      friend void B::printOn(C&);
      int a,b,c;
 public:
      C(int i, int j, int k) { a=i; b=j; c=k;}
};
void A::printOn(C& t)
{
      printf("\n Thành phần a của lớp A là a = %d",a).
      printf("\n Thành phần c của lớp C là c = %d", t.c);
```

```
    }
void B::printOn(C& t)
{
    printf("\n Thành phần b của lớp B là b = %d", b);
    printf("\n Thành phần c của lớp C là c = %d", t.c);
}
void Print(A& m, C& t)
{
    m.printOn(t);
}
// Hàm chính
void main()
{
    A a(10);
    B b(10,20);
    C c(10,20,30);
    Print(a,c);
    Print(b,c);
};
```

Trong ví dụ này có thể nhận thấy hai hàm *printOn(...)* của hai lớp A và B được khai báo như một hàm bạn của lớp C và do đó có thể truy nhập không hạn chế đến tất cả các thành phần của lớp C. Thêm vào đó hai hàm này lại là hai hàm ảo do được khai báo bằng từ khóa *virtual* vì vậy có thể được gọi một cách chính xác tùy theo đối tượng được truyền như tham chiếu cho hàm *Print(..)*.

10.2.5. Toán tử ảo (Virtual operator)

Cũng như các hàm thành phần, các toán tử của lớp cũng có thể được khai báo theo kiểu *virtual*. Ví dụ 10.8 dưới đây sẽ mô tả chương trình minh họa cho việc sử dụng các toán tử của một lớp như các toán tử *virtual*.

Ví dụ 10.8

```
class A
{
public:
      int value_a;
      A(int i) { value_a = i;}
      virtual A& operator + (A&);
      virtual A& operator = (A&);
      virtual int a() { return value_a;}
      virtual int b() { return 0;}
      virtual int c() { return 0;}
};
class B:: public A
{
      public:
      int value_b;
      B(int i, int j ): A(i) { value_b = j;}
      virtual A& operator + (A&);
      virtual A& operator = (A&);
      virtual int b() { return value_b;}
};
class C:public B
{
```

```
        int value_c;
        C(int i, int j, int k) : B(i,j) { value_b = k;}
        virtual A& operator + (A&);
        virtual A& operator = (A&);
        virtual int c() { return value_c;}
};
A& A::operator + (A& t)
{
        value_a += t.a();
        return *this;
}
A& A:: operator = (A& t)
{
        value_a = t.a();
        return *this;
}
A& B::operator + (A& t)
{
        value_a += t.a();
        value_b + t.b();
        return *this;
}
A& B:: operator = (A& t)
{
        value_a = t.a();
        value_b = t.b();
        return *this;
}
A& C::operator + (A& t)
```

```
{
    value_a += t.a();
    value_b += t.b();
    value_c += t.c();
    return *this:
}
A& C::operator = (A& t)
{
    value_a = t.a();
    value_b = t.b();
    value_c = t.c();
    return *this;
}
void AddObjects(A& object1, A& object2)
{
    object1 = object1 + object2;
}
void main()
{
    A a(10);
    b b(10,20);
    C c(10, 20, 30);
    AddObjects(a, b);
    AddObjects(b, c);
    AddObjects(c, b);
    a = b + c;
    c = c + a;
}
```

Trong ví dụ này, tất cả các toán tử thành phần ảo đều được

khai báo theo kiểu tham chiếu đến đối tượng của lớp *A (A&)*. Điều này là bắt buộc vì các hàm được khai báo theo kiểu *virtual* trong các lớp khác nhau đòi hỏi phải có cùng kiểu, cùng số lượng và kiểu của các đối số. Các toán tử ảo được sử dụng trong hàm *main* qua hai cách - truyền đối tượng như đối số của hàm và sau đó sử dụng cơ cấu tương ứng bội bên trong hàm *AddObject(A&, A&)*, hoặc sử dụng một cách rõ ràng các toán tử này như hai dòng cuối của lệnh. Để xem xét cách sử dụng các toán tử *virtual* này một cách cụ thể, hãy xem cách thực hiện và kết quả của câu lệnh cuối cùng *c = c + a*; Khi thực hiện câu lệnh này, đầu tiên trình biên dịch sẽ tính kết quả của biểu thức *c + a*. Khi tính giá trị của biểu thức này, trình biên dịch sẽ sử dụng toán tử *A& C::operator +* do toán tử bên trái của biểu thức là đối tượng của lớp C. Kết quả nhận được sẽ là value_a = 10 + 10 = 20, value_b = 20 + 0 - 20 và value_c = 0 + 30 = 30. Giá trị tính được của biểu thức sẽ được gán cho c qua việc sử dụng toán tử ảo A& *C: :operator =*.

Khi sử dụng từ khóa *virtual* để khai báo cho các hàm bạn, cần chú ý đến một số điểm sau:

- Chỉ có các hàm thành phần mới được khai báo theo kiểu *virtual*, không sử dụng từ khóa này cho các hàm được khai báo theo kiểu toàn cục.
- Các hàm thành phần kiểu *static* không được khai báo theo kiểu virtual.
- Các constructor không được khai báo theo kiểu *virtual*, trong khi có thể sử dụng từ khóa này cho *destructor* và bởi vì *destructor* không có tham số nên chỉ có duy nhất một *destructor* được khai báo theo kiểu *virtual* cho một lớp.

Chương XI

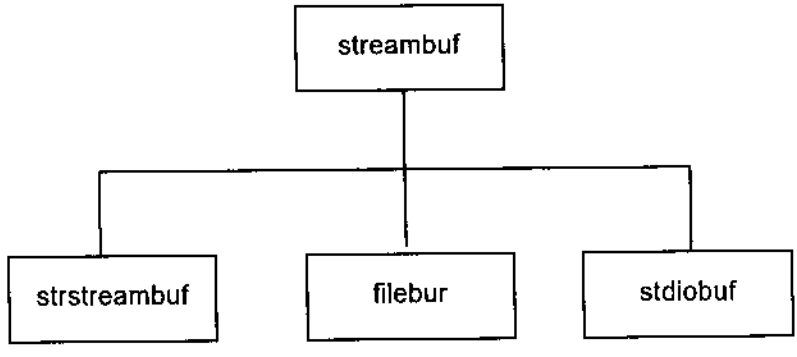
THƯ VIỆN CÁC DÒNG NHẬP XUẤT

Để thuận tiện cho người sử dụng, Borland C++ đã được cung cấp một số lớp cho phép người lập trình có thể sử dụng trực tiếp các hàm thành phần của lớp đó hoặc thừa kế chúng trong các lớp dẫn xuất khi cần thiết. Nhìn chung các lớp này được thiết kế và sử dụng cho trình biên dịch với mục đích làm tăng hiệu quả của việc sử dụng bộ nhớ cũng như tốc độ thực hiện của chương trình. Các dòng nhập xuất đã được đề cập đến như cin, cout, clog và cerr chính là các thể hiện của các lớp đã được xây dựng sẵn trong C++.

Việc nghiên cứu kỹ và sử dụng các lớp đã được thiết kế sẵn trong C++ đòi hỏi người sử dụng phải nắm vững kỹ thuật và phương pháp lập trình định hướng đối tượng mà cụ thể là các khái niệm như tính thừa kế, tính tương ứng bội v.v. Nếu khi nghiên cứu các lớp này bạn vẫn còn cảm thấy khó khăn, đề nghị các bạn hãy tham khảo lại các chương V và VI để thuận tiện cho việc nghiên cứu tiếp theo. Phần này của cuốn sách không có tham vọng giới thiệu tất cả các lớp của C++, mà chỉ đề cập đến một số lớp đại diện trong số các lớp đó. Các lớp khác, bạn có thể tham khảo ngay trong trình biên dịch của C++.

11.1. CẤU TRÚC CÂY TỪ STREAMBUF

Streambuf và các lớp dẫn xuất từ nó là các lớp hỗ trợ cho các thao tác nhập xuất dữ liệu có sử dụng bộ đệm - *buffer* (để tiện cho việc theo dõi, dưới đây sẽ dùng từ buffer thay cho bộ đệm). Tất cả các thao tác nhằm di chuyển dữ liệu từ một vùng này (nguồn) sang một vùng khác (đích) đều được hỗ trợ bởi một cấu trúc cây của các lớp với sự phân chia thứ bậc (*hierachy)* được bắt đầu từ *streambuf*. Có thể nói rằng *streambuf* và các lớp dẫn xuất từ lớp này đã được thiết kế để điều khiển các thao tác nhập xuất ở mức thấp nhất. Tuy vậy cũng cần lưu ý rằng *streambuf* chưa được thiết kế để làm việc với một đối tượng cụ thể, do vậy khi sử dụng trực tiếp *streambuf* ta chỉ có thể làm việc với *buffer* mà không thể thực hiện các thao tác nhập xuất trên một đối tượng cụ thể khác như file, màn hình v.v. Để có thể làm việc với các đối tượng này, cần phải làm việc với các lớp khác được dẫn xuất từ *streambuf*. Hình dưới đây mô tả cấu trúc cây được dẫn xuất từ *streambuf*:



Như vậy mục đích chính của *streambuf* khi được thiết kế để cho các lớp khác có thể được dẫn xuất từ lớp này. Xin nhắc lại là việc thể hiện trực tiếp một đối tượng từ lớp *streambuf* là hầu

như không có ý nghĩa trên thực tế. Dưới đây sẽ nghiên cứu một số lớp được dẫn xuất từ *streambuf* cũng như chính bản thân lớp này.

11.1.1. Lớp streambuf

11.1.1.1. Giới thiệu về streambuf

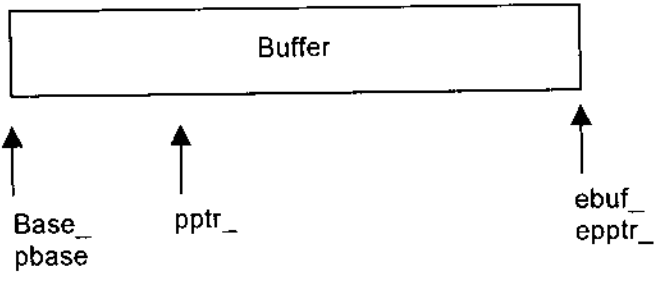
Như đã đề cập đến ở trên, *streambuf* dùng để điều khiển các thao tác nhập xuất ở mức thấp có sử dụng *buffer*. Việc nhập xuất ở đây được tiến hành theo nguyên tắc *First In Last Out* (theo nguyên tắc này, việc xuất các ký tự từ *buffer* sẽ được thực hiện theo thứ tự khi nhập chúng vào *buffer*). Cần lưu ý một số vấn đề sau khi làm việc với các đối tượng thuộc kiểu *streambuf*:

- Đối với các *buffer* xuất (*output buffer*), các ký tự có thể được xuất vào *buffer* cho đến khi lấp đầy vùng xuất. Khi vùng xuất được lấp đầy (trường hợp này được gọi là *overflow*) để có thể xuất các ký tự tiếp theo, vùng xuất cần được làm sạch (*flushing*).
- Đối với *buffer* nhập (*input buffer*), các ký tự có thể được đọc từ buffer cho đến khi *buffer* trở thành rỗng. Trường hợp này được gọi là *underflow*.

Bản thân lớp *streambuf* tuy vậy lại không hỗ trợ cho các thao tác xảy ra khi gặp các trường hợp *buffer* bị đầy (khi xuất) hoặc bị rỗng (khi nhập) ngoài việc tự động gọi hai hàm được định nghĩa *virtual* tương ứng là *oveflow* và *underflow*. Hai hàm này đến lượt mình lại không thực hiện bất cứ một công việc gì khi được thiết kế trong lớp *streambuf*. Nguyên nhân là do hoạt động của hai hàm này hoàn toàn phụ thuộc vào kiểu đối tượng cụ thể

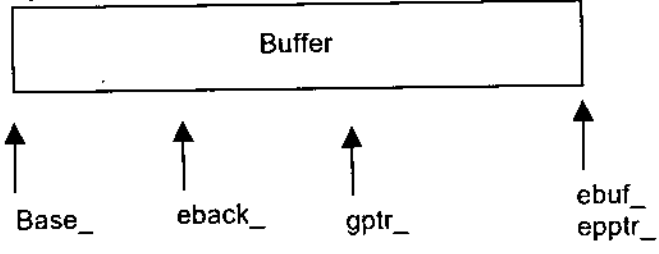
đang làm việc (chẳng hạn file hoặc chuỗi v.v) trong khi đối tượng này lại không thể được xác định trong khi làm việc với *streambuf*. Các hàm này được thiết kế theo kiểu *virtual* để có thể được định nghĩa chồng trong các lớp dẫn xuất (khi các đối tượng cụ thể đã được xác định) và có thể sử dụng tính tương ứng bội khi các trường hợp *overflow* và *underflow* xảy ra.

Lớp *streambuf* có sử dụng một số biến con trỏ để điều khiển hoạt động của *buffer*. Hình 11.1 và 11.2 chỉ ra cách sắp xếp của các con trỏ này trong hai trường hợp sử dụng *output buffer* (vùng xuất) và *input buffer* (vùng nhập).



Hình 11.1. Streambuf cho các thao tác xuất

Trong các hình vẽ này, các con trỏ được bắt đầu bởi ký tự *p* được sử dụng cho các thao tác xuất trong khi các con trỏ *gptr_* và *eback_* được sử dụng để nhập một ký tự vào bộ đệm.



Hình 11.2. Streambuf cho các thao tác nhập

Dưới đây là chức năng của các con trỏ được mô tả trên các hình 11.1 và 11.2:

*base_* : Con trỏ chỉ đến vị trí đầu của *buffer* (địa chỉ byte đầu tiên của *buffer*).

*pbase_* : Con trỏ chỉ đến vị trí đầu tiên của vùng xuất trong *buffer*...

*pptr_* : Con trỏ chỉ đến vị trí tiếp theo sẽ xuất ký tự vào *buffer*.

*ebuf_* : Con trỏ chỉ đến vị trí ngay sau điểm cuối của *buffer*.

*gptr_* : Con trỏ chỉ đến ký tự tiếp theo sẽ được đọc từ *buffer*. Cùng với việc đọc ký tự, ký tự này sẽ bị loại khỏi *buffer*.

*eback_* : Sau khi ký tự được đọc từ *buffer*, ký tự đó có thể được xuất trở lại vào *buffer* bởi các thao tác riêng (*putback operation*). Con trỏ *eback_* trong trường hợp này sẽ chỉ ra khoảng cách để con trỏ *gptr_* có thể được dịch chuyển trở lại.

Sau đây là mô tả chi tiết các loại dữ liệu và hàm thành phần của lớp *streambuf* trong BorlandC^{++}.

Các thành phần thuộc kiểu public:

streambuf(): Constructor ngầm định. Các đối tượng được khởi tạo bằng *constructor* này sẽ không được thiết lập *buffer* để làm việc. Tất cả các con trỏ được thiết lập bởi giá trị 0.

streambuf(char, int)*: Constructor dùng để khởi tạo các đối tượng và thiết lập *buffer* để làm việc với đối tượng đó. Các tham số *char** và *int* là địa chỉ đầu và kích thước của *buffer*.

virtual ~streambuf(): Destructor này được xây dựng với từ khóa *virtual* nhằm tránh việc nhầm lẫn khi loại bỏ một đối tượng của các lớp được thừa kế từ *streambuf*. Trong lớp

streambuf có rất nhiều hàm thành phần sử dụng con trỏ chỉ đến *streambuf*. Khi con trỏ hoặc các tham chiếu này được truyền bởi các đối tượng đã được xác định (chẳng hạn *filebuf* hoặc *streambuf*), *destructor* này đảm bảo rằng đối tượng tương ứng sẽ bị xóa.

streambuf setbuf(unsigned char*, int)*: Hàm này dùng để thiết lập *buffer* liên kết với các dòng nhập xuất và không được phép định nghĩa chồng.

Khi được gọi, đến lượt mình hàm sẽ gọi hàm *virtual streambuf:: setbuf(signed char*, int)*.

virtual streambuf setbuf(signed char*, int)*: Được thiết kế theo kiểu *virtual* với mục đích có thể được định nghĩa lại trong lớp dẫn xuất. Cần chú ý là tham số đầu của hàm được khai báo theo kiểu *signed char** để phân biệt hàm này với hàm *non-virtual setbuf* vừa được trình bày ở trên. Tham số được truyền cho hàm bao gồm địa chỉ và độ dài của *buffer* sẽ được liên kết với đối tượng. Nếu đối tượng đã được thiết lập *buffer*, *buffer* này sẽ bị loại bỏ và được thay thế bởi *buffer* mới.

int sgetc(): Nhận ký tự tiếp theo từ vùng nhập *get(input buffer)* nhưng không làm thay đổi các con trỏ của *buffer* trừ khi *buffer* đã được đọc hết. Trong trường hợp này, hàm *virtual streambuf::underflow()* sẽ tự động được gọi và người sử dụng cần phải thiết lập lại các con trỏ trong hàm. Nếu khi đọc có lỗi *sgetc()* sẽ trả lại giá trị *EOF*.

int snextc(): Con trỏ *get pointer* được tăng trước khi đọc ký tự từ *buffer*.

int sgetn(char, int n)*: Đọc *n* ký tự tiếp theo vào *buffer s*.

Cần kiểm tra trường hợp *underflow* khi đọc. Hàm này gọi *streambuf::do_sgetn(...)* nếu số ký tự còn lại trong *buffer* nhỏ hơn *n*. Giá trị trả lại của hàm là số lượng các ký tự đã được đọc.

virtual int do_sgetn(char s, int):* Hàm *virtual* này dùng để điều khiển *buffer underflow* khi thực hiện hàm *sgetn(...)* đã được trình bày ở trên. Hàm sẽ sao chép *n* ký tự từ *buffer* vào *s*. Giá trị trả lại của hàm là số ký tự đã được đọc.

virtual int underflow(): Hàm được tự động gọi khi xảy ra trường hợp *underflow* khi đọc một ký tự từ *buffer (buffer* rỗng). Hàm *underflow* trong lớp *streambuf* tuy vậy không làm bất cứ điều gì khác ngoại trừ việc trả lại giá trị *EOF*. Các công việc cụ thể của *underflow* phải được thiết lập trong các hàm tương ứng trong các lớp dẫn xuất.

int sputback(char): Hàm cho phép đưa một ký tự vào vùng *get* (vùng đọc) nếu có chỗ. Ký tự này có thể là ký tự bất kỳ kể cả ký tự đã được đọc ra từ *buffer* trước đó. Nếu thực hiện thành công, ký tự này sẽ được nhận như giá trị được trả lại bởi hàm *sputback()*. Nếu trong vùng nhập dữ liệu không còn chỗ, hàm *virtual int streambuf::pbackfail(int)* sẽ được gọi với tham số là ký tự đang được làm việc.

virtual int pbackfail(int): Hàm được thiết kế dành cho các lớp được dẫn xuất từ *streambuf*. Hàm được gọi bởi *streambf::sputback(char)* khi cần đưa lại một ký tự vào vùng nhập nhưng không còn chỗ. Giá trị ngầm định được trả lại của *pbackfail(int)* là *EOF*. Để có thể được sử dụng, hàm cần được định nghĩa lại trong lớp dẫn xuất.

int *in_avail():* Hàm trả lại số ký tự hiện có trong vùng nhập (*get buffer*).

int sputc(int): Hàm sử dụng con trỏ *pptr_* để xuất ký tự vào *buffer*. Giá trị trả lại của hàm là ký tự được xuất nếu thao tác thực hiện thành công. Trong trường hợp ngược lại, giá trị trả lại của hàm là *EOF*. Sau khi thực hiện, con trỏ *pptr_* sẽ được điều chỉnh lại.

int sputn(const char s, int n):* Sao chép *n* ký tự đầu tiên của mảng *s* vào *buffer*. Còn trỏ *pptr_* được điều chỉnh lại sau khi thực hiện sao chép. Giá trị trả lại của hàm là số lượng ký tự được sao chép nếu độ dài của *buffer* lớn hơn hoặc bằng *n*. Trong trường hợp ngược lại, hàm *do_spuntn(const c har*, int)* sẽ được gọi.

virtual int overflow(int - EOF): Hàm này cần được định nghĩa lại trong các lớp dẫn xuất từ *streambuf*. Hàm được gọi khi thực hiện một thao tác xuất vào *buffer* trong khi *buffer* đã bị đầy. Thông thường trong các lớp được dẫn xuất, hàm được thiết kế để có thể sao chép toàn bộ nội dung cũ của *buffer* đến các thiết bị hoặc các *buffer* khác và xuất ký tự đang làm việc vào *buffer*. Trước khi xuất ký tự này, cần điều chỉnh lại vị trí của các con trỏ để *buffer* có thể nhận thêm các ký tự mới. Giá trị *EOF* là giá trị được trả lại nếu hàm thực hiện có lỗi.

int out_waiting(): Trả lại số ký tự hiện có trong vùng xuất.

virtual streampos seekoff(streamoff seek_dir, int = (ios: :in | ios:: out) : Hàm sẽ dịch chuyển các con trỏ của *streambuf buffer* với khoảng cách bằng số các ký tự được định nghĩa bởi tham số truyền thứ nhất. Tham số truyền thứ hai dùng để xác định vị trí

bắt đầu và hướng dịch chuyển được khai báo theo kiểu *enum*. Loại dữ liệu này được định nghĩa trong lớp *ios* theo cách sau:

enum seek_dir { beg =0, cur =2, end =2} ;

Trong đó :

beg =0 - Xác định hướng dịch chuyển từ đầu *buffer*.

cur = 1 - Xác định hướng dịch chuyển từ điểm hiện thời đến cuối *buffer*.

end =2 - Xác định hướng dịch chuyển từ cuối *buffer*.

Tham số truyền thứ ba sẽ xác định loại con trỏ cần dịch chuyển. Nếu giá trị của tham số là *ios: :in* thì *pptr_* sẽ được dịch chuyển, nếu giá trị của tham số là *ios::out* thì con trỏ *gptr_* sẽ được dịch chuyển. Ngầm định thì cả hai con trỏ này đều được dịch chuyển.

Cần chú ý là hàm *streambuf::seekoff(...)* không làm điều gì khác ngoài việc trả lại giá trị *EOF* và cần phải được thiết kế lại trong các lớp dẫn xuất.

virtual streampos seekpos(streampos, int = (ios::in | ios:out): Hàm này tương tự như hàm *seekoff* được mô tả ở trên nhưng không sử dụng tham số xác định hướng dịch chuyển của các con trỏ. Thay vào đó tham số thứ nhất sẽ dùng để xác định vị trí bất kỳ trong *buffer* cần dịch chuyển đến. Tham số thứ hai cũng có tác dụng như tham số thứ ba của hàm *seekoff*.

virtual int sync() : Hàm trả lại giá trị khác 0 nếu dữ liệu có sẵn trong vùng xuất (*put area*) hoặc vùng nhập (*get area*). Trong trường hợp ngược lại, giá trị trả lại của hàm là *0*.

Các thành phần thuộc kiểu protected:

Các hàm sau đây dùng để truy nhập đến các biến kiểu

private của *streambuf:*

char base()* : Trả lại giá trị của con trỏ *base_*.

char ebuf()* : Trả lại giá trị của con trỏ *ebuf_*.

char blen()* : Trả lại giá trị *ebuf_ - base_).*

char pbase()* : Trả lại giá trị của con trỏ *pbase_*.

char pptr()* : Trả lại giá trị của con trỏ *pptr_*.

char epptr()* : Trả lại giá trị của con trỏ *epptr_*.

char eback()* : Trả lại giá trị của con trỏ *eback_*.

char gptr()* : Trả lại giá trị của con trỏ *gptr_*.

char egptr()* : Trả lại giá trị của con trỏ *egptr_*.

void setp(char char*)* : Hàm *setp* thiết lập các con trỏ cho vùng xuất của *buffer*. Tham số truyền thứ nhất khởi tạo *pptr_* và *pbase_*, trong khi tham số thứ hai khởi tạo *epptr_*. Hai tham số này phải chỉ đến hai vị trí khác nhau.

void setg(char, char*, char*) : setg* dùng để khởi tạo các con trỏ cho vùng nhập. Tham số truyền thứ nhất là giá trị của *eback_*. Tham số thứ hai là giá trị của *gptr_*. Tham số thứ ba là giá trị của *egptr_*.

void pbump(int amount) : Hàm này có thể dịch chuyển con trỏ *pptr_* theo hai hướng khác nhau với khoảng cách bằng *amount*. Hướng dịch chuyển của con trỏ phụ thuộc vào dấu của *amount*. Nếu *amount* > 0, hướng dịch chuyển là chiều tăng của con trỏ, nếu *amount* < 0, hướng dịch chuyển là chiều giảm dần của con trỏ.

void gbump(int amount) : Dịch chuyển con trỏ *gptr_*. Hướng

dịch chuyển cùng phụ thuộc vào dấu của *amount*.

void setb(char, char*, int =0)* : Hàm dùng để thiết lập các con trỏ cơ sở *base_* và *ebuf_* cho *buffer* nào đó, *buffer* này sẽ bị loại bỏ và được thay thế bởi *buffer* mới. Tham số thứ nhất của hàm là giá trị của *base_*, tham số thứ hai là giá trị của *ebuf_*. Tham số thứ ba dùng để xác định có sử dụng nội dung có sẵn của *buffer* đang được thiết lập hay không. Giá trị của tham số này bằng số ký tự có sẵn trong *buffer*. Giá trị ngầm định bằng 0 chỉ ra rằng *buffer* không chứa dữ liệu.

void unbuffered(int) : Hàm thiết lập chế độ làm việc của *streambuf* với *buffer*. Chế độ này được thiết lập thông qua biến *unbuf_* biến được khai báo với mức độ truy nhập *private* trong lớp *streambuf_*. nếu giá trị của tham số truyền khi gọi hàm *unbuffered* khác 0, *unbuf_* sẽ được thiết lập và *buffer* không được sử dụng. Trong trường hợp này có thể trực tiếp nhập hoặc xuất ký tự đến thiết bị đầu cuối mà không cần sử dụng *buffer*. Mỗi khi ký tự được xuất vào *streambuf*, hàm *overflow* sẽ được gọi. Ngược lại khi ký tự được đọc từ *streambuf*, hàm *underflow* sẽ được gọi. Các hàm này phải được thiết kế để có thể gửi ký tự đến thiết bị đầu cuối. Để thiết lập lại *buffer*, hàm *unbuffered* cần được gọi với tham số bằng 0.

int unbuffered() : Trả lại giá trị khác 0 nếu *streambuf* được sử dụng trong chế độ không có bộ đệm.

int allocate() : Hàm sử dụng con trỏ *base_* và *ebuf_* để thiết lập *buffer* mới. Hàm trả lại giá trị bằng 0 nếu *buffer* trước đó đã được phân bố hoặc chế độ làm việc là *unbuffered*. Trong trường hợp ngược lại hàm sẽ nhận giá trị được trả lại bởi

streambuf::doallocate().

virtual int doallocate() : Đây là hàm ảo được xây dựng cho các lớp dẫn xuất từ *streambuf*. Hàm trả lại giá trị là *EOF* khi có lỗi và giá trị 0 trong trường hợp ngược lại.

Các thành phần private:

int do_snextc() : Hàm được sử dụng để nhận ký tự từ *streambuf*. Hàm được gọi bởi *streambuf::snextc()* khi cần nhập ký tự nhưng con trỏ *gptr_* chưa được khởi tạo, hoặc khi vùng nhập không còn ký tự.

streambuf(streambuf &) : Hàm được khai báo nhưng không được định nghĩa với mục đích tránh việc sử dụng *constructor* sao chép cho các đối tượng thuộc lớp *streambuf*.

void operator = (streambuf &) : Toán tử này cũng được khai báo nhưng không được định nghĩa để tránh việc sử dụng phép gán cho các đối tượng được khởi tạo từ *streambuf*.

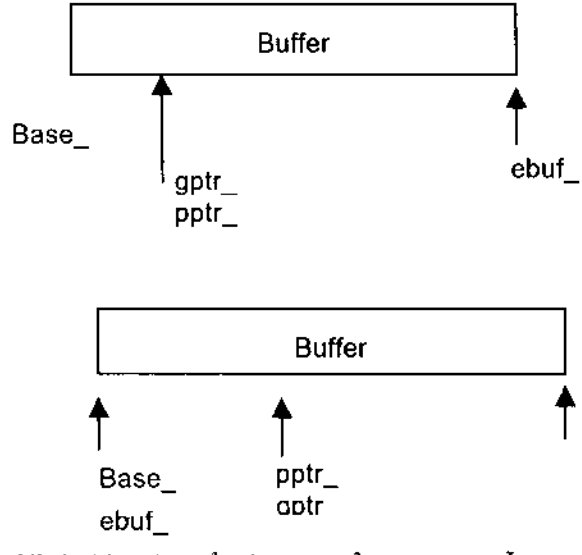
11.1.1.2. Sử dụng con trỏ put (pptr_) và con trỏ get (gptr_)

Có thể nhận thấy việc sử dụng con trỏ *put* và con trỏ *get* sẽ rất dễ dàng nếu hai con trỏ này làm việc độc lập với nhau, tức là *buffer* hoặc chỉ được sử dụng cho thao tác xuất hoặc chỉ được sử dụng cho thao tác nhập.

Nếu chỉ sử dụng con trỏ *put (pptr_)* thì vùng xuất sẽ được xem là bị tràn (*oveflow*) nếu con trỏ *pptr_* đạt đến cuối *buffer* (*pptr_* bằng *ebuf_* và khi đó *buffer* cần được làm sạch để có thể nhận thêm ký tự mới. Ngược lại, nếu chỉ sử dụng con trỏ get (*gptr_*) thì *buffer* sẽ bị xem là rộng (*undeflow*) nếu con trỏ *gptr_*

đạt đến cuối *buffer* (*gptr_* bằng *ebuf_*) và cần được xuất thêm dữ liệu vào *buffer*.

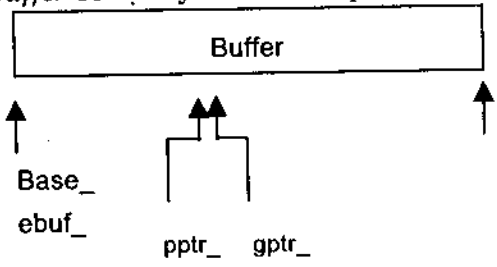
Việc xác định hiện tượng *overflow* và *underflow* tuy vậy sẽ trở nên phức tạp hơn nếu *buffer* được sử dụng cho thao tác xuất và nhập - tức là việc nhập và xuất các ký tự trên *buffer* được tiến hành song song với nhau. Vì nguyên tắc làm việc với *buffer* là *FIFO (First - In, First - Out)*, do đó con trỏ *get* và *put* phải di chuyển theo một trình tự nhất định. Nếu con trỏ *get* đuổi kịp con trỏ *put*, tức là *pptr_* và *gptr_* cùng chỉ đến một vị trí trong *buffer* (vị trí này không cần thiết là điểm xuất phát của *buffer*) khi đó *buffer* được xem là rộng. Nếu con trỏ *put* đuổi kịp con trỏ *get* khi đó có thể xem như *buffer* bị đầy.



Hình 11.3. Mô tả các con trỏ khi buffer rỗng

Vấn đề có vẻ khó hiểu ở đây là tại sao con trỏ *put* lại có thể đuổi kịp con trỏ *get* ? Điều này có thể xảy ra là do hai con trỏ *get* và *put* được sử dụng cùng lúc nên khi hiện tượng *overflow* xảy ra

(con trỏ *put* đạt đến cuối *buffer*) con trỏ của vùng xuất sẽ được thiết lập lại (trở về vị trí đầu của *buffer*) trong khi dữ liệu trong *buffer* và các con trỏ của vùng nhập vẫn ở trạng thái cũ vì vậy dễ dàng nhận thấy là khi quá trình xuất dữ liệu vào *buffer* được tiếp tục thì *buffer* sẽ bị đầy khi con trỏ *put* đuổi kịp con trỏ *get*.



Hình 11.4. Mô tả các con trỏ khi buffer bị đầy

Trên thực tế, việc sử dụng *buffer* một cách đầy đủ nhất còn đòi hỏi phải điều khiển nhiều loại con trỏ hơn so với các con trỏ được mô tả trên hình 11.4. Chẳng hạn việc sử dụng hàm *streambuf:sputback(char)* sẽ đòi hỏi các con trỏ *epptr_* và *eback_* và để đảm bảo sự độc lập hoàn toàn giữa hai vùng nhập và xuất thì việc sử dụng con trỏ *egptr_* cũng sẽ là điều cần thiết. Tuy vậy cũng cần lưu ý là việc sử dụng tất cả các con trỏ không phải là điều bắt buộc trong tất cả các trường hợp.

Để có thể hiểu một cách cặn kẽ hơn việc sử dụng lớp *streambuf* trong quá trình nhập và xuất dữ liệu với *buffer*, hãy xét một số ví dụ dưới đây.

Ví dụ 11.1

```
#include <iostream.h>
#include<conio.h>
int const SIZE_BUFFER = 20;
class custom_buffer:public streambuf
```

```
{
    char buffer[SIZE_buffer];
    int status_overflow;
    int status_underflow;
public:
    custom_buffer():streambuf(buffer.SIZE_buffer)
    {
        status_overflow =0;
        status_underflow =0;
    } ;
    void SendTestPacket(char c);
    void GetTestPacket(char *);
    int underflow();
    int overflow(int); } ;
void custom_buffer::SendTestPacket(char c)
{
    for(int i=0;i<blen();i++)
    {
        sputc(c);
        if(status_overflow ==1)
        {
            status_overflow = 0
            break;
        }
    }
}
int custom_buffer::overflow(int i)
{
    status_overflow = 1;
```

```
    if (i==EOF) return EOF;
    if(!out_waiting()) // Bắt đầu làm việc với buffer xuất
    {
        setp(base().ebuf());
        return sputc(i);
    }
    cout << "buffer bi day, va da bi lam sach "<< endl;
    char *cp = pbase();
    for (int j=0;j<blen();j++,cp++)
        *cp = " ";
        setp(base().ebuf());
        return(unsigned char) i; }
void custom_buffer:GetTestPacket(char* cp)
{
    for(int i=0;i<blen();i++)
    {
        *(cp+i) = sbumpc();
        if (status_underflow ==1)
        {
            status_underflow =0;
            break;
        }
    }
    *(cp+i)=' ';
}

int custom_buffer::underflow()
{
    if (gptr()!= NULL)
```

```
            // Underflow xảy ra sau khi thiết lập buffer
            {
                status_underflow =1;
                cout << "da doc het cac ky tu trong buffer"<< endl;
            }
            setg(base(),base(),ebuf());
            return sgetc();
        }
void main()
{
        char cp[SIZE_buffer];
        costom_buffer buf;
        clrscr();
        // Xuất blen() ký tự vào buffer
        buf.SendTestPacket('a');
        // Xảy ra overflow - Xuất thêm ký tự khi buffer đã bị đầy
        buf.SendTestPacket('b');
        // Xuất các ký tự mới vào buffer
        buf.SendTesPacket('c');
        // Xuất.toàn bộ nội dung của buffer vào mảng
        buf.GetTestPacket(cp);
        // Xuất nội dung của mảng lên màn hình
        cout<< cp<< endl;
       // Xảy ra underflow - Đọc ký tự từ buffer khi buffer đã bị rỗng
        buf.GetTestPacket(cp);
        // Xuất nội dung của mảng lên màn hình
        cout<< cp<< endl;
}
```

Ví dụ này mô tả việc đọc và xuất các ký tự trên *buffer* và các

thao tác xử lý đối với *buffer* khi xảy ra trường hợp *buffer* bị tràn (*overflow*) và khi *buffer* rỗng (*underflow*). Khi làm việc với các đối tượng thuộc lớp *streambuf, overflow* có thể bị xảy ra trong hai trường hợp - khi bắt đầu làm việc với vùng xuất, khi đó các con trỏ của vùng xuất chưa được thiết lập và đều chỉ đến *NULL* và trong trường hợp các con trỏ này đã được thiết lập và đã đạt đến cuối *buffer*. Biến *status_overflow* được thiết lập giá trị 1 khi xảy ra trường hợp thứ hai. Cũng tương tự như vậy, *underflow* cũng có thể xảy ra trong hai trường hợp - khi bắt đầu làm việc với vùng xuất và khi các con trỏ đã được khởi tạo và thao tác đọc được thực hiện khi vùng nhập đã rỗng. Biến *status_underflow* được thiết lập giá trị 1 khi trường hợp thứ hai xảy ra. Trong ví dụ có sử dụng một số hàm thành phần của lớp *streambuf:*

out_waiting() : Kiểm tra *buffer* xuất đang rỗng hoặc đã có ký tự.

setp() : Thiết lập các con trỏ *put.*

setg() : Thiết lập các con trỏ *get*

blen() : Xác định kích thước của *buffer.*

base(), ebuf(), pbase() : Làm việc với các con trỏ với mức độ truy nhập private của lớp *streambuf.*

Ngoài các hàm thành phần này, trong chương trình còn xây dựng thêm một số hàm thành phần khác của lớp *custom_buffer.*

void SendTestPacket(char c) : Xuất ký tự *c* vào *buffer* cho đến khi *buffer* được làm đầy. Khi xuất ký tự, nếu xảy ra *overflow*, quá trình xuất ký tự sẽ bị dừng lại và *buffer* sẽ được làm sạch.

void *GetTestPacket(char *)* : Đọc tất cả các ký tự từ *buffer*

vào mảng. Trong quá trình đọc, nếu *underflow* xảy ra *GetTesPacket* sẽ đặt ký tự ' \0' vào cuối mảng.

int underflow() : Thiết lập lại các con trỏ *get* khi *buffer* bị rỗng.

int overflow(int) : Thiết lập lại các con trỏ *put* khi *buffer* bị làm đầy.

Trong ví dụ 11.1 có thể nhận thấy, việc sử dụng con trỏ *put* và con trỏ *get* là hoàn toàn độc lập với nhau. Khi xảy ra *overflow*, mặc dù *buffer* còn có thể chứa nhiều ký tự, *buffer* vẫn bị làm sạch, do đó việc sử dụng con trỏ *get* để đọc tiếp các ký tự từ *buffer* là không được phép. Ngược lại khi đọc đến ký tự ở cuối *buffer* và xảy ra underflow, chương trình cũng mặc nhiên xem *buffer* đã bị rỗng mặc dù con trỏ *put* trước đó đã được thiết lập lại và đã có thể xuất ký tự vào vùng đầu của *buffer*. Như vậy nếu yêu cầu được đặt ra là phải đọc tất cả các ký tự đã được xuất vào *buffer* thì chương trình trong ví dụ này sẽ không thể thực hiện được điều này.

Để sử dụng một cách triệt để *buffer* trong quá trình nhập và xuất dữ liệu, việc đánh giá *buffer* rỗng hoặc đã bị làm đầy đòi hỏi phải kết hợp kiểm tra cả hai con trỏ *put* và *get*. Như đã đề cập đến ở trên, *buffer* được xem là đầy trong các trường hợp sau:

- Con trỏ *put* đạt đến cuối *buffer* và con trỏ *get* còn ở đầu *buffer*.
- Con trỏ *put* đạt đến cuối *buffer* và đã được thiết lập lại, quá trình xuất dữ liệu được tiếp tục cho đến khi con trỏ *put* đuổi kịp con trỏ *get* (Lúc này *overflow* xảy ra nhưng *buffer* vẫn có thể tiếp tục).

Ngược lại *buffer* được xem là rỗng trong các trường hợp sau:

- Con trỏ *get* đạt đến cuối *buffer* và con trỏ *put* ở đầu *buffer* (hiện tượng *underflow* bị xảy ra).
- Con trỏ *get* đạt đến cuối *buffer* và được thiết lập lại, quá trình nhập được tiếp tục cho đến khi con trỏ *get* vượt qua con trỏ *put (pptr_ = gptr_ -1).*

Ngoài ra cũng cần lưu ý là trong cả hai trường hợp, *overflow* và *underflow* đều xảy ra khi lần đầu tiên các thao tác nhập và xuất được thực hiện.

Như vậy nếu quá trình đọc và xuất dữ liệu được thực hiện theo một vòng khép kín và nếu sử dụng tốt các qui tắc xác định trạng thái của *buffer*, dữ liệu được xuất vào *buffer* sẽ không bị mất mát và có thể được đọc tất cả theo thứ tự được xuất vào. Thêm vào đó nếu tốc độ xuất và nhập dữ liệu trên *buffer* không khác nhau lắm thì quá trình nhập và xuất này có thể chỉ cần sử dụng *buffer* với kích thước tương đối nhỏ. Ví dụ 11.2 dưới đây sẽ mô tả việc sử dụng *buffer* theo ý tưởng được trình bày ở trên:

Ví dụ 11.2

```
#include <istream.h>
class CircularFIFO:public streambuf
{
        char* buffer;
        int size;
        int overflow(int);
        int underflow();
public:
        circularFIFO(int size);
        ~CircularFIFO() { delete buffer;}
```

```
        IsEmpty();
        IsFull();
};
circularFIFO::CircularFIFO(int s) :streambuf()
{
    // Phân bố kích thước cho buffer và liên kết buffer với streambuf
    sie = s;
    buffer = new char [size];
    setbuf(buffer,size);
    // Khởi tạo các con trỏ put và get, không thiết lập vùng
    // putback trong vùng nhập
    setp(buffer,&buffer[size];
    setg(buffer,buffer,&buffer[size]);
}
    // Hàm này tự động được gọi khi con trỏ xuất đạt đến
    // cuối buffer
int CircularFIFO::overflow(int c)
{
    // Thông báo lỗi nếu FIFO bị đầy.
    if(IsFull()) return EOF;
    // Thiết lập các con trỏ put về đầu buffer.
    setp(base(),ebuf());
    // Ghi ký tự vào buffer và tăng con trỏ put
    return sputc(c);
}
// Hàm này tự động gọi khi con trỏ get đạt đến cuối buffer
int CircularFIFO::underflow()
{
    // Thông báo lỗi nếu FIFO rỗng.
```

```
    is(IsEmpty()) return EOF;
    // Thiết lập con trỏ get về đầu buffer và không thiết
    // lập vùng puback
    setg(base(),base(),ebuf());
    // Nhận ký tự tiếp theo và tăng con trỏ get
    return sgetc();
}
// Kiểm tra FIFO rỗng
int CircularFIFO::IsEmpty()
{
    return(gptr() ==pptr()) ? 1:0;
}
// Kiểm tra FIFO đầy.
CircularFIFO::IsFull()
{
    if(gptr() ==base())
        return (pptr() ==ebuf())? 1:0;
    else return (pptr() ==gptr() -1)? 1:0;
}
void main()
{
//
int j;
//
for(int i= '0';i<'0' +120;i++)
{
    // FIFO đầy nhưng không xảy ra overflow,
    // pptr_ = gptr_ -1
    if(fifo,IsFull() ) return;
```

```
        // Xuất ký tự vào buffer
        fifo.sputc(i);
        // FIFO rỗng nhưng không xảy ra underflow,
        //gptr_ = pptr_.
        if (fifo.IsEmpty() ) return;
        // Đọc ký tự từ buffer
        j = fifo.sbumpc();
        // Xuất ký tự lên màn hình
        cout << '\n'<< j;
    }
}
```

11.1.2. Lớp strstreambuf

Lớp *streambuf* đã trình bày ở trên không có các phương pháp cụ thể để xử lý khi *buffer* bị tràn(*buffer overflow*) hoặc khi *buffer* bị rỗng (*buffer underflow*). *Streambuf* là một lớp được thiết kế chỉ để làm việc với *buffer*, *buffer* được thiết lập cho các đối tượng thuộc lớp này lại không được liên kết với các thiết bị nhập xuất.

Để có thể làm việc một cách hiệu quả hơn với các *buffer* được thiết lập trong bộ nhớ (*memory buffer*). BorlandC^{++} đưa ra một lớp mới có tên là *strstreambuf* (viết tắt của *String Streambuf*). *Streambuf* được thiết kế với các hàm thành phần riêng *overflow* và *underflow* để xử lý *buffer* khi *buffer* bị tràn hoặc bị rỗng khi xuất và nhập dữ liệu. Tuy vậy *strstreambuf* cũng thường được sử dụng như một lớp cơ sở cho các lớp khác nhằm thực hiện các thao tác đã quen biết với bộ nhớ như sao chép, khởi tạo chuỗi v.v. Bởi vì được dẫn xuất từ lớp *streambuf* do đó rất nhiều các hàm thành phần của *streambuf* được sử

dụng trong lớp này.

Dưới đây xin giới thiệu chi tiết về các hàm thành phần của lớp *strstreambuf* trong Borland C++ để các bạn tiện sử dụng.

```
class strstreambuf : public streambuf
{
// Các hàm và dữ liệu thành phần.
}
```

Các thành phần kiểu private

void (*allocf)(long)* : Con trỏ chỉ đến một hàm với giá trị trả lại là kiểu con trỏ. Giá trị ngầm định của con trỏ này là *NULL*. Nếu được định nghĩa con trỏ này sẽ chỉ đếm hàm dùng để phân bố bộ nhớ cho *buffer*. Khi sử dụng hàm này, cần lưu ý thiết lập các con trỏ cho *buffer*.

*void (*freef)(void*)* : Con trỏ chỉ đến hàm dùng để giải phóng bộ nhớ đã được cấp phát cho *buffer* khi bộ nhớ này không còn cần thiết. Biến này luôn được dùng cùng với biến *allocf* đã trình bày ở trên. Giá trị ngầm định của con trỏ *freef* là con trỏ *NULL*. Khi được sử dụng cần thiết lập lại các con trỏ cho *buffer*, trong trường hợp này tất cả các con trỏ đều phải chỉ đến *NULL*.

short ssbflags : Đây là biến có thể nhận các giá trị tương ứng với các chế độ làm việc khác nhau của *buffer*. Giá trị của *ssbflags* là một trong các giá trị của dữ liệu được định nghĩa theo kiểu *enum* trong *strstream.h:*

```
enum { dynamic = 1,frozen = 2,unlimited =4} ssbflags;
```

Nếu cờ *dynamic* được thiết lập, *buffer* sẽ được phân bố theo kiểu động (*dynamic*) và có thể tăng kích thước khi cần thiết nếu

kích thước đã được cấp cấp phát trước đó không còn phù hợp (trong trường hợp này hàm thành phần *overflow* sẽ được gọi khi *buffer* bị tràn và sẽ tăng kích thước cho *buffer*). *Buffer* cũng sẽ tự động bị loại bỏ khi đối tượng liên kết với nó ra khỏi phạm vi hoạt động. Khi một đối tượng của lớp *strstreambuf* được khởi tạo mà không cung cấp bộ nhớ cho *buffer*, *buffer* này sẽ được phân bố theo kiểu *dynamic* và cờ được thiết lập. Trong trường hợp ngược lại - nếu *buffer* được phân bố bộ nhớ khi sử dụng *constructor* để thể hiện một đối tượng của *streambuf*, *buffer* này sẽ không thuộc kiểu *dynamic* và mặc dù hàm *overflow* sẽ được gọi khi thao tác xuất được thực hiện khi *buffer* đã bị đầy. Thêm vào đó *buffer* này sẽ không tự động bị loại bỏ khi đối tượng liên kết với nó thoát khỏi phạm vi hoạt động.

Nếu cờ *frozen* được thiết lập, *buffer* sẽ không có khả năng mở rộng kích thước khi bị tràn. Cờ này có thể được xử lý thông qua hàm thành phần *strstreambuf::freeze(int)* và chỉ có các *buffer* theo kiểu *dynamic* mới có thể được thiết lập trạng thái này.

Nếu kích thước dùng để phân bố bộ nhớ cho *buffer* có giá trị nhỏ hơn 0, *ssbflags* sẽ được thiết lập giá trị *unlimit* và kích thước của buffer sẽ là Ox7FFF. Mặc dù kích thước này được thiết lập khi cờ *unlimited* được thiết lập, điểm cuối của buffer vẫn không được xác định. Đây là điều cần lưu ý khi sử dụng hàm *streambuf::streamoff(...)* để di chuyển các con trỏ của *buffer* đến một vị trí nào đó trong *buffer*. Ngoài ra nếu cờ *unlimited* được thiết lập, *buffer* sẽ không thể tăng kích thước khi bị tràn.

int next_alloc : Giá trị của kích thước bộ nhớ được phân bố cho *buffer* ở lần gọi tiếp theo của *streambuf::doallocate();*

void init(signed char start, int size, signed char* offset)* : Hàm thành phần này sẽ tự động được gọi bởi các *constructor strstreambuf(signed char*, int, signed char*)* và *strstreambuf(unsigned char*, int, unsigned char*)*. Trong các trường hợp này, cờ *dynamic* sẽ không được thiết lập và các con trỏ của *buffer* sẽ được khởi tạo bởi các giá trị sau (hình 11.5):

base_ = start

ebuf_ = start +size

pbase_ = offset

pptr_ = offset

epptr_ = start + size

eback_ = start

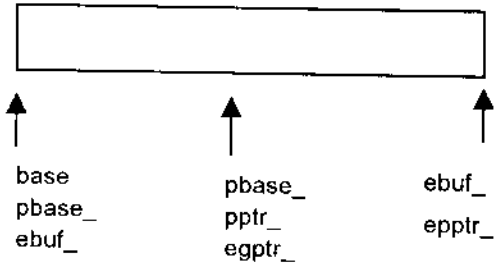
gptr_ = start

egptr_ = offset.

Khi gọi *strstreambuf: :init(...)*, nếu tham số thứ ba được truyền giá trị 0, vùng xuất (*put area*) sẽ không được thiết lập và toàn bộ *buffer* sẽ được dành cho vùng nhập. Trong trường hợp này, con trỏ của vùng xuất sẽ có giá trị 0 trong khi các con trỏ *egptr_* và *ebuf_* sẽ có giá trị bằng nhau.

Giá trị của tham số *size* không được lớn hơn 0. Nếu giá trị này bằng 0, nội dung của *buffer* sẽ được xem như một chuỗi ký tự được kết thúc bởi *NULL*.

Kích thước của *buffer* trong trường hợp này được phân bố động và chính bằng độ dài của xâu ký tự này.



Hình 11.5. Buffer được thiết lập bởi strstreambuf::int(...)

Nếu giá trị của *size* nhỏ hơn 0, cờ *unlimited* của *buffer* sẽ được thiết lập, trong trường hợp này kích thước của *buffer* sẽ là 0x7FFF.

Các thành phần kiểu public

strstreambuf() : Constructor mặc định. Khi một đối tượng được khởi tạo bởi *constructor* này, *buffer* sẽ không được thiết lập và tất cả các con trỏ sẽ được thiết lập bởi giá trị *NULL*.

strstreambuf(int n) : Với *constructor*, một đối tượng thuộc lớp *strstreambuf* khi được khởi tạo sẽ được thiết lập *buffer* thuộc kiểu *dynamic* với kích thước ban đầu bằng n. Ngoại trừ hai con trỏ *base_* và *ebuf_* chỉ đến điểm đầu và cuối của *buffer* còn tất cả các con trỏ khác đều có giá trị *NULL*.

strstreambuf(void(*allocate)(long),void(*deallocate)(void*):* Constructor này khi được sử dụng sẽ không thiết lập *buffer* cho đối tượng, thay vào đó *constructor* sẽ cho phép xác định các hàm dùng để phân bố bộ nhớ cho *buffer* cũng như để giải phóng *buffer*. Cách sử dụng của *constructor* này sẽ giới thiệu trong ví dụ 11.4.

strstreambuf(signed char s, init i, signed char* t=0)* ;

strstreambuf(unsnigned char s, int i, unsigned char* t=0);*

Hai *constructor* này được sử dụng nếu trước đó *buffer* đã được phân bố kích thước. Với các *constructor* này, các đối tượng được khởi tạo sẽ được phép sử dụng *buffer* khi làm việc. Cả hai *constructor* đều gọi hàm thành phần *strstreambuf::init(char*,int,char*)* khi khởi tạo các đối tượng.

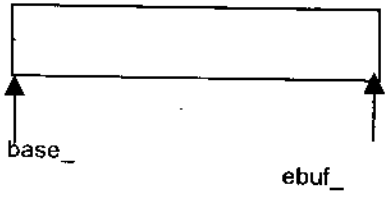
~strstreambuf() : *Destructor* dùng để giải phóng bộ nhớ đã cấp phát cho *buffer* nếu *buffer* thuộc kiểu *dynamic* và cờ *frozen* không được thiết lập. Trong trường hợp hàm giải phóng bộ nhớ đã được thiết lập trước (qua việc sử dụng con trỏ void *(*freef)(void*)* hoặc constructor *strstreambuf(void* (*allocate)(long), void (*doallocate(void*))* hàm này sẽ được gọi, ngược lại trình biên dịch sẽ gọi *::operator delete*.

void freeze(int i=1) : Hàm thành phần này sẽ cho phép xóa hoặc thiết lập cờ *frozen* trong *strstreambuf::ssbflags*. Giá trị ngầm định i=1 sẽ dùng để thiết lập cờ *frozen*. Khi cờ được thiết lập, *buffer* sẽ không có khả năng thay đổi kích thước hoặc loại bỏ ở bên trong lớp *strstreambuf*. Khi *buffer* đang ở trong trạng thái *frozen*, nếu đối tượng thuộc lớp *strstreambuf* thoát khỏi phạm vi hoạt động của nó thì người lập trình phải chịu trách nhiệm xóa bộ nhớ đã được cấp phát cho *buffer*.

Char str()* : Hàm thành phần này trả lại giá trị là kiểu con trỏ chỉ đến địa chỉ đầu của *buffer* và thêm vào đó cờ *frozen* sẽ được thiết lập. Trong trường hợp này việc thay đổi kích thước của *buffer* hoặc giải phóng vùng nhớ đã cấp phát cho nó cũng thuộc về trách nhiệm của người lập trình. Thông qua hàm *strstreambuf::str()* ta sẽ được phép truy nhập đến *buffer* đã được

thiết lập cho đối tượng.

virtual int doallocate() : Hàm ảo này được thiết lập để thay thế cho hàm tương ứng trong lớp cơ sở *streambuf*. Hàm *strstreambuf::doallocate(0)* được dùng để phân bố bộ nhớ cho *buffer*. Hàm nhận giá trị *EOF* nếu việc cấp phát có lỗi và giá trị 1 trong trường hợp ngược lại. Thông thường *::operator new* sẽ được gọi. Tuy vậy, nếu hàm phân bố bộ nhớ đã được thiết lập trước đó, hàm này sẽ được gọi bởi *strstreambuf::doallocate()*. Hình 11.6, mô tả vị trí của các con trỏ *base_* và *ebuf_* sau khi hàm thực hiện, tất cả các con trỏ khác đều không có gì thay đổi.



Hình 11.6

virtual int overflow(int) : Hàm được gọi bởi hàm *sputc(int)* của lớp cơ sở *streambuf* trong trường hợp xuất ký tự vào *buffer* nhưng *buffer* không còn chỗ. Nếu *buffer* thuộc kiểu *dynamic* cờ *frozen* không được thiết lập, kích thước của *buffer* được tăng lên bằng cách tạo ra một vùng nhớ mới và sao chép nội dung của *buffer* cũ sang vùng này sau đó các con trỏ được thiết lập lại. Nếu *buffer* thuộc kiểu *unlimited* thì *buffer* sẽ không có khả năng thay đổi kích thước khi bị tràn.

virtual underflow(): Hàm virtual này gọi được bởi hàm *sgetc()* của lớp cơ sở *streambuf* trong trường hợp nhập ký tự từ *buffer* nhưng không có ký tự trong *buffer*. Vùng nhập lúc này được thiết lập để phù hợp với vùng xuất.

Nếu vùng xuất không tồn tại, hàm sẽ trả lại giá trị *EOF*, trong trường hợp ngược lại, ký tự tiếp theo trong vùng *get* mới được thiết lập lại sẽ được đọc.

virtual streambuf setbuf(char* s, int i)* : Hàm thiết lập giá trị của biến kiểu *private next_alloc* bởi giá trị của *i*. Con trỏ *s* không được sử dụng và cần chỉ đến *NULL*. Nếu con trỏ này khác *NULL*, hàm sẽ trả lại con trỏ *NULL*, trong trường hợp ngược lại, hàm sẽ trả lại con trỏ thuộc kiểu *streambuf*.

virtual long seekoff(long offset, seek_dir dir, int mode) : Hàm dùng để di chuyển các con trỏ của *buffer*. Các con trỏ nào được di chuyển sẽ phụ thuộc vào tham số truyền *mode*. Nếu tham số truyền này là *ios::in*, hàm sẽ làm việc với con trỏ *get*, nếu tham số truyền này là *ios: :in*, hàm sẽ làm việc với con trỏ *put*, nếu tham số truyền là *ios::in | ios::out*, hàm sẽ làm việc với cả hai loại con trỏ. Tham số *dir* sẽ xác định vị trí mà từ đó các con trỏ sẽ được di chuyển với khoảng cách *offset*. Các giá trị của *dir* bao gồm *ios:beg* (vị trí bắt đầu của *buffer*), *ios::cur* (vị trí hiện thời của con trỏ) và *ios:end* (vị trí của *buffer*). Tuy vậy, như đã đề cập đến ở trên, việc di chuyển con trỏ với vị trí xuất phát là điểm cuối của *buffer* là không được phép nếu *buffer* thuộc kiểu *unlimited*. Nếu *offset* có giá trị dương con trỏ sẽ di chuyển về phía cuối của *buffer* trong khi giá trị âm sẽ di chuyển con trỏ về đầu *buffer*.

Ví dụ 11.3

Ví dụ này sẽ sử dụng lớp *strstreambuf* như lớp cơ sở để xây dựng một lớp mới có tên là *CharacterBuffer*. Lớp này sẽ chứa các hàm thành phần để làm việc với dữ liệu kiểu chuỗi. Các hàm thành phần này sẽ mô phỏng và thay thế cho các hàm của C

chuẩn như *memset(), strcpy(), memcpy().*

```
#include<strstrea.h>
int const CHRACTER_buffer_SIZE = 50;
class CharacterBuffer:public strstreambuf
    {
        int capsPlock;
    public:
    CharcterBuffer():strstreambuf(CHARaCTER_buffer_
                                        SIZE){ } ;
        void Set(char);
        void Copy(char*);
        void StringCopy(char*);
        char* Contens (return base();}
    };
// Thiết lập tất cả nội dung của buffer với ký tự cho trước
// Mô phỏng hàm memset()
void CharacterBuffer::Set(char c)
    {
    // Khởi tạo các con trỏ puts
    setp(base(),ebuf());
    // Xuất ký tự vào buffer
    for(int i=0;i<blen();i++)
        sputc(c);
}
// Mô phỏng memcpy()
void CharacterBuffer::Copy(char* cp)
    {
    setp(base(),ebuf());
    for(int i=0;i<blen()-1;i++)
```

```
            sputc(*cp++);
    sputc(0);
    }
    // Mô phỏng strcpy()
    void CharacterBuffer::StringCopy(char* cp)
          {
          setp(base(),ebuf());
          for(int i=0;i<blen();i++)
              {
              sputc(*cp++);
              if(*cp==0) return;
          }
    }
    void main()
          {
          char* string = { "abcdefghijk"} ;
          char stuff[] = { '1','2','3','4'} ;
          CharacterBuffer buffer; // Tạo buffer rỗng
          buffer.Set('a'); // menset()
          buffer.Copy(stuff); // memcpy()
          cout<< buffer.Contens();
          buffer.StringCopy(string); //strcpy()
          cout<< buffer.Contents();
    }
```

Ví dụ 11.4

Dưới đây là một ví dụ đơn giản mô tả cho việc sử dụng constructor *(voi*(*allocate)(long), void(*doallocate)(void*))* của lớp *strstreambuf.*

```
#include <strstrea.h>
void * phanbo(long);
void xoa(void *);
char* buffer;
int s;
class vidu:public strstreambuf
{
public:
      vidu();
          char* Contents() { return base();}
          void set(char);
      };
 void vidu::set(char c)
 {
       setp(buffer,&buffer[s]);
       for(int i=0;i<blen()-1;i++)
           sputc(c);
       sputc(0);
 }
 void *vidu:vidu():strstreambuf(*phanbo, *xoa)
 {
 }
 void* phanbo(long size)
 {
       s = size;
       buffer = new char [size];
       return buffer;
 }
 void xoa(void *p)
```

```
{
    delete p;
}
void main()
{
    int i;
vidu vd;
clrscr();
vd.doallocate();
vd.set('a');
cout<< vd.Contents();
}
```

Trong ví dụ này, hai hàm void **phanbo(long)* và *void xoa(void*)* là hai hàm phân bố và giải phóng bộ nhớ cho *buffer*. Hai hàm này sẽ được thiết lập khi khởi tạo đối tượng bằng constructor strstreambuf::(void* (*)(long, void(*)(void*)). Tuy các hàm này được thiết lập nhưng cũng cần lưu ý là trong khi khởi tạo đối tượng thì *buffer* vẫn chưa được cấp phát bộ nhớ. Việc cấp phát bộ nhớ cho *buffer* sẽ được tiến hành khi hàm *strstreambuf:: doallocate()* được gọi, khi đó hàm *vidu::phanbo* sẽ được thực hiện, kích thước mặc định mà trình biên dịch truyền cho hàm này là 16.

11.1.3. Lớp filebuf

Filebuf là một lớp được dẫn xuất từ lớp *streambuf* và có sử dụng các hàm thành phần của *streambuf* để hỗ trợ cho các thao tác làm việc với file. Khác với *streambuf, filebuf* được thiết kế để làm việc với một đối tượng cụ thể - *file*. Trong lớp này có rất nhiều hàm đã được thiết kế sẵn để làm việc với *file* như mở file, đóng file, đọc dữ liệu từ file và ghi dữ liệu lên file v.v. Có một số

đặc điểm chính của lớp *filebuf* mà ta cần lưu ý khi làm việc với lớp này.

- Ngoài các hàm của lớp cơ sở *streambuf* để xử lý *buffer*, trong lớp *filebuf* còn được thiết kế thêm các hàm khác để làm việc với file.
- Lớp *filebuf* sử dụng các hàm mức thấp của ANSI C như : *::open(...), :: read(...), ::write(...)* để tạo ra các ứng dụng cho phép làm việc với file.
- Các hàm *virtual* như *streambuf::overflow* và *streambuf::underflow* được xây dựng lại trong *filebuf* để hỗ trợ cho các thao tác đọc và ghi dữ liệu lên file.

Dưới đây sẽ lần lượt xem xét các loại dữ liệu và các hàm thành phần được xây dựng trong lớp *filebuf*.

Các thành phần thuộc loại protected:

int xfd : Biến *xfd* là một đại lượng dùng để mô tả file và có kiểu là *int*. Đại lượng này được gọi là *file descriptor* hay số chỉ định file. Sử dụng biến, ta có thể truy nhập đến file đã được gắn với một đối tượng thuộc kiểu *filebuf*. File *descriptor* sử dụng trong C^{++} cũng giống như biến tương ứng được dùng trong các hàm làm việc với File ở mức thấp của ANSI C. Nếu không có file nào được gắn với *filebuf*, giá trị của biến sẽ là *EOF*.

int mode : Qua giá trị của biến, có thể nhận được chế độ đã được sử dụng khi mở một file đã được gắn với một đối tượng thuộc kiểu *filebuf*. Biến *mode*, nhận một trong các giá trị sau:

ios::in // File chỉ được phép đọc.

ios::out // File chỉ được phép ghi.

ios::atc // Đưa con trỏ file đến cuối file khi mở.

ios::app // Cộng dữ liệu vào cuối file khi thực hiện thao tác ghi.

ios::trunc // Xóa nội dung cũ nếu file đã tồn tại.

ios::nocreate	// File được mở phải tồn tại trên đĩa.
ios::noreplace	// File được mở chưa tồn tại trên đĩa.
ios::binary	// Mở file trong chế độ nhị phân. Nếu cờ này không được sử dụng, file sẽ được mở trong chế độ text.

Tất cả các giá trị này được định nghĩa trong file tiêu đề chuẩn *iostream.h* và có thể được sử dụng theo các tổ hợp cùng nhau.

short openned : Biến này sẽ cho giá trị khác 0 nếu file đã được mở. Ngược lại giá trị của nó sẽ là 0.

long_last_seek : Biến thành phần này được sử dụng đối với vị trí được xác định qua thao tác cuối cùng seek.

char in_start;*

char last()* : Các biến này được khai báo nhằm đảm bảo sự tương thích đối với AT&T C++.

char lahead [2] : Đây là mảng được xây dựng cho các thao tác xuất không sử dụng buffer.

Các thành phần kiểu public :

static const int openprot : Đây là biến được dùng chung cho tất cả các đối tượng, không phụ thuộc số lượng các đối tượng được khởi tạo do biến được khai báo theo kiểu *static*. Qua biến *filebuf::openprot* có thể xác định quyền truy nhập đến file. Các

quyền truy nhập này phụ thuộc rất nhiều vào hệ điều hành đang được sử dụng. Đối với hệ điều hành DOS, quyền truy nhập ngầm định đến file là *S_IREAD | S_IWRITE* (đọc và ghi). Do được khai báo theo kiểu *static*, biến *filebuf::openprot* phải được khởi tạo ở bên ngoài tất cả các hàm thành phần và trước khi bất cứ một đoạn mã nào của chương trình được thực hiện.

filebuf() : Constructor để khởi tạo đối tượng thuộc kiểu *filebuf*. *Constructor* này phân bố filebuf thuộc kiểu *dynamic* cho đối tượng nhưng không mở bất cứ một file nào. Với *constructor* này, cần phải sử dụng các hàm thành phần khác để mở và gắn file vào đối tượng đã được khởi tạo.

filebuf(int fd) : *Constructor* này gắn file đã được mở với file *descriptor* là *fd* với đối tượng được khởi tạo và thiết lập *buffer* thuộc kiểu *dynamic*. Chế độ truy nhập (đọc, ghi, v.v) chưa được xác định và có giá trị bằng 0. Giá trị này cũng là dấu hiệu cho biết file không thuộc *filebuf* và do đó ngăn việc xóa file trong *destructor*.

filebuf(int fd, char buffer, int s)* : *Constructor* này cho phép khởi tạo đối tượng và gắn file với *descriptor fd* với đối tượng đó. Khi khởi tạo đối tượng, *buffer* với kích thước s cũng được thiết lập nhưng không thuộc loại *dynamic*.

~filebuf() : *Destructor* có thể thực hiện một trong hai nhiệm vụ sau:

Nếu biến *mode* có giá trị bằng 0 (không có file được mở hoặc file đã được mở nhưng không thuộc vào *filebuf*) nội dung của *buffer* sẽ được xuất vào file. Nếu giá trị của *mode* khác 0, nội dung của *buffer* sẽ được xuất vào file và file sẽ được đóng lại.

int is_open() : Hàm này nhận giá trị khác 0 nếu file đã được

mở. Hàm cũng cho giá trị khác 0 thậm chí khi *filebuf* không tự mở file mà nhận file *descriptor* như tham số truyền trong *constructor.*

int fd() : Nhận giá trị là *file descriptor* đang được sử dụng. Nếu không có file nào được mở, giá trị trả lại của hàm là EOF.

filebuf open(const char* n, int m, int p=filebuf::openprot)* : Hàm này mở file có tên là n trong chế độ được chỉ ra bởi tham số m và gắn với đối tượng tương ứng. Nếu thao tác mở file không thành công, hàm trả lại giá trị 0. Hàm sử dụng *::open(...)* để thực hiện thao tác mở file.

filebuf close()* : Hàm này xuất (*flushes*) tất cả nội dung của *buffer* vào file đã được gắn với đối tượng và đóng file. Hàm trả lại giá trị 0 nếu khi thực hiện có lỗi.

filebuf attach(int fd)* : Hàm này gắn file có *descriptor* là *fd* với đối tượng. *Buffer* thuộc kiểu *dynamic* sẽ được thiết lập nếu trước đó nó chưa được khởi tạo. Nếu *filebuf* đã được liên kết với file, hàm chỉ đơn thuần trả lại giá trị bằng 0. Trong trường hợp *filebuf::attach(int)* thực hiện thành công hàm sẽ trả lại giá trị là con trỏ chỉ đến một đối tượng thuộc kiểu *filebuf.*

virtual int overflow(int c = EOF) : Đây là hàm ảo và có tác dụng xuất dữ liệu trong *buffer* của *filebuf* vào file. Hàm trả lại giá trị là *EOF* nếu khi thực hiện có lỗi và 1 nếu thành công. Khi thực hiện, hàm sử dụng hàm mức thấp *::write(...)* để ghi dữ liệu lên file. Tham số truyền c là ký tự được ghi vào *filebuf buffer* sau khi thực hiện *flushing*. Nếu giá trị này là EOF sẽ không có ký tự nào được ghi vào *buffer* sau khi thực hiện *flushing.*

virtual int underfloww() : Hàm sẽ đọc dữ liệu từ file vào *buffer* của *filebuf* được gắn với file này. Nếu có lỗi xảy ra, hàm sẽ trả lại giá trị *EOF*, trong trường hợp ngược lại, ký tự đầu tiên được đọc sẽ là giá trị của hàm. Khi thực hiện hàm sử dụng hàm mức thấp *::read(...)* để đọc dữ liệu của file.

virtual int sync() : Hàm sẽ làm rỗng cả hai vùng xuất và nhập của *filebuf*. Nội dung của vùng xuất sẽ được xuất vào file và các con trỏ *put* và *get* được thiết lập về trạng thái của *buffer* rỗng. Biến *last_index* được thiết lập và cho phép đọc lại tất cả các ký tự đã được loại bỏ trong vùng *get*. Hàm nhận giá trị bằng 0 nếu khi thực hiện không có lỗi, trong trường hợp ngược lại, giá trị trả lại của hàm là *EOF*.

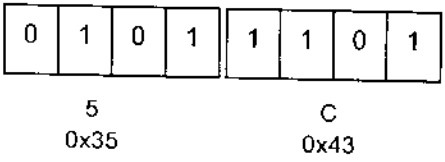
virtual long seekoff(long offset, seek_dir, int mode) : Hàm này di chuyển con trỏ *last_seek* đến vị trí nào đó trong file. Các vị trí mà từ đó con trỏ này sẽ di chuyển có thể là đầu, cuối hoặc vị trí hiện thời của file. Giá trị trả lại của hàm là vị trí mới được xác định sau khi thực hiện hàm. Trước khi di chuyển con trỏ, hàm sẽ xuất dữ liệu trong vùng xuất của *filebuf buffer* vào file bằng hàm *:write(...)*. Việc xác định vị trí mới sẽ được thực hiện qua hàm *::lseek(...)*. Giá trị trả lại của hàm sẽ là *EOF* trong trường hợp thực hiện có lỗi.

virtual streambuf setbuf(char*, int)* : Hàm này có thể sử dụng để gắn *filebuf* với *buffer*. Nếu *filebuf* đã được thiết lập *buffer* và được gắn với một file đã được mở, hàm sẽ ngừng thực hiện và trả lại giá trị 0. Nếu file chưa được mở, *buffer* cũ sẽ bị xóa (nếu có) và được thay thế bởi *buffer* mới, các con trỏ *put* và *get* sẽ được thiết lập lại.

Ví dụ dưới dây sẽ mô tả việc sử dụng lớp *filebuf* trong các

thao tác nhập xuất dữ liệu với *file*. Trong ví dụ có sử dụng lớp *HexFile* được dẫn xuất từ lớp *filebuf* dùng để lưu trữ dữ liệu lên đĩa theo phương pháp đặc biệt. Thông thường có thể nhận thấy rằng, dữ liệu được lưu trữ dưới dạng nhị phân không thể được đọc bởi các chương trình dùng để đọc ký tự (*Text Editor*). Nguyên nhân là do các chương trình *Text Editor* sẽ đọc và xem dữ liệu chứa trong các byte như mã *ASCII* của ký tự cần đọc, trong khi đó dữ liệu được lưu trữ dưới dạng nhị phân trong các byte lại có thể chứa các giá trị tương ứng với mã *ASCII* của các ký tự không được phép in ra màn hình. Chẳng hạn byte chứa dữ liệu dưới dạng nhị phân *00* là mã của ký tự *NULL*, giá trị *0x13* dưới dạng nhị phân là mã của ký tự xuống dòng v.v. Tất cả các ký tự này đều không thể được đọc bởi các chương trình *Text Editor*. Lớp *HexFile* được thiết kế dưới đây sẽ chuyển đổi tất cả các dữ liệu thành dạng *ASCII* được biểu diễn dưới dạng thập lục. Chẳng hạn byte nhị phân *00* sẽ được lưu trữ trong hai byte *0x30 0x30* để có thể in lên màn hình. Giá trị *CF* trong byte được chuyển đổi và lưu trữ lên file bằng hai byte *0x43* (mã *ASCII* của C) *0x46* (mã *ASCII* của F) và do đó ký tự này có thể được đọc và hiển thị bằng bất cứ một chương trình *Text Editor* bất kỳ. Như vậy khi đọc dữ liệu bằng *HexFile*, các giá trị nhị phân trong từng byte sẽ không được xem như mã *ASCII* của từng ký tự mà đầu tiên sẽ được tách thành hai phần 4 bit và dữ liệu trong từng phần ở đây sẽ được xem như một số được biểu diễn dưới dạng nhị phân (từ 0 đến F). Các số này sau đó sẽ được thay thế bởi các mã *ASCII* tương ứng (chẳng hạn số 1 được thay bằng *0x31*, số *12* (c trong dạng thập lục được chuyển đổi thành *0x43*) để sau đó có thể được đọc bởi các chương trình *Editor* và hiển thị lên màn hình. Bằng cách chuyển đổi này lớp *HexFile* sẽ cho phép dữ liệu được lưu trữ dưới dạng nhị phân được ghi lên trên file theo

khuôn dạng *Text*. Chẳng hạn byte 01011101 dưới đây có thể được *Hexfile* chuyển thành hai byte *0x35 0x43:*



Trong lớp *Hexfile* có sử dụng hai hàm thành phần: hàm *Ascii(char)* được khai báo với từ khóa *private* và dùng để chuyển đổi giá trị trong 4 bit của một byte thành mã *ASCII* của ký tự tương ứng với giá trị của số được lưu trữ ở 4 bit này. Hàm *virtual streambuf::overflow* được định nghĩa lại nhằm thực hiện việc chuyển đổi và ghi dữ liệu lên đĩa. Hàm *Hexfile::overflow* sẽ chèn thêm ký tự xuống dòng sau khi chuyển đổi 16 byte. Hàm mức thấp *::write* được sử dụng để ghi dữ liệu lên đĩa.

Ví dụ 11.5

```
#include <io.h>
#include <fstream.h>
// Thiết lập kích thước của buffer.
const int BUFFER_SIZE = 200;
class Hexfile:public filebuf
    {
        char Ascii(char);
    public:
        Hexfile():filebut(){ }
        virtual int overflow(int);
    };
char Hexfile::Ascii(char c)
    {
    static char character[] = { "123456789ABCDEF""} ;
```

```
    return character[c & 0x0F];
    }
int Hexfile::oveflow(int c)
    {
    // Buffer cho dữ liệu ASCII dưới dạng thập lục
    char hex_buffer[3*BUFFER_SIZE];
    if(!opened)
        return EOF;
    int count = out_waiting(); // Bao nhiêu ký tự có trong buffer.
    if(count)
        {
        // Thực hiện việc chuyển đổi ký tự
        for(int i=0,j=0;i<count;i++j+=2)
            {
            if(i%0x16)==0)
                hex_buffer[j++] = '\n ';
            hex_buffer[j] = Ascii(*base() +i)> 4);
            hex_buffer[j+1] = Ascii *(base()+i));
            }
        count =j;
        if(::write(xfd,hex_buffer,count) !=count)
        return EOF;
     }
     // Thiết lập các con trỏ của buffer.
     setp(base(),ebuf();
     setg(base(), base(), ebuf());
     return(c);
}
void main()
```

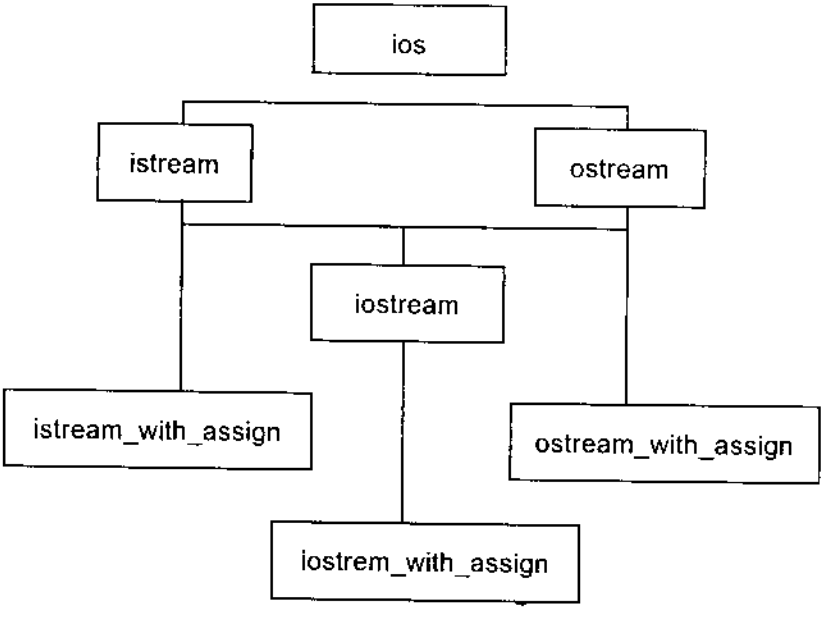
```
{
char *filename = { "test.hex");
// Khởi tạo đối tượng với dynamic buffer
HexFile file;
char buffer [BUFFER_SIZE];
// Mở file
file.open(filename,ios::out);
// Sao chép dữ liệu vào buffer
for(int i=0;i<BUFFER_SIZE;i++)
    buffer[i] = i;
// Sao chép dữ liệu từ buffer vào file buffer
file.sputn(buffer,sizeof(buffer));
// Đóng File.
file.close();
}
```

11.2. CẤU TRÚC CÂY TỪ LỚP IOS

Cấu trúc cây từ lớp *ios* được xây dựng theo kiểu thừa kế bội. Điều này không chỉ đem lại một phương pháp làm việc chuẩn tắc hơn mà còn đưa lại hiệu quả cao hơn khi thực hiện các đoạn mã chương trình. Trong cấu trúc cây mang tính thừa kế bội này thì *istream* và *ostream* có thể được xem như các lớp cơ sở và hỗ trợ cho các thao tác nhập và xuất dữ liệu. Tuy vậy cả hai thao tác nhập và xuất dữ liệu lại chứa rất nhiều đặc điểm chung như chế độ nhập xuất, khuôn dạng nhập xuất v.v. Điều này cho phép cả hai lớp *istream* và *ostream* được dẫn xuất từ một lớp cơ sở khác chứa các đặc điểm chung này - lớp ios. Trên thực tế cấu trúc cây *ios* được thiết kế cho các dòng nhập xuất để tương thích với AT&T 2.0.

11.2.1. Lớp ios

Lớp *ios* là lớp cơ sở nhất trong cấu trúc cây này và nó có tác dụng cung cấp tất cả các lệnh để điều khiển lớp với mức độ thấp nhất là *streambuf*. Mặc dù lớp *streambuf* có thể được sử dụng bất cứ lúc nào như một lớp cơ sở khi ta cần làm việc với *buffer*, nhưng để sử dụng hiệu quả hơn, một đối tượng thuộc kiểu *ios*. Khác với việc sử dụng lớp *streambuf* một cách độc lập, khi được liên kết với một đối tượng kiểu *ios* thì các thao tác làm việc với *buffer* như đọc, ghi, lấp đầy và làm rỗng *buffer* có thể được thực hiện với các dữ liệu đã được định dạng. Hình 11.7 dưới đây mô tả cho cấu trúc cây được xây dựng từ lớp *ios*.



Hình 11.7

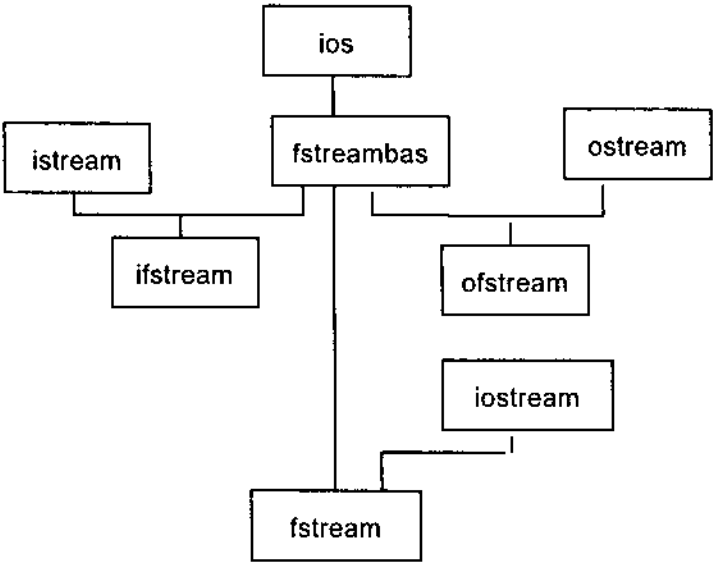
Như đã đề cập đến ở trên, bản thân lớp *ios* hầu như không thực hiện bất cứ một công việc gì ngoài việc định nghĩa và khởi tạo giá trị cho các cờ phục vụ cho các quá trình nhập xuất. Các cờ này sẽ cho biết một số thông tin được sử dụng và có thể xuất hiện trong quá trình nhập xuất dữ liệu:

- Lỗi làm việc với File.
- Các cờ định dạng.
- Các điều kiện trạng thái (status condition)

Việc sử dụng lớp *ios* chỉ có ý nghĩa khi được gắn liền với một đối tượng của một lớp khác được thừa kế từ lớp *streambuf*. Chẳng hạn, để xây dựng một đối tượng lớp *ios* dùng để làm việc với file, ta cần liên kết *ios* với đối tượng của lớp *filebuf*, vì đối tượng của lớp này có chứa các hàm thành phần để điều khiển quá trình làm việc với file như mở, đọc, ghi và đóng file. Để thực hiện được điều này, khi sử dụng các hàm thành phần của lớp *ios* cần phải được truyền tham số là con trỏ thuộc kiểu của lớp *filebuf*. Tính tương ứng bội của các hàm thành phần được thiết kế trong lớp *streambuf* sẽ cho phép làm việc với các đối tượng thuộc kiểu *filebuf* khi cần thiết.

Để tăng hiệu quả làm việc của lớp *ios*, BorlandC^{++} đưa ra một cơ cấu rất mềm dẻo để làm việc với các cờ do người sử dụng định nghĩa. Các cờ này sau đó có thể được xử lý trực tiếp bởi người sử dụng thông qua lớp *ios*. Các cờ này được lưu trữ trong một mảng theo cách cấp phát bộ nhớ động, vì vậy về nguyên tắc, số lượng các cờ (kích thước của mảng) được định nghĩa là không hạn chế.

Dưới đây là cấu trúc nhánh của lớp *ios* được mô tả ở một mức thấp hơn



Hình 11.8

Phần tiếp theo giới thiệu về các thành phần được định nghĩa trong lớp *ios*.

Các thành phần kiểu private

static long nexbit: Biến thành phần này chỉ ra giá trị của bít cờ được phân bố sau cùng.

static int usercount: Biến này được khai báo theo kiểu *static* và chỉ ra số lượng từ được phân bố cho mảng *userwords*. Các từ này (mảng *userword*) được phân bố bên trong lớp *ios* và luôn được liên kết với lớp nhằm phục vụ cho các mục đích riêng của người sử dụng. Trong mảng này mỗi một từ do người sử dụng định nghĩa được xem như một thành phần của mảng và được

truy nhập thông qua chỉ số của mảng *userword*. Giá trị của các chỉ số này được giới hạn trong khoảng từ 0 đến *0x7FFF*. Khi bắt đầu được sử dụng, lớp *ios* luôn làm việc với giá trị ban đầu của *usercount* là 0 và giá trị đó sẽ được tăng lên một đơn vị mỗi khi thêm một từ mới được phân bố. Chi tiết về việc sử dụng của các từ này sẽ được giới thiệu trong ví dụ.

*union ios_user_union *userwords* : Đây là biến thuộc kiểu con trỏ chỉ đến địa chỉ cơ sở của mảng được phân bố dành cho việc lưu trữ các từ của người sử dụng (*userword*), *ios_user_union* được khai báo theo cách sau:

```
union ios_user_union
{
long lword;
void *pword;
}
```

Như vậy dễ dàng nhận thấy rằng các thành phần của mảng có thể là một giá trị thuộc kiểu *long integer* hoặc một con trỏ chưa được xác định kiểu (kiểu *void*) và có thể được truy nhập thông qua biến *userword*.

int nword : Biến thành phần này chỉ ra kích thước của mảng được phân bố cho các *userword*. Giá trị này sẽ bằng - nếu không có một từ nào được tạo ra.

void usersize(int size) : Hàm này dùng để phân bố bộ nhớ cho mảng *userword*. Tham số *size* được truyền cho hàm chính là kích thước của vùng nhớ cần phân bố cho mảng. Nếu mảng đã tồn tại và giá trị của *size* lớn hơn kích thước cũ của mảng, mảng sẽ được phân bố lại kích thước mới cho phù hợp và giá trị cũ của

mảng sẽ được sao chép sang vùng mới.

ios(ios &) : *Constructor* này được khai báo nhưng không được định nghĩa.

Mục đích của việc khai báo này là tránh việc sử dụng *constructor* sao chép trong chương trình. Việc khai báo nhưng không định nghĩa hàm là một kỹ thuật chung hay được sử dụng nhằm hạn chế thay đổi một lớp đã được thiết kế bởi người sử dụng.

void operator =(ios&) : Toán tử này được khai báo nhưng không định nghĩa để tránh việc sử dụng toán tử gán đối với các đối tượng thuộc kiểu *ios*.

Các thành phần thuộc kiểu public

Các bit dưới đây chỉ ra trạng thái của thao tác cuối cùng được thực hiện trên *stream*. Các bit này được sử dụng cùng với *ios: :state, ios:: ispecial* và *ios::ospecial*.

```
enum ios_state
    {
    gootbit = 0x00 // Thao tác thực hiện tốt
    eofbit = 0x01 // Đã đến cuối file
    failbit = 0x02 // Thao tác cuối cùng phát sinh lỗi
                   do một lý do nào đó
    badbit = 0x04 // Thực hiện một thao tác không được phép
    hardfail = 0x80 // Một số lỗi nặng khi thực hiện thao tác
}
```

Các bit biểu thị cách mà *stream* được mở và được lưu trữ trong biến *protected filebuf::mode*. Để truy nhập đến *filebuf*, lớp *ios* có sử dụng biến bp.

enum open_mode{

in = 0x01 // Stream chỉ có tác dụng nhập

out = 0x02 // Stream chỉ có tác dụng xuất.

ate = 0x04 // Con trỏ của *stream* sẽ được chỉ đến cuối *stream* đó. Việc nhập ký tự sẽ không được phép, trong khi các thao tác xuất sẽ cho phép cộng thêm dữ liệu vào cuối *stream*.

app = 0x08 // *Stream* được mở trong chế độ appending (cộng thêm dữ liệu). Các thao tác ghi sẽ có tác dụng cộng thêm dữ liệu vào cuối *stream*.

trunc = 0x10 // Xoá tất cả dữ liệu cũ trong stream nếu có.

nocreate = 0x20 // Thao tác mở file sẽ phát sinh lỗi nếu file đó chưa tồn tại trên đĩa.

noreplace = *0x40* // Thao tác mở file sẽ phát sinh lỗi nếu file đã tồn tại trên đĩa.

binary = 0x80 // *Stream* được mở trong chế độ nhị phân. Trong trường hợp này dữ liệu chứa trong *stream* được xem là dữ liệu không phải *text*. Nếu cờ này không được thiết lập, dữ liệu trong *stream* được xem như kiểu *text*.

};

Các bit xác định hướng dịch chuyển của con trỏ trong các thao tác của *stream seek*. Các bit này không được lưu trữ trong lớp *ios*.

enum seek_dir

beg = 0 // Di chuyển tính từ đầu stream

cur = 1 // Di chuyển tính từ vị trí hiện thời

end = 2 // Di chuyển tính từ cuối stream

} ;

Các cờ dưới đây sẽ điều khiển việc tạo khuôn dạng cho các thao tác nhập xuất. Các cờ này được lưu trữ trong biến *ios::x_flags*.

enum {

skipws = 0x000 // Bỏ qua các ký tự trắng trong các thao tác nhập. Các ký tự trắng bao gồm dấu cách, dấu canh cột, dấu xuống dòng v.v.

left = 0x022 // Dóng dữ liệu trong thao tác xuất về bên trái

right = 0x004 // Dóng dữ liệu trong thao tác nhập về bên phải.

internal = 0x0008 // Cờ định dạng này chỉ có tác dụng khi đang làm việc với dữ liệu kiểu dấu phẩy động. Sử dụng cờ này sẽ cho phép chèn các ký tự sau dấu hoặc cơ số nhưng trước giá trị của số đó.

dec = 0x0010 // Chỉ ra rằng dữ liệu đang được nhập là trong cơ số 10, dữ liệu trong thao tác xuất cũng được xem ở dạng cơ số 10.

oct = 0x0020 // Chỉ ra rằng các thao nhập và xuất đang làm việc với dữ liệu trên cơ số 8.

hex = 0x0040 // Chỉ ra rằng các thao nhập và xuất đang làm

việc với dữ liệu trên cơ số 16.

showbase = 0x0080 // Bắt buộc hiển thị cơ số trong thao tác xuất dữ liệu khi đang làm việc với dữ liệu kiểu số.

showpoint = 0x0100 // Bắt buộc hiển thị dấu chấm thập phân khi làm việc với dữ liệu dấu phẩy động.

uppercase = 0x0200 // Bắt buộc dữ liệu không phải dạng số phải hiển thị dưới dạng chữ in. Dữ liệu này bao gồm cả các ký tự x khi dùng để hiển thị cơ số của dữ liệu kiểu thập lục, e khi sử dụng đối với dữ liệu được hiển thị theo kiểu số mũ v.v.

showpos = 0x0400 // Bắt buộc dấu + phải được hiển thị làm việc với các số dương.

scientific = 0x0800 // Bắt buộc phải sử dụng các số dưới dạng khoa học trong các thao tác nhập xuất. Chẳng hạn số 12345 sẽ được với khuôn dạng 1.2345e3.

fixed = 0x1000 // Bắt buộc phải sử dụng dữ liệu kiểu thập phân với độ chính xác cố định của phần thập phân.

unitbuf = 0x2000 // Bắt buộc stream sẽ được làm rỗng sau mỗi thao tác xuất.

stdio = 0x4000 // Bắt buộc làm rỗng *(flushing) cout, cerr* và *clog* sau các phép xuất

};

static const long basefield; dec | oct | hex

static const long adjustfield; left | right | internal

static const long floatfield; scientific | fixed

Các biến này dùng để biểu thị các trường bit lưu trữ các cờ của biến *ios::flags*. Các hằng số này được sử dụng như tham số truyền thứ hai khi gọi hàm *setf(long,long)*.

if(streambuf)* : *Constructor* này sử dụng con trỏ của *streambuf* như tham số truyền. Thông thường tham số truyền này là các đối tượng thuộc kiểu *strestrambuf* hoặc *filebuf*. *Stream* được khởi tạo và gắn với đối tượng thuộc kiểu *streambuf* được truyền.

virtual ~ios() : *Destructor* này chỉ làm một việc duy nhất là loại bỏ bộ nhớ đã được cấp phát cho các cờ của người sử dụng.

long flags(long l) : Thiết lập *ios::flags* với giá trị mới *f* và trả lại giá trị trước đó của *x_flags*.

long setf(long s, long f) : Sử dụng tham số *s* dưới dạng trường bit. Hàm sẽ thiết lập giá trị 1 cho các bit của *ios:x_flag* nếu hai bit tương ứng trong *s* và *f* có giá trị 1. Các bit có giá trị 0 là các bit được thiết lập trong *f* (giá trị 1) nhưng không được thiết lập trong *s* (giá trị 0). Các bit còn lại sẽ chứa giá trị cũ. Ví dụ trong trường hợp *s = 0x0080* và *0x0FF0*, hàm *setf(long, long)* sẽ thiết lập bit 7 của *ios::x_flag*. Các bit 4...6 và 8...1 sẽ bị xoá vì các bit tương ứng của s có giá trị 0 trong khi các bit tương ứng của f có giá trị 1. Công việc của *setf(long s, long f)* có thể được thực hiện bởi hai dòng lệnh dưới đây:

```
x_flags &= -f; // Xoá tất cả các bit trong f;

x_flags | = (s&f); // Thiết lập các bit cần thiết.
```

Hàm *setf(long, long)* trả lại giá trị cũ của *ios::x_flag* trong phép toán & với *f*. Tham số *f* thường là các trường bit sau đây:

basefield // Thiết lập cơ số 10, 8 hoặc 16

adjustfield // Thiết lập cách dóng hàng.

floatfield // Thiết lập cách hiển thị số thập phân (dưới dạng khoa học hay *fixed*)

long setf(long s) : Thiết lập các bit của *x_flags* tùy theo giá trị của các bit trong s. Giá trị của một bit nào đó của *x_flags* sẽ là 1 nếu bit tương ứng của s là 1, giá trị đó sẽ là 0 nếu bit tương ứng của s có giá trị 0. Hàm *setf(long s)* trả lại giá trị cũ của *ios::x_flags*. Hàm *unsetf(long s)* trả lại giá trị cũ của *ios::x_flag* trong phép toán & với s.

int width() : Trả lại giá trị của *ios::xwidth*.

int width(int) : Thiết lập *ios::xwidth* với giá trị mới và trả lại giá trị cũ. Độ dài ngầm định là 0 biểu thị không dùng ký tự độn.

char fill() : Trả lại ký tự vừa được sử dụng như ký tự độn. Thao tác độn ký tự được dùng khi kích thước của biến được chèn vào *stream* nhỏ hơn kích thước của vùng hiển thị.

char fill(char) : Thiết lập *ios::x_fill* với ký tự được sử dụng như tham số truyền. Ký tự này sẽ được sử dụng như ký tự độn khi cần thiết.

int precision() : Trả lại số lượng các số đang sử dụng trong các thao tác xuất đối với các số phẩy động.

int precision(int) : Thiết lập số lượng các số cần sử dụng trong các thao tác xuất đối với các số phẩy động.

ostream tie(ostream*)* : Gắn đối tượng của *ios* với dòng xuất. Hàm trả lại giá trị kiểu con trỏ thuộc kiểu *ostream*, con trỏ này chỉ đến dòng xuất đã được sử dụng trước đó. Dòng xuất được gắn với *ios* sẽ được xuất (*flushing*) khi *streambuf* của *ios* đòi hỏi ký tự hoặc khi *streambuf* được xuất (*flushing*).

ostream(tie)* : Trả lại dòng được gắn với *ios* đang được sử dụng.

int rdstate() : Trả lại giá trị của *ios::state*.

int eof() : Hàm nhận giá trị khác 0 nếu bit eofbit được thiết lập trong *ios:state*. Có thể sử dụng *eof()* trong việc kiểm tra khi làm việc với các dòng nhập xuất theo cách sau:

```
if(some_stream.eof()) // Đã đạt đến cuối stream

// Các lệnh.
```

int fail() : Hàm nhận giá trị khác 0 nếu ít nhất một trong các bit *failbit, babbit, hardfail* trong *ios::state* được thiết lập.

int bad() : Hàm nhận giá trị khác 0 nếu *badbit* hoặc *hardbit* trong *ios::state* được thiết lập. Có thể sử dụng *bad()* theo cách sau:

```
if(some_stream.bab()    // Thực hiện thao tác không được
                        // phép trên stream

// Các hàm
```

int good() : Có giá trị khác 0 nếu không có biến trạng thái nào được thiết lập. Trong trường hợp này thao tác trên *stream* được thực hiện tốt. Có thể sử dụng *good()* theo cách sau:

if(some_stream.good())//Không có lỗi khi làm việc với *stream*

//Các hàm

void clear(int s=0) : Hàm này thiết lập các cờ trong *ios::state* theo giá trị của tham số s mà không làm thay đổi *hardfail bit.*

operator void()* : Toán tử trả lại giá trị 0 nếu hàm ios: fail() trả lại giá trị 0. Trong trường hợp ngược lại giá trị trả lại của hàm là con trỏ *this* của đối tượng thuộc kiểu *ios*. Với toán tử này, có thể trực tiếp sử dụng tên của *stream* như một biểu thức:

if(some_stream)

// Không có lỗi trên stream

int operator! () : Toán tử này được định nghĩa chồng và có giá trị là giá trị được trả về bởi *ios: :fail()*. Với toán tử này, có thể trực tiếp sử dụng tên của *stream* như một biểu thức:

if(some_stream)

// Các lệnh

streambuf rdbuf()* : Giá trị của hàm là con trỏ chỉ đến *streambuf* được gắn với đối tượng thuộc kiểu *ios*. Con trỏ này được lưu trữ trong lớp *ios:: bp.*

static long bitalloc() : Hàm này xác định giá trị của bit tiếp theo được sử dụng cho cờ của người sử dụng. Số lượng bit được sử dụng nhiều nhất là 31. Nếu tất cả các bít đều đã được sử dụng, hàm cho giá trị là 0. Trong trường hợp ngược lại, số hiệu của bit (từ 15 đến 31) sẽ được nhận như giá trị của hàm. Một điều cần lưu ý là 16 bit đầu đã được dành riêng cho lớp *ios*. Tất cả các bit cờ (kể cả các cờ của *ios* và của người sử dụng) đều được

phân bố trong biến *ios::flags*.

static int xalloc() : Hàm này dùng để phân bố bộ nhớ cho phần tử mới mỗi khi người sử dụng cần *userword*. Hàm sẽ trả lại chỉ số của phần tử mới và có thể được sử dụng để truy cập đến phần tử này.

long& iword(int i) : Giá trị trả lại của hàm là biến tham chiếu đến userword thứ i trong mảng. Trong trường hợp này mảng *userword* dùng để chứa các giá trị thuộc kiểu *long*. Nếu giá trị của i vượt quá kích thước của mảng thì một mảng với kích thước mới sẽ được phân bố và nội dung của mảng cũ sẽ được sao chép sang. Giá trị của chỉ số i cần phải được nhận qua hàm *ios::xalloc()* và mọi phép truy nhập đến phần tử chưa được phân bố bởi *ios::alloc()* đều phát sinh lỗi. Trong trường hợp này hàm trả lại giá trị *NULL*.

void& puwod(int)* : Hàm này cũng giống như hàm *iword* với một khác biệt là giá trị trả lại không phải là *long integer* mà là con trỏ. (Con trỏ chỉ đến biến tham chiếu).

static void sync_with_stdio() : Hàm này có tác dụng làm rỗng các *stream* chuẩn *(flushing)*. Thao tác này chỉ được thực hiện một lần và do đó các thao tác tiếp theo sẽ bị bỏ qua. Mục đích của hàm là đồng bộ các dòng nhập xuất của ANSI C stdio và các dòng nhập xuất của *iostream* khi trong chương trình cả hai kiểu nhập xuất này đều được sử dụng.

int skip(int) : Mục đích của hàm này là để thiết lập hoặc xoá cờ *ios:: skipws* trong *ios::x_flags* và cờ *ios::skipping* trong *ios:ispecial*.

Các thành phần kiểu protected

streambuf bp* : Biến thành phần này là con trỏ chỉ đến *filebuf* hoặc *strstreambuf* (nếu có). Cần lưu ý là lớp *ios* chỉ có tác dụng nếu được gắn liền với một lớp thuộc kiểu *streambuf.*

ostream x_tie* : Con trỏ chỉ đến dòng xuất được gắn liền với đối tượng thuộc kiểu *ios.* Nếu dòng xuất được gắn với *ios*, dòng xuất sẽ được làm rỗng *(flushing)* khi *streambuf* của lớp ios bị tràn (*underflow*) hoặc bị làm rỗng (*flushing*).

int state : Đây là biến có chứa các cờ biểu thị kết quả của tác động cuối cùng trên đối tượng kiểu *ios.* Các cờ này được định nghĩa trong *enum* ios::io_state và bao gồm:

goodbit

eofbit

failbit

badbit

hardbit

int ispecial:

int ospecial:

Đây là các bit trạng thái dành cho 1.2 *streams.*

long x_flags : Biến này xác định giá trị cho các cờ.

Các cờ sau dùng cho các thao tác định dạng trong các phép nhập và xuất. Các bit này được định nghĩa trong biến thuộc kiểu *enum* (không có tên đích) và bao gồm:

skipws

left

right
internal
dec
oct
hex
showbase
showpoint
uppercase
showpos
scientific
fixed
unitbuf
stdio

int x_precision : Chỉ ra số lượng các số được sử dụng khi cần xuất một số thuộc kiểu phẩy động.

int x_width : Chỉ ra độ rộng của trường được sử dụng khi xuất một biến.

int x_fill : Chỉ ra ký tự độn. Ký tự độn được sử dụng khi độ dài của biến có kích thước nhỏ hơn độ rộng của trường được thiết lập cho nó.

ios() : *Constructor* để xây dựng các đối tượng thuộc lớp *ios*. Tuy vậy đối tượng của lớp ios không được liên kết với bất cứ một đối tượng nào khác được thừa kế từ *filebuf* hay *streambuf*. Cũng cần nhắc lại là trong trường hợp này ta không thể thực hiện bất cứ một thao tác nào với *ios*, thậm chí các thành phần dữ liệu của *ios* cũng không được khởi tạo bởi *constructor* này.

void init(streambuf)* : Hàm này sẽ gắn *filebuf* hoặc

streambuf với đối tượng thuộc kiểu *ios* và khởi tạo các thành phần không phải kiểu *static* của lớp.

void state(nit s) : Hàm này sẽ gán giá trị của s cho biến *ios::state*. Hàm này có điểm khác với *ios::clear(int)* ở một điểm là nó cũng tác động lên bit hardfail trong *ios::state*:

*static void(*stdioflush)()* : Đây là con trỏ chỉ đến hàm. Hàm này được sử dụng để *flushing* (làm rỗng) các dòng nhập xuất chuẩn. Tuy vậy việc sử dụng con trỏ này cũng gặp khó khăn vì không có tài liệu về cách sử dụng con trỏ này.

Trên thực tế, việc khởi tạo một đối tượng thuộc kiểu *ios* là một công việc ít gặp. Tuy vậy các thao tác nhằm sử dụng các cờ của người sử dụng lại hay xảy ra. Ví dụ dưới đây sẽ mô tả các phương pháp mà qua đó có thể xử lý trên các *userword* cũng như các cờ của người sử dụng (*user flags*). Lớp *CharacterStream* được xây dựng dưới đây có thể được mở rộng để thực hiện các thao tác với file như đọc và ghi. Lớp này có khả năng tự động chuyển đổi dữ liệu từ kiểu *text* sang dạng chữ in hoặc chữ thường. Thêm vào nó còn có khả năng lọc một ký tự cho trước trong *stream*. Ký tự này sau đó có thể được di chuyển và lưu trữ trong một xâu nào đó, xâu này sẽ được tham chiếu đến trong vùng của lớp *ios* dành cho người sử dụng.

Dưới đây là đoạn mã của lớp này:

Ví dụ 11.6

```
#include <fstream.h>
#include <string.h>
class CharacterStream:public ios
{
    int character_index;        //Index cho user flag
```

```
    int filter_index;          //Index cho user flag
    long special_stream_flag; // Chỉ sử dụng một bit
public:
    enum cs_case
        {
            any_case,
            uppercace,
            lower_case,
            filter
        } ;
CharacterStream(filebuf*, cs_case = any_case, char* =0);
};
CharacterStream::CharacterStream(filebuf* fb, cs_case
                                 mode, char* filter_string_::ios(fb)
{
// Tìm chỉ số mới cho user flag
character_index = xalloc();
// Tăng kích thước vùng nhớ sử dụng cho user flag và
// thiết lập userr flag
iword(character_index) = mode;
// Thiết lập flag cho trạng thái thích hợp
special_stream_flag = bitalloc();
if(special_stream_flag)
    if(mode !=any_case)
        setf(special_stream_flag);
    else
    // Không cần thiết nhưng đưa vào để minh họa
        unsetf(special_stream_flag);
 }
```

```
// Thiết lập ký tự cần lọc
if (mode == filter)
{
    // Tìm chỉ số cho user flag
    filter_index = xalloc();
    // Mở rộng user flags và lưu lại con trỏ của chuỗi.
    pword(filter_index) = filter_string;
}
}
```

Trong ví dụ trên, hàm *ios::xalloc* trả lại chỉ số của *userword* sẽ được sử dụng và giá trị này được sử dụng như tham số truyền khi gọi hàm *ios:: iword(int)* để tạo vùng nhớ lưu trữ *user flag*. Từ này sau đó sẽ được thiết lập với một giá trị cụ thể. *Userword* có thể là một trong hai loại dữ liệu như đã đề cập ở trên là *long integer* hoặc con trỏ chỉ đến địa chỉ của loại dữ liệu chưa được xác định trước (*void* *). Cả hai loại dữ liệu này của *userword* đều được sử dụng trong ví dụ 11.6 qua hai hàm *ios::iword(int)* và *ios::pword(int)*. Một điều cần lưu ý là giá trị trả lại từ hai hàm này đều là tham chiếu mặc dù giá trị này không được sử dụng ở trong chương trình. Vùng nhớ chứa các dữ liệu này được phân bố động và có thể được thay đổi kích thước khi cần dưa thêm các *userword*, tốt nhất vùng nhớ này nên được phân bố động bên trong lớp và chỉ cần lưu lại con trỏ chỉ đến vùng này trong vùng của *userword*. Chỉ số lớn nhất cho vùng *userword* được giới hạn đến *0x7fff* do giá trị của biến này được khai báo theo kiểu *signed int*.

Lớp *ios* sử dụng một số lượng các cờ có sẵn nhằm hỗ trợ cho các thao tác định dạng, điều khiển file và phát hiện lỗi. Các cờ

này được lưu trữ trong một biến thuộc kiểu *static long integer* và vì vậy tất cả các đối tượng thuộc lớp *ios* đều sử dụng các cờ này. Cũng cần lưu ý là trong số các cờ này chỉ có 16 cờ được dành riêng cho người sử dụng, trong khi 16 cờ còn lại 9 từ 0 đến 31) được sử dụng bởi lớp *ios*. Để nhận bit cờ có thể được sử dụng, hàm *ios::bitalloc()* cần được gọi. Giá trị trả lại khác 0 từ hàm chính là giá trị của cờ mà chúng ta có thể sử dụng. Trong ví dụ 11.6, để giải quyết vấn đề, về thực chất có thể không cần phải sử dụng *user flag* bởi vì tất cả các thông tin cần thiết cho việc chuyển đổi đã được lưu trữ trong *userword*, do đó việc sử dụng các *user flag* chỉ nhằm minh họa cho phương pháp sử dụng chúng. Để thiết lập hoặc xoá các cờ có thể sử dụng các hàm *Ios::setf(long)* và *unsetf(long)*.

11.2.2. Lớp istream

Đây là lớp được thiết kế để hỗ trợ cho các thao tác nhập dữ liệu trên các dòng nhập xuất. Các dòng nhập xuất này thông thường được liên kết với file thông qua *filebuf* hoặc các bộ đệm trong vùng nhớ (*memory buffer*) thông qua *strstreambuf*. Qua hình 11.5, có thể nhận thấy rằng *istream* được thiết kế trong một cơ cấu thừa kế bội. Đây là nguyên nhân để *istream* được dẫn xuất từ *ios* thông qua cơ cấu ảo. Chức năng chính của *istream* là xuất dữ liệu từ *stream*. Qua việc sử dụng toán tử được định nghĩa chồng >, *istream* sẽ cho phép làm việc với các loại dữ liệu có sẵn như *char, long, double* v.v. Có thể sử dụng lớp *istream* để làm việc trực tiếp với các đối tượng kiểu file được mở trong chế độ chỉ được phép đọc. Dưới đây là mô tả của lớp *istream*.

class istream : virtual public ios

Các thành phần thuộc kiểu private

*int gcount_* : Chỉ ra số lượng các ký tự được đọc trong thao tác nhập dữ liệu không có khuôn dạng cuối cùng được thực hiện trên *stream.*

signed char do_get() : Hàm này nhận ký tự kế tiếp từ dòng nhập. Đây là hàm được thiết kế ở mức độ thấp phục vụ cho các thao tác nhập dữ liệu.

Các thành phần thuộc kiểu protected

istream(): Constructor này cho phép thiết lập một dòng nhập nhưng không gắn dòng nhập này với bất cứ một đối tượng nào thuộc kiểu *streambuf.* Cần lưu ý rằng một đối tượng thuộc kiểu *istream* chỉ phát huy tác dụng và làm việc có hiệu quả nếu nó được gắn liền với một đối tượng khác được dẫn xuất từ *streambuf* mà cụ thể là *filebuf* và *streambuf.*

void eatwhite() : Hàm này sẽ loại bỏ các ký tự trắng từ *istream.* Các ký tự trắng này có thể là dấu canh cột, xuống dòng v.v.

istream(streambuf s) : Constructor* này dùng để khởi tạo một đối tượng mới và gắn đối tượng s thuộc kiểu *streambuf* với đối tượng. Con trỏ s trong trường hợp này có thể là con trỏ chỉ đến đối tượng thuộc kiểu *filebuf* hoặc *streambuf.*

virtua ~istream() : Destructor này không thực hiện bất cứ một công việc gì vì không có một vùng nhớ nào được phân bố động khi tạo một lớp từ *istream. Destructor* của lớp nào được dẫn xuất từ *istream* sẽ được thực hiện sẽ được xác định thông qua cơ cấu tương ứng bội của C++.

int infx(int count =0) : Đây là hàm nhập thuộc kiểu tiền tố (*prefix function*). Hàm này sẽ được gọi mỗi khi có một nội dung cần xuất từ stream. Hàm hay được sử dụng để thực hiện thao tác *flushing* cho các dòng xuất được gắn với đối tượng thuộc kiểu *istream*. Điều gì sẽ xảy ra khi thực hiện flushing trên *stream* phụ thuộc vào kiểu của *streambuf* được gắn với *stream*. Đối với *file stream, flushing* sẽ bắt buộc các ký tự nằm trong *buffer* phải ghi vào file. Nếu tham số *count =0, flushing* sẽ xảy ra không điều kiện, nếu tham số này khác 0, *flushing* chỉ xảy ra nếu vùng nhập (*get area*) của *streambuf* của *istream* có chứa ít nhất *count* ký tự. Hàm được gọi là tiền tố vì *flushing* được thực hiện trước khi các ký tự được xuất từ *istream*.

int ipfx0() : Hàm này sẽ gọi *istream: :ipfx(int)* với tham số truyền bằng 0. Như vậy thao tác *flushing* luôn được thực hiện trên *output stream* được liên kết.

int ipfx1() : Hàm này sẽ gọi *istream::ipfx(int)* với tham số truyền là 1. Như vậy thao tác *flushing* chỉ được thực hiện trên *output stream* được liên kết nếu không có bất cứ ký tự nào trong *streambuf* của *istream*.

void isfx() : Hàm này ít khi được sử dụng và cũng không có tài liệu nào đề cập đến một cách chi tiết.

stream seekg(long p)* : Hàm này đặt con trỏ *get* của *stream* vào vị trí *p* tính từ vị trí bắt đầu của *stream*. Nếu *istream* được liên kết với file, thao tác này sẽ được thực hiện trên file (*file seek*). Nếu *istream* được liên kết với *strstream*, con trỏ của vùng nhớ sẽ được điều chỉnh lại.

istream seekg(long p, seek_dir point)* : Hàm thực hiện chức năng tương tự như hàm trên với một điều khác biệt là vị trí được

thiết lập sẽ phụ thuộc vào tham số *p*. Tham số này có thể là *ios:beg, ios:cur* hoặc *ios::end*.

long tellg() : Trả lại giá trị hiện hành của con trỏ *get*.

int sync() : Nhiệm vụ mà hàm thực hiện sẽ phụ thuộc vào kiểu của đối tượng *streambuf* được liên kết với *istream::sync()* sẽ thực hiện thao tác *flushing* trên vùng xuất vào đĩa, các ký tự đang chờ trong vùng nhập sẽ bị bỏ qua. Hàm sẽ nhận giá trị khác 0 nếu thao tác *flushing* lên đĩa thực hiện không có lỗi. Trong trường hợp ngược lại hàm sẽ nhận giá trị *EOF*. Nếu đối tượng được liên kết với *istream* thuộc kiểu *streambuf*, hàm *istream::sync()* chỉ thực hiện thao tác kiểm tra trạng thái của *buffer*. Hàm sẽ nhận giá trị 0, nếu không có ký tự ở vùng nhập hoặc vùng xuất, ngược lại hàm sẽ nhận giá trị *EOF*.

istream get(signed char* buffer, int size, char delimiter = '\n'* : Hàm này đọc các ký tự từ *istream* và sao chép vào *buffer* mà không để ý đến khuôn dạng của dữ liệu. Việc sao chép sẽ dừng lại khi đạt đến cuối file, khi *size* ký tự đã được đọc hoặc khi gặp ký tự được xác định bởi *delimiter*. Tuy vậy ký tự này sẽ vẫn được giữ trong *streambuf* của *istream* mà không được sao chép sang *buffer* và *buffer* thuộc kiểu *unsigned*.

istream& read(signed char buffer, int size)* : Hàm thực hiện nhiệm vụ giống như hàm trên với một điều khác biệt là không sử dụng ký tự *delimiter* và *buffer* không được kết thúc bởi ký tự '\0 '. Thông thường hàm được sử dụng để đọc các loại dữ liệu thuộc kiểu nhị phân.

istream& read(unsigned char buffer, int size)* : Giống như hàm trên nhưng làm việc với *buffer* kiểu *unsigned*.

istream& getline(signed char buffer, int size, char delimiter = '\n')* : Giống hàm *get* với các đối số tương ứng. Tuy vậy, ký tự *delimiter* sẽ được loại khỏi *streambuf* của *istream* nhưng cũng không được sao chép sang *buffer*.

istream& getline(unsigned char buffer, int size, char delimiter = '\n ')* : Giống như hàm trên nhưng làm việc với *buffer* thuộc kiểu *unsigned*.

istream& get(streambuf& s, char delimiter = '\n'): Sao chép dữ liệu từ *istream* vào *streambuf* cho đến khi gặp ký tự được xác định bởi *delimiter*. Ký tự được xác định bởi *delimiter* không được sao chép và vẫn được giữ trong istream. Dữ liệu được sao chép và lưu giữ trong *streambuf* không được kết thúc bởi ký tự '\0'.

istream& get(unsigned char& c): Đọc một ký tự từ *istream* vào một biến thuộc loại *char*. Khi thực hiện có lỗi, giá trị của *c* sẽ được thiết lập bằng *0xFF*.

istream& get(signed char&): Giống như hàm trên nhưng tham số truyền thuộc loại *signed*.

int get(): Hàm này đọc một ký tự tiếp theo từ *istream* hoặc *EOF* nếu gặp cuối file.

int peek(): Hàm trả lại ký tự tiếp theo trong *istream* nhưng không loại ký tự này khỏi istream. Hàm có tác dụng kiểm tra ký tự tiếp theo của *istream* và trả lại *EOF* khi gặp cuối file.

int gcount(): Giá trị trả lại của hàm là số lượng ký tự được

đọc cuối cùng qua thao tác xuất không định dạng từ *istream*. Cần lưu ý là tất cả các hàm xuất dữ liệu từ istream đã được trình bày như *istream::get(), istream:getline(signed char*, int, char)* v.v. đều phục vụ cho thao tác dẫn xuất dữ liệu không định dạng.

istream& putback(char c): Ký tự *c* được đưa lại vào vùng nhập của streambuf của istream nếu có chỗ.

istream& ignore(int count = 1, int target = EOF): Hàm này sẽ xuất các ký tự từ istream và loại chúng nếu một trong các điều kiện sau xảy ra:

- Hàm đã xuất *count* ký tự
- Ký tự *target* được tìm thấy.
- Đã đạt đến cuối file

*istream& operator>(istream& (*_f)(istream&))*: Đây là toán tử được dùng với mục đích chung. Toán tử được cung cấp để hỗ trợ cho các thao tác xử lý dữ liệu trên *stream* đối với các việc thiết lập istream.

*istream& operator >(ios&(*_f(ios&))*: Toán tử này cũng giống như toán tử được trình bày ở trên với một điểm khác biệt là nó hỗ trợ cho các thao tác thiết lập liên quan đến *ios*.

istream operator >(streambuf s)*: Toán tử này sao chép các ký tự từ *istream* vào *streambuf s* tính từ ký tự hiện thời của stream. Quá trình sao chép sẽ bị ngừng lại nếu đạt đến vị trí cuối cùng của *istream* hoặc *streambuf*.

Ngoài các toán tử trên, các toán tử sau sẽ hỗ trợ cho các thao tác xuất dữ liệu chuẩn từ *stream*:

istream& operator>>(signed char*)

istreambuf operator>>(unsigned char&)

istream& operator>>(unsigned char*)

istream& operator>>(signed char&)

istream& operator>>(short&)

istream& operator>>(int&)

istream& operator>>(long&)

istream& operator>>(unsigned short&)

istream& operator>>(unsigned int&)

istream& operator>>(insigned long&)

istream& operator>>(float&)

istream& operator>>(double&)

istream& operator>>(long double&)

Ví dụ 11.7

Ví dụ này mô tả một chương trình đơn giản nhằm minh họa cho cách sử dụng lớp *stream*. Trong ví dụ, một đối tượng kiểu *filebuf* được khởi tạo và sau đó được gắn với *istream*. Các thao tác thực hiện trên file tuy vậy lại không phải là mục đích của ví dụ vì chúng đều có thể thực hiện bằng cách sử dụng lớp ifstream. Ví dụ chỉ nhằm mục đích mô tả cách gắn một đối tượng thuộc kiểu *filebuf* với *istream*. Nếu bạn tạo một lớp được dẫn xuất từ *streambuf*, *filebuf* hoặc *strstreambuf*, bạn có thể xây dựng một lớp phục vụ cho các thao tác nhập dữ liệu có định dạng thông qua *istream*. Điều này sẽ không thể được thực hiện với *ifstream* vì lớp này được thiết kế từ lớp *filebuf*.

```
#include <fstream.h>
void main()
{
char buffer[80];
filebuf directory_file;
// Gắn file với filebuf.
if(directory_file.open("dir.txt",ios::in))
      {
      cout << "\n ********************************"
      << "\n * ERROR... Hãy tạo file có tên là DIR.TXT*"
      << "\n * sử dụng lệnh của Dos: md DIR.TXT và *"
      << "\n * Thử lại một lần nữa                  *"
      << "\n ********************************\n ";
      return;
      }
// Tạo input stream liên kết với file
istream directory_stream(&directory_file);
cout << "\n ************************"
<< "\n\t* Chương trình này đọc file có tên là DIT.TXT và *"
<< "\n  * hiển thị nội dung từng dòng lên màn hình  *"
<< "\n  * ************************************";
while (1)
{
      directory_stream.getline(buffer,sizeof(buffer) );
      if(directory_stream.eof() )
          break;
      cout << "\n" << bufferr;
      }
}
```

Trong ví dụ, hàm *getline* trong câu lệnh directory_stream.getline *(buffer,sizeof(buffer));* được dùng để đọc dữ liệu theo từng dòng file vào buffer. Khi đọc từng dòng của file vào buffer, các ký tự xuống dòng sẽ được thay thế bởi các ký tự '\0'. Việc đọc dữ liệu stream vào buffer cũng có thể được thực hiện trực tiếp, chẳng hạn có thể sử dụng câu lệnh sau:

```
directory_stream>buffer;
```

Như đã đề cập, cách nhập dữ liệu này có một hạn chế là việc nhập dữ liệu sẽ bị kết thúc khi gặp một ký tự trong stream. Để đọc dữ liệu từ stream, cũng có thể sử dụng hàm *get*(...). Tuy vậy, khác với *getline*, hàm *get* lại để lại ký tự *delimiter* trong stream và vì vậy ta phải thực hiện thao tác tiếp theo để loại ký tự này ra khỏi stream. Hàm get(...) có thể sử dụng bằng cách sau:

```
directory_stream.get(buffer,sizeof(buffer) );
// Loại bỏ ký tự delimiter khỏi stream
directory_stream.ignore(1,'\n ');
```

Ngoài việc sử dụng các hàm để sao chép từng dòng của file, có thể sử dụng một phương pháp khác để sao chép toàn bộ nội dung của lớp file. Lớp istream cung cấp cho chúng ta một toán tử được định nghĩa chồng để thực hiện điều này - istream& operator >(stream* s). Chẳng hạn nội dung của file có thể được xuất lên màn hình thông qua toán lệnh sau:

```
directory_stream >> cout.rdbuf();
```

trong dòng lệnh này, *cout.rdbuf()* sẽ trả lại con trỏ kiểu *streambuf* được sử dụng cho dòng xuất chuẩn và toàn bộ nội dung của file sẽ được hiển thị lên màn hình.

11.2.3. Lớp ostream

Trong khi lớp cơ sở *ios* chỉ dành riêng cho các thao tác xử lý các cờ của dòng nhập xuất như định dạng, xử lý lỗi v.v. Lớp *ostream* là một lớp được thiết kế dành riêng cho các thao tác xuất dữ liệu. *Ostream* cung cấp các toán tử để hỗ trợ cho các thao tác xuất có định dạng đối với các loại dữ liệu chuẩn cũng như các thao tác xuất dữ liệu ở dạng nhị phân. Sử dụng *ostream* có thể xuất dữ liệu vào một bộ đệm mà tham chiếu đến bộ đệm này có thể là con trỏ thuộc kiểu *streambuf*. Trên thực tế *ostream* thường được sử dụng để xuất dữ liệu vào file thông qua *filebuf* hoặc vào vùng đệm của bộ nhớ thông qua strstreambuf. Dưới đây là chi tiết về lớp *ostream*

class ostream : virutal public ios { ....}

Các thành phần thuộc kiểu private

void outstr(const signed char data, const signed char* prefix)* : Hàm thành phần này được sử dụng bởi các hàm thành phần khác để xuất dữ liệu kiểu text vào *stream*. Dữ liệu *data* được xuất sau chuỗi *prefix*. Khi xuất dữ liệu có thể thực hiện thao tác động với ký tự được định nghĩa trong *ios:: fill* và có thể đóng dữ liệu không tùy thuộc vào *ios::x_flag*.

Các thành phần thuộc kiểu *protected*:

int do_opfx(): Thông thường hàm này được gọi bởi hàm *ostream::opfx()* trước khi thực hiện thao tác xuất dữ liệu vào stream. Hàm sẽ thực hiện *flushing* trên *stream* được gắn liền với đối tượng thuộc kiểu *ostream*. Tên ngắn gọn của hàm là xuất tiền tố *(output prefix)*.

void do_osfx() : Hàm này được gọi bởi *ostream::osfx()* sau khi

dữ liệu đã được xuất vào *stream*. Hàm sẽ thực hiện *flushing* đối với *buffer* của *ostream* và các *stream* chuẩn nếu các cờ của ios được thiết lập với mục đích này. Tên ngắn gọn của hàm là *output suffix* (xuất hậu tố).

ostream() : *Constructor* này không thực hiện bất cứ điều gì.

Các thành phần thuộc kiểu public

ostream(streambuf s)* : Hàm này sẽ sử dụng *streambuf s* và gắn *s* với đối tượng thuộc kiểu *ostream*. Thông thường con trỏ *s* sẽ tham chiếu đến *filebuf* hoặc *streambuf*.

virtual ~ostream(): *Destructor* này không thực hiện bất cứ điều gì ngoài việc thực hiện *flushing* trên *stream* được liên kết với đối tượng kiểu *ostream*.

int opfx(): Hàm này gọi *ostream::do_opfx()* trước khi ghi dữ liệu lên đối tượng kiểu *ostream*.

void osffix(): Gọi *ostream::do_osfx()* sau khi ghi dữ liệu lên đối tượng kiểu *ostream*.

ostream& flush() : Hàm thực hiện *flushing* trên *streambuf buffer* được liên kết với đối tượng của *stream*. *Flushing* được thực hiện thông qua việc gọi hàm thành phần *sync()* đối với đối tượng kiểu *streambuf* được liên kết với *ostream*. Nếu hàm thực hiện có lỗi, thông qua *sync()*, *bit nos::badbit* sẽ được thiết lập trong biến *ios::state*.

ostream& seekp(long p): Hàm này sẽ tác động lên con trỏ put của vùng xuất. Hàm sẽ định vị con trỏ tại vị trí p kể từ đầu của *streambuf buffer*.

ostream& seek(long displacement, see_dir s): Có tác dụng

như hàm được mô tả ở trên với một điều khác biệt là khoảng cách định vị *displacement* của con trỏ put sẽ phụ thuộc vào giá trị của s.

long tellp(): Trả lại khoảng cách của con trỏ put kể từ vị trí đầu của vùng xuất.

ostream& put(char c) : Ghi ký tự *c vào ostream*. Hàm có thể thực hiện trên dữ liệu kiểu ký tự hoặc nhị phân và không thực hiện việc định dạng dữ liệu.

ostream& write(const signed char buffer, int size)*: Hàm có tác dụng ghi dữ liệu từ *buffer* lên *ostream*. Việc ghi dữ liệu sẽ được thực hiện cho đến khi đọc hết *size* ký tự hoặc gặp lỗi. Hàm thực hiện thao tác ghi dữ liệu không định dạng và ghi dữ liệu theo khối và không phân biệt ký tự *NULL*.

ostream& write(const unsigned char buffer, int size)*: Có tác dụng như hàm trên nhưng làm việc với dữ liệu kiểu *unsigned*.

Các toán tử hỗ trợ việc xuất dữ liệu có định dạng đối với các loại dữ liệu chuẩn:

ostream& operator<< (signed char)

ostream& operator<< (unsigned char)

ostream& operator<< (short)

ostream& operator<< (unsigned short)

ostream& operator<< (nit)

ostream& operator<< (long)

ostream& operator<< (unsigned long)

ostream& operator<< (flot)

ostream& operator<< (double)

ostream& operator<< (const signed char*)

ostream& operator<< (const unsigned char*)

ostream& operator<< (void)* : Sử dụng để ghi giá trị của con trỏ kiểu bất kỳ (*generic pointer*) vào ostream (*generic pointer*).

ostream& operator << (streambuf)*: Nhập dữ liệu vào *ostream* sau khi xuất *streambuf*. Có thể sử dụng toán tử này để sao chép toàn bộ nội dung của file.

*ostream& operator << (ostream& (*_f)(ostream&)* : Toán tử với mục đích chung dùng để hỗ trợ cho các thao tác xử lý trên *ostream*.

*ostream& operator << (ios& (*_f)(ios&)* : Giống như hàm trên nhưng dùng để hỗ trợ cho các thao tác xử lý trên *ios*.

Ví dụ 11.8

Ví dụ này là một chương trình ngắn dùng để lưu trữ thư mục hiện thời vào một file có tên là *dir.txt*. Trong chương trình có sử dụng các hàm đã quen biết của *C* như *findfirst*(...) *findnext*(...).

```
#include <fstream.h>
#include <dir.h>
void main()
{
    char buffer[80];
```

```
filebuf directory_file;
// Liên kết file với đối tượng kiểu filebuf
if(!directory_file.open("dir.txt", ios:out) )
{
    cout << "Lỗi khi làm việc với dirtxt";
    return:
}
// Tạo stream xuất liên kết với file
ostream directory_stream(&directory_file);
cout << "\n  ***************************"
    << "\n  *Chương trình này ghi nội dung của thư
                    mục hiện thời*"
    << "\n  * vào file có tên là dir.txt***************"
// Tìm đường dẫn của thư mục hiện thời.
if(getcwd(buffer,sizeof(buffet) ) == 0)
{
    cout << "Lỗi khi đọc file ...\n ";
    return;
}
// Lưu đường dẫn của thư mục
directory_stream << "\n Thư mục của "
<< buffer << "\n";
// Lưu đường dẫn vào file
struct ffblk file_block;
if(findfirst*"*,*.*", &file_block,0) )
{
directory_stream << "Có lỗi khi truy nhập đến thư mục...";
    return;
}
```

```
    do
        directory_stream << file_block.ff_name << "\n ";
    while (!findnext(&file_block) );
// Không cần dùng lệnh để đóng file
// Destructor của directory_file sẽ thực hiện điều này
}
```

11.2.4. Lớp iostream

Iostream là lớp của dòng nhập xuất được sử dụng với mục đích chung cho cả hai thao tác nhập/xuất có và không định dạng trên các đối tượng được thừa kế từ *streambuf*. Về thực chất, đơn giản có thể xam đây là một lớp được kết hợp từ hai lớp đã được thiết kế trước đó là *istream* và *ostream*. Dưới đây là chi tiết về việc khai báo của lớp này.

class iostream:public istream, public ostream { ....} ;

Các thành phần protected

iostream() : Constructor này sẽ khởi tạo đối tượng thuộc lớp *iostream*.

Đối tượng này chỉ có tác dụng khi được liên kết với một đối tượng thuộc họ *streambuf*. *Constructor* được thiết kế để sử dụng cùng với các lớp của dòng nhập xuất chuẩn được dẫn xuất từ *iostream*.

Các thành phần public

iostream(streambuf*s): Khởi tạo đối tượng thuộc lớp iostream và gắn đối tượng này với streambuf để sử dụng. Thông thường con trỏ s là tham chiếu của filebuf hoặc strstreambuf.

TÀI LIỆU THAM KHẢO

1. Sams Publishing - Borland C++ 3.1 Object Oriented Programing - 1992
2. Charles Petzold - Programming Windows 3.1 - Published by Microsoft Press a Division of Microsoft Corporation, 1992
3. Sams Publishing - Graphical User interfaces with Turbo C++
4. Miller, Larry and Alex Quilici. The Official Borland Turbo C Survival Guide, New York, NY:john Wiley &Soms, 1989
5. Dương Tử Cường. Ngôn ngữ lập trình C, học và sử dụng - Nhà xuất bản Khoa học và Kỹ thuật, Hà Nội 2001.
6. Dương Tử Cường. Lập trình bằng C++. Nhà xuất bản Khoa học và Kỹ thuật, Hà Nội 1998.

MỤC LỤC

Trang

Chương III
CÁC PHÉP TOÁN VÀ
CÂU LỆNH ĐIỀU KHIỂN

Chương IV. BỘ TIỀN XỬ LÝ

Chương V
BIẾN CON TRỎ, BIẾN THAM CHIẾU VÀ HÀM

Chương VI
CÁC KIỂU DỮ LIỆU PHỨC TẠP

PHẦN II: LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Chương VII
LỚP VÀ ĐỐI TƯỢNG

Chương VIII
TÍNH THỪA KẾ

Chương IX
ĐỊNH NGHĨA CHỒNG CÁC HÀM
VÀ TOÁN TỬ

Chương X

TÍNH TƯƠNG ỨNG BỘI

Chương XI

THƯ VIỆN CÁC DÒNG NHẬP XUẤT

205264

8 935048 952641

Giá: 50.000đ