

NGÔN NGỮ LẬP TRÌNH C++

**Lập trình nâng cao với C++
Lập trình hướng đối tượng với C++**

MỤC LỤC

MỤC LỤC	2
CHƯƠNG 1	5
GIỚI THIỆU VỀ NGÔN NGỮ LẬP TRÌNH C++	5
1.1 MỘT SỐ KIẾN THỨC CƠ SỞ	5
1.2 CÁC CẤU TRÚC LỆNH ĐIỀU KHIỂN	6
1.2.1 Câu lệnh khối	6
1.2.2 Cấu trúc lệnh if.....	6
1.2.3 Cấu trúc lệnh switch.....	7
1.2.4 Vòng lặp.....	7
TỔNG KẾT CHƯƠNG 1	8
CHƯƠNG 2 CON TRỎ VÀ MẢNG	11
2.1 KHÁI NIỆM CON TRỎ	11
2.1.1 Khai báo con trỏ	11
2.1.2 Sử dụng con trỏ	11
2.2 CON TRỎ VÀ MẢNG	14
2.2.1 Con trỏ và mảng một chiều	14
2.2.2 Con trỏ và mảng nhiều chiều.....	17
2.3 CON TRỎ HÀM	18
2.4 CẤP PHÁT BỘ NHỚ ĐỘNG.....	20
2.4.1 Cấp phát bộ nhớ động cho biến.....	21
2.4.2 Cấp phát bộ nhớ cho mảng động một chiều	22
2.4.3 Cấp phát bộ nhớ cho mảng động nhiều chiều	23
TỔNG KẾT CHƯƠNG 2	25
CHƯƠNG 3	30
KIÊU DỮ LIỆU CẤU TRÚC	30
3.1 ĐỊNH NGHĨA CẤU TRÚC	30
3.1.1 Khai báo cấu trúc.....	30
3.1.2 Cấu trúc lồng nhau	31
3.1.3 Định nghĩa cấu trúc với từ khoá typedef	32
3.2 THAO TÁC TRÊN CẤU TRÚC	33
3.2.1 Khởi tạo giá trị ban đầu cho cấu trúc.....	33
3.2.2 Truy nhập đến thuộc tính của cấu trúc	34
3.3 CON TRỎ CẤU TRÚC VÀ MẢNG CẤU TRÚC	38
3.3.1 Con trỏ cấu trúc	38
3.3.2 Mảng cấu trúc.....	41
3.4 MỘT SỐ KIÊU DỮ LIỆU TRÙU TƯỢNG	44
3.4.1 Ngăn xếp	45
3.4.2 Hàng đợi.....	48
3.4.3 Danh sách liên kết	53
TỔNG KẾT CHƯƠNG 3	60
4.1 KHÁI NIỆM TỆP	64
4.1.1 Tệp dữ liệu	64
4.1.2 Tệp văn bản	65
4.1.3 Tệp nhị phân	65
4.2 VÀO RA TRÊN TỆP	66
4.2.2 Vào ra tệp nhị phân bằng read và write	70
4.3 TRUY NHẬP TỆP TRỰC TIẾP	74
4.3.1 Con trỏ tệp tin	74
4.3.2 Truy nhập vị trí hiện tại của con trỏ tệp	74
4.3.3 Dịch chuyển con trỏ tệp	76
TỔNG KẾT CHƯƠNG 4	78

CHƯƠNG 5 LỚP	82
5.1 KHÁI NIỆM LỚP ĐỐI TƯỢNG.....	82
5.1.1 Định nghĩa lớp đối tượng	82
5.1.2 Sử dụng lớp đối tượng	83
5.2 CÁC THÀNH PHẦN CỦA LỚP	83
5.2.1 Thuộc tính của lớp.....	84
5.2.2 Phương thức của lớp.....	85
5.3 PHẠM VI TRUY NHẬP LỚP	90
5.3.1 Phạm vi truy nhập lớp	90
5.3.2 Hàm bạn	91
5.3.3 Lớp bạn.....	96
5.4 HÀM KHỞI TẠO VÀ HỦY BỎ	97
5.4.1 Hàm khởi tạo	97
5.4.2 Hàm hủy bỏ.....	101
5.5 CON TRỎ ĐỐI TƯỢNG VÀ MẢNG ĐỐI TƯỢNG	103
5.5.1 Con trỏ đối tượng	103
5.5.2 Mảng các đối tượng.....	106
TỔNG KẾT CHƯƠNG 5	110
CHƯƠNG 6	115
6.1 KHÁI NIỆM KÉ THÙA.....	115
6.1.1 Khai báo thừa kế	115
6.1.2 Tính chất dẫn xuất	116
6.2 HÀM KHỞI TẠO VÀ HỦY BỎ TRONG KÉ THÙA.....	117
6.2.1 Hàm khởi tạo trong kế thừa.....	117
6.2.2 Hàm hủy bỏ trong kế thừa.....	119
6.3 TRUY NHẬP TÓI CÁC THÀNH PHẦN TRONG KÉ THÙA LỚP	120
6.3.1 Phạm vi truy nhập	120
6.3.2 Sử dụng các thành phần của lớp cơ sở từ lớp dẫn xuất	122
6.3.3 Định nghĩa chồng các phương thức của lớp cơ sở	125
6.3.4 Chuyển đổi kiểu giữa lớp cơ sở và lớp dẫn xuất	128
6.4 ĐA KÉ THÙA.....	131
6.4.1 Khai báo đa kế thừa.....	131
6.4.2 Hàm khởi tạo và hàm hủy bỏ trong đa kế thừa.....	132
6.4.3 Truy nhập các thành phần lớp trong đa kế thừa.....	134
6.5 LỚP CƠ SỞ TRÙU TƯỢNG	138
6.5.1 Đặt vấn đề	138
6.5.2 Khai báo lớp cơ sở trừu tượng	138
6.5.3 Hàm khởi tạo lớp cơ sở trừu tượng	139
6.6 ĐA HÌNH	143
6.6.1 Đặt vấn đề	143
6.6.2 Khai báo phương thức trừu tượng	144
6.6.3 Sử dụng phương thức trừu tượng – đa hình.....	144
TỔNG KẾT CHƯƠNG 6	147
CHƯƠNG 7	153
7.1 LỚP VẬT CHỨA	153
7.1.1 Giao diện của lớp Container.....	153
7.1.2 Con chay Iterator	154
7.2 LỚP TẬP HỢP	155
7.2.1 Hàm khởi tạo.....	155
7.2.2 Toán tử	155
7.2.3 Phương thức	156
7.2.4 Áp dụng	158
7.3 LỚP CHUỖI	159
7.3.1 Hàm khởi tạo	159
7.3.2 Toán tử	160

7.3.3 Phương thức	161
7.3.4 Áp dụng.....	163
7.4.1 Lớp ngăn xếp.....	165
7.4.2 Lớp hàng đợi	166
7.5 LỚP DANH SÁCH LIÊN KẾT	169
7.5.1 Hàm khởi tạo.....	169
7.5.2 Toán tử	169
7.5.3 Phương thức	170
7.5.4 Áp dụng	171
TỔNG KẾT CHƯƠNG 7	173
TÀI LIỆU THAM KHẢO	174

CHƯƠNG 1

GIỚI THIỆU VỀ NGÔN NGỮ LẬP TRÌNH C++

1.1 MỘT SỐ KIẾN THỨC CƠ SỞ.

Cấu trúc tổng quát của chương trình trong C/C++

Chương trình tổng quát viết bằng ngôn ngữ C được chia thành 6 phần, trong đó có một số phần có thể có hoặc không có tùy thuộc vào nội dung chương trình và ý đồ của mỗi lập trình viên.:

- **Phần 1:** Khai báo các chỉ thị đầu tệp và định nghĩa các hằng sử dụng trong chương trình.
- **Phần 2:** Định nghĩa các cấu trúc dữ liệu mới (user type) để sử dụng trong khi viết chương trình.
- **Phần 3:** Khai báo các biến ngoài (biến toàn cục) được sử dụng trong chương trình.
- **Phần 4:** Khai báo nguyên mẫu cho hàm (Function Prototype). Nếu khai báo qui cách xây dựng và chuyên tham biến cho hàm, compiler sẽ tự động kiểm tra giữa nguyên mẫu của hàm có phù hợp với phương thức xây dựng hàm hay không trong văn bản chương trình.
- **Phần 5:** Mô tả chi tiết các hàm, các hàm được mô tả phải phù hợp với nguyên mẫu đã được khai báo cho hàm.
- **Phần 6:** Hàm main(), hàm xác định điểm bắt đầu thực hiện chương trình và điểm kết thúc thực hiện chương trình.

Các kiểu dữ liệu cơ sở

Sau đây là bảng các giá trị có thể của các kiểu dữ liệu cơ bản

Kiểu	Miền xác định	Kích thước
char	0.. 255	1 byte
int	-2^31..2^31-1	4 byte
long	-2^63..2^63-1	8 byte
unsigned int	0 .. 2^32-1	4 byte
unsigned long	0 .. 2^64-1	8 byte
float	3. 4e-38 .. 3.4e + 38	4 byte
double	1.7e-308 .. 1.7e + 308	8 byte

Các phép toán cơ bản

Các phép toán số học: Gồm có: +, -, *, / (cộng, trừ, nhân, chia), % (lấy phần dư). Phép chia (/) sẽ cho lại một số nguyên giống như phép chia nguyên nếu chúng ta thực hiện chia hai đối tượng kiểu nguyên. Để tiện lợi trong viết chương trình cũng như giảm thiểu các kí hiệu sử dụng trong các biểu thức số học. C trang bị một số phép toán tăng và giảm mở rộng cho các số nguyên.

Các phép toán so sánh: Gồm có các phép >, =, <=, ==, != (lớn hơn, nhỏ hơn, lớn hơn hoặc bằng, nhỏ hơn hoặc bằng, đúng bằng, khác).

Các phép toán logic:

- && : Phép và logic chỉ cho giá trị đúng khi hai biểu thức tham gia đều có giá trị đúng (giá trị đúng của một biểu thức trong C đợt ngược lại là biểu thức có giá trị khác 0).
- || : Phép hoặc logic chỉ cho giá trị sai khi cả hai biểu thức tham gia đều có giá trị sai.
- ! : Phép phủ định cho giá trị đúng nếu biểu thức có giá trị sai và ngược lại cho giá trị sai khi biểu thức có giá trị đúng

Các toán tử thao tác bít: Các toán tử thao tác bít không sử dụng cho float và double:

- & : Phép và (and) các bít.
- | : Phép hoặc (or) các bít.
- ^ : Phép hoặc loại trừ bít (XOR).
- << : Phép dịch trái (dịch sang trái n bít giá trị 0)
- >> : Phép dịch phải (dịch sang phải n bít có giá trị 0)
- ~ : Phép lấy phần bù.

1.2 CÁC CÂU TRÚC LỆNH ĐIỀU KHIỂN

1.2.1 Câu lệnh khối

Tập các câu lệnh đợt ngược bao bởi hai dấu { . . . } đợt ngược gọi là một câu lệnh khối. Về cú pháp, ta có thể đặt câu lệnh khối ở một vị trí bất kì trong chương trình. Tuy nhiên, nên đặt các câu lệnh khối ứng với các chu trình điều khiển lệnh như for, while, do . . while, if . . else, switch để hiển thị rõ cấu trúc của chương trình. Ví dụ:

```
if (a > b) {  
    câu_lệnh;  
}.
```

1.2.2 Câu trúc lệnh if

Dạng 1:

```
if (biểu_thức)  
    câu_lệnh;
```

Nếu biểu thức có giá trị đúng thì thực hiện câu lệnh; Câu lệnh có thể hiểu là câu lệnh đơn hoặc câu lệnh khối, nếu câu lệnh là lệnh khối thì nó phải đợt ngược bao trong { . . }.

Dạng 2:

```
if (biểu_thức)  
    câu_lệnh_A;  
else  
    câu_lệnh_B;
```

Nếu biểu thức có giá trị đúng thì câu_lệnh_A sẽ được thực hiện, nếu biểu thức có giá trị sai thì câu_lệnh_B sẽ được thực hiện

Dạng 3:

```
if (biểu_thức_1)
    câu_lệnh_1
else if (biểu_thức_2)
    câu_lệnh_2;
.....
else if (biểu_thức_k)
    câu_lệnh_k;
else
    câu_lệnh_k+1;
```

1.2.3 Cấu trúc lệnh switch

Cú pháp

```
switch(biểu_thức_nguyên) {
    case hằng_nguyên_1: câu_lệnh_1; break;
    case hằng_nguyên_2: câu_lệnh_2; break;
    case hằng_nguyên_3: câu_lệnh_3; break;
    .....
    case hằng_nguyên_n: câu_lệnh_n; break;
    default: câu_lệnh_n+1; break;
}
```

Thực hiện: Nếu biểu_thức_nguyên có giá trị trùng với hằng_nguyên_i thì câu_lệnh_i trở đi sẽ được thực hiện cho tới khi nào gặp từ khoá break để thoát khỏi chu trình.

1.2.4 Vòng lặp

Vòng lặp for:

Cú pháp:
for(biểu_thức_1; biểu_thức_2; biểu_thức_3)
 câu_lệnh

Vòng lặp while:

Cú pháp:
while(biểu_thức)
 câu_lệnh

Vòng lặp do – while

Cú pháp:
do {
 câu_lệnh;
} while(biểu_thức);

TỔNG KẾT CHƯƠNG 1

Nội dung tập trung vào các kiến thức cơ bản có liên quan trực tiếp đến ngôn ngữ lập trình C++:

- Các kiến thức cơ sở
- Các cấu trúc lệnh điều khiển

CHƯƠNG 2

CON TRỎ VÀ MẢNG

Nội dung của chương này tập trung trình bày các vấn đề cơ bản liên quan đến các thao tác trên kiểu dữ liệu con trỏ và mảng trong C++:

- Khái niệm con trỏ, cách khai báo và sử dụng con trỏ.
- Mối quan hệ giữa con trỏ và mảng
- Con trỏ hàm
- Cấp phát bộ nhớ cho con trỏ

2.1 KHÁI NIỆM CON TRỎ

2.1.1 Khai báo con trỏ

Con trỏ là một biến đặc biệt chứa địa chỉ của một biến khác. Con trỏ có cùng kiểu dữ liệu với kiểu dữ liệu của biến mà nó trỏ tới. Cú pháp khai báo một con trỏ như sau:

<Kiểu dữ liệu> *<Tên con trỏ>;

Trong đó:

- **Kiểu dữ liệu:** Có thể là các kiểu dữ liệu cơ bản của C++, hoặc là kiểu dữ liệu có cấu trúc, hoặc là kiểu đối tượng do người dùng tự định nghĩa.
- **Tên con trỏ:** Tuân theo qui tắc đặt tên biến của C++:
 - Chỉ được bắt đầu bằng một kí tự (chữ), hoặc dấu gạch dưới “_”.
 - Bắt đầu từ kí tự thứ hai, có thể có kiểu kí tự số.
 - Không có dấu trống (space bar) trong tên biến.
 - Có phân biệt chữ hoa và chữ thường.
 - Không giới hạn độ dài tên biến.

Ví dụ, để khai báo một biến con trỏ có kiểu là int và tên là pointerInt, ta viết như sau:

```
int *pointerInt;
```

Lưu ý

- Có thể viết dấu con trỏ “*” ngay sau kiểu dữ liệu, nghĩa là hai cách khai báo sau là tương đương:

```
int *pointerInt; int*  
pointerInt;
```
- Các cách khai báo con trỏ như sau là sai cú pháp:

```
*int pointerInt; // Khai báo sai con trỏ int  
pointerInt*; // Khai báo sai con trỏ
```

2.1.2 Sử dụng con trỏ

Con trỏ được sử dụng theo hai cách:

- Dùng con trỏ để lưu địa chỉ của biến để thao tác
- Lấy giá trị của biến do con trỏ trỏ đến để thao tác

Dùng con trỏ để lưu địa chỉ của biến

Bản thân con trỏ sẽ được trỏ vào địa chỉ của một biến có cùng kiểu dữ liệu với nó. Cú pháp của phép gán như sau:

`<Tên con trỏ> = &<tên biến>;`

Lưu ý

- Trong phép toán này, tên con trỏ không có dấu “*”.

Ví dụ:

```
int x, *px; px =
&x;
```

sẽ cho con trỏ px có kiểu int trỏ vào địa chỉ của biến x có kiểu nguyên. Phép toán `&<Tên biến>` sẽ cho địa chỉ của biến tương ứng.

Lấy giá trị của biến do con trỏ trỏ đến

Phép lấy giá trị của biến do con trỏ trỏ đến được thực hiện bằng cách gọi tên:

`*<Tên con trỏ>;`

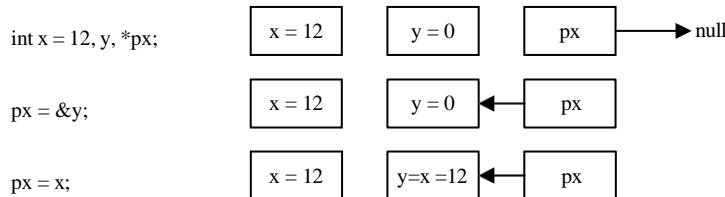
Lưu ý

- Trong phép toán này, phải có dấu con trỏ “*”. Nếu không có dấu con trỏ, sẽ trở thành phép lấy địa chỉ của biến do con trỏ trỏ tới.

Ví dụ:

```
int x = 12, y, *px; px =
&y;
*px = x;
```

Quá trình diễn ra như sau:



con trỏ px vẫn trỏ tới địa chỉ biến y và giá trị của biến y sẽ là 12.

Phép gán giữa các con trỏ

Các con trỏ cùng kiểu có thể gán cho nhau thông qua phép gán và lấy địa chỉ con trỏ:

`<Tên con trỏ 1> = <Tên con trỏ 2>;`

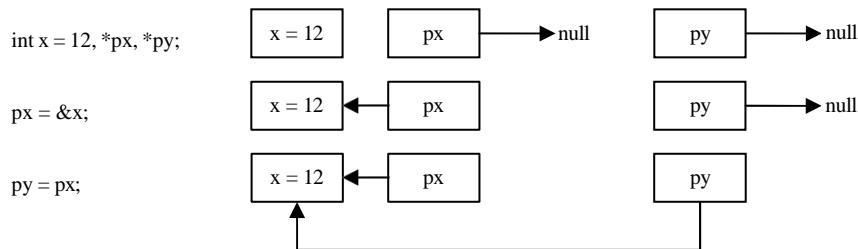
Lưu ý

- Trong phép gán giữa các con trỏ, bắt buộc phải dùng phép lấy **địa chỉ của biến** do con trỏ trỏ tới (không có dấu “*” trong tên con trỏ) mà không được dùng phép lấy **giá trị của biến** do con trỏ trỏ tới.

- Hai con trỏ phải cùng kiểu. Trong trường hợp hai con trỏ khác kiểu, phải sử dụng các phương thức ép kiểu tương tự như trong phép gán các biến thông thường có kiểu khác nhau.

Ví dụ:

```
int x = 12, *px, *py; px =
&x;
py = px;
```



con trỏ py cũng trỏ vào địa chỉ của biến x như con trỏ px. Khi đó *py cũng có giá trị 12 giống như *px và là giá trị của biến x.

Chương trình 2.1 minh họa việc dùng con trỏ giữa các biến của một chương trình C++.

Chương trình 2.1

```
#include <stdio.h>
#include <conio.h>
void main(void) {
    int x = 12, *px, *py;
    cout << "x = " << x << endl;

    px = &x;           // Con trỏ px trỏ tới địa chỉ của x
    cout << "px = &x, *px = " << *px << endl;

    *px = *px + 20;   // Nội dung của px là 32
    cout << "*px = *px+20, x = " << x << endl;

    py = px;          // Cho py trỏ tới chỗ mà px trỏ: địa chỉ của x
    *py += 15;         // Nội dung của py là 47
    cout << "py = px, *py +=15, x = " << x << endl;
}
```

Trong chương trình 2.1, ban đầu biến x có giá trị 12. Sau đó, con trỏ px trỏ vào địa chỉ của biến x nên con trỏ px cũng có giá trị 12. Tiếp theo, ta tăng giá trị của con trỏ px thêm 20, giá trị của con trỏ px là 32. Vì px đang trỏ đến địa chỉ của x nên x cũng có giá trị là 32. Sau đó, ta cho con trỏ py trỏ đến vị trí mà px đang trỏ tới (địa chỉ của biến x) nên py cũng có giá trị 32. Cuối cùng, ta tăng giá trị của con trỏ py thêm 15, py sẽ có giá trị 37. Vì py cũng đang trỏ đến địa chỉ của x nên x cũng có giá trị 37. Do đó, ví dụ 2.1 sẽ in ra kết quả như sau:

```
x = 12
px = &x, *px = 12
*px = *px + 20, x = 32
py = px, *py += 15, x = 37
```

2.2 CON TRỎ VÀ MẢNG

2.2.1 Con trỏ và mảng một chiều

Mảng một chiều

Trong C++, tên một mảng được coi là một kiểu con trỏ hằng, được định vị tại một vùng nhớ xác định và địa chỉ của tên mảng trùng với địa chỉ của phần tử đầu tiên của mảng.

Ví dụ khai báo:

```
int A[5];
```

thì địa chỉ của mảng A (cũng viết là A) sẽ trùng với địa chỉ phần tử đầu tiên của mảng A (là &A[0]) nghĩa là:

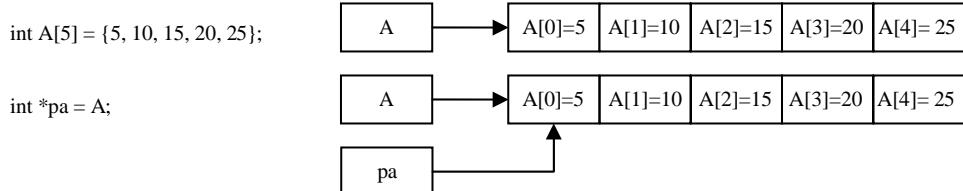
```
A = &A[0];
```

Quan hệ giữa con trỏ và mảng

Vì tên của mảng được coi như một con trỏ hằng, nên nó có thể được gán cho một con trỏ có cùng kiểu.

Ví dụ khai báo:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = A;
```



thì con trỏ pa sẽ trỏ đến mảng A, tức là trỏ đến địa chỉ của phần tử A[0], cho nên hai khai báo sau là tương đương:

```
pa = A;
pa = &A[0];
```

Với khai báo này, thì địa chỉ trỏ tới của con trỏ pa là địa chỉ của phần tử A[0] và giá trị của con trỏ pa là giá trị của phần tử A[0], tức là *pa = 5;

Phép toán trên con trỏ và mảng

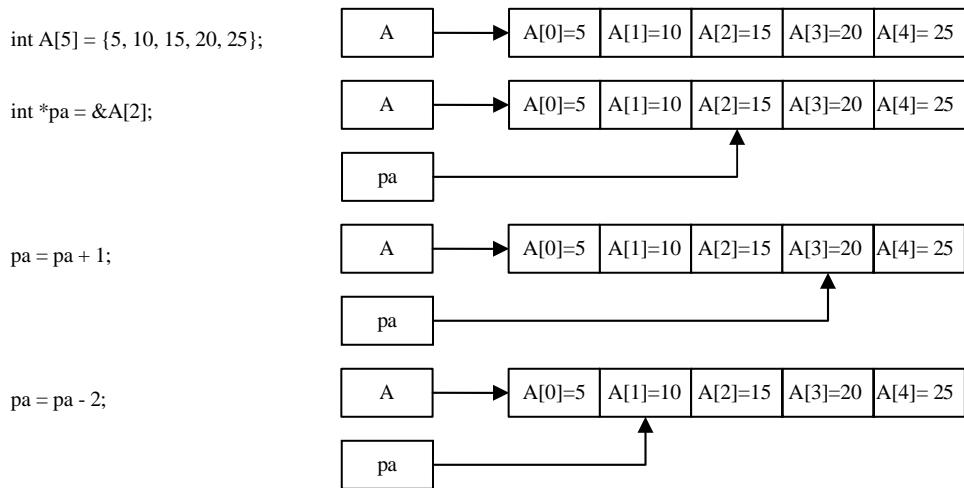
Khi một con trỏ trỏ đến mảng, thì các phép toán tăng hay giảm trên con trỏ sẽ tương ứng với phép dịch chuyển trên mảng.

Ví dụ khai báo:

```
int A[5] = {5, 10, 15, 20, 25};
```

int *pa = &A[2];
 thì con trỏ pa sẽ trỏ đến địa chỉ của phần tử A[2] và giá trị của pa là: *pa = A[2] = 15. Khi đó, phép toán:

pa = pa + 1;
 sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử tiếp theo của mảng A, đó là địa chỉ của A[3]. Sau đó, phép toán:
 pa = pa - 2;
 sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử A[1].



Lưu ý:

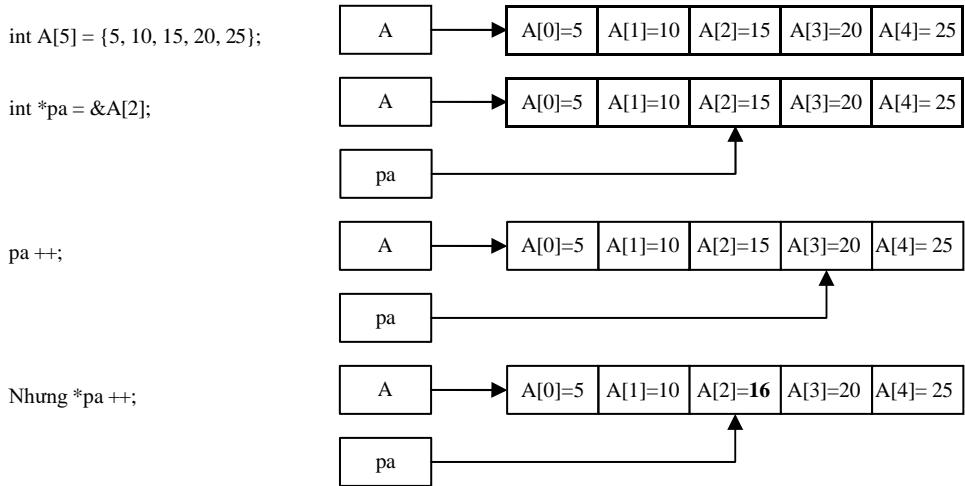
- Hai phép toán pa++ và *pa++ có tác dụng hoàn toàn khác nhau trên mảng, pa++ là thao tác trên con trỏ, tức là trên bộ nhớ, nó sẽ đưa con trỏ pa trỏ đến địa chỉ của phần tử tiếp theo của mảng. *pa++ là phép toán trên giá trị, nó tăng giá trị hiện tại của phần tử mảng lên một đơn vị.

Ví dụ:

```
int A[5] = {5, 10, 15, 20, 25};  
int *pa = &A[2];
```

thì pa++ là tương đương với pa = &A[3] và *pa = 20.

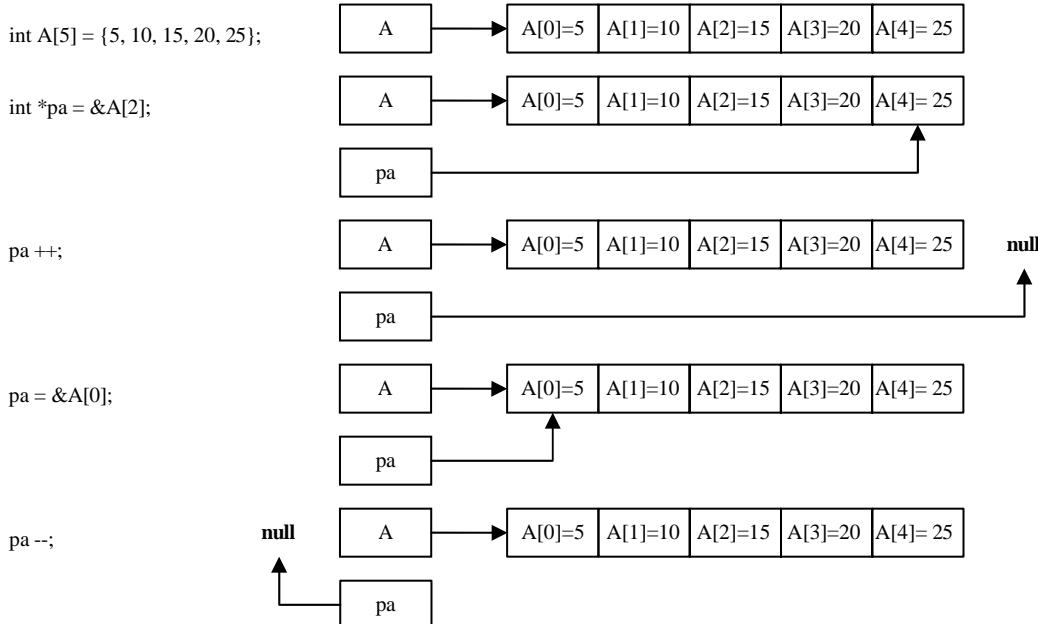
nhưng *pa++ lại tương đương với pa = &A[2] và *pa = 15+1 = 16, A[2] = 16.



- Trong trường hợp:

```
int A[5] = {5, 10, 15, 20, 25};  
int *pa = &A[4];
```

thì phép toán pa++ sẽ đưa con trỏ pa trỏ đến một địa chỉ không xác định. Lý do là A[4] là phần tử cuối của mảng A, nên pa++ sẽ trỏ đến địa chỉ ngay sau địa chỉ của A[4], địa chỉ này nằm ngoài vùng chỉ số của mảng A nên không xác định. Tương tự với trường hợp pa=&A[0], phép toán pa-- cũng đưa pa trỏ đến một địa chỉ không xác định.



- Vì mảng A là con trỏ hằng, cho nên không thể thực hiện các phép toán trên A mà chỉ có thể thực hiện trên các con trỏ trỏ đến A: các phép toán pa++ hoặc pa-- là hợp lệ, nhưng các phép toán A++ hoặc A-- là không hợp lệ.

Chương trình 2.2a minh họa việc cài đặt một thủ tục sắp xếp các phần tử của một mảng theo cách thông thường.

Chương trình 2.2a

```
void SortArray(int A[], int n) {
    int temp;
    for(int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(A[i] > A[j]) {
                temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
}
```

Chương trình 2.2b cài đặt một thủ tục tương tự bằng con trỏ. Hai thủ tục này có chức năng hoàn toàn giống nhau.

Chương trình 2.2b

```
void SortArray(int *A, int n) {
    int temp;
    for(int i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(* (A+i) > * (A+j)) {
                temp = * (A+i);
                * (A+i) = * (A+j);
                * (A+j) = temp;
            }
}
```

Trong chương trình 2.2b, thay vì dùng một mảng, ta dùng một con trỏ để trỏ đến mảng cần sắp xếp. Khi đó, ta có thể dùng các thao tác trên con trỏ thay vì các thao tác trên các phần tử mảng.

2.2.2 Con trỏ và mảng nhiều chiều

Con trỏ và mảng nhiều chiều

Một câu hỏi đặt ra là nếu một ma trận một chiều thì tương đương với một con trỏ, vậy một mảng nhiều chiều thì tương đương với con trỏ như thế nào?

Xét ví dụ:

```
int A[3][3] = {
    {5, 10, 15},
    {20, 25, 30},
    {35, 40, 45}
};
```

Khi đó, địa chỉ của ma trận A chính là địa chỉ của hàng đầu tiên của ma trận A, và cũng là địa chỉ của phần tử đầu tiên của hàng đầu tiên của ma trận A:

- Địa chỉ của ma trận A: $A = A[0] = *(A+0) = &A[0][0];$
- Địa chỉ của hàng thứ nhất: $A[1] = *(A+1) = &A[1][0];$
- Địa chỉ của hàng thứ i: $A[i] = *(A+i) = &A[i][0];$
- Địa chỉ phần tử $\&A[i][j] = (*(A+i)) + j;$
- Giá trị phần tử $A[i][j] = *((*(A+i)) + j);$

Như vậy, một mảng hai chiều có thể thay thế bằng một mảng một chiều các con trỏ cùng kiểu:

```
int A[3][3];
```

có thể thay thế bằng:

```
int (*A)[3];
```

Con trỏ trả tới con trỏ

Vì một mảng hai chiều int A[3][3] có thể thay thế bằng một mảng các con trỏ int (*A)[3]. Hơn nữa, một mảng int A[3] lại có thể thay thế bằng một con trỏ int *A. Do vậy, một mảng hai chiều có thể thay thế bằng một mảng các con trỏ, hoặc một con trỏ trả đến con trỏ. Nghĩa là các cách viết sau là tương đương:

```
int A[3][3];
int (*A)[3]; int
**A;
```

2.3 CON TRỎ HÀM

Mặc dù hàm không phải là một biến cụ thể nên không có một địa chỉ xác định. Nhưng trong khi chạy, mỗi một hàm trong C++ cũng có một vùng nhớ xác định, do vậy, C++ cho phép dùng con trỏ để trả đến hàm. Con trỏ hàm được dùng để truyền tham số có dạng hàm.

Khai báo con trỏ hàm

Con trỏ hàm được khai báo tương tự như khai báo nguyên mẫu hàm thông thường trong C++, ngoại trừ việc có thêm kí hiệu con trỏ “*” trước tên hàm. Cú pháp khai báo con trỏ hàm như sau:

```
<Kiểu dữ liệu trả về> (*<Tên hàm>)(<Các tham số>);
```

Trong đó:

- **Kiểu dữ liệu trả về:** là các kiểu dữ liệu thông thường của C++ hoặc kiểu do người dùng tự định nghĩa.
- **Tên hàm:** tên do người dùng tự định nghĩa, tuân thủ theo quy tắc đặt tên biến trong C++.
- **Các tham số:** có thể có hoặc không (phần trong dấu “[]” là tùy chọn). Nếu có nhiều tham số, mỗi tham số được phân cách nhau bởi dấu phẩy.

Ví dụ khai báo:

```
int (*Calcul)(int a, int b);
```

là khai báo một con trỏ hàm, tên là Calcul, có kiểu int và có hai tham số cũng là kiểu int.

Lưu ý:

- Dấu “()” bao bọc tên hàm là cần thiết để chỉ ra rằng ta đang khai báo một con trỏ hàm. Nếu không có dấu ngoặc đơn này, trình biên dịch sẽ hiểu rằng ta đang khai báo một hàm thông thường và có giá trị trả về là một con trỏ.

Ví dụ, hai khai báo sau là khác nhau hoàn toàn:

```
// Khai báo một con trỏ hàm int
(*Calcul)(int a, int b);
// Khai báo một hàm trả về kiểu con trỏ int
*Calcul(int a, int b);
```

Sử dụng con trỏ hàm

Con trỏ hàm được dùng khi cần gọi một hàm như là tham số của một hàm khác. Khi đó, một hàm được gọi phải có khuôn mẫu giống với con trỏ hàm đã được khai báo.

Ví dụ, với khai báo:

```
int (*Calcul)(int a, int b);
```

thì có thể gọi các hàm có hai tham số kiểu int và trả về cũng kiểu int như sau:

```
int add(int a, int b); int sub(int
a, int b);
```

nhưng không được gọi các hàm khác kiểu tham số hoặc kiểu trả về như sau:

```
int add(float a, int b); int add(int
a);
char* sub(char* a, char* b);
```

Chương trình 2.3 minh họa việc khai báo và sử dụng con trỏ hàm.

Chương trình 2.3

```
#include <ctype.h>
#include <string>

// Hàm có sử dụng con trỏ hàm như tham số
void Display(char[] str, int (*Xtype) (int c)) {
    int index = 0;
    while(str[index] != '\0') {
        cout << (*Xtype) (str[index]); // Sử dụng con trỏ hàm
        index++;
    }
    return;
}

// Hàm main, dùng lời gọi hàm đến con trỏ hàm
void main() {
    char input[500];
    cout << "Enter the string: ";
    cin >> input;
```

```
char reply;
cout << "Display the string in uppercase or lowercase (u,l): ";
cin >> reply;
if(reply == 'l') // Hiển thị theo dạng lowercase
    Display(str, tolower);
else // Hiển thị theo dạng uppercase
    Display(str, toupper);
return;
}
```

Chương trình 2.3 khai báo hàm Display() có sử dụng con trỏ hàm có khuôn mẫu

```
int (*Xtype)(int c);
```

Trong hàm main, con trỏ hàm này được gọi bởi hai thẻ hiện là các hàm tolower() và hàm toupper(). Hai hàm này được khai báo trong thư viện ctype.h với mẫu như sau:

```
int tolower(int c); int
toupper(int c);
```

Hai khuôn mẫu này phù hợp với con trỏ hàm Xtype trong hàm Display() nên lời gọi hàm Display() trong hàm main là hợp lệ.

2.4 CẤP PHÁT BỘ NHỚ ĐỘNG

Xét hai trường hợp sau đây:

- Trường hợp 1, khai báo một con trỏ và gán giá trị cho nó:

```
int *pa = 12;
```

- Trường hợp 2, khai báo con trỏ đến phần tử cuối cùng của mảng rồi tăng thêm một đơn vị cho nó:

```
int A[5] = {5, 10, 15, 20, 25};
int *pa = &A[4]; pa++;
```

Trong cả hai trường hợp, ta đều không biết thực sự con trỏ pa đang trỏ đến địa chỉ nào trong bộ nhớ: trường hợp 1 chỉ ra rằng con trỏ pa đang trỏ tới một địa chỉ không xác định, nhưng lại chứa giá trị là 12 do được gán vào. Trường hợp 2, con trỏ pa đã trỏ đến địa chỉ ngay sau địa chỉ phần tử cuối cùng của mảng A, đó cũng là một địa chỉ không xác định. Các địa chỉ không xác định này là các địa chỉ nằm ở vùng nhớ tự do còn thừa của bộ nhớ. Vùng nhớ này có thể bị chiếm dụng bởi bất kỳ một chương trình nào đang chạy.

Do đó, rất có thể các chương trình khác sẽ chiếm mất các địa chỉ mà con trỏ pa đang trỏ tới. Khi đó, nếu các chương trình thay đổi giá trị của địa chỉ đó, giá trị pa cũng bị thay đổi theo mà ta không thể kiểm soát được. Để tránh các rủi ro có thể gặp phải, C++ yêu cầu phải cấp phát bộ nhớ một cách tường minh cho con trỏ trước khi sử dụng chúng.

2.4.1 Cấp phát bộ nhớ động cho biến

Cấp phát bộ nhớ động

Thao tác cấp phát bộ nhớ cho con trỏ thực chất là gán cho con trỏ một địa chỉ xác định và đưa địa chỉ đó vào vùng đã bị chiếm dụng, các chương trình khác không thể sử dụng địa chỉ đó. Cú pháp cấp phát bộ nhớ cho con trỏ như sau:

```
<tên con trỏ> = new <kiểu con trỏ>;
```

Ví dụ, khai báo:

```
int *pa;  
pa = new int;
```

sẽ cấp phát bộ nhớ hợp lệ cho con trỏ pa.

Lưu ý:

- Ta có thể vừa cấp phát bộ nhớ, vừa khởi tạo giá trị cho con trỏ theo cú pháp sau:

```
int *pa;  
pa = new int(12);
```

sẽ cấp phát cho con trỏ pa một địa chỉ xác định, đồng thời gán giá trị của con trỏ *pa = 12.

Giải phóng bộ nhớ động

Địa chỉ của con trỏ sau khi được cấp phát bởi thao tác **new** sẽ trở thành vùng nhớ đã bị chiếm dụng, các chương trình khác không thể sử dụng vùng nhớ đó ngay cả khi ta không dùng con trỏ nữa. Để tiết kiệm bộ nhớ, ta phải huỷ bỏ vùng nhớ của con trỏ ngay sau khi không dùng đến con trỏ nữa. Cú pháp huỷ bỏ vùng nhớ của con trỏ như sau:

```
delete <tên con trỏ>;
```

Ví dụ:

```
int *pa = new int(12);           // Khai báo con trỏ pa, cấp phát bộ nhớ  
                                // và gán giá trị ban đầu cho pa là 12.  
delete pa;                      // Giải phóng vùng nhớ vừa cấp cho pa.
```

Lưu ý:

- Một con trỏ, sau khi bị giải phóng địa chỉ, vẫn có thể được cấp phát một vùng nhớ mới hoặc trỏ đến một địa chỉ mới:

```
int *pa = new int(12);           // Khai báo con trỏ pa, cấp phát bộ nhớ  
                                // và gán giá trị ban đầu cho pa là 12.  
delete pa;                      // Giải phóng vùng nhớ vừa cấp cho pa. int A[5] =  
                                {5, 10, 15, 20, 25};  
pa = A;                         // Cho pa trỏ đến địa chỉ của mảng A
```

- Nếu có nhiều con trỏ cùng trỏ vào một địa chỉ, thì chỉ cần giải phóng bộ nhớ của một con trỏ, tất cả các con trỏ còn lại cũng bị giải phóng bộ nhớ:

```
int *pa = new int(12);           // *pa = 12  
int *pb = pa;                  // pb trỏ đến cùng địa chỉ pa.  
*pb += 5;                      // *pa = *pb = 17  
delete pa;                     // Giải phóng cả pa lẫn pb
```

- Một con trỏ sau khi cấp phát bộ nhớ động bằng thao tác **new**, cần phải phóng bộ nhớ trước khi trỏ đến một địa chỉ mới hoặc cấp phát bộ nhớ mới:

```
int *pa = new int(12);           // pa được cấp bộ nhớ và *pa = 12  
*pa = new int(15);             // pa trỏ đến địa chỉ khác và *pa = 15.  
                                // địa chỉ cũ của pa vẫn bị coi là bận
```

2.4.2 Cấp phát bộ nhớ cho mảng động một chiều

Cấp phát bộ nhớ cho mảng động một chiều

Mảng một chiều được coi là tương ứng với một con trỏ cùng kiểu. Tuy nhiên, cú pháp cấp phát bộ nhớ cho mảng động một chiều là khác với cú pháp cấp phát bộ nhớ cho con trỏ thông thường:

<Tên con trỏ> = new <Kiểu con trỏ>[<Độ dài mảng>];

Ví dụ:

- Tên con trỏ:** tên do người dùng đặt, tuân thủ theo quy tắc đặt tên biến của C++.
- Kiểu con trỏ:** Kiểu dữ liệu cơ bản của C++ hoặc là kiểu do người dùng tự định nghĩa.
- Độ dài mảng:** số lượng các phần tử cần cấp phát bộ nhớ của mảng.

Ví dụ:

```
int *A = new int[5];
```

sẽ khai báo một mảng A có 5 phần tử kiểu int được cấp phát bộ nhớ động.

Lưu ý:

- Khi cấp phát bộ nhớ cho con trỏ có khởi tạo thông thường, ta dùng dấu “()”, khi cấp phát bộ nhớ cho mảng, ta dùng dấu “[]”. Hai lệnh cấp phát sau là hoàn toàn khác nhau:

```
// Cấp phát bộ nhớ và khởi tạo cho một con trỏ int int *A = new  
int(5);  
// Cấp phát bộ nhớ cho một mảng 5 phần tử kiểu int int *A = new  
int[5];
```

Giải phóng bộ nhớ của mảng động một chiều

Để giải phóng vùng nhớ đã được cấp phát cho một mảng động, ta dùng cú pháp sau:

```
delete [] <tên con trỏ>;
```

Ví dụ:

```
// Cấp phát bộ nhớ cho một mảng có 5 phần tử kiểu int int *A = new  
int[5];  
// Giải phóng vùng nhớ do mảng A đang chiếm giữ. delete [] A;
```

Chương trình 2.4 minh họa hai thủ tục khởi tạo và giải phóng một mảng động một chiều.

Chương trình 2.4

```
void InitArray(int *A, int length) {  
    A = new int[length];  
    for(int i=0; i<length; i++)
```

```
        A[i] = 0;
    return;
}

void DeleteArray(int *A) {
    delete [] A;
    return;
}
```

2.4.3 Cấp phát bộ nhớ cho mảng động nhiều chiều

Cấp phát bộ nhớ cho mảng động nhiều chiều

Một mảng hai chiều là một con trỏ đến một con trỏ. Do vậy, ta phải cấp phát bộ nhớ theo từng chiều theo cú pháp cấp phát bộ nhớ cho mảng động một chiều.

Ví dụ:

```
int **A;
const int length = 10;
A = new int*[length];           // Cấp phát bộ nhớ cho số dòng của ma trận A for(int i=0;
i<length; i++)
    // Cấp phát bộ nhớ cho các phần tử của mỗi dòng A[i] = new
    int[length];
```

Sẽ cấp phát bộ nhớ cho một mảng động hai chiều, tương đương với một ma trận có kích thước 10*10.

Lưu ý:

- Trong lệnh cấp phát `A = new int*[length]`, cần phải có dấu “*” để chỉ ra rằng cần cấp phát bộ nhớ cho một mảng các phần tử có kiểu là con trỏ `int (int*)`, khác với kiểu `int` bình thường.

Giải phóng bộ nhớ của mảng động nhiều chiều

Ngược lại với khi cấp phát, ta phải giải phóng lần lượt bộ nhớ cho con trỏ tương ứng với cột và hàng của mảng động.

Ví dụ:

```
int **A;
...;                         // cấp phát bộ nhớ
...
for(int i=0; i<length; i++)
    delete [] A[i];          // Giải phóng bộ nhớ cho mỗi dòng delete [] A; //
```

Giải phóng bộ nhớ cho mảng các dòng

Sẽ giải phóng bộ nhớ cho một mảng động hai chiều.

Chương trình 2.5 minh họa việc dùng mảng động hai chiều để tính tổng của hai ma trận.

Chương trình 2.5

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

/* Khai báo nguyên mẫu hàm */
void InitArray(int **A, int row, int colum);
void AddArray(int **A, int **B, int row, int colum); void
DisplayArray(int **A, int row, int colum); void DeleteArray(int
**A, int row);

void InitArray(int **A, int row, int colum){ A = new
int*[row];
for(int i=0; i<row; i++){ A[i] = new
int[colum];
for(int j=0; j<colum; j++){
cout << "Phan tu [" << i << "," << j << "] = ";
cin >>
A[i][j];
}
return;
}

void AddArray(int **A, int **B, int row, int colum){ for(int i=0; i<row;
i++)
for(int j=0; j<colum; j++) A[i][j] +=
B[i][j];
return;
}

void DisplayArray(int **A, int row, int colum){ for(int i=0; i<row;
i++){
for(int j=0; j<colum; j++) cout <<
A[i][j] << " ";
cout << endl; // Xuống dòng return;
}

void DeleteArray(int **A, int row){ for(int i=0; i<row;
i++)
delete [] A[i]; delete []
A;
return;
}
```

```
void main() {
    clrscr();
    int **A, **B, row, colum;
    cout << "So dong: ";
    cin >> row;
    cout << "So cot: ";
    cin >> colum;

    /* Khởi tạo các ma trận */
    cout << "Khoi tao mang A:" << endl;
    InitArray(A, row, colum);

    cout << "Khoi tao mang B:" << endl;
    InitArray(B, row, colum);

    // Cộng hai ma trận
    AddArray(A, B, row, colum);

    // Hiển thị ma trận kết quả
    cout << "Tong hai mang A va mang B:" << endl;
    DisplayArray(A, row, colum);

    // Giải phóng bộ nhớ
    DeleteArray(A, row);
    DeleteArray(B, row);
    return;
}
```

TỔNG KẾT CHƯƠNG 2

Nội dung chương 2 đã trình bày các vấn đề liên quan đến việc khai báo và sử dụng con trỏ và mảng trong ngôn ngữ C++:

- Con trỏ là một kiểu biến đặc biệt, nó trỏ đến địa chỉ của một biến khác. Có hai cách truy nhập đến con trỏ là truy nhập đến địa chỉ hoặc truy nhập đến giá trị của địa chỉ mà con trỏ trỏ đến.
- Con trỏ có thể tham gia vào các phép toán như các biến thông thường bằng phép lấy giá trị.
- Một con trỏ có sự tương ứng với một mảng một chiều có cùng kiểu.
- Một ma trận hai chiều có thể thay thế bằng một mảng các con trỏ hoặc một con trỏ trỏ đến con trỏ.
- Một con trỏ có thể trỏ đến một hàm, khi đó, nó được dùng để gọi một hàm như là một tham số cho hàm khác.

- Một con trỏ cần phải trỏ vào một địa chỉ xác định hoặc phải được cấp phát bộ nhớ qua phép toán **new** và giải phóng bộ nhớ sau khi dùng bằng thao tác **delete**.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 2

1. Trong các khai báo con trỏ sau, những khai báo nào là đúng:

- a. int A*;
- b. *int A;
- c. int* A, B;
- d. int* A, *B;
- e. int *A, *B;

2. Với khai báo:

```
int a = 12; int  
*pa;
```

Các phép gán nào sau đây là hợp lệ:

- a. pa = &a;
- b. pa = a;
- c. *pa = &a;
- d. *pa = a;

3. Với khai báo:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A+2;
```

Khi đó, *pa = ?

- a. 10
- b. 20
- c. 30
- d. 40
- e. 50

4. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
*pa += 2;
```

Khi đó, *pa = ?

- a. 10
- b. 12
- c. 30
- d. 32

5. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;
```

pa += 2;

Khi đó, *pa = ?

- a. 10
- b. 12
- c. 30
- d. 32

6. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A; pa  
+= 2;
```

Khi đó, pa = ?

- a. &A[0]
- b. A[2]
- c. &A[2]
- d. Không xác định

7. Với đoạn chương trình:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A; pa -  
= 2;
```

Khi đó, pa = ?

- a. &A[0]
- b. &A[2]
- c. &A[4]
- d. Không xác định

8. Với đoạn chương trình:

```
int A[3][3] = {  
    {10, 20, 30},  
    {40, 50, 60},  
    {70, 80, 90}  
};  
  
int *pa;
```

Khi đó, để có được kết quả *pa = 50, các lệnh nào sau đây là đúng?

- a. pa = A + 4;
- b. pa = (*(A+1)) + 1;
- c. pa = &A[1][1];
- d. pa = *((*(A+1)) + 1);

9. Giả sử ta khai báo một hàm có sử dụng con trỏ hàm với khuôn mẫu như sau:

```
int Calcul(int a, int b, int (*Xcalcul)(int x, int y)){}
```

Và ta có cài đặt một số hàm như sau:

```
int add(int a, int b); void  
cal(int a, int b); int square(int  
a);
```

Khi đó, lời gọi hàm nào sau đây là đúng:

- a. Calcul(5, 10, add);
- b. Calcul(5, 10, add(2, 3));
- c. Calcul(5, 10, cal);
- d. Calcul(5, 10, square);

10. Ta muốn cấp phát bộ nhớ cho một con trỏ kiểu int và khởi đầu giá trị cho nó là 20. Lệnh nào sau đây là đúng:

- a. int *pa = 20;
- b. int *pa = new int{20};
- c. int *pa = new int(20);
- d. int *pa = new int[20];

11. Ta muốn cấp phát bộ nhớ cho một mảng động kiểu int có chiều dài là 20. Lệnh nào sau đây là đúng:

- a. int *pa = 20;
- b. int *pa = new int{20};
- c. int *pa = new int(20);
- d. int *pa = new int[20];

12. Xét đoạn chương trình sau:

```
int A[5] = {10, 20, 30, 40, 50};  
int *pa = A;  
pa = new int(2);
```

Khi đó, *pa = ?

- a. 10
- b. 30
- c. 2
- d. Không xác định

13. Xét đoạn chương trình sau:

```
1>     int A[5] = {10, 20, 30, 40, 50};  
2>     int *pa = A;  
3>     pa += 15;  
4>     delete pa;
```

Đoạn chương trình trên có lỗi ở dòng nào?

- a. 1
- b. 2
- c. 3
- d. 4

14. Viết chương trình thực hiện các phép toán cộng, trừ, nhân, chia trên đa thức. Các đa thức được biểu diễn bằng mảng động một chiều. Độ của đa thức và các hệ số tương ứng được nhập từ bàn phím.
15. Viết chương trình thực hiện các phép toán cộng, trừ, nhân hai ma trận kích thước $m \times n$. Các ma trận được biểu diễn bằng mảng động hai chiều. Giá trị kích cỡ ma trận (m, n) và giá trị các phần tử của ma trận được nhập từ bàn phím.

CHƯƠNG 3

KIỂU DỮ LIỆU CẤU TRÚC

Nội dung chương này tập trung trình bày các vấn đề liên quan đến kiểu dữ liệu có cấu trúc trong C++:

- Định nghĩa một cấu trúc
- Sử dụng một cấu trúc bằng các phép toán cơ bản trên cấu trúc
- Con trỏ cấu trúc, khai báo và sử dụng con trỏ cấu trúc
- Mảng các cấu trúc, khai báo và sử dụng mảng các cấu trúc
- Một số kiểu dữ liệu trừu tượng khác như ngăn xếp, hàng đợi, danh sách liên kết.

3.1 ĐỊNH NGHĨA CẤU TRÚC

Kiểu dữ liệu có cấu trúc được dùng khi ta cần nhóm một số biến dữ liệu luôn đi kèm với nhau. Khi đó, việc xử lý trên một nhóm các biến được thực hiện như trên các biến cơ bản thông thường.

3.1.1 Khai báo cấu trúc

Trong C++, một cấu trúc do người dùng tự định nghĩa được khai báo thông qua từ khoá **struct**:

```
struct <Tên cấu trúc>{  
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;  
    <Kiểu dữ liệu 2> <Tên thuộc tính 2>;  
    ...  
    <Kiểu dữ liệu n> <Tên thuộc tính n>;  
};
```

Trong đó:

- **struct**: là tên từ khoá để khai báo một cấu trúc, bắt buộc phải có khi định nghĩa cấu trúc.
- **Tên cấu trúc**: là tên do người dùng tự định nghĩa, tuân thủ theo quy tắc đặt tên biến trong C++. Tên này sẽ trở thành tên của kiểu dữ liệu có cấu trúc tương ứng.
- **Thuộc tính**: mỗi thuộc tính của cấu trúc được khai báo như khai báo một biến thuộc kiểu dữ liệu thông thường, gồm có kiểu dữ liệu và tên biến tương ứng. Mỗi khai báo thuộc tính phải kết thúc bằng dấu chấm phẩy ";" như một câu lệnh C++ thông thường.

Ví dụ, để quản lí nhân viên của một công ty, khi xử lý thông tin về mỗi nhân viên, ta luôn phải xử lý các thông tin liên quan như:

- Tên
- Tuổi
- Chức vụ
- Lương

Do đó, ta sẽ dùng cấu trúc để lưu giữ thông tin về mỗi nhân viên bằng cách định nghĩa một cấu trúc có tên là Employee với các thuộc tính như sau:

```
struct Employee{
    char name[20];           // Tên nhân viên int
    age;                     // Tuổi nhân viên
    char role[20];           // Chức vụ của nhân viên float
    salary;                  // Lương của nhân viên
};
```

Lưu ý:

- Cấu trúc chỉ cần định nghĩa một lần trong chương trình và có thể được khai báo biến cấu trúc nhiều lần. Khi cấu trúc đã được định nghĩa, việc khai báo biến ở lần khác trong chương trình được thực hiện như khai báo biến thông thường:

<Tên cấu trúc> <tên biến 1>, <tên biến 2>;

Ví dụ, sau khi đã định nghĩa cấu trúc Employee, muốn có biến myEmployee, ta khai báo như sau:

```
Employee myEmployee;
```

3.1.2 Cấu trúc lồng nhau

Các cấu trúc có thể được định nghĩa lồng nhau khi một thuộc tính của một cấu trúc cũng cần có kiểu là một cấu trúc khác. Khi đó, việc định nghĩa cấu trúc cha được thực hiện như một cấu trúc bình thường, với khai báo về thuộc tính đó là một cấu trúc con:

```
struct <Tên cấu trúc cha>{
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;
    // Có kiểu cấu trúc
    cKiểu cấu trúc conS cTên thuộc tính 2S;
    ...
    <Kiểu dữ liệu n> <Tên thuộc tính n>;
};
```

Ví dụ, với kiểu cấu trúc Employee, ta không quan tâm đến tuổi nhân viên nữa, mà quan tâm đến ngày sinh của nhân viên. Vì ngày sinh cần có các thông tin luôn đi với nhau là ngày sinh, tháng sinh, năm sinh. Do đó, ta định nghĩa một kiểu cấu trúc con cho kiểu ngày sinh:

```
struct Date{
    int day; int
    month; int
    year;
};
```

khi đó, cấu trúc Employee trở thành:

```
struct Employee{
    char name[20];           // Tên nhân viên
    Date birthDay;           // Ngày sinh của nhân viên char
    role[20];                 // Chức vụ của nhân viên
    float salary;              // Lương của nhân viên
};
```

Lưu ý:

- Trong định nghĩa các cấu trúc lồng nhau, cấu trúc con phải được định nghĩa trước cấu trúc cha để đảm bảo các kiểu dữ liệu của các thuộc tính của cấu trúc cha là tường minh tại thời điểm nó được định nghĩa.

3.1.3 Định nghĩa cấu trúc với từ khoá **typedef**

Để tránh phải dùng từ khoá **struct** mỗi khi khai báo biến cấu trúc, ta có thể dùng từ khoá **typedef** khi định nghĩa cấu trúc:

```
typedef struct {
    <Kiểu dữ liệu 1> <Tên thuộc tính 1>;
    <Kiểu dữ liệu 2> <Tên thuộc tính 2>;
    ...
    <Kiểu dữ liệu n> <Tên thuộc tính n>;
} <Tên kiểu dữ liệu cấu trúc>;
```

Trong đó:

- Tên kiểu dữ liệu cấu trúc:** là tên kiểu dữ liệu của cấu trúc vừa định nghĩa. Tên này sẽ được dùng như một kiểu dữ liệu thông thường khi khai báo biến cấu trúc.

Ví dụ, muốn có kiểu dữ liệu có cấu trúc nhân viên, có tên là Employee, ta dùng từ khoá **typedef** để định nghĩa cấu trúc như sau:

```
typedef struct {
    char name[20];           // Tên nhân viên int
    age;                    // Tuổi nhân viên
    char role[20];          // Chức vụ của nhân viên float
    salary;                 // Lương của nhân viên
} Employee;
```

Khi đó, muốn có hai biến là myEmployee1 và myEmployee2 có kiểu cấu trúc Employee, ta chỉ cần khai báo như sau mà không cần từ khoá **struct**:

```
Employee myEmployee1, myEmployee2;
```

Trong ví dụ khai báo lồng cấu trúc Employee, dùng từ khoá **typedef** cho kiểu Date:

```
typedef struct {
    int day; int
    month; int
    year;
} Date;
```

cấu trúc Employee trở thành:

```
typedef struct {
    char name[20];           // Tên nhân viên
    Date birthDay;          // Ngày sinh của nhân viên char
    role[20];                // Chức vụ của nhân viên float
    salary;                  // Lương của nhân viên
} Employee;
```

Lưu ý:

- Khi không dùng từ khoá **typedef**, tên cấu trúc (nằm sau từ khoá struct) được dùng để khai báo biến. Trong khi đó, khi có từ khoá **typedef**, tên kiểu dữ liệu cấu trúc (dòng cuối cùng trong định nghĩa) mới được dùng để khai báo biến.
- Khi dùng từ khoá **typedef** thì không thể khai báo biến đồng thời với định nghĩa cấu trúc.

3.2 THAO TÁC TRÊN CẤU TRÚC

Các thao tác trên cấu trúc bao gồm:

- Khai báo và khởi tạo giá trị ban đầu cho biến cấu trúc
- Truy nhập đến các thuộc tính của cấu trúc

3.2.1 Khởi tạo giá trị ban đầu cho cấu trúc

Khởi tạo biến có cấu trúc đơn

Biến cấu trúc được khai báo theo các cách sau:

<Tên kiểu dữ liệu cấu trúc> <tên biến>;

Ngoài ra, ta có thể khởi tạo các giá trị cho các thuộc tính của cấu trúc ngay khi khai báo bằng các cú pháp sau:

```
<Tên kiểu dữ liệu cấu trúc> <tên biến> = {  
    <giá trị thuộc tính 1>,  
    <giá trị thuộc tính 2>,  
    ...  
    <giá trị thuộc tính n>  
};
```

Trong đó:

- **Giá trị thuộc tính:** là giá trị khởi đầu cho mỗi thuộc tính, có kiểu phù hợp với kiểu dữ liệu của thuộc tính. Mỗi giá trị của thuộc tính được phân cách bằng dấu phẩy “,”.

Ví dụ, với định nghĩa cấu trúc:

```
typedef struct {  
    char name[20];           // Tên nhân viên int  
    age;                   // Tuổi nhân viên  
    char role[20];          // Chức vụ của nhân viên float  
    salary;                // Lương của nhân viên  
} Employee;
```

thì có thể khai báo và khởi tạo cho một biến như sau:

```
Employee myEmployee1 = {  
    "Nguyen Van A", 27,  
    "Nhan vien", 300f
```

};

Khởi tạo các biến có cấu trúc lồng nhau

Trong trường hợp các cấu trúc lồng nhau, phép khởi tạo cũng thực hiện như thông thường với phép khởi tạo cho tất cả các cấu trúc con.

Ví dụ với khai báo cấu trúc như sau:

```
typedef struct {  
    int day; int  
    month; int  
    year;  
} Date;
```

và:

```
typedef struct {  
    char name[20];           // Tên nhân viên  
    Date birthDay;          // Ngày sinh của nhân viên char  
    role[20];                // Chức vụ của nhân viên float  
    salary;                  // Lương của nhân viên  
} Employee;
```

Thì khai báo và khởi tạo một biến có kiểu Employee có thể thực hiện như sau:

```
Employee myEmployee1 = {"Nguyen  
Van A",  
{15, 05, 1980},           // Khởi tạo cấu trúc con "Nhan  
vien",  
300f  
};
```

3.2.2 Truy nhập đến thuộc tính của cấu trúc

Việc truy nhập đến thuộc tính của cấu trúc được thực hiện bằng cú pháp:

<Tên biến cấu trúc>.<tên thuộc tính>

Ví dụ, với một biến cấu trúc kiểu Employee đơn:

```
Employee myEmployee1 = {  
    "Nguyen Van A", 27,  
    "Nhan vien", 300f  
};
```

ta có thể truy xuất như sau:

```
cout << myEmployee1.name;           // hiển thị ra "Nguyen Van A"  
myEmployee1.age += 1;                 // Tăng số tuổi lên 1
```

Đối với kiểu cấu trúc lồng nhau, phép truy nhập đến thuộc tính được thực hiện lần lượt từ cấu trúc cha đến cấu trúc con.

Ví dụ, với một biến cấu trúc kiểu Employee lồng nhau:

```
Employee myEmployee1 = {  
    "Nguyen Van A",  
    {15, 05, 1980},  
    "Nhan vien", 300f  
};
```

ta có thể truy xuất như sau:

```
cout << myEmployee1.name; // hiển thị ra "Nguyen Van A"  
myEmployee1.birthDay.day = 16; // Sửa lại ngày sinh thành 16  
myEmployee1.birthDay.month = 07; // Sửa lại tháng sinh thành 07
```

Chương trình 3.1a minh họa việc tạo lập và sử dụng cấu trúc Employee đơn, không dùng từ khoá **typedef**.

Chương trình 3.1a

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
struct Employee{  
    char name[20]; // Tên nhân viên  
    int age; // Tuổi nhân viên  
    char role[20]; // Chức vụ của nhân viên  
    float salary; // Lương của nhân viên  
};  
  
/* Khai báo khuôn mẫu hàm */  
void Display(Employee myEmployee);  
  
void Display(Employee myEmployee){  
    cout << "Name: " << myEmployee.name << endl;  
    cout << "Age: " << myEmployee.age << endl;  
    cout << "Role: " << myEmployee.role << endl;  
    cout << "Salary: " << myEmployee.salary << endl;  
    return;  
}  
  
void main(){  
    clrscr();  
    // Hiển thị giá trị mặc định  
    Employee myEmployee =
```

Chương 3: Kiểu dữ liệu cấu trúc

```
{ "Nguyen Van A", 27, "Nhan vien", 300f};  
cout << "Thông tin mặc định:" << endl;  
Display(myEmployee);  
  
// Thay đổi giá trị cho các thuộc tính  
cout << "Name: ";  
cin >> myEmployee.name;  
cout << "Age: ";  
cin >> myEmployee.age;  
cout << "Role: ";  
cin >> myEmployee.role;  
cout << "Salary: ";  
cin >> myEmployee.salary;  
  
cout << "Thông tin sau khi thay đổi:" << endl;  
Display(myEmployee);  
return;  
}
```

Chương trình 3.1b minh họa việc tạo lập và sử dụng cấu trúc Employee lồng nhau, có dùng từ khoá **typedef**.

Chương trình 3.1b

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
typedef struct {  
    int day;  
    int month;  
    int year;  
} Date;  
  
typedef struct {  
    char name[20];      // Tên nhân viên  
    Date birthDay;     // Ngày sinh của nhân viên  
    char role[20];      // Chức vụ của nhân viên  
    float salary;       // Lương của nhân viên  
} Employee;  
  
/* Khai báo khuôn mẫu hàm */  
void Display(Employee myEmployee);
```

```
void Display(Employee myEmployee){  
    cout << "Name: " << myEmployee.name << endl;  
    cout << "Birth day: " << myEmployee.birthDay.day << "/"  
        << myEmployee.birthDay.month << "/"  
        << myEmployee.birthDay.year << endl; cout << "Role:  
    " << myEmployee.role << endl;  
    cout << "Salary: " << myEmployee.salary << endl; return;  
}  
  
void main(){  
    clrscr();  
    // Hiển thị giá trị mặc định Employee  
    myEmployee =  
        {"Nguyen Van A", {15, 5, 1980}, "Nhan vien", 300f}; cout <<  
    "Thông tin mặc định:" << endl; Display(myEmployee);  
  
    // Thay đổi giá trị cho các thuộc tính cout << "Name:  
    ";  
    cin >> myEmployee.name; cout <<  
    "Day of birth: ";  
    cin >> myEmployee.birthDay.day; cout <<  
    "Month of birth: ";  
    cin >> myEmployee.birthDay.month; cout <<  
    "Year of birth: ";  
    cin >> myEmployee.birthDay.year; cout <<  
    "Role: ";  
    cin >> myEmployee.role; cout  
    << "Salary: ";  
    cin >> myEmployee.salary;  
  
    cout << "Thông tin sau khi thay đổi:" << endl;  
    Display(myEmployee);  
    return;  
}
```

3.3 CON TRỎ CẤU TRÚC VÀ MẢNG CẤU TRÚC

3.3.1 Con trả cấu trúc

Con trả cấu trúc là một con trả trỏ đến địa chỉ của một biến có kiểu cấu trúc. Cách khai báo và sử dụng con trả cấu trúc được thực hiện như con trả thông thường.

Khai báo con trả cấu trúc

Con trả cấu trúc được khai báo theo cú pháp:

<Tên kiểu cấu trúc> *<Tên biến>;

Ví dụ, với kiểu khai báo cấu trúc:

```
typedef struct {  
    int day; int  
    month; int  
    year;  
} Date;
```

và:

```
typedef struct {  
    char name[20];           // Tên nhân viên  
    Date birthDay;          // Ngày sinh của nhân viên char  
    role[20];                // Chức vụ của nhân viên float  
    salary;                  // Lương của nhân viên  
} Employee;
```

thì ta có thể khai báo một con trả cấu trúc như sau:

```
Employee *ptrEmployee;
```

Lưu ý:

- Cũng như khai báo con trả thông thường, dấu con trả “*” có thể nằm ngay trước tên biến hoặc nằm ngay sau tên kiểu cấu trúc.

Cũng giống con trả thông thường, con trả cấu trúc được sử dụng khi:

- Cho nó trả đến địa chỉ của một biến cấu trúc
- Cấp phát cho nó một vùng nhớ xác định.

Gán địa chỉ cho con trả cấu trúc

Một con trả cấu trúc có thể trả đến địa chỉ của một biến cấu trúc có cùng kiểu thông qua phép gán:

<Tên biến con trả> = &<Tên biến thường>;

Ví dụ, khai báo và phép gán:

```
Employee *ptrEmployee, myEmployee; ptrEmployee =  
&myEmployee;
```

sẽ đưa con trả ptrEmployee trả đến địa chỉ của biến cấu trúc myEmployee.

Cấp phát bộ nhớ động cho con trỏ cấu trúc

Trong trường hợp ta muốn tạo ra một con trỏ cấu trúc mới, không trỏ vào một biến cấu trúc có sẵn nào, để sử dụng con trỏ mới này, ta phải cấp phát vùng nhớ cho nó. Cú pháp cấp phát vùng nhớ cho con trỏ cấu trúc:

```
<Tên biến con trỏ> = new <Kiểu cấu trúc>;
```

Ví dụ, cấu trúc Employee được khai báo bằng từ khoá **typedef**, ta có thể cấp phát vùng nhớ cho con trỏ cấu trúc như sau:

```
Employee *ptrEmployee; ptrEmployee =  
    new Employee;
```

hoặc cấp phát ngay khi khai báo:

```
Employee *ptrEmployee = new Employee;
```

Sau khi cấp phát vùng nhớ cho con trỏ bằng thao tác **new**, khi con trỏ không được dùng nữa, hoặc cần trỏ sang một địa chỉ khác, ta phải giải phóng vùng nhớ vừa được cấp phát cho con trỏ bằng thao tác:

```
delete <Tên biến con trỏ>;
```

Ví dụ:

```
Employee *ptrEmployee = new Employee;  
...  
// Thực hiện các thao tác trên con trỏ  
...  
delete ptrEmployee;
```

Lưu ý:

- Thao tác **delete** chỉ được thực hiện đối với con trỏ mà trước đó, nó được cấp phát bộ nhớ động thông qua thao tác **new**:

```
Employee *ptrEmployee = new Employee; delete  
ptrEmployee; //đúng
```

mà không thể thực hiện với con trỏ chỉ trỏ đến địa chỉ của một biến cấu trúc khác:

```
Employee *ptrEmployee, myEmployee; ptrEmployee =  
&myEmployee;  
delete ptrEmployee; //lỗi
```

Truy nhập thuộc tính của con trỏ cấu trúc

Thuộc tính của con trỏ cấu trúc có thể được truy nhập thông qua hai cách:

Cách 1:

```
<Tên biến con trỏ> -> <Tên thuộc tính>;
```

Cách 2:

```
(*<Tên biến con trỏ>).<Tên thuộc tính>;
```

Ví dụ, thuộc tính tên nhân viên của cấu trúc Employee có thể được truy nhập thông qua hai cách:

```
Employee *ptrEmployee = new Employee; cin >>
```

```
ptrEmployee -> name;
```

hoặc:

```
cin >> (*ptrEmployee).name;
```

Lưu ý:

- Trong cách truy nhập thứ hai, phải có dấu ngoặc đơn “()” quanh tên con trỏ vì phép toán truy nhập thuộc tính “.” có độ ưu tiên cao hơn phép toán lấy giá trị con trỏ “*”.
- Thông thường, ta dùng cách thứ nhất cho đơn giản và thuận tiện.

Chương trình 3.2 cài đặt việc khởi tạo và hiển thị nội dung của một con trỏ cấu trúc.

Chương trình 3.2

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

typedef struct {
    int day;
    int month;
    int year;
} Date;

typedef struct {
    char name[20];      // Tên nhân viên
    Date birthDay;      // Ngày sinh của nhân viên
    char role[20];       // Chức vụ của nhân viên
    float salary;        // Lương của nhân viên
} Employee;

/* Khai báo khuôn mẫu hàm */
void InitStruct(Employee *myEmployee);
void Display(Employee *myEmployee);

void InitStruct(Employee *myEmployee) {
    myEmployee = new Employee;
    cout << "Name: ";
    cin >> myEmployee->name;
    cout << "Day of birth: ";
    cin >> myEmployee->birthDay.day;
    cout << "Month of birth: ";
    cin >> myEmployee->birthDay.month;
    cout << "Year of birth: ";
    cin >> myEmployee->birthDay.year;
    cout << "Role: ";
    cin >> myEmployee->role;
    cout << "Salary: ";
```

```
    cin >> myEmployee->salary;
}

void Display(Employee myEmployee){
    cout << "Name: " << myEmployee->name << endl;
    cout << "Birth day: " << myEmployee->birthDay.day << "/"
        << myEmployee->birthDay.month << "/"
        << myEmployee->birthDay.year << endl;
    cout << "Role: " << myEmployee->role << endl;
    cout << "Salary: " << myEmployee->salary << endl;
    return;
}

void main(){
    clrscr();
    Employee *myEmployee;
    InitStruct(myEmployee);
    Display(myEmployee);
    return;
}
```

3.3.2 Mảng cấu trúc

Khi cần xử lí nhiều đối tượng có dùng kiểu dữ liệu cấu trúc, ta có thể sử dụng mảng các cấu trúc. Vì một mảng một chiều là tương đương với một con trỏ có cùng kiểu. Do đó, có thể khai báo mảng theo hai cách: Khai báo mảng tĩnh như thông thường hoặc khai báo mảng động thông qua con trỏ.

Khai báo mảng tĩnh các cấu trúc

Khai báo mảng tĩnh các cấu trúc theo cú pháp:

<Tên kiểu cấu trúc> <Tên biến mảng>[<Số phần tử mảng>];

Ví dụ:

Employee employees[10];

là khai báo một mảng tên là employees gồm 10 phần tử có kiểu là cấu trúc Employee.

Khai báo mảng động các cấu trúc

Khai báo một mảng động các cấu trúc hoàn toàn tương tự khai báo một con trỏ cấu trúc cùng kiểu:

<Tên kiểu cấu trúc> *<Tên biến>;

Ví dụ, khai báo:

Employee *employees;

vừa có thể coi là khai báo một con trỏ thông thường có cấu trúc Employee, vừa có thể coi là khai báo một mảng động các cấu trúc có kiểu cấu trúc Employee.

Chương 3: Kiểu dữ liệu cấu trúc

Tuy nhiên, cách cấp phát bộ nhớ động cho mảng các cấu trúc khác với một con trỏ. Đây là cách để chương trình nhận biết ta đang dùng một con trỏ cấu trúc hay một mảng động có cấu trúc. Cú pháp cấp phát bộ nhớ cho mảng động như sau:

<Tên biến mảng> = new <Kiểu cấu trúc>[<Số lượng phần tử>];

Ví dụ, khai báo:

Employee *employees = new Employee[10];

Sẽ cấp phát bộ nhớ cho một mảng động employees có 10 phần tử kiểu cấu trúc Employee.

Truy nhập đến phần tử của mảng cấu trúc

Việc truy nhập đến các phần tử của mảng cấu trúc được thực hiện như truy cập đến phần tử của mảng thông thường. Ví dụ muốn truy nhập đến thuộc tính tên nhân viên phần tử nhân viên thứ i trong mảng cấu trúc, ta viết như sau:

Employee *employees = new Employee[10]; employees[i].name;

Chương trình 3.3 cài đặt việc khởi tạo một mảng các nhân viên của một phòng trong một công ty. Sau đó, chương trình sẽ tìm và in ra thông tin về nhân viên có lương cao nhất và nhân viên có lương thấp nhất trong phòng.

Chương trình 3.3

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

typedef struct {
    int day;
    int month;
    int year;
} Date;

typedef struct {
    char name[20];      // Tên nhân viên
    Date birthDay;      // Ngày sinh của nhân viên
    char role[20];       // Chức vụ của nhân viên
    float salary;        // Lương của nhân viên
} Employee;

/* Khai báo khuôn mẫu hàm */
void InitArray(Employee *myEmployee, int length);
Employee searchSalaryMax(Employee *myEmployee, int length);
Employee searchSalaryMin(Employee *myEmployee, int length);
void Display(Employee myEmployee);
```

Chương 3: Kiểu dữ liệu cấu trúc

```
void InitArray(Employee *myEmployee, int length){ myEmployee =  
    new Employee[length];  
    for(int i=0; i<length; i++){  
        cout << "Nhan vien thu " << i << endl; cout <<  
        "Name: ";  
        cin >> myEmployee[i].name; cout <<  
        "Day of birth: ";  
        cin >> myEmployee[i].birthDay.day; cout <<  
        "Month of birth: ";  
        cin >> myEmployee[i].birthDay.month; cout <<  
        "Year of birth: ";  
        cin >> myEmployee[i].birthDay.year; cout << "Role:  
        ";  
        cin >> myEmployee[i].role; cout <<  
        "Salary: ";  
        cin >> myEmployee[i].salary;  
    }  
    return;  
}
```

```
Employee searchSalaryMax(Employee *myEmployee, int length){ int index = 0;  
    int maxSalary = myEmployee[0].salary; for(int i=1;  
    i<length; i++)  
        if(myEmployee[i].salary > maxSalary){ maxSalary =  
            myEmployee[i].salary; index = i;  
        }  
    return myEmployee[index];  
}
```

```
Employee searchSalaryMin(Employee *myEmployee, int length){ int index = 0;  
    int minSalary = myEmployee[0].salary; for(int i=1;  
    i<length; i++)  
        if(myEmployee[i].salary < minSalary){ minSalary =  
            myEmployee[i].salary; index = i;  
        }  
    return myEmployee[index];  
}
```

```
void Display(Employee myEmployee){
```

Chương 3: Kiểu dữ liệu cấu trúc

```
cout << "Name: " << myEmployee.name << endl;
cout << "Birth day: " << myEmployee.birthDay.day << "/"
    << myEmployee.birthDay.month << "/"
    << myEmployee.birthDay.year << endl;
cout << "Role: " << myEmployee.role << endl;
cout << "Salary: " << myEmployee.salary << endl;
return;
}

void main(){
    clrscr();
    Employee *myEmployee, tmpEmployee;
    int length = 0;
    cout << "So luong nhan vien: ";
    cin >> length;

    // Khởi tạo danh sách nhân viên
    InitArray(myEmployee);

    // Nhân viên có lương cao nhất
    tmpEmployee = searchSalaryMax(myEmployee, length);
    Display(tmpEmployee);

    // Nhân viên có lương thấp nhất
    tmpEmployee = searchSalaryMin(myEmployee, length);
    Display(tmpEmployee);

    // Giải phóng vùng nhớ
    delete [] myEmployee;
    return;
}
```

3.4 MỘT SỐ KIỂU DỮ LIỆU TRÙU TƯỢNG

Nội dung phần này tập trung trình bày việc cài đặt một số cấu trúc dữ liệu trùu tượng, bao gồm:

- Ngăn xếp (stack)
- Hàng đợi (queue)
- Danh sách liên kết (list)

3.4.1 Ngăn xếp

Ngăn xếp (stack) là một kiểu danh sách cho phép thêm và bớt các phần tử ở một đầu danh sách, gọi là đỉnh của ngăn xếp. Ngăn xếp hoạt động theo nguyên lý: phần tử nào được đưa vào sau, sẽ được lấy ra trước.

Định nghĩa cấu trúc ngăn xếp

Vì ta chỉ cần quan tâm đến hai thuộc tính của ngăn xếp là:

- Danh sách các phần tử của ngăn xếp
- Vị trí đỉnh của ngăn xếp

nên ta có thể định nghĩa cấu trúc ngăn xếp như sau (các phần tử của ngăn xếp có kiểu int):

```
typedef SIZE 100; typedef  
struct {  
    int top;           // Vị trí của đỉnh  
    int nodes[SIZE];  // Danh sách các phần tử  
} Stack;
```

Tuy nhiên, định nghĩa này tồn tại một vấn đề, đó là kích thước (SIZE) của danh sách chứa các phần tử là tĩnh. Do đó:

- Nếu ta chọn SIZE lớn, nhưng khi gặp ứng dụng chỉ cần một số ít phần tử cho ngăn xếp thì rất tốn bộ nhớ.
- Nếu ta khai báo SIZE nhỏ, thì khi gặp bài toán cần ngăn xếp có nhiều phần tử, ta sẽ không thêm được các phần tử mới vào, chương trình sẽ có lỗi.

Để khắc phục hạn chế này, ta có thể sử dụng bộ nhớ động (mảng động thông qua con trỏ) để lưu danh sách các phần tử của ngăn xếp. Khi đó, định nghĩa cấu trúc ngăn xếp sẽ có dạng như sau:

```
typedef struct {  
    int top;           // Vị trí của đỉnh  
    int *nodes;        // Danh sách các phần tử  
} Stack;
```

Ta sẽ sử dụng định nghĩa này trong các chương trình ứng dụng ngăn xếp.

Các thao tác trên ngăn xếp

Đối với các thao tác trên ngăn xếp, ta quan tâm đến hai thao tác cơ bản:

- Thêm một phần tử mới vào đỉnh ngăn xếp, gọi là **push**.
- Lấy ra một phần tử từ đỉnh ngăn xếp, gọi là **pop**.

Khi thêm một phần tử mới vào ngăn xếp, ta làm các bước như sau:

1. Só phần tử trong ngăn xếp cũ là $(top+1)$. Do đó, ta cấp phát một vùng nhớ mới để lưu được $(top+1+1) = (top+2)$ phần tử.
2. Sao chép $(top+1)$ phần tử cũ sang vùng mới. Nếu danh sách ban đầu rỗng ($top = -1$) thì không cần thực hiện bước này.
3. Thêm phần tử mới vào cuối vùng nhớ mới
4. Giải phóng vùng nhớ của danh sách cũ

5. Cho danh sách nodes trỏ vào vùng nhớ mới.

Chương trình 3.4a cài đặt thủ tục thêm một phần tử mới vào ngăn xếp.

Chương trình 3.4a

```
void push(Stack *stack, int node) {
    int *tmpNodes = new int[stack->top + 2]; // Cấp phát vùng nhớ mới
    stack->top++; // Tăng chỉ số của node đỉnh
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    tmpNodes[stack->top] = node; // Thêm node mới vào đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trỏ vào vùng nhớ mới
    return;
}
```

Khi lấy ra một phần tử của ngăn xếp, ta làm các bước như sau:

- Kiểm tra xem ngăn xếp có rỗng (top = -1) hay không. Nếu không rỗng thì thực hiện các bước tiếp theo.
- Lấy phần tử ở đỉnh ngăn xếp ra
- Cấp phát một vùng nhớ mới có $(top+1) - 1 = top$ phần tử
- Sao chép top phần tử từ danh sách cũ sang vùng nhớ mới (trừ phần tử ở đỉnh).
- Giải phóng vùng nhớ cũ
- Cho con trỏ danh sách trỏ vào vùng nhớ mới.
- Trả về giá trị phần tử ở đỉnh đã lấy ra.

Chương trình 3.4b cài đặt thủ tục lấy một phần tử từ ngăn xếp.

Chương trình 3.4b

```
int pop(Stack *stack) {
    if(stack->top < 0) { // Kiểm tra ngăn xếp rỗng
        cout << "Stack is empty!" << endl;
        return 0;
    }
    int result = stack->nodes[stack->top]; // Lưu giữ giá trị đỉnh
    int *tmpNodes = new int[stack->top]; // Cấp phát vùng nhớ mới
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    stack->top--; // Giảm chỉ số của node đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trỏ vào vùng nhớ mới
    return result; // Trả về giá trị node đỉnh
}
```

```
}
```

Áp dụng

Ngăn xếp được sử dụng trong các ứng dụng thỏa mãn nguyên tắc: cái nào đặt vào trước sẽ được lấy ra sau. Chương trình 3.4c minh họa việc dùng ngăn xếp để đảo ngược một xâu kí tự được nhập vào từ bàn phím.

Chương trình 3.4c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct {
    int top;                                // Vị trí node đỉnh
    int *nodes;                             // Danh sách phần tử
} Stack;

/* Khai báo nguyên mẫu hàm */
void init(Stack *stack);
void push(Stack *stack, int node);
int pop(Stack *stack);
void release(Stack *stack);

void init(Stack *stack) {
    stack = new Stack;                      // Cấp phát vùng nhớ cho con trỏ
    stack->top = -1;                        // Khởi tạo ngăn xếp rỗng
}

void push(Stack *stack, int node) {
    int *tmpNodes = new int[stack->top + 2]; // Cấp phát vùng nhớ mới
    stack->top++;                          // Tăng chỉ số của node đỉnh
    for(int i=0; i<stack->top; i++)        // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    tmpNodes[stack->top] = node;            // Thêm node mới vào đỉnh
    delete [] stack->nodes;                // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes;                // Trỏ vào vùng nhớ mới
    return;
}

int pop(Stack *stack) {
    if(stack->top < 0){                  // Kiểm tra ngăn xếp rỗng
        return -1;
    }
    int result = stack->nodes[stack->top];
    stack->top--;
    return result;
}
```

```

        cout << "Stack is empty!" << endl;
        return 0;
    }

    int result = stack->nodes[stack->top];// Lưu giữ giá trị đỉnh
    int *tmpNodes = new int[stack->top];// Cấp phát vùng nhớ mới
    for(int i=0; i<stack->top; i++) // Sao chép sang vùng nhớ mới
        tmpNodes[i] = stack->nodes[i];
    stack->top --; // Giảm chỉ số của node đỉnh
    delete [] stack->nodes; // Giải phóng vùng nhớ cũ
    stack->nodes = tmpNodes; // Trò vào vùng nhớ mới
    return result; // Trả về giá trị node đỉnh
}

void release(Stack *stack) {
    delete [] stack->nodes; // Giải phóng vùng danh sách
    delete stack; // Giải phóng con trỏ
    return;
}

void main() {
    clrscr();
    Stack *stack;
    init(stack); // Khởi tạo ngăn xếp
    char strIn[250];
    // Nhập chuỗi kí tự từ bàn phím
    cout << "Nhập chuỗi: ";
    cin >> strIn;
    for(int i=0; i<strlen(strIn); i++) // Đặt vào ngăn xếp
        push(stack, strIn[i]);
    while(stack->top > -1) // Lấy ra từ ngăn xếp
        cout << pop(stack);
    release(stack); // Giải phóng bộ nhớ
    return;
}

```

3.4.2 Hàng đợi

Hàng đợi (queue) cũng là một cấu trúc tuyến tính các phần tử. Trong đó, các phần tử luôn được thêm vào ở một đầu, gọi là đầu cuối hàng đợi, và việc lấy ra các phần tử luôn được thực hiện ở đầu còn lại, gọi là đầu mặt của hàng đợi. Hàng đợi hoạt động theo nguyên lý: phần tử nào được đưa vào trước, sẽ được lấy ra trước.

Định nghĩa cấu trúc hàng đợi

Hàng đợi có các thuộc tính:

- Một danh sách các phần tử có mặt trong hàng đợi.
- Chỉ số của phần tử đứng đầu của danh sách (front).
- Chỉ số phần tử cuối của danh sách (rear).

Nếu dùng cấu trúc tĩnh để định nghĩa, hàng đợi có cấu trúc như sau:

```
typedef SIZE 100; typedef  
struct {  
    int front, rear;           // Vị trí của đỉnh đầu, đỉnh cuối int  
    nodes[SIZE];              // Danh sách các phần tử  
} Queue;
```

Nếu dùng bộ nhớ động để lưu giữ hàng đợi, thì phần tử front luôn là phần tử thứ 0 của danh sách. Và rear sẽ bằng độ dài danh sách trừ đi 1. Cấu trúc động của hàng đợi:

```
typedef struct {  
    int front, rear;           // Vị trí của đỉnh đầu, đỉnh cuối int *nodes;  
    // Danh sách các phần tử  
} Queue;
```

Thao tác trên hàng đợi

- Thêm một phần tử vào cuối hàng đợi
- Lấy một phần tử ở vị trí đầu của hàng đợi

Thao tác thêm một phần tử vào cuối hàng đợi với bộ nhớ động được thực hiện tương tự với ngăn xếp. Chương trình 3.5a cài đặt thủ tục thêm một phần tử vào cuối hàng đợi động.

Chương trình 3.5a

```
void insert(Queue *queue, int node) {  
    int *tmpNodes = new int[queue->rear + 2]; // Cấp phát vùng nhớ mới  
    queue->rear++;                         // Tăng chỉ số của node đuôi  
    if(queue->front == -1)                   // Nếu hàng đợi cũ rỗng  
        queue->front = 0;                     // thì cập nhật front  
    for(int i=0; i<queue->rear; i++)         // Sao chép sang vùng nhớ mới  
        tmpNodes[i] = queue->nodes[i];  
    tmpNodes[queue->rear] = node;             // Thêm node mới vào đuôi  
    delete [] queue->nodes;                  // Giải phóng vùng nhớ cũ  
    queue->nodes = tmpNodes;                 // Trỏ vào vùng nhớ mới  
    return;  
}
```

Thao tác lấy ra một phần tử đầu của hàng đợi thực hiện theo các bước:

1. Kiểm tra tính rỗng ($front = rear = -1$) của hàng đợi. Nếu không rỗng mới thực hiện tiếp

2. Lấy phần tử nodes[0] ra.
3. Sao chép danh sách còn lại sang vùng nhớ mới
4. Giải phóng vùng nhớ cũ
5. Đưa danh sách trả vào vùng nhớ mới
6. Trả về giá trị phần tử lấy ra

Chương trình 3.5b cài đặt thủ tục lấy ra một phần tử của hàng đợi động.

Chương trình 3.5b

```
int remove(Queue *queue) {
    if((queue->front < 0) || (queue->rear < 0)){// Kiểm tra hàng đợi rỗng
        cout << "Queue is empty!" << endl;
        return 0;
    }
    // Lưu giữ giá trị phần tử đầu
    int result = queue->nodes[queue->front];
    int *tmpNodes;
    if(queue->rear > 0){           // Nếu có hơn 1 phần tử
        tmpNodes = new int[queue->rear];// Cấp phát vùng nhớ mới
        for(int i=0; i<queue->rear; i++)// Sao chép sang vùng nhớ mới
            tmpNodes[i] = queue->nodes[i];
    }else                         // Nếu chỉ có 1 phần tử
        queue->front --;          // Hàng đợi thành rỗng
    queue->rear --;              // Giảm chỉ số của node đuôi
    delete [] queue->nodes;      // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes;     // Trả vào vùng nhớ mới
    return result;                // Trả về giá trị node đầu
}
```

Áp dụng

Hàng đợi được áp dụng trong các bài toán cần cơ chế quản lý cái nào vào trước sẽ được lấy ra trước. Chương trình 3.5c minh họa cơ chế quản lý tiến trình đơn giản nhất của hệ điều hành: các tiến trình được quản lý theo mã tiến trình, khi xuất hiện, tiến trình được đưa vào cuối của một hàng đợi. Khi nào CPU rảnh thì sẽ lấy tiến trình đầu hàng đợi ra để thực hiện.

Chương trình 3.5c

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

typedef struct {
    int front, rear;           // Vị trí của đỉnh đầu, đỉnh cuối
```

Chương 3: Kiểu dữ liệu cấu trúc

```
int *nodes;                                // Danh sách các phần tử
} Queue;

/* Khai báo các nguyên mẫu hàm */ void
init(Queue *queue);
void insert(Queue *queue, int node); int
remove(Queue *queue);
void traverse(Queue *queue); void
release(Queue *queue);

void init(Queue *queue){
    queue = new Queue;                      // Cấp phát bộ nhớ cho con trỏ queue-
    >front = -1;                            // Khởi tạo danh sách rỗng queue->rear =
    -1;
    return;
}

void insert(Queue *queue, int node){
    int *tmpNodes = new int[queue->rear + 2];// Cấp phát vùng nhớ mới queue->rear++;
                                                // Tăng chỉ số của node cuối
    if(queue->front == -1)                  // Nếu hàng đợi cũ rỗng queue-
        >front = 0;                          // thì cập nhật front
    for(int i=0; i<queue->rear; i++)       // Sao chép sang vùng nhớ mới
        tmpNodes[i] = queue->nodes[i];
    tmpNodes[queue->rear] = node;           // Thêm node mới vào cuối delete
    [] queue->nodes;                      // Giải phóng vùng nhớ cũ
    queue->nodes = tmpNodes;               // Trả vào vùng nhớ mới return;
}

int remove(Queue *queue){
    if((queue->front < 0)||(queue->rear < 0)){// Kiểm tra hàng đợi rỗng cout << "Queue is empty!""
        << endl;
        return 0;
    }
    // Lưu giữ giá trị phần tử đầu
    int result = queue->nodes[queue->front]; int *tmpNodes;
    if(queue->rear > 0){                  // Nếu có hơn 1 phần tử tmpNodes =
        new int[queue->rear];// Cấp phát vùng nhớ mới for(int i=0; i<queue->rear; i++)// Sao
        chép sang vùng nhớ mới
        tmpNodes[i] = queue->nodes[i];
    }else                                // Nếu chỉ có 1 phần tử
}
```

```

        queue->front --;                                // Hàng đợi thành rỗng
        queue->rear --;                               // Giảm chỉ số của node đuôi
        delete [] queue->nodes;                      // Giải phóng vùng nhớ cũ
        queue->nodes = tmpNodes;                     // Trỏ vào vùng nhớ mới
        return result;                                // Trả về giá trị node đầu
    }

void travese(Queue *queue){
    if(queue->front < 0){                         // Khi danh sách rỗng cout
        << "Danh sách rỗng!" << endl;
        return;
    }
    for(int i=queue->front; i<=queue.rear; i++)
        cout << queue->nodes[i] << " "; // Liệt kê các phần tử cout << endl;
    return;
}

void release(Queue *queue){
    if(queue->front > -1)                         // Nếu danh sách không rỗng thì delete []
        delete [] queue->nodes; // giải phóng vùng nhớ của danh sách
    delete queue;                                    // Giải phóng vùng nhớ của con trỏ
}

void main(){
    clrscr(); Queue
    *queue;
    init(queue);                                     // Khởi tạo hàng đợi int
    function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot tien trinh vao hang doi" << endl;
        cout << "2: Dua mot tien trinh vao thuc hien" << endl; cout << "3: Xem tat ca cac
        tien trinh trong hang doi" << endl; cout << "5: Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl; cin >>
        function;
        switch(function){
            case '1': // Thêm vào hàng đợi int maso;
                cout << "Ma so tien trinh vao hang doi: ";

```

```

        cin >> maso;
        insert(queue, maso);
        break;
    case '2': // Lấy ra khỏi hàng đợi
        cout << "Tien trinh duoc thuc hien: " <<
            remove(queue) << endl;
        break;
    case '3': // Duyệt hàng đợi
        cout << "Cac tien trinh dang o trong hang doi la:" << endl;
        traverse(queue);
        break;
    }while(function != '5');
    release(queue); // Giải phóng hàng đợi
    return;
}

```

3.4.3 Danh sách liên kết

Danh sách liên kết là một kiểu dữ liệu bao gồm một dãy các phần tử có thứ tự, các phần tử có cùng cấu trúc dữ liệu, ngoại trừ node đầu tiên của danh sách là node lưu thông tin về danh sách. Có hai loại danh sách liên kết:

- **Danh sách liên kết đơn:** mỗi node có một con trỏ trỏ đến node tiếp theo trong danh sách
- **Danh sách liên kết kép:** mỗi node có hai con trỏ, một trỏ vào node trước, một trỏ vào node tiếp theo trong danh sách.

Trong phần này sẽ trình bày danh sách liên kết đơn. Danh sách liên kết kép được coi như là một bài tập mở rộng từ danh sách liên kết đơn.

Định nghĩa danh sách đơn

Mỗi node của danh sách đơn chứa dữ liệu của nó, đồng thời trỏ đến node tiếp theo trong danh sách, cấu trúc một node như sau:

```

struct simple{
    Employee employee;           // Dữ liệu của node có kiểu Employee struct
    simple *next;                // Trỏ đến node kế tiếp
};

typedef struct simple SimpleNode;

```

Node đầu của danh sách đơn có cấu trúc riêng, nó không chứa dữ liệu như node thường mà chứa các thông tin:

- Số lượng node trong danh sách (không kể bản thân nó – node đầu)
- Con trỏ đến node đầu tiên của danh sách
- Con trỏ đến node cuối cùng của danh sách

Do vậy, cấu trúc node đầu của danh sách đơn là:

```
typedef struct{
    int nodeNumber;           // Số lượng các node
    SimpleNode *front, *rear; // Trỏ đến node đầu và cuối danh sách
} SimpleHeader;
```

Các thao tác trên danh sách liên kết đơn

Các thao tác cơ bản trên danh sách đơn bao gồm:

- Chèn thêm một node vào vị trí thứ n trong danh sách
- Loại ra một node ở vị trí thứ n trong danh sách

Việc chèn thêm một node vào vị trí thứ n trong danh sách được thực hiện theo các bước:

1. Nếu $n \leq 0$, chèn vào đầu. Nếu $n > \text{số phần tử của danh sách}$, chèn vào cuối. Trường hợp còn lại, chèn vào giữa.
2. Tìm node thứ n: giữ vết của hai node thứ $n-1$ và n .
3. Tạo một node mới: cho node thứ $n-1$ trỏ tiếp vào node mới và node mới trỏ tiếp vào node thứ n .

Chương trình 3.6a cài đặt thủ tục chèn một node vào vị trí thứ n của danh sách.

Chương trình 3.6a

```
void insert(SimpleHeader *list, int position, int value) {
    SimpleNode *newNode = new SimpleNode;
    newNode->value = value;
    if(position <= 0){           // Chèn vào đầu ds
        newNode->next = list->front; // Chèn vào trước node đầu
        list->front = newNode;      // Cập nhật lại node đầu ds
        if(list->nodeNumber == 0)   // Nếu ds ban đầu rỗng thì
            list->rear = newNode;  // node đuôi trùng với node đầu
    }else if(position >= list->nodeNumber){ // Chèn vào cuối ds
        list->rear->next = newNode; // Chèn vào sau node cuối
        list->rear = newNode;      // Cập nhật lại node cuối ds
        if(list->nodeNumber == 0)   // Nếu ds ban đầu rỗng thì
            list->front = newNode; // node đầu trùng node đuôi
    }else{                      // Chèn vào giữa ds
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){ // tìm node n-1 và n
            prev = curr;
            curr = curr->next;
            index++;
        }
        newNode->next = curr;      // chèn vào trước node n
        prev->next = newNode;      // và chèn vào sau node n-1
    }
}
```

```

        list->nodeNumber++;
                                // Cập nhật số lượng node
        return;
    }
}

```

Việc xoá một node ở vị trí thứ n trong danh sách được thực hiện theo các bước:

1. Nếu $n < 0$ hoặc $n > \text{số phần tử của danh sách}$, không xoá node nào.
2. Tìm node thứ n: giữ vết của ba node thứ $n-1$, thứ n và thứ $n+1$.
3. Cho node thứ $n-1$ trỏ tiếp vào node thứ $n+1$, xoá con trỏ của node thứ n .
4. Trả về node thứ n .

Chương trình 3.6b cài đặt thủ tục xoá một node ở vị trí thứ n của danh sách.

Chương trình 3.6b

```

SimpleNode* remove(SimpleHeader *list, int position){
    if((position < 0) || (position >= list->nodeNumber))
        return NULL;                      // Không xoá node nào cả
    SimpleNode* result;
    if(position == 0){                  // Xoá node đầu
        result = list->front;          // Giữ node cần xoá
        list->front = list->front->next; // Cập nhật node đầu
        if(list->nodeNumber == 1)       // Nếu ds chỉ có 1 node thì
            list->rear = list->front; // Cập nhật node cuối ds
        }else if(position == list->nodeNumber - 1){
            result = list->rear;      // Giữ node cần xoá
            SimpleNode *curr = list->front;
            while(curr->next != list->rear)
                curr = curr->next;     // Tìm node trước của node cuối
            curr->next = NULL;         // Xoá node rear hiện tại
            list->rear = curr;        // Cập nhật node cuối ds
        }
    }else{
        SimpleNode *prev = list->front, *curr = list->front;
        int index = 0;
        while(index < position){        // Tìm node  $n-1$  và  $n$ 
            prev = curr;
            curr = curr->next;
            index++;
        }
        result = curr;                  // Giữ node cần xoá
        prev->next = curr->next;       // Cho node  $n-1$  trỏ đến node  $n+1$ 
    }
    list->nodeNumber--;              // Cập nhật số lượng node
    return result;                   // Trả về node cần xoá
}

```

```
}
```

Áp dụng

Chương trình 3.6c minh họa việc dùng danh sách liên kết đơn để quản lý nhân viên văn phòng với các thông tin rất đơn giản: tên, tuổi và tiền lương của mỗi nhân viên.

Chương trình 3.6c

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

typedef struct{
    char name[25];           // Tên nhân viên
    int age;                 // Tuổi nhân viên
    float salary;            // Lương nhân viên
} Employee;

struct simple{
    Employee employee;       // Dữ liệu của node
    struct simple *next;     // Trỏ đến node kế tiếp
};

typedef struct simple SimpleNode;

typedef struct{
    int nodeNumber;          // Số lượng các node
    SimpleNode *front, *rear; // Trỏ đến node đầu và cuối ds
} SimpleHeader;

/* Khai báo các nguyên mẫu hàm */
void init(SimpleHeader *list);
void insert(SimpleHeader *list, int position, Employee employee);
SimpleNode* remove(SimpleHeader *list);
void travese(SimpleHeader *list);
void release(SimpleHeader *list);

void init(SimpleHeader *list){
    list = new list;           // Cấp phát bộ nhớ cho con trỏ
    list->front = NULL;        // Khởi tạo danh sách rỗng
    list->rear = NULL;
```

```

list->nodeNumber = 0; return;
}

void insert(SimpleHeader *list, int position, Employee employee){ SimpleNode *newNode = new
SimpleNode;
newNode->employee = employee;
if(position <= 0){                                     // Chèn vào đầu ds
    newNode->next = list->front;                      // Chèn vào trước node đầu list->front
    = newNode;                                         // Cập nhật lại node đầu ds if(list-
>nodeNumber == 0)                                    // Nếu ds ban đầu rỗng thì
    list->rear = newNode;                             // node đuôi trùng với node đầu
}else if(position >= list->nodeNumber){// Chèn vào cuối ds list->rear->next =
    newNode;                                         // Chèn vào sau node cuối
    list->rear = newNode;                            // Cập nhật lại node cuối ds if(list-
>nodeNumber == 0)// Nếu ds ban đầu rỗng thì
    list->front = newNode;                           // node đầu trùng node đuôi
}else{ //     Chèn     vào     giữa     ds SimpleNode
    *prev = list->front, *curr = list->front; int index = 0;
    while(index < position){// tìm node n-1 và n prev = curr;
        curr = curr->next; index++;
    }
    newNode->next = curr;                          // chèn vào trước node n prev->next
    = newNode;                                       // và chèn vào sau node n-1
}
list->nodeNumber++;                                // Cập nhật số lượng node return;
}

SimpleNode* remove(SimpleHeader *list, int position){ if((position < 0)||(position
>= list->nodeNumber))
    return NULL;                                     // Không xoá node nào cả
SimpleNode* result;
if(position == 0){                                   // Xoá node đầu result =
    list->front;                                  // Giữ node cần xoá
    list->front = list->front->next;// Cập nhật node đầu if(list->nodeNumber == 1)
                                         // Nếu ds chỉ có 1 node thì
    list->rear = list->front;// Cập nhật node cuối ds
}else if(position == list->nodeNumber - 1){ result = list-
>rear;                                         // Giữ node cần xoá
}
}

```

```

SimpleNode *curr = list->front; while(curr->next != list->rear)
    curr = curr->next;// Tìm node trước của node cuối curr->next =
    NULL; // Xoá node rear hiện tại
    list->rear = curr; // Cập nhật node cuối ds
} else{
    SimpleNode *prev = list->front, *curr = list->front; int index = 0;
    while(index < position){// Tìm node n-1 và n prev = curr;
        curr = curr->next; index++;
    }
    result = curr; // Giữ node cần xoá
    prev->next = curr->next;// Cho node n-1 trỏ đến node n+1
}
list->nodeNumber --; // Cập nhật số lượng node return
result; // Trả về node cần xoá
}

void traverse(SimpleHeader *list){
    if(list->nodeNumber <= 0){ // Khi danh sách rỗng cout
        << "Danh sách rỗng!" << endl;
        return;
    }
    SimpleNode *curr = list->front; while(curr != NULL){
        cout << curr->employee.name << " "
            << curr->employee.age << " "
            << curr->employee.salary << endl;// Liệt kê các phần tử curr = curr->next;
    }
    return;
}

void release(SimpleHeader *list){ SimpleNode* curr =
remove(list, 0); while(curr != NULL){
    delete curr; //Giải phóng vùng nhớ của node curr =
    remove(list, 0);
}
delete list; //Giải phóng vùng nhớ của con trỏ
}

```

```
void main(){
    clrscr(); SimpleHeader *list;
    init(list); // Khởi tạo ds int
    function;
    do{
        clrscr();
        cout << "CAC CHUC NANG:" << endl;
        cout << "1: Them mot nhan vien" << endl; cout <<
        "2: Xoa mot nhan vien" << endl;
        cout << "3: Xem tat ca cac nhan vien trong phong" << endl; cout << "5:
        Thoat!" << endl;
        cout << "===== " << endl;
        cout << "Chon chuc nang: " << endl; cin >>
        function;
        switch(function){
            case '1':// Thêm vào ds int position;
                Employee employee;
                cout << "Vi tri can chen: "; cin >>
                position;
                cout << "Ten nhan vien: "; cin >>
                employee.name;
                cout << "Tuoi nhan vien: "; cin >>
                employee.age;
                cout << "Luong nhan vien: "; cin >>
                employee.salary;
                insert(list, position, employee); break;
            case '2':// Lấy ra khỏi ds int position;
                cout << "Vi tri can xoa: "; cin >>
                position;
                SimpleNode* result = remove(list, position); if(result != NULL){
                    cout << "Nhan vien bi loai: " << endl; cout << "Ten:
                    " << result->employee.name
                    << endl;
                    cout << "Tuoi: " << result->employee.age
                    << endl; cout
                    << "Luong: "
                    << result->employee.salary << endl;
                }
        }
    }
}
```

```
        }
        break;
    case '3':           // Duyệt ds
        cout<<"Cac nhan vien cua phong:"<<endl;
        traverse(list);
        break;
    }while(function != '5');
release(list);          // Giải phóng ds
return;
}
```

TỔNG KẾT CHƯƠNG 3

Nội dung chương 3 đã trình bày các vấn đề liên quan đến các kiểu dữ liệu có cấu trúc trong C++:

- Khai báo cấu trúc thông qua từ khoá **struct**
- Tự định nghĩa kiểu dữ liệu cấu trúc bằng từ khoá **typedef**.
- Khai báo một biến có kiểu dữ liệu cấu trúc
- Khai báo các cấu trúc lồng nhau.
- Truy nhập đến các thuộc tính của cấu trúc
- Khai báo con trỏ cấu trúc, cấp phát và giải phóng bộ nhớ động của con trỏ cấu trúc. Truy nhập đến các thuộc tính của con trỏ cấu trúc.
- Khai báo và sử dụng mảng cấu trúc
- Khai báo mảng cấu trúc bằng con trỏ cấu trúc. Cấp phát và giải phóng vùng nhớ của mảng động các cấu trúc.
- Cài đặt một số cấu trúc đặc biệt:
 - Ngăn xếp
 - Hàng đợi
 - Danh sách liên kết

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 3

1. Để định nghĩa một cấu trúc sinh viên có tên là Sinhvien, gồm có tên và tuổi sinh viên. Định nghĩa nào sau đây là đúng:

- a. struct Sinhvien{ char
name[20]; int age;
};
- b. struct {
char name[20]; int
age;
} Sinh vien;

```
c.     typedef struct Sinhvien{  
            char name[20]; int age;  
};
```

2. Một cấu trúc được định nghĩa như sau:

```
struct Employee{  
    char name[20]; int  
    age;  
};
```

Khi đó, cách khai báo biến nào sau đây là đúng:

- a. struct Employee myEmployee;
- b. struct employee myEmployee;
- c. Employee myEmployee;
- d. employee myEmployee;

3. Một cấu trúc được định nghĩa như sau:

```
typedef struct employee{ char  
    name[20]; int age;  
} Employee;
```

Khi đó, cách khai báo biến nào sau đây là đúng:

- a. Employee myEmployee;
- b. employee myEmployee;
- c. struct Employee myEmployee;
- d. struct employee myEmployee;

4. Với cấu trúc được định nghĩa như trong bài 3. Khi đó, cách khởi tạo biến nào sau đây là đúng:

- a. Employee myEmployee = {'A', 27};
- b. Employee myEmployee = {"A", 27};
- c. Employee myEmployee = ('A', 27);
- d. Employee myEmployee = ("A", 27);

5. Với cấu trúc được định nghĩa như trong bài 3. Khi đó, các cách cấp phát bộ nhớ cho biến con trỏ nào sau đây là đúng:

- a. Employee *myEmployee = new Employee;
- b. Employee *myEmployee = new Employee();
- c. Employee *myEmployee = new Employee(10);
- d. Employee *myEmployee = new Employee[10];

6. Định nghĩa một cấu trúc về môn học của một học sinh có tên Subject, bao gồm các thông tin:

- Tên môn học, kiểu char[];
- Điểm tổng kết môn học, kiểu float;

7. Định nghĩa cấu trúc về học sinh tên là Student bao gồm các thông tin sau:

- Tên học sinh, kiểu char[];
- Tuổi học sinh, kiểu int;
- Lớp học sinh, kiểu char[];
- Danh sách điểm các môn học của học sinh, kiểu là một mảng các cấu trúc Subject đã được định nghĩa trong bài tập 6.
- Xếp loại học lực, kiểu char[];

8. Khai báo một biến có cấu trúc là Student đã định nghĩa trong bài 7. Sau đó, thực hiện tính điểm trung bình của tất cả các môn học của học sinh đó, và viết một thủ tục xếp loại học sinh dựa vào điểm trung bình các môn học:

- Nếu điểm tb nhỏ hơn 5.0, xếp loại kém
- Nếu điểm tb từ 5.0 đến dưới 6.5, xếp loại trung bình.
- Nếu điểm tb từ 6.5 đến dưới 8.0, xếp loại khá
- Nếu điểm tb từ 8.0 trở lên, xếp loại giỏi.

9. Viết một chương trình quản lí các học sinh của một lớp, là một dãy các cấu trúc có kiểu Stupid định nghĩa trong bài 7. Sử dụng thủ tục đã cài đặt trong bài 8 để thực hiện các thao tác sau:

- Khởi tạo danh sách và điểm của các học sinh trong lớp.
- Tính điểm trung bình và xếp loại cho tất cả các học sinh.
- Tìm tất cả các học sinh theo một loại nhất định

10. Sử dụng cấu trúc ngăn xếp đã định nghĩa trong bài để đổi một số từ kiểu thập phân sang kiểu nhị phân: Chỉ số nguyên cho 2, mãi cho đến khi thương <2, lưu các số dư vào ngăn xếp. Sau đó, đọc các giá trị dư từ ngăn xếp ra, ta sẽ thu được chuỗi nhị phân tương ứng.

11. Mở rộng cấu trúc hàng đợi đã định nghĩa trong bài để trở thành hàng đợi có độ ưu tiên:

- Cho mỗi node thêm một thuộc tính là độ ưu tiên của node đó
- Khi thêm một node vào hàng đợi, thay vì thêm vào cuối hàng đợi như thông thường, ta tìm vị trí có độ ưu tiên phù hợp để chèn node vào, sao cho dãy các node trong hàng đợi là một danh sách có độ ưu tiên của các node là giảm dần.
- Việc lấy ra là không thay đổi: lấy ra phần tử ở đầu hàng đợi, chính là phần tử có độ ưu tiên cao nhất.

12. Áp dụng hàng đợi có độ ưu tiên trong bài 11 để xây dựng chương trình quản lí tiến trình có độ ưu tiên của hệ điều hành, mở rộng ứng dụng trong bài ngăn xếp.

13. Mở rộng cấu trúc danh sách liên kết đơn trong bài thành danh sách liên kết kép:

- Mỗi node có thêm một con trỏ prev để trỏ đến node trước nó
- Đối với node header, cũng cần 2 con trỏ: trỏ đến node đầu tiên và node cuối cùng của danh sách

- Riêng với node đầu tiên (front) của danh sách, con trỏ prev của nó sẽ trỏ đến NULL. Giống như con trỏ next của node rear.
14. Cài đặt lại hai thao tác thêm vào một node và xoá một node ở một vị trí xác định trong một cấu trúc danh sách liên kết kép định nghĩa trong bài 13.
 15. Áp dụng các định nghĩa và thao tác trong các bài 13 và 14. Cài đặt lại chương trình quản lý nhân viên ở chương trình 3.6c bằng danh sách liên kết kép.

CHƯƠNG 4

VÀO RA TRÊN TỆP

Nội dung chương này tập trung trình bày các vấn đề liên quan đến các thao tác trên tệp dữ liệu trong ngôn ngữ C++:

- Khái niệm tệp, tệp văn bản và tệp nhị phân
- Các thao tác vào ra trên tệp
- Phương thức truy nhập tệp trực tiếp

4.1 KHÁI NIỆM TỆP

4.1.1 Tệp dữ liệu

Trong C++, khi thao tác với một tệp dữ liệu, cần thực hiện tuần tự theo các bước như sau:

1. Mở tệp tin
2. Thực hiện các thao tác đọc, ghi trên tệp tin đang mở
3. Đóng tệp tin

Để thực hiện các thao tác liên quan đến tệp dữ liệu, C++ cung cấp một thư viện `<fstream.h>` chứa các lớp và các hàm phục vụ cho các thao tác này. Do vậy, trong các chương trình làm việc với tệp tin, ta cần khai báo chỉ thị dùng thư viện này ngay từ đầu chương trình:

```
#include<fstream.h>
```

Khai báo biến tệp

Trong C++, khi khai báo một biến tệp, đồng thời ta sẽ mở tệp tương ứng theo cú pháp tổng quát bằng cách dùng kiểu **fstream** như sau:

```
fstream <Tên biến tệp>(<Tên tệp>, <Chế độ mở tệp>);
```

Trong đó:

- **Tên biến tệp:** có tính chất như một tên biến thông thường, nó sẽ được dùng để thực hiện các thao tác với tệp gắn với nó. Tên biến tệp cũng phải tuân thủ theo quy tắc đặt tên biến trong C++.
- **Tên tệp:** là tên tệp dữ liệu mà ta cần thao tác trên nó.
- **Chế độ mở tệp:** là các hằng kiểu bít đã được định nghĩa sẵn bởi C++. Nó chỉ ra rằng ta đang mở tệp tin ở chế độ nào: đọc hoặc ghi, hoặc cả đọc lẫn ghi.

Ví dụ, khai báo:

```
fstream myFile("abc.txt", ios::in);
```

là khai báo một biến tệp, có tên là myFile, dùng để mở tệp tin có tên là abc.txt và tệp tin này được mở ở chế độ để đọc dữ liệu (bít chỉ thị ios::in).

Lưu ý:

- Tên tệp tin có dạng một chuỗi kí tự, nếu khai báo tên tệp có đường dẫn thư mục “\” thì mỗi dấu “\” phải được viết thành “\\” để tránh bị nhầm lẫn với các kí tự đặc biệt trong C như “\n”, “\d”...

Ví dụ, muốn mở một tệp tên là abc.txt trong thư mục myDir để đọc, ta phải khai báo như sau:

```
fstream myFile("myDir\\abc.txt", ios::in);
```

Các chế độ mở tệp tin

Các chế độ mở tệp tin được định nghĩa bởi các bít chỉ thị:

- **ios::in:** Mở một tệp tin để đọc.
- **ios::out:** Mở một tệp tin có sẵn để ghi.
- **ios::app:** Mở một tệp tin có sẵn để thêm dữ liệu vào cuối tệp.
- **ios::ate:** Mở tệp tin và đặt con trỏ tệp tin vào cuối tệp.
- **ios::trunc:** Nếu tệp tin đã có sẵn thì dữ liệu của nó sẽ bị mất.
- **ios::nocreate:** Mở một tệp tin, tệp tin này bắt buộc phải tồn tại.
- **ios::noreplace:** Chỉ mở tệp tin khi tệp tin chưa tồn tại.
- **ios::binary:** Mở một tệp tin ở chế độ nhị phân.
- **ios::text:** Mở một tệp tin ở chế độ văn bản.

Lưu ý:

- Khi muốn mở một tệp tin đồng thời ở nhiều chế độ khác nhau, ta kết hợp các bít chỉ thị tương ứng bằng phép toán hợp bít “|”.

Ví dụ, muốn mở một tệp tin abc.txt để đọc (ios::in) đồng thời với để ghi (ios::out) dưới chế độ văn bản (ios::text), ta khai báo như sau:

```
fstream myFile("abc.txt", ios::in|ios::out|ios::text);
```

4.1.2 Tệp văn bản

Để mở một tệp tin dưới chế độ văn bản, ta dùng cú pháp sau:

```
fstream <Tên biến tệp>(<Tên tệp>, ios::text);
```

Khi đó, các thao tác đọc, ghi trên biến tệp được thực hiện theo đơn vị là các từ, được phân cách bởi dấu trống (space bar) hoặc dấu xuống dòng (enter).

Ví dụ, muốn mở tệp tin baitho.txt dưới chế độ văn bản, ta khai báo như sau:

```
fstream myBaiTho("baitho.txt", ios::text);
```

4.1.3 Tệp nhị phân

Để mở một tệp tin dưới chế độ nhị phân, ta dùng cú pháp sau:

```
fstream <Tên biến tệp>(<Tên tệp>, ios::binary);
```

Khi đó, các thao tác đọc, ghi trên biến tệp được thực hiện theo đơn vị byte theo kích thước các bản ghi (cấu trúc) được ghi trong tệp.

Ví dụ, muốn mở tệp tin baitho.txt dưới chế độ nhị phân, ta khai báo như sau:

```
fstream myBaiTho("baitho.txt", ios::binary);
```

4.2 VÀO RA TRÊN TỆP

4.2.1 Vào ra tệp văn bản bằng “>>” và “<<”

Ghi tệp văn bản bằng “<<”

Các bước thực hiện để ghi dữ liệu vào một tệp tin như sau:

1. Mở tệp tin theo chế độ để ghi bằng đối tượng ofstream (mở tệp tin chỉ để ghi):

ofstream <Tên biến tệp>(<Tên tệp tin>, ios::out);

2. Ghi dữ liệu vào tệp bằng thao tác “<<”:

<Tên biến tệp> << <Dữ liệu>;

3. Đóng tệp tin bằng lệnh close():

<Tên biến tệp>.close();

Chương trình 4.1 minh họa việc ghi dữ liệu vào tệp tin:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ ghi vào tệp các kí tự do người dùng gõ vào từ bàn phím, mỗi kí tự được phân cách nhau bởi dấu trắng (space bar).
- Chương trình dừng lại khi người dùng nhập kí tự ‘e’. Và tệp tin được kết thúc bằng một dấu xuống dòng “endl”.

Chương trình 4.1

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

const int length = 25; // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], input;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin

    /* Mở tệp tin */
    ofstream fileOut(fileName, ios::out); // Khai báo và mở tệp tin
    if(!fileOut){ // Không mở được tệp
        cout << "Khong the tao duoc tep tin " << fileName << endl;
        exit(1);
    }

    /* Ghi dữ liệu vào tệp tin */
```

Chương 4: Vào ra trên tệp

```
do{  
    cin >> input; // Đọc kí tự từ bàn phím  
    fileOut << input << ' '; // Ghi kí tự vào tệp tin  
}while((input != 'e') && (fileOut));  
fileOut << endl; // Xuống dòng cuối tệp tin  
  
/* Đóng tệp tin */  
fileOut.close(); // Đóng tệp tin  
return;  
}
```

Đọc dữ liệu từ tệp văn bản bằng “>>”

Các bước thực hiện để đọc dữ liệu từ một tệp tin như sau:

1. Mở tệp tin theo chế độ để đọc bằng đối tượng ifstream (mở tệp tin chỉ để đọc):
 ifstream <Tên biến tệp>(<Tên tệp tin>, ios::in);
2. Đọc dữ liệu từ tệp bằng thao tác “>>”:
 <Tên biến tệp> SS <Biến dữ liệu>;
3. Đóng tệp tin bằng lệnh close():
 <Tên biến tệp>.close();

Chương trình 4.2 minh họa việc đọc dữ liệu từ tệp tin vừa sử dụng trong chương trình 4.1 ra màn hình:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ đọc các kí tự trong tệp và hiển thị ra màn hình, mỗi kí tự được phân cách nhau bởi dấu trống (space bar).
- Chương trình dừng lại khi kết thúc tệp tin.

Chương trình 4.2

```
#include<stdlib.h>  
#include<iostream.h>  
#include<fstream.h>  
#include<conio.h>  
  
const int length = 25; // Độ dài tối đa tên tệp tin  
  
void main(){  
    clrscr();  
    char fileName[length], output;  
    cout << "Tên tệp tin: ";  
    cin >> setw(length) >> fileName; // Nhập tên tệp tin  
  
    /* Mở tệp tin */
```

Chương 4: Vào ra trên tệp

```
ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin
if(!fileIn){ // Không mở được tệp
    cout << "Khong the mo duoc tep tin " << fileName << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra màn hình */
while(fileIn){
    fileIn >> output; // Đọc kí tự từ tệp tin
    cout << output; // Ghi kí tự ra màn hình
}
cout << endl; // Xuống dòng trên màn hình

/* Đóng tệp tin */
fileIn.close(); // Đóng tệp tin
return;
}
```

Chương trình 4.3 minh họa việc copy toàn bộ nội dung của một tệp tin sang một tệp tin mới:

- Tên tệp tin nguồn và tệp tin đích được nhập từ bàn phím bởi người dùng.
- Tệp tin nguồn được mở ở chế độ đọc.
- Tệp tin đích được mở ở chế độ ghi.
- Đọc từng kí tự từ tệp tin nguồn và ghi ngay vào tệp tin đích.
- Đóng các tệp tin khi kết thúc.

Chương trình 4.3

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

const int length = 25; // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char sourceFile[length], targetFile[length], data;
    cout << "Ten tep tin nguon: ";
    cin >> setw(length) >> sourceFile; // Nhập tên tệp tin nguồn

    cout << "Ten tep tin dich: ";
    cin >> setw(length) >> targetFile; // Nhập tên tệp tin đích
```

```
/* Mở tệp tin nguồn */
ifstream fileIn(sourceFile, ios::in); // Khai báo và mở tệp nguồn
if(!fileIn){ // Không mở được tệp nguồn
    cout << "Khong the mo duoc tep tin nguon "
    << sourceFile << endl;
    exit(1);
}

/* Mở tệp tin đích */
ofstream fileOut(targetFile, ios::out); // Khai báo và mở tệp đích
if(!fileOut){ // Không mở được tệp đích
    cout << "Khong the tao duoc tep tin dich "
    << targetFile << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra tệp đích */
while(fileIn){
    fileIn >> data; // Đọc kí tự từ tệp nguồn
    fileOut << data; // Ghi kí tự ra tệp đích
}

/* Đóng các tệp tin */
fileIn.close(); // Đóng tệp tin nguồn
fileOut.close(); // Đóng tệp tin đích
return;
}
```

Lưu ý:

- Tên biến tệp, sau khi dùng xong với một tệp xác định, có thể sử dụng để mở một tệp khác, với một chế độ mở tệp khác bằng phép toán **open()** của biến tệp.

<Tên biến tệp>.open(<Tên tệp mới>, <chế độ mở mới>);

Ví dụ, đoạn chương trình:

```
ofstream myFile("abc.txt", ios::out);
...
// ghi vào file abc.txt
myFile.close();
myFile.open("xyz.txt", ios::out|ios::app);
...
// Them vao cuoi file xyz.txt
myFile.close();
```

sẽ dùng biến tệp myFile (có kiểu ofstream) hai lần: một lần là dùng với tệp tin abc.txt ở chế độ mở để ghi từ đầu. Một lần khác là với tệp tin xyz.txt ở chế độ mở để ghi thêm vào cuối.

4.2.2 Vào ra tệp nhị phân bằng read và write

Ghi vào tệp nhị phân bằng write

Các bước thực hiện để ghi dữ liệu vào một tệp nhị phân như sau:

1. Mở tệp tin theo chế độ để ghi nhị phân bằng đối tượng fstream:

```
fstream <Tên biến tệp>(<Tên tệp tin>, ios::out|ios::binary);
```

2. Ghi dữ liệu vào tệp bằng thao tác “**write()**”:

```
<Tên biến tệp>.write(char* <Đữ liệu>,  
int <Kích thước dữ liệu>);
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Trong đó, thao tác **write** nhận hai tham số đầu vào như sau:

- Tham số thứ nhất là con trỏ kiểu char trả về đến vùng dữ liệu cần ghi vào tệp. Vì con trỏ bắt buộc có kiểu char nên khi muốn ghi dữ liệu có kiểu khác vào tệp, ta dùng hàm chuyển kiểu:

```
reinterpret_cast<char*>(<Đữ liệu>);
```

- Tham số thứ hai là kích cỡ dữ liệu được ghi vào tệp. Kích cỡ này được tính theo byte, nên thông thường ta dùng toán tử:

```
sizeof(<Kiểu dữ liệu>);
```

Lưu ý:

- Khi muốn đọc, ghi các dữ liệu có cấu trúc (struct) vào tệp thì ta phải dùng ở chế độ đọc/ghi tệp nhị phân mà không thể dùng chế độ đọc/ghi ở chế độ văn bản.
- Khi đọc/ghi dữ liệu có kiểu cấu trúc, để toán tử sizeof() thực hiện chính xác thì các thành viên của cấu trúc không được là kiểu con trỏ. Vì toán tử sizeof() đối với con trỏ chỉ cho kích cỡ của con trỏ mà không cho kích cỡ thật của vùng dữ liệu mà con trỏ trả tới.

Chương trình 4.4 minh họa việc ghi dữ liệu vào tệp tin nhị phân, dữ liệu là kiểu cấu trúc:

- Tên tệp tin và số lượng bản ghi được người dùng nhập vào từ bàn phím.
- Chương trình sẽ ghi vào tệp các bản ghi có cấu trúc do người dùng gõ vào từ bàn phím.

Chương trình 4.4

```
#include<stdlib.h>  
#include<iostream.h>  
#include<fstream.h>  
#include<conio.h>  
#include<type.h>  
  
const int length = 25; // Độ dài tối đa tên tệp tin  
  
typedef struct {  
    int day; // Ngày  
    int month; // Tháng
```

Chương 4: Vào ra trên tệp

```
int year;                                // Năm
} Date;

typedef struct {
    char name[20];                         // Tên nhân viên
    Date birthDay;                          // Ngày sinh của nhân viên
    char role[20];                          // Chức vụ của nhân viên
    float salary;                           // Lương của nhân viên
} Employee;

void main(){
    clrscr();
    char fileName[length];                  // Tên tệp tin cout
    << "Ten tep tin: ";
    cin >> setw(length) >> fileName;       // Nhập tên tệp tin

    int recordNumber;                      // Số lượng bản ghi cout
    << "So luong ban ghi: ";
    cin >> recordNumber;                  // Nhập số lượng bản ghi

    /* Mở tệp tin */
    // Khai báo và mở tệp tin
    fstream fileOut(fileName, ios::out|ios::binary); if(!fileOut){ // Không mở được
        tệp
        cout << "Khong the tao duoc tep tin " << fileName << endl; exit(1);
    }

    /* Ghi dữ liệu vào tệp tin */ Employee
myEmployee;
for(int i=0; i<recordNumber; i++){
    cout << "Ban ghi thu " << i+1 << endl; cout <<
    "Name: ";
    cin >> myEmployee.name;                // Nhập tên nhân viên cout
    << "Day of birth: ";
    cin >> myEmployee.birthDay.day;// Nhập ngày sinh cout << "Month
    of birth: ";
    cin >> myEmployee.birthDay.month;// Nhập tháng sinh cout << "Year
    of birth: ";
    cin >> myEmployee.birthDay.year;// Nhập năm sinh cout << "Role:
    ";
    cin >> myEmployee.role;                 // Nhập chức vụ
    cout << "Salary: ";
```

Chương 4: Vào ra trên tệp

```
    cin >> myEmployee.salary;           // Nhập tiền lương

    // Ghi dữ liệu vào tệp
    fileOut.write(reinterpret_cast<char*>(&myEmployee),
                  sizeof(Employee));
}

/* Đóng tệp tin */
fileOut.close();                      // Đóng tệp tin
return;
}
```

Đọc dữ liệu từ tệp nhị phân bằng read

Các bước thực hiện để đọc dữ liệu từ một tệp tin nhị phân như sau:

1. Mở tệp tin theo chế độ để đọc nhị phân bằng đối tượng fstream (mở tệp tin chỉ để ghi):

```
fstream <Tên biến tệp>(<Tên tệp tin>, ios::in|ios::binary);
```

2. Đọc dữ liệu từ tệp bằng thao tác “**read()**”:

```
<Tên biến tệp>.read(char* <Đữ liệu ra>,
                     int <Kích thước dữ liệu>);
```

3. Đóng tệp tin bằng lệnh close():

```
<Tên biến tệp>.close();
```

Chương trình 4.5 minh họa việc đọc dữ liệu từ tệp tin vào biến có cấu trúc:

- Tên tệp tin được người dùng tự nhập vào từ bàn phím.
- Chương trình sẽ đọc các cấu trúc nhân viên trong tệp và hiển thị ra màn hình.
- Chương trình dừng lại khi kết thúc tệp tin.

Chương trình 4.5

```
#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<type.h>

const int length = 25;                   // Độ dài tối đa tên tệp tin

typedef struct {
    int day;                            // Ngày
    int month;                           // Tháng
    int year;                            // Năm
} Date;
```

```
typedef struct {
    char name[20];                                // Tên nhân viên
    Date birthDay;                                 // Ngày sinh của nhân viên
    char role[20];                                // Chức vụ của nhân viên
    float salary;                                  // Lương của nhân viên
} Employee;

void main(){
    clrscr();
    char fileName[length];                         // Tên tệp tin cout
    << "Ten tep tin: ";
    cin >> setw(length) >> fileName;             // Nhập tên tệp tin

    /* Mở tệp tin */
    // Khai báo và mở tệp tin
    fstream fileIn(fileName, ios::in|ios::binary); if(!fileIn){      // Không mở được
        tệp
        cout << "Khong the mo duoc tep tin " << fileName << endl; exit(1);
    }

    /* Đọc dữ liệu từ tệp tin ra màn hình */ Employee
    myEmployee;
    while(fileIn){
        fileIn.read(reinterpret_cast<char *>(&myEmployee), sizeof(Employee));      // Đọc
            kí tự từ tệp tin
        cout << myEmployee.name << " "
            << myEmployee.birthDay.day << "/"
            << myEmployee.birthDay.month << "/"
            << myEmployee.birthDay.year << " "
            << myEmployee.role << " "
            << myEmployee.salary << endl; // Ghi kí tự ra màn hình
    }

    /* Đóng tệp tin */
    fileIn.close();                               // Đóng tệp tin
    return;
}
```

4.3 TRUY NHẬP TỆP TRỰC TIẾP

4.3.1 Con trỏ tệp tin

Con trỏ tệp tin có vai trò như một đầu đọc trỏ vào một vị trí xác định của tệp và thao tác truy nhập tệp diễn ra tuần tự:

- Tại mỗi thời điểm, con trỏ tệp tin xác định một vị trí trên tệp mà tại đó, thao tác truy nhập tệp (đọc/ghi) được thực hiện.
- Sau thao tác truy nhập, con trỏ tệp tự động chuyển đến vị trí tiếp theo dựa vào kích thước đơn vị dữ liệu được truy nhập.

Cách truy nhập tệp tuần tự có nhược điểm là bao giờ cũng phải bắt đầu từ đầu tệp tin, đi tuần tự cho đến vị trí cần truy nhập. Khi tệp tin có kích thước lớn thì cách truy nhập này rất tốn thời gian.

Để tránh nhược điểm này, C++ cho phép truy nhập trực tiếp đến một vị trí xác định trên tệp tin bằng các phép toán:

- Truy nhập vị trí hiện tại của con trỏ tệp tin
- Dịch chuyển con trỏ tệp tin đến một vị trí xác định

4.3.2 Truy nhập vị trí hiện tại của con trỏ tệp

Cú pháp truy nhập đến vị trí hiện thời của con trỏ tệp phụ thuộc vào kiểu biến tệp đang dùng là để đọc hay ghi.

- Nếu biến tệp là kiểu mở tệp để đọc **ifstream** thì cú pháp là:
`<Tên biến tệp>.tellg();`
- Nếu biến tệp là kiểu mở tệp để ghi **ofstream** thì cú pháp là:
`<Tên biến tệp>.telli();`

Chương trình 4.6a minh họa việc xác định vị trí hiện thời của con trỏ tệp sau một số thao tác đọc tệp trước đó.

Chương trình 4.6a

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

const int length = 25; // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], output;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin
```

Chương 4: Vào ra trên tệp

```
/* Mở tệp tin */
ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin
if(!fileIn){ // Không mở được tệp
    cout << "Khong the mo duoc tep tin " << fileName << endl;
    exit(1);
}

/* Đọc dữ liệu từ tệp tin ra màn hình
 * Ghi vị trí con trỏ tệp ra màn hình cứ sau 5 lần đọc kí tự */
int index = 0;
while(fileIn){
    fileIn >> output; // Đọc kí tự từ tệp tin
    cout << output; // Ghi kí tự ra màn hình
    if(index % 5 == 0) // Ghi ra vị trí con trỏ tệp
        cout << endl << "Vi tri con tro tep: "
        << fileIn.tellg() << endl;
    index++;
}
cout << endl; // Xuống dòng trên màn hình

/* Đóng tệp tin */
fileIn.close(); // Đóng tệp tin
return;
}
```

Chương trình 4.6b minh họa việc xác định vị trí hiện thời của con trỏ tệp sau một số thao tác ghi vào tệp trước đó.

Chương trình 4.6b

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

const int length = 25; // Độ dài tối đa tên tệp tin

void main(){
    clrscr();
    char fileName[length], input;
    cout << "Ten tep tin: ";
    cin >> setw(length) >> fileName; // Nhập tên tệp tin
```

```
/* Mở tệp tin */
ofstream fileOut(fileName, ios::out); // Khai báo và mở tệp tin
if(!fileOut){ // Không mở được tệp
    cout << "Khong the tao duoc tep tin " << fileName << endl;
    exit(1);
}

/* Ghi dữ liệu vào tệp tin
 * Hiện ra màn hình vị trí con trỏ tệp sau khi ghi được 5 kí tự*/
int index = 0;
do{
    cin >> input; // Đọc kí tự từ bàn phím
    fileOut << input << ' ';
    if(index%5 == 0) // Hiển thị vị trí con trỏ tệp
        cout << "Vi tri con tro tep: "
        << fileOut.tellp() << endl;
    index++;
}while((input != 'e') && (fileOut));
fileOut << endl; // Xuống dòng cuối tệp tin

/* Đóng tệp tin */
fileOut.close(); // Đóng tệp tin
return;
}
```

4.3.3 Dịch chuyển con trỏ tệp

Ngoài việc xác định vị trí hiện thời của con trỏ tệp, C++ còn cho phép dịch chuyển con trỏ tệp đến một vị trí bất kỳ trên tệp. Cú pháp dịch chuyển phụ thuộc vào kiểu biến tệp là đọc hay ghi.

- Nếu biến tệp có kiểu là mở tệp tin để đọc **ifstream**, cú pháp sẽ là:

<Tên biến tệp>.seekg(<Kích thước>, <Mốc dịch chuyển>);

- Nếu biến tệp có kiểu là mở tệp để ghi **ofstream**, cú pháp sẽ là:

<Tên biến tệp>.seekp(<Kích thước>, <Mốc dịch chuyển>);

Trong đó:

- **Kích thước**: là tham số mô tả khoảng cách dịch chuyển so với vị trí mốc dịch chuyển. Đơn vị tính của kích thước là byte, có kiểu là số nguyên.
- **Mốc dịch chuyển**: là vị trí gốc để xác định khoảng cách dịch chuyển của con trỏ tệp. Có ba tham số hằng về kiểu mốc dịch chuyển:
 - **ios::beg**: Mốc dịch chuyển là đầu tệp tin.
 - **ios::cur**: Mốc dịch chuyển là vị trí hiện thời của con trỏ tệp.
 - **ios::end**: Mốc dịch chuyển là vị trí cuối cùng của tệp tin.

Ví dụ:

```
ifstream fileIn("abc.txt", ios::in); fileIn.seekg(sizeof(char)*7,  
ios::beg);
```

sẽ dịch chuyển con trỏ tệp tin đến kí tự (kiểu char) thứ $7+1 = 8$ trong tệp tin abc.txt để đọc (giả sử tệp tin abc.txt lưu các kí tự kiểu char).

Lưu ý:

- Vì khoảng cách dịch chuyển có kiểu số nguyên (int) cho nên có thể nhận giá trị âm hoặc dương. Nếu giá trị dương, dịch chuyển về phía sau vị trí làm mốc, nếu giá trị âm, dịch chuyển về phía trước vị trí làm mốc.
- Nếu vị trí dịch chuyển đến nằm ngoài phạm vi tệp tin (phía sau vị trí cuối cùng của tệp hoặc phía trước vị trí đầu tiên của tệp) sẽ nảy sinh lỗi, khi đó <Tên biến tệp> = false.

Chương trình 4.7 cài đặt chương trình truy nhập tệp tin trực tiếp để đọc giá trị kí tự (kiểu char) trong tệp:

- Tên tệp tin (chứa dữ liệu kiểu char) do người dùng nhập vào từ bàn phím.
- Sau đó, mỗi khi người dùng nhập vào một số nguyên, chương trình sẽ dịch chuyển đến vị trí mới, cách vị trí cũ đúng bằng từng ấy kí tự, tính từ vị trí hiện thời của con trỏ tệp.
- Chương trình sẽ kết thúc khi người dùng nhập vào số 0.

Chương trình 4.7

```
#include<stdlib.h>  
#include<iostream.h>  
#include<fstream.h>  
#include<conio.h>  
  
const int length = 25; // Độ dài tối đa tên tệp tin  
  
void main(){  
    clrscr();  
    char fileName[length], output;  
    cout << "Ten tep tin: ";  
    cin >> setw(length) >> fileName; // Nhập tên tệp tin  
  
    /* Mở tệp tin */  
    ifstream fileIn(fileName, ios::in); // Khai báo và mở tệp tin  
    if(!fileIn){ // Không mở được tệp  
        cout << "Khong the mo duoc tep tin " << fileName << endl;  
        exit(1);  
    }  
  
    /* Đọc dữ liệu từ tệp tin ra màn hình  
     * Ghi vị trí con trỏ tệp ra màn hình cứ sau 5 lần đọc kí tự */
```

```
int index = 1;
do{
    cout << "So ki tu dich chuyen: ";
    cin >> index;

    // Dịch chuyển con trỏ tệp từ vị trí hiện thời
    fileIn.seekg(sizeof(char)*index, ios::cur);
    if(fileIn){           // Đúng
        fileIn >> output;      // Đọc kí tự từ tệp tin

        // Ghi kí tự ra màn hình
        cout << "Vi tri: " << fileIn.tellg() << output;
    }else{// Ra khỏi phạm vi tệp
        fileIn.clear();      // Về vị trí đầu tệp
    }
}while(index);

/* Đóng tệp tin */
fileIn.close();           // Đóng tệp tin
return;
}
```

TỔNG KẾT CHƯƠNG 4

Nội dung chương 4 đã tập trung trình bày các vấn đề liên quan đến các thao tác trên tệp tin trong ngôn ngữ C++. Bao gồm:

- Các bước tuần tự khi thao tác với một tệp tin:
 - Mở tệp tin
 - Đọc/ghi dữ liệu trên tệp tin
 - Đóng tệp tin
- Thao tác mở tệp tin với nhiều chế độ bằng kiểu fstream.
- Thao tác mở tệp tin chỉ để đọc với kiểu ifstream
- Thao tác mở tệp tin chỉ để ghi với thao tác ofstream.
- Đọc dữ liệu từ tệp tin văn bản với thao tác “>>”.
- Ghi dữ liệu vào tệp tin văn bản bằng thao tác “<<”.
- Đọc tệp tin nhị phân bằng thao tác read().
- Ghi vào tệp tin nhị phân bằng thao tác write().
- Xác định vị trí hiện thời của con trỏ tệp tin với các thao tác tellg() và tellp().
- Dịch chuyển vị trí của con trỏ tệp với các thao tác seekg() và seekp().
- Thiết lập lại trạng thái cho con trỏ tệp bằng thao tác clear().

- Đóng tệp tin đã sử dụng bằng thao tác close().

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 4

- Muốn mở một tệp tin tên là abc.txt để đọc dữ liệu, lệnh mở tệp nào sau đây là đúng:
 - fstream myFile("abc.txt", ios::in);
 - fstream myFile("abc.txt", ios::out);
 - fstream myFile("abc.txt", ios::app);
 - fstream myFile("abc.txt", ios::ate);
- Muốn mở một tệp tin abc.txt nằm trong thư mục xyz để ghi dữ liệu vào. Lệnh mở nào sau đây là đúng:
 - fstream myFile("xyz\abc.txt", ios::out);
 - fstream myFile("xyz\\abc.txt", ios::out);
 - fstream myFile("xyz/abc.txt", ios::out);
 - fstream myFile("xyz//abc.txt", ios::out);
- Muốn mở một tệp tin abc.txt để ghi thêm dữ liệu vào cuối tệp, lệnh nào sau đây là đúng:
 - fstream myFile("abc.txt", ios::out);
 - fstream myFile("abc.txt", ios::app);
 - fstream myFile("abc.txt", ios::out|ios::app);
 - fstream myFile("abc.txt", ios::out|ios::app);
- Xét hai lệnh khai báo sau:

```
fstream myFile1("abc.txt", ios::out); ofstream  
myFile2("abc.txt", ios::out);
```

Nhận định nào sau đây là đúng:
 - myFile1 và myFile2 có chức năng giống nhau.
 - myFile1 và myFile2 có chức năng khác nhau
- Xét hai lệnh khai báo sau:

```
fstream myFile1("abc.txt", ios::in); ifstream  
myFile2("abc.txt", ios::in);
```

Nhận định nào sau đây là đúng:
 - myFile1 và myFile2 có chức năng giống nhau.
 - myFile1 và myFile2 có chức năng khác nhau
- Xét đoạn chương trình sau:

```
ofstream myFile("abc.txt", ios::out); if(myFile) myFile <<  
"abc.txt";
```

Chương trình sẽ làm gì?
 - Ghi ra màn hình dòng chữ "abc.txt"
 - Ghi vào tệp tin abc.txt dòng chữ "abc.txt"
 - Đọc từ tệp tin abc.txt dòng chữ "abc.txt"

d. Chương trình sẽ báo lỗi.

7. Xét đoạn chương trình sau:

```
ifstream myFile("abc.txt", ios::in); char text[20];
if(myFile) myFile >> text;
```

Chương trình sẽ làm gì, nếu tệp tin abc.txt có nội dung là dòng chữ “abc.txt”?

- a. Ghi ra màn hình dòng chữ “abc.txt”
- b. Ghi vào tệp tin abc.txt dòng chữ “abc.txt”
- c. Đọc từ tệp tin abc.txt dòng chữ “abc.txt”
- d. Chương trình sẽ báo lỗi.

8. Xét đoạn chương trình sau:

```
fstream myFile("abc.txt", ios::out); if(myFile)
myFile << "abc.txt"; myFile.close();
myFile.open("abc.txt", ios::in); char text[20];
if(myFile) myFile >> text; cout << text;
```

Chương trình sẽ làm gì, nếu tệp tin abc.txt có nội dung là dòng chữ “abc.txt”?

- a. Ghi vào tệp tin abc.txt dòng chữ “abc.txt”
- b. Đọc từ tệp tin abc.txt dòng chữ “abc.txt”
- c. Ghi ra màn hình dòng chữ “abc.txt”
- d. Cả ba đáp án trên.
- e. Chương trình sẽ báo lỗi.

9. Xét đoạn chương trình sau:

```
ifstream myFile("abc.txt", ios::in); if(myFile) cout
<< myFile.tellg();
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0
- b. 1
- c. 8
- d. 16

10. Xét đoạn chương trình sau, nếu tệp abc.txt chứa một số lượng ký tự đủ lớn:

```
ifstream myFile("abc.txt", ios::in); if(myFile){
    char c; myFile
    >> c;
    cout << myFile.tellg();
}
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0

- b. 1
- c. 8
- d. 16

11. Xét đoạn chương trình sau, nếu tệp abc.txt chứa một số lượng kí tự đủ lớn:

```
ifstream myFile("abc.txt", ios::in); if(myFile){  
    myFile.seekg(sizeof(char)*5, ios::beg);  
    myFile.seekg(sizeof(char)*5, ios::cur); cout <<  
    myFile.tellg();  
}
```

Chương trình sẽ in ra màn hình kết quả gì?

- a. 0
- b. 5
- c. 10
- d. 80

12. Viết một chương trình gộp nội dung của hai tệp tin có sẵn vào một tệp tin thứ ba. Tên các tệp tin được nhập vào từ bàn phím.
13. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có tên là X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
14. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có năm sinh là X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
15. Viết một chương trình tìm kiếm trên tệp nhị phân có cấu trúc được tạo bởi chương trình 4.4: Tìm tất cả các nhân viên có lương cao hơn hoặc bằng một giá trị X, X được nhập từ bàn phím. Hiển thị kết quả là tất cả các thông tin về các nhân viên được tìm thấy.
16. Viết một chương trình sao chép một đoạn đầu nội dung của một tệp tin vào một tệp tin thứ hai. Tên các tệp tin và độ dài đoạn nội dung cần sao chép được nhập từ bàn phím.
17. Viết một chương trình sao chép một đoạn cuối nội dung của một tệp tin vào một tệp tin thứ hai. Tên các tệp tin và độ dài đoạn nội dung cần sao chép được nhập từ bàn phím.

CHƯƠNG 5

LỚP

Nội dung chương này tập trung trình bày các vấn đề liên quan đến lớp đối tượng trong C++:

- Khái niệm, khai báo và sử dụng lớp
- Khai báo và sử dụng các thành phần của lớp: các thuộc tính và các phương thức của lớp
- Phạm vi truy nhập lớp
- Khai báo và sử dụng các phương thức khởi tạo và huỷ bỏ của lớp
- Sử dụng lớp thông qua con trỏ đối tượng, mảng các đối tượng.

5.1 KHÁI NIỆM LỚP ĐỐI TƯỢNG

C++ coi lớp là sự trừu tượng hóa các đối tượng, là một khuôn mẫu để biểu diễn các đối tượng thông qua các thuộc tính và các hành động đặc trưng của đối tượng.

5.1.1 Định nghĩa lớp đối tượng

Để định nghĩa một lớp trong C++, ta dùng từ khóa **class** với cú pháp:

```
class <Tên lớp>{  
};
```

Trong đó:

- **class**: là tên từ khóa bắt buộc để định nghĩa một lớp đối tượng trong C++.
- **Tên lớp**: do người dùng tự định nghĩa. Tên lớp có tính chất như tên kiểu dữ liệu để sử dụng sau này. Cách đặt tên lớp phải tuân thủ theo quy tắc đặt tên biến trong C++.

Ví dụ:

```
class Car{  
};
```

là định nghĩa một lớp xe ô tô (Car). Lớp này chưa có bất kì một thành phần nào, việc định nghĩa các thành phần cho lớp sẽ được trình bày trong mục 5.2.

Lưu ý:

- Từ khóa **class** là bắt buộc để định nghĩa một lớp đối tượng trong C++. Hơn nữa, C++ có phân biệt chữ hoa chữ thường trong khai báo cho nên chữ **class** phải được viết bằng chữ thường.

Ví dụ:

```
class Car{ // Định nghĩa đúng  
};
```

nhưng:

```
Class Car{ // Lỗi từ khóa  
};
```

- Bắt buộc phải có dấu chấm phẩy “;” ở cuối định nghĩa lớp vì C++ coi định nghĩa một lớp như định nghĩa một kiểu dữ liệu, cho nên phải có dấu chấm phẩy cuối định nghĩa (tương tự định nghĩa kiểu dữ liệu kiểu cấu trúc).
- Để phân biệt với tên biến thông thường, ta nên (nhưng không bắt buộc) đặt *tên lớp bắt đầu bằng một chữ in hoa và các tên biến bắt đầu bằng một chữ in thường*.

5.1.2 Sử dụng lớp đối tượng

Lớp đối tượng được sử dụng khi ta khai báo các thể hiện của lớp đó. Một thể hiện của một lớp chính là một đối tượng cụ thể của lớp đó. Việc khai báo một thể hiện của một lớp được thực hiện như cú pháp khai báo một biến có kiểu lớp:

<Tên lớp> <Tên biến lớp>;

Trong đó:

- **Tên lớp:** là tên lớp đối tượng đã được định nghĩa trước khi khai báo biến.
- **Tên biến lớp:** là tên đối tượng cụ thể. Tên biến lớp sẽ được sử dụng như các biến thông thường trong C++, ngoại trừ việc nó có kiểu lớp đối tượng.

Ví dụ, muốn khai báo một thể hiện (biến) của lớp Car đã được định nghĩa trong mục 5.1.1, ta khai báo như sau:

Car myCar;

Sau đó, ta có thể sử dụng biến myCar trong chương trình như các biến thông thường: truyền tham số cho hàm, gán cho biến khác ...

Lưu ý:

- Khi khai báo biến lớp, ta không dùng lại từ khóa **class** nữa. Từ khóa **class** chỉ được sử dụng khi định nghĩa lớp mà không dùng khi khai báo biến lớp.

Ví dụ, khai báo:

Car myCar; // đúng

là đúng, nhưng khai báo:

class Car myCar; // Lỗi cú pháp

là sai cú pháp.

5.2 CÁC THÀNH PHẦN CỦA LỚP

Việc khai báo các thành phần của lớp có dạng như sau:

```
class <Tên lớp>{ private:  
    <Khai báo các thành phần riêng> protected:  
    <Khai báo các thành phần được bảo vệ> public:  
    <Khai báo các thành phần công cộng>  
};
```

Trong đó:

- **private:** là từ khóa chỉ tính chất của C++ để chỉ ra rằng các thành phần được khai báo trong phạm vi từ khóa này là riêng tư đối với lớp đối tượng. Các đối tượng của các lớp khác không truy nhập được các thành phần này.
- **protected:** các thành phần được khai báo trong phạm vi từ khóa này đều được bảo vệ. Qui định loại đối tượng nào được truy nhập đến các thành phần được bảo vệ sẽ được mô tả chi tiết trong mục 5.3.
- **public:** các thành phần công cộng. Các đối tượng của các lớp khác đều có thể truy nhập đến các thành phần công cộng của một đối tượng bất kì.

Các thành phần của lớp được chia làm hai loại:

- Các thành phần chỉ dữ liệu của lớp, được gọi là *thuộc tính* của lớp
- Các thành phần chỉ hành động của lớp, được gọi là *phương thức* của lớp.

5.2.1 Thuộc tính của lớp

Khai báo thuộc tính

Thuộc tính của lớp là thành phần chứa dữ liệu, đặc trưng cho các tính chất của lớp. Thuộc tính của lớp được khai báo theo cú pháp sau:

<Kiểu dữ liệu> <Tên thuộc tính>;

Trong đó:

- **Kiểu dữ liệu:** có thể là các kiểu dữ liệu cơ bản của C++, cũng có thể là các kiểu dữ liệu phức tạp do người dùng tự định nghĩa như struct, hoặc kiểu là một lớp đã được định nghĩa trước đó.
- **Tên thuộc tính:** là tên thuộc tính của lớp, có tính chất như một biến thông thường. Tên thuộc tính phải tuân theo quy tắc đặt tên biến của C++.

Ví dụ, khai báo:

```
class Car{  
    private:  
        int speed; public:  
        char mark[20];  
    };
```

là khai báo một lớp xe ô tô (Car), có hai thuộc tính: thuộc tính tốc độ (speed) có tính chất private, thuộc tính nhãn hiệu xe (mark) có tính chất public.

Lưu ý:

- Không được khởi tạo giá trị ban đầu cho các thuộc tính ngay trong lớp. Vì các thuộc tính chỉ có giá trị khi nó gắn với một đối tượng cụ thể, là một thể hiện (biến) của lớp.

Ví dụ:

```
class Car{ private:  
    int speed;           // đúng  
    int weight = 500;   // lỗi
```

};

- Khả năng truy nhập thuộc tính của lớp là phụ thuộc vào thuộc tính áy được khai báo trong phạm vi của từ khóa nào: private, protected hay public.
- Thông thường, do yêu cầu đóng gói dữ liệu của hướng đối tượng, ta nên khai báo các thuộc tính có tính chất riêng tư (private). Nếu muốn các đối tượng khác truy nhập được vào các thuộc tính này, ta xây dựng các hàm public truy nhập (get / set) đến thuộc tính đó.

Sử dụng thuộc tính

Thuộc tính có thể được sử dụng cho các chương trình nằm ngoài lớp thông qua tên biến lớp hoặc sử dụng ngay trong lớp bởi các phương thức của lớp.

- Nếu thuộc tính được dùng bên ngoài phạm vi lớp, cú pháp phải thông qua tên biến lớp (cách này chỉ sử dụng được với các biến có tính chất public):
`<Tên biến lớp>.<tên thuộc tính>;`
- Nếu thuộc tính được dùng bên trong lớp, cú pháp đơn giản hơn:
`<Tên thuộc tính>;`

Ví dụ, với định nghĩa lớp:

```
class Car{ private:  
    int speed; public:  
    char mark[20];  
};
```

ta khai báo một biến lớp:

```
Car myCar;
```

Thì có thể sử dụng thuộc tính nhãn hiệu xe khi in ra màn hình như sau:

```
cout << myCar.mark;
```

Lưu ý:

- Khi dùng thuộc tính bên trong các phương thức của lớp, mà tên thuộc tính lại bị trùng với tên biến toàn cục (tự do) của chương trình, ta phải chỉ rõ việc dùng tên thuộc tính của lớp (mà không phải tên biến toàn cục) bằng cách dùng chỉ thị phạm vi lớp “::” với cú pháp:

```
<Tên lớp>::<Tên thuộc tính>;
```

5.2.2 Phương thức của lớp

Khai báo khuôn mẫu phương thức

Một phương thức là một thao tác thực hiện một số hành động đặc trưng của lớp đối tượng. Phương thức được khai báo tương tự như các hàm trong C++:

```
<Kiểu trả về> <Tên phương thức>([<Các tham số>]);
```

Trong đó:

- **Kiểu trả về:** là kiểu dữ liệu trả về của phương thức. Kiểu có thể là các kiểu dữ liệu cơ bản của C++, cũng có thể là kiểu do người dùng định nghĩa, hoặc kiểu lớp đã được định nghĩa.

- **Tên phương thức:** do người dùng tự đặt tên, tuân theo quy tắc đặt tên biến của C++.
- **Các tham số:** Các tham số đầu vào của phương thức, được biểu diễn bằng kiểu dữ liệu tương ứng. Các tham số được phân cách bởi dấu phẩy “,”. Các tham số là tùy chọn (Phần trong dấu ngoặc vuông “[]” là tùy chọn).

Ví dụ, khai báo:

```
class Car{  
private:  
    int speed; char  
    mark[20];  
public:  
    void show();  
};
```

là định nghĩa một lớp Car có hai thuộc tính cục bộ là speed và mark, và khai báo một phương thức show() để mô tả đối tượng xe tương ứng. Show() là một phương thức không cần tham số và kiểu trả về là void.

Lưu ý:

- Khả năng truy nhập phương thức từ bên ngoài là phụ thuộc vào phương thức được khai báo trong phạm vi của từ khóa nào: private, protected hay public.

Định nghĩa phương thức

Trong C++, việc cài đặt chi tiết nội dung của phương thức có thể tiến hành ngay trong phạm vi lớp hoặc bên ngoài phạm vi định nghĩa lớp. Cú pháp chỉ khác nhau ở dòng khai báo tên phương thức.

- Nếu cài đặt phương thức ngay trong phạm vi định nghĩa lớp, cú pháp là:

```
<Kiểu trả về> <Tên phương thức>([<Các tham số>]) {  
    ... // Cài đặt chi tiết  
}
```
- Nếu cài đặt phương thức bên ngoài phạm vi định nghĩa lớp, ta phải dùng chỉ thị phạm vi “::” để chỉ ra rằng đây là một phương thức của lớp mà không phải là một hàm tự do trong chương trình:

```
<Kiểu trả về> <Tên lớp>::<Tên phương thức>([<Các tham số>]) {  
    ... // Cài đặt chi tiết  
}
```

Ví dụ, nếu cài đặt phương thức show() của lớp Car ngay trong phạm vi định nghĩa lớp, ta cài đặt như sau:

```
class Car{  
private:  
    int      speed;          // Tốc độ  
    char    mark[20];         // Nhãn hiệu  
public:  
    void show() {            // Khai báo phương thức ngay trong lớp cout << "This  
                                is a " << mark << " having a speed of "
```

```

        << speed << "km/h!" << endl; return;
    }
};


```

Nếu muốn cài đặt bên ngoài lớp, ta cài đặt như sau:

```

class Car{ private:
    int      speed;           // Tốc độ
    char    mark[20];         // Nhãn hiệu

public:
    void show();             // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */ void
Car::show(){
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h!" << endl; return;
}

```

Lưu ý:

- Nếu phương thức được cài đặt ngay trong lớp thì các tham số phải tường minh, nghĩa là mỗi tham số phải được biểu diễn bằng một cặp <Kiểu dữ liệu> <Tên tham số> như khi cài đặt chi tiết một hàm tự do trong chương trình.
- Thông thường, chỉ các phương thức ngắn (trên một dòng) là nên cài đặt ngay trong lớp. Còn lại nên cài đặt các phương thức bên ngoài lớp để chương trình được sáng sủa, rõ ràng và dễ theo dõi.

Sử dụng phương thức

Cũng tương tự như các thuộc tính của lớp, các phương thức cũng có thể được sử dụng bên ngoài lớp thông qua tên biến lớp, hoặc có thể được dùng ngay trong lớp bởi các phương thức khác của lớp định nghĩa nó.

- Nếu phương thức được dùng bên ngoài phạm vi lớp, cú pháp phải thông qua tên biến lớp (cách này chỉ sử dụng được với các phương thức có tính chất public):

 <Tên biến lớp>.<Tên phương thức>([<Các đối số>]);
- Nếu thuộc tính được dùng bên trong lớp, cú pháp đơn giản hơn:

 <Tên phương thức>([<Các đối số>]);

Ví dụ, với định nghĩa lớp:

```

class Car{ private:
    int      speed;           // Tốc độ
    char    mark[20];         // Nhãn hiệu

public:
    void show();             // Giới thiệu xe
}

```

```
};

/* Khai báo phương thức bên ngoài lớp */ void
Car::show(){
    cout << "This is a " << mark << " having a speed of "
    << speed << "km/h!" << endl; return;
}
```

ta khai báo một biến lớp:

```
Car myCar;
```

Thì có thể sử dụng phương thức giới thiệu xe như sau:

```
myCar.show();
```

Lưu ý:

- Khi dùng phương thức bên trong các phương thức khác của lớp, mà phương thức lại bị trùng với các phương thức tự do của chương trình, ta phải chỉ rõ việc dùng phương thức của lớp (mà không phải dùng phương thức tự do) bằng cách dùng chỉ thị phạm vi lớp “::” với cú pháp:

```
<Tên lớp>::<Tên phương thức>([<Các đối số>]);
```

Chương trình 5.1 cài đặt đầy đủ một lớp xe ô tô (Car) với các thuộc tính có tính chất cục bộ:

- Tốc độ xe (speed)
- Nhãn hiệu xe (mark)
- Giá xe (price)

Và các phương thức có tính chất public:

- Khởi tạo các tham số (init)
- Giới thiệu xe (show)
- Các phương thức truy nhập (get/set) các thuộc tính

Sau đó, chương trình main sẽ sử dụng lớp Car này để định nghĩa các đối tượng cụ thể và sử dụng các phương thức của lớp này.

Chương trình 5.1

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp */
class Car{
private:
    int speed; // Tốc độ
    char mark[20]; // Nhãn hiệu
    float price; // Giá xe
public:
    void setSpeed(int); // Gán tốc độ cho xe
```

Chương 5: Lớp

```
int      getSpeed();           // Đọc tốc độ xe
void     setMark(char);       // Gán nhãn cho xe
char[]   getMark();          // Đọc nhãn xe void
          setPrice(float);    // Gán giá cho xe float
getPrice();                  // Đọc giá xe
void     init(int, char[], float); // Khởi tạo thông tin về xe void     show(); //
Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
void Car::setSpeed(int speedIn){           // Gán tốc độ cho xe speed
    = speedIn;
}
int Car::getSpeed(){                      // Đọc tốc độ xe return
    speed;
}
void Car::setMark(char markIn){           // Gán nhãn cho xe
    strcpy(mark, markIn);
}
char[] Car::getMark(){                   // Đọc nhãn xe
    return mark;
}
void Car::setPrice(float priceIn){        // Gán giá cho xe price
    = priceIn;
}
float Car::getPrice(){                  // Đọc giá xe
    return price;
}

void Car::init(int speedIn, char markIn[], float priceIn){ speed = speedIn;
    strcpy(mark, markIn); price =
    priceIn; return;
}

void Car::show(){                       // Phương thức giới thiệu xe cout <<
    "This is a " << mark << " having a speed of "
    << speed << "km/h and its price is $" << price << endl; return;
}

// Hàm main, chương trình chính
```

```
void main() {
    clrscr();
    Car myCar; // Khai báo biến lớp

    // Khởi tạo lần thứ nhất
    cout << "Xe thu nhat: " << endl;
    myCar.init(100, "Ford", 3000);
    cout << "Toc do (km/h): " << myCar.getSpeed() << endl;
    cout << "Nhan hieu : " << myCar.getMark() << endl;
    cout << "Gia ($) : " << myCar.getPrice() << endl;

    // Thay đổi thuộc tính xe
    cout << "Xe thu hai: " << endl;
    myCar.setSpeed(150);
    myCar.setMark("Mercedes");
    myCar.setPrice(5000);
    myCar.show();
    return;
}
```

Chương trình 5.1 sẽ in ra kết quả như sau:

```
Xe thu nhat:  
Toc do (km/h): 100  
Nhan hieu: Ford Gia  
($): 3000  
Xe thu hai:  
This is a Mercedes having a speed of 150km/h and its price is $5000
```

5.3 PHẠM VI TRUY NHẬP LỚP

5.3.1 Phạm vi truy nhập lớp

Trong C++, có một số khái niệm về phạm vi, xếp từ bé đến lớn như sau:

- **Phạm vi khối lệnh:** Trong phạm vi giữa hai dấu giới hạn “{}” của một khối lệnh. Ví dụ các lệnh trong khối lệnh lặp while(){} sẽ có cùng phạm vi khối lệnh.
- **Phạm vi hàm:** Các lệnh trong cùng một hàm có cùng mức phạm vi hàm.
- **Phạm vi lớp:** Các thành phần của cùng một lớp có cùng phạm vi lớp với nhau: các thuộc tính và các phương thức của cùng một lớp.
- **Phạm vi chương trình (còn gọi là phạm vi tệp):** Các lớp, các hàm, các biến được khai báo và định nghĩa trong cùng một tệp chương trình thì có cùng phạm vi chương trình.

Trong phạm vi truy nhập lớp, ta chỉ quan tâm đến hai phạm vi lớn nhất, đó là phạm vi lớp và phạm vi chương trình. Trong C++, phạm vi truy nhập lớp được quy định bởi các từ khóa về thuộc tính truy nhập:

- **private:** Các thành phần của lớp có thuộc tính private thì chỉ có thể được truy nhập trong phạm vi lớp.
- **protected:** Trong cùng một lớp, thuộc tính protected cũng có ảnh hưởng tương tự như thuộc tính private: các thành phần lớp có thuộc tính protected chỉ có thể được truy nhập trong phạm vi lớp. Ngoài ra nó còn có thể được truy nhập trong các lớp con khi có kế thừa (sẽ được trình bày trong chương 6).
- **public:** các thành phần lớp có thuộc tính public thì có thể được truy nhập trong phạm vi chương trình, có nghĩa là nó có thể được truy nhập trong các hàm tự do, các phương thức bên trong các lớp khác...

Ví dụ, thuộc tính price của lớp Car có tính chất private nên chỉ có thể truy nhập bởi các phương thức của lớp Car. Không thể truy nhập từ bên ngoài lớp (phạm vi chương trình), chẳng hạn trong một hàm tự do ngoài lớp Car.

```
void Car::setPrice(float priceIn){
    price = priceIn;           // Đúng, vì setPrice là một phương thức
                                // của lớp Car
}
```

nhưng:

```
void freeFunction(Car myCar){
    myCar.price = 3000; // Lỗi, vì freeFunction là một hàm tự do
                        // nằm ngoài phạm vi lớp Car
}
```

Khi đó, hàm freeFunction phải truy nhập gián tiếp đến thuộc tính price thông qua phương thức truy nhập có tính chất public như sau:

```
void freeFunction(Car myCar){
    myCar.setPrice(3000); // Đúng, vì setPrice là một phương thức
                          // của
                          // lớp Car có thuộc tính public
}
```

Tuy nhiên, C++ cho phép một cách đặc biệt để truy nhập đến các thành phần private và protected của một lớp bằng khái niệm *hàm bạn* và *lớp bạn* của một lớp: trong các hàm bạn và lớp bạn của một lớp, có thể truy nhập đến các thành phần private và protected như bên trong phạm vi lớp đó.

5.3.2 Hàm bạn

Có hai kiểu hàm bạn cơ bản trong C++:

- Một hàm tự do là hàm bạn của một lớp
- Một hàm thành phần (phương thức) của một lớp là bạn của một lớp khác

Ngoài ra còn có một số kiểu hàm bạn mở rộng từ hai kiểu này:

- Một hàm là bạn của nhiều lớp

- Tất cả các hàm của một lớp là bạn của lớp khác (lớp bạn)

Hàm tự do bạn của một lớp

Một hàm bạn của một lớp được khai báo bằng từ khóa **friend** khi khai báo khuôn mẫu hàm trong lớp tương ứng.

```
class <Tên lớp>{
    ...      // Khai báo các thành phần lớp như thông thường
    // Khai báo hàm bạn
    friend <Kiểu trả về> <Tên hàm bạn>(<Các tham số>);
}
```

Khi đó, định nghĩa chi tiết hàm bạn được thực hiện như định nghĩa một hàm tự do thông thường:

```
<Kiểu trả về> <Tên hàm bạn>(<Các tham số>){
    ...      // Có thể truy nhập trực tiếp các thành phần private
            // của lớp đã khai báo
}
```

Lưu ý:

- Mặc dù hàm bạn được khai báo khuôn mẫu hàm trong phạm vi lớp, nhưng hàm bạn tự do lại không phải là một phương thức của lớp. Nó là hàm tự do, việc định nghĩa và sử dụng hàm này hoàn toàn tương tự như các hàm tự do khác.
- Việc khai báo khuôn mẫu hàm bạn trong phạm vi lớp ở vị trí nào cũng được: hàm bạn không bị ảnh hưởng bởi các từ khóa private, protected hay public trong lớp.
- Trong hàm bạn, có thể truy nhập trực tiếp đến các thành phần private và protected của đối tượng có kiểu lớp mà nó làm bạn (truy nhập thông qua đối tượng cụ thể).

Chương trình 5.2 minh họa việc định nghĩa một hàm bạn của lớp Car, hàm này so sánh xem hai chiếc xe, chiếc nào đắt hơn.

Chương trình 5.2

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp */
class Car{
    private:
        int speed;                  // Tốc độ
        char mark[20];              // Nhãn hiệu
        float price;                // Giá xe
    public:
        void init(int, char[], float); // Khởi tạo thông tin về xe
        // Khai báo hàm bạn của lớp
        friend void moreExpensive(Car, Car);
};
```

```
/* Khai báo phương thức bên ngoài lớp */
void Car::init(int speedIn, char markIn[], float priceIn) {
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
    return;
}

/* Định nghĩa hàm bạn tự do */
void moreExpensive(Car car1, Car car2){
    if(car1.price > car2.price)//Truy nhập đến các thuộc tính private
        cout << "xe thu nhat dat hon" << endl;
    else if(car1.price < car2.price)
        cout << "xe thu nhat dat hon" << endl;
    else
        cout << "hai xe dat nhu nhau" << endl;
    return;
}

// Hàm main, chương trình chính
void main(){
    clrscr();
    Car car1, car2; // Khai báo biến lớp

    // Khởi tạo xe thứ nhất, thứ hai
    car1.init(100, "Ford", 3000);
    car2.init(150, "Mercedes", 3500);

    // So sánh giá hai xe
    moreExpensive(car1, car2); // Sử dụng hàm bạn tự do
    return;
}
```

Chương trình 5.2 sẽ in ra thông báo:

xe thu hai dat hon
vì xe thứ hai có giá \$3500, trong khi xe thứ nhất được khởi tạo giá là \$3000.

Phương thức lớp là bạn của một lớp khác

Trong C++, một phương thức của lớp này cũng có thể làm bạn của một lớp kia. Để khai báo một phương thức f của lớp B là bạn của lớp A và f nhận một tham số có kiểu lớp A, ta phải khai báo tuân tự như sau (trong cùng một chương trình):

- Khai báo khuôn mẫu lớp A, để làm tham số cho hàm f của lớp B:

```
class A;
```

- Khai báo lớp B với hàm f như khai báo các lớp thông thường:

```
class B{
```

```
    ...      // Khai báo các thành phần khác của lớp B void f(A);
```

```
};
```

- Khai báo chi tiết lớp A với hàm f của lớp B là bạn

```
class A{
```

```
    ...      // Khai báo các thành phần khác của lớp A friend void
```

```
        B::f(A);
```

```
};
```

- Định nghĩa chi tiết hàm f của lớp B:

```
void B::f(A){
```

```
    ...      // Định nghĩa chi tiết hàm f
```

```
}
```

Lưu ý:

- Trong trường hợp này, hàm f chỉ được định nghĩa chi tiết một khi lớp A đã được định nghĩa chi tiết. Do vậy, chỉ có thể định nghĩa chi tiết hàm f ngay trong lớp A (ở bước 3) hoặc sau khi định nghĩa lớp A (ở bước 4), mà không thể định nghĩa chi tiết hàm f ngay trong lớp B (ở bước 2).
- Hàm f có thể truy nhập đến các thành phần private và protected của cả hai lớp A và B. Tuy nhiên, muốn f truy nhập đến các thành phần của lớp A thì phải thông qua một đối tượng cụ thể có kiểu lớp A.

Chương trình 5.3 minh họa việc cài đặt và sử dụng một hàm permission() của lớp Person, là hàm bạn của lớp Car. Hàm này thực hiện việc kiểm tra xem một người có đủ quyền điều khiển xe hay không, theo luật sau:

- Với các loại xe thông thường, người điều khiển phải đủ 18 tuổi.
- Với các loại xe có tốc độ cao hơn 150km/h, người điều khiển phải đủ 21 tuổi.

Chương trình 5.3

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

class Car;                                // Khai báo nguyên mẫu lớp

/* Định nghĩa lớp Person */
class Person{
    private:
        char    name[25];                  // Tên
        int     age;                      // Tuổi
```

Chương 5: Lớp

```
public:
    void init(char[], int); // Khởi tạo thông tin về người int
    permission(Car); // Xác định quyền điều khiển xe
};

/* Định nghĩa lớp Car */
class Car{
private:
    int speed; // Tốc độ
    char mark[20]; // Nhãn hiệu
    float price; // Giá xe
public:
    void init(int, char[], float); // Khởi tạo thông tin về xe
    // Khai báo hàm bạn của lớp
    friend int Person::permission(Car);
};

/* Khai báo phương thức bên ngoài lớp */
void Person::init(char nameIn[], int ageIn){
    strcpy(name, nameIn); age =
    ageIn;
    return;
}

void Car::init(int speedIn, char markIn[], float priceIn){ speed = speedIn;
    strcpy(mark, markIn); price =
    priceIn; return;
}

/* Định nghĩa hàm bạn */
int Person::permission(Car car){ if(age < 18)
    return 0;
    // Truy nhập thuộc tính private thông qua đối tượng car if((age <
    21)&&(car.speed > 150))
    return 0;
    return 1;
}

// Hàm main, chương trình chính void
main(){
```

```
clrscr();
// Khai báo các biến lớp
Car car;
Person person;

// Khởi tạo các đối tượng
car.init(100, "Ford", 3000);
person.init("Vinh", 20);

// Xác định quyền điều khiển xe
if(person.permission(car))           // Sử dụng hàm bạn
    cout << "Co quyen dieu khien" << endl;
else
    cout << "Khong co quyen dieu khien" << endl;
return;
}
```

Chương trình 5.3 sẽ hiển thị thông báo:

Co quyen dieu khien

Vì người chủ xe đã 20 tuổi và xe chỉ có tốc độ 100km/h.

5.3.3 Lớp bạn

Khi tất cả các phương thức của một lớp là bạn của một lớp khác, thì lớp của các phương thức đó cũng trở thành lớp bạn của lớp kia. Muốn khai báo một lớp B là lớp bạn của lớp A, ta khai báo theo tuần tự sau:

- Khai báo khuôn mẫu lớp B:

```
class B;
```

- Định nghĩa lớp A, với khai báo B là lớp bạn:

```
class A{
    ...
    // Khai báo các thành phần của lớp A
    // Khai báo lớp bạn B friend
    class B;
};
```

- Định nghĩa chi tiết lớp B:

```
class B{
    ...
    // Khai báo các thành phần của lớp B
};
```

Lưu ý:

- Trong trường hợp này, lớp B là lớp bạn của lớp A nhưng không có nghĩa là lớp A cũng là bạn của lớp B: tất cả các phương thức của lớp B có thể truy nhập các thành phần private của lớp A (qua các đối tượng có kiểu lớp A) nhưng các phương thức của lớp A lại không thể truy nhập đến các thành phần private của lớp B.

- Muốn các phương thức của lớp A cũng truy nhập được đến các thành phần private của lớp B, thì phải khai báo thêm là lớp A cũng là lớp bạn của lớp B.

5.4 HÀM KHỞI TẠO VÀ HỦY BỎ

5.4.1 Hàm khởi tạo

Hàm khởi tạo được gọi mỗi khi khai báo một đối tượng của lớp. Ngay cả khi không được khai báo tường minh, C++ cũng gọi hàm khởi tạo ngầm định khi đối tượng được khai báo.

Khai báo hàm khởi tạo

Hàm khởi tạo của một lớp được khai báo tường minh theo cú pháp sau:

```
class <Tên lớp>{
    public:
        cTên LớpS([cCác tham sốS]); // Khai báo hàm khởi tạo
};
```

Ví dụ:

```
class Car{
    int      speed; char
            mark[20];
    float   price;
public:
    Car(int speedIn, char markIn[], float priceIn){ speed = speedIn;
        strcpy(mark, markIn); price =
        priceIn;
    }
};
```

là khai báo một hàm khởi tạo với ba tham số của lớp Car.

Lưu ý:

- Hàm khởi tạo phải có tên trùng với tên của lớp
- Hàm khởi tạo không có giá trị trả về
- Hàm khởi tạo có tính chất public
- Có thể có nhiều hàm khởi tạo của cùng một lớp

Sử dụng hàm khởi tạo của lớp

Hàm khởi tạo được sử dụng khi khai báo biến lớp. Khi đó, ta có thể vừa khai báo, vừa khởi tạo giá trị các thuộc tính của đối tượng lớp theo cú pháp sau:

```
<Tên lớp> <Tên đối tượng>(<Các đối số khởi tạo>);
```

Trong đó:

- **Tên lớp:** là tên kiểu lớp đã được định nghĩa
- **Tên đối tượng:** là tên biến có kiểu lớp, tuân thủ theo quy tắc đặt tên biến của C++

- **Các đối số khởi tạo:** Là các đối số tương ứng với hàm khởi tạo của lớp tương ứng. Tương tự như việc truyền đối số khi dùng các lời gọi hàm thông thường.

Ví dụ, nếu lớp Car có hàm khởi tạo Car(int, char[], float) thì khi khai báo biến có kiểu lớp Car, ta có thể sử dụng hàm khởi tạo này như sau:

```
Car myCar(100, "Ford", 3000);
```

Lưu ý:

- Khi sử dụng hàm khởi tạo, phải truyền đúng số lượng và đúng kiểu của các tham số của các hàm khởi tạo đã được định nghĩa của lớp.
- Khi một lớp đã có ít nhất một hàm khởi tạo tường minh, thì không được sử dụng hàm khởi tạo ngầm định của C++. Do đó, khi đã khai báo ít nhất một hàm khởi tạo, nếu muốn khai báo biến mà không cần tham số, lớp tương ứng phải có ít nhất một hàm khởi tạo không có tham số.

Ví dụ, nếu lớp Car chỉ có duy nhất một hàm khởi tạo như sau:

```
class Car{ public:  
    Car(int, char[], float);  
};
```

thì không thể khai báo một biến như sau:

```
Car myCar; // Khai báo lỗi
```

- Trong trường hợp dùng hàm khởi tạo không có tham số, ta không cần phải sử dụng cặp dấu ngoặc đơn “()” sau tên biến đối tượng. Việc khai báo trở thành cách khai báo thông thường.

Chương trình 5.4a minh họa việc định nghĩa và sử dụng lớp Car với hai hàm khởi tạo khác nhau.

Chương trình 5.4a

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp */  
class Car{  
private:  
    int speed; // Tốc độ  
    char mark[20]; // Nhãn hiệu  
    float price; // Giá xe  
public:  
    Car(); // Khởi tạo không tham số  
    Car(int, char[], float); // Khởi tạo đầy đủ tham số  
    void show(); // Giới thiệu xe  
};
```

Chương 5: Lớp

```
/* Khai báo phương thức bên ngoài lớp */
Car::Car() {                                     // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo có đầy đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn) {
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::show() {                                // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main() {
    clrscr();
    Car myCar1;                                    // Sử dụng hàm khởi tạo không tham số
    Car myCar2(150, "Mercedes", 5000); // Dùng hàm khởi tạo đầy đủ tham số

    // Giới thiệu xe thứ nhất
    cout << "Xe thu nhat: " << endl;
    myCar1.show();

    // Giới thiệu xe thứ hai
    cout << "Xe thu hai: " << endl;
    myCar2.show();
    return;
}
```

Chương trình 5.4a sẽ hiển thị các thông tin như sau:

Xe thu nhat:

This is a having a speed of 0km/h and its price is \$0 Xe thu hai:

This is a Mercedes having a speed of 150km/h and its price is \$5000

Lí do là xe thứ nhất sử dụng hàm khởi tạo không có tham số nên xe không có tên, tốc độ và giá đều là mặc định (0). Trong khi đó, xe thứ hai được khởi tạo đầy đủ cả ba tham số nên thông tin giới thiệu xe được đầy đủ.

Tuy nhiên, khi đối tượng có nhiều thuộc tính riêng, để tránh trường hợp phải định nghĩa nhiều hàm khởi tạo cho các trường hợp thiếu vắng một vài tham số khác nhau. Ta có thể sử dụng hàm khởi tạo với các giá trị khởi đầu ngầm định. Chương trình 5.4b cho kết quả hoàn toàn giống chương trình 5.4a, nhưng đơn giản hơn vì chỉ cần định nghĩa một hàm khởi tạo với các tham số có giá trị ngầm định.

Chương trình 5.4b

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp */
class Car{
    private:
        int speed;           // Tốc độ
        char mark[20];       // Nhãn hiệu
        float price;         // Giá xe
    public:
        // Khởi tạo với các giá trị ngầm định cho các tham số
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        void show();          // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

void Car::show() {                                // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main(){
    clrscr();
    Car myCar1;                                     // Các tham số nhận giá trị mặc định
```

```
Car myCar2(150, "Mercedes", 5000); // Dùng hàm khởi tạo đủ tham số

    // Giới thiệu xe thứ nhất
    cout << "Xe thu nhat: " << endl;
    myCar1.show();

    // Giới thiệu xe thứ hai
    cout << "Xe thu hai: " << endl;
    myCar2.show();
    return;
}
```

5.4.2 Hàm hủy bỏ

Hàm hủy bỏ được tự động gọi đến khi mà đối tượng được giải phóng khỏi bộ nhớ. Nhiệm vụ của hàm hủy bỏ là dọn dẹp bộ nhớ trước khi đối tượng bị giải phóng. Cú pháp khai báo hàm hủy bỏ như sau:

```
class <Tên lớp>{
    public:
        -cTên LớpS([cCác tham số]); // Khai báo hàm khởi tạo
};
```

Ví dụ:

```
class Car{
    int speed; char
    *mark; float
    price;
public:
    ~Car(){
        delete [] mark;
    };
};
```

là khai báo một hàm hủy bỏ của lớp Car với thuộc tính mark có kiểu con trỏ. Hàm này sẽ giải phóng vùng nhớ đã cấp phát cho con trỏ kiểu char của thuộc tính nhãn hiệu xe.

Lưu ý:

- Hàm hủy bỏ phải có tên bắt đầu bằng dấu “~”, theo sau là tên của lớp tương ứng.
- Hàm hủy bỏ không có giá trị trả về.
- Hàm hủy bỏ phải có tính chất public
- Mỗi lớp chỉ có nhiều nhất một hàm hủy bỏ. Trong trường hợp không khai báo tường minh hàm hủy bỏ, C++ sẽ sử dụng hàm hủy bỏ ngầm định.

- Nói chung, khi có ít nhất một trong các thuộc tính của lớp là con trỏ, thì nên dùng hàm hủy bỏ tường minh để giải phóng triệt để các vùng nhớ của các thuộc tính, trước khi đối tượng bị giải phóng khỏi bộ nhớ.

Chương trình 5.5 minh họa việc định nghĩa lớp Car với một hàm khởi tạo có các tham số với giá trị mặc định và một hàm hủy bỏ tường minh.

Chương trình 5.5

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp */
class Car{
    private:
        int speed;           // Tốc độ
        char *mark;          // Nhãn hiệu
        float price;         // Giá xe
    public:
        // Khởi tạo với các giá trị ngầm định cho các tham số
        Car(int speedIn=0, char *markIn=NULL, float priceIn=0);
        void show();          // Giới thiệu xe
        ~Car();              // Hàm hủy bỏ tường minh
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char *markIn, float priceIn){
    speed = speedIn;
    mark = markIn;
    price = priceIn;
}

void Car::show() {                                // Phương thức giới thiệu xe
    cout << "This is a " << *mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

Car::~Car() {                                     // Hàm hủy bỏ tường minh
    delete [] mark;
    cout << "The object has been destroyed!" << endl;
}
```

```
// Hàm main, chương trình chính
void main() {
    clrscr();
    Car myCar(150, "Mercedes", 5000); // Dùng hàm khởi tạo đủ tham số

    // Giới thiệu xe
    cout << "Gioi thieu xe: " << endl;
    myCar.show();
    return;
}
```

Chương trình 5.5 sẽ in ra thông báo như sau:

```
Gioi thieu xe:
This is a Mercedes having a speed of 150km/h and its price is $5000 The object has
been destroyed!
```

Dòng cuối cùng là của hàm hủy bỏ, mặc dù ta không gọi hàm hủy bỏ trực tiếp, nhưng khi thoát khỏi hàm main() của chương trình chính, đối tượng myCar bị giải phóng khỏi bộ nhớ và khi đó, C++ tự động gọi đến hàm hủy bỏ mà ta đã định nghĩa tường minh. Do vậy, mà có dòng thông báo cuối cùng này.

5.5 CON TRỎ ĐỐI TƯỢNG VÀ MẢNG ĐỐI TƯỢNG

5.5.1 Con trỏ đối tượng

Con trỏ đối tượng là con trỏ trỏ đến địa chỉ của một đối tượng có kiểu lớp. Các thao tác liên quan đến con trỏ đối tượng bao gồm:

- Khai báo con trỏ đối tượng
- Cấp phát bộ nhớ cho con trỏ đối tượng
- Sử dụng con trỏ đối tượng
- Giải phóng bộ nhớ cho con trỏ đối tượng

Khai báo con trỏ đối tượng

Con trỏ đối tượng được khai báo tương tự như khai báo các con trỏ có kiểu thông thường:

<Tên lớp> *<Tên con trỏ đối tượng>;

Ví dụ, muốn khai báo một con trỏ đối tượng có kiểu của lớp Car, ta khai báo như sau:

Car *myCar;

Khi đó, myCar là một con trỏ đối tượng có kiểu lớp Car.

Cấp phát bộ nhớ cho con trỏ đối tượng

Con trỏ đối tượng cũng cần phải cấp phát bộ nhớ hoặc trỏ vào một địa chỉ của một đối tượng lớp xác định trước khi được sử dụng. Cấp phát bộ nhớ cho con trỏ đối tượng cũng bằng thao tác **new**:

<Tên con trỏ đối tượng> = new <Tên lớp>(<Các đối số>);

Ví dụ, nếu lớp Car có hai hàm khởi tạo như sau:

```
class Car{ public:  
    Car();  
    Car(int, char[], float);  
};
```

thì ta có thể cấp phát bộ nhớ theo hai cách, tương ứng với hai hàm khởi tạo của lớp:

```
myCar = new Car(); // Khởi tạo không tham số myCar  
= new Car(100, "Ford", 3000); // Khởi tạo đủ tham số
```

Lưu ý:

- Các đối số truyền phải tương ứng với ít nhất một trong các hàm khởi tạo của lớp.
- Khi sử dụng hàm khởi tạo không có tham số, ta vẫn phải sử dụng cặp ngoặc đơn “()” trong thao tác **new**.
- Khi lớp không có một hàm khởi tạo tường minh nào, sẽ dùng hàm khởi tạo ngầm định của C++ và cú pháp tương tự như sử dụng hàm khởi tạo tường minh không có tham số.
- Có thể vừa khai báo, vừa cấp phát bộ nhớ cho con trỏ đối tượng.

Ví dụ:

```
Car myCar = new Car(); // Khởi tạo không tham số
```

Sử dụng con trỏ đối tượng

Con trỏ đối tượng được sử dụng qua các thao tác:

- Trỏ đến địa chỉ của một đối tượng cùng lớp
- Truy nhập đến các phương thức của lớp

Con trỏ đối tượng có thể trỏ đến địa chỉ của một đối tượng có sẵn, cùng lớp theo cú pháp sau:

```
<Tên con trỏ đối tượng> = &<Tên đối tượng có sẵn>;
```

Ví dụ, ta có một con trỏ và một đối tượng của lớp Car:

```
Car *ptrCar, myCar(100, "Ford", 3000);
```

Khi đó, có thể cho con trỏ ptrCar trỏ vào địa chỉ của đối tượng myCar như sau:

```
ptrCar = &myCar;
```

Khi muốn truy nhập đến các thành phần của con trỏ đối tượng, ta dùng cú pháp sau:

```
<Tên con trỏ đối tượng> -> <Tên thành phần lớp>([<Các đối số>]);
```

Ví dụ, đoạn chương trình sau sẽ thực hiện phương thức giới thiệu xe của lớp Car thông qua con trỏ ptrCar:

```
Car *ptrCar = new Car(100, "Ford", 3000); ptrCar->show();
```

Lưu ý:

- Danh sách các đối số phải tương thích với tên phương thức tương ứng.
- Các quy tắc phạm vi truy nhập vẫn áp dụng trong truy nhập các thành phần lớp thông qua con trỏ.

Giải phóng bộ nhớ cho con trỏ đối tượng

Con trỏ đối tượng cũng được giải phóng thông qua thao tác **delete**:

```
delete <Tên con trỏ đối tượng>;
```

Ví dụ:

```
Car *ptrCar = new Car(); // Khai báo và cấp phát bộ nhớ  
... // Sử dụng con trỏ ptrCar  
delete ptrCar; // Giải phóng bộ nhớ.
```

Lưu ý:

- Thao tác **delete** chỉ được dùng khi trước đó, con trỏ được cấp phát bộ nhớ qua thao tác **new**:

```
Car *ptrCar = new Car();  
delete ptrCar; // Đúng.
```

Nhưng không được dùng **delete** khi trước đó, con trỏ chỉ trỏ vào một địa chỉ của đối tượng có sẵn (tĩnh):

```
Car *ptrCar, myCar(100, "Ford", 3000); ptrCar =  
&myCar;  
delete ptrCar; // Không được
```

Chương trình 5.6 minh họa việc dùng con trỏ đối tượng có kiểu lớp là Car.

Chương trình 5.6

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp */  
class Car{  
    private:  
        int speed; // Tốc độ  
        char mark[20]; // Nhãn hiệu  
        float price; // Giá xe  
    public:  
        // Khởi tạo với các giá trị ngầm định cho các tham số  
        Car(int speedIn=0, char markIn[]="", float priceIn=0);  
        void show(); // Giới thiệu xe  
};  
  
/* Khai báo phương thức bên ngoài lớp */  
Car::Car(int speedIn, char markIn[], float priceIn){  
    speed = speedIn;  
    strcpy(mark, markIn);  
    price = priceIn;
```

```
}

void Car::show() { // Phương thức giới thiệu xe
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}

// Hàm main, chương trình chính
void main(){
    clrscr();
    // Khai báo con trỏ, cấp phát bộ nhớ dùng hàm khởi tạo đủ tham số
    Car *myCar = new Car(150, "Mercedes", 5000);

    // Giới thiệu xe
    cout << "Gioi thieu xe: " << endl;
    myCar->show();

    // Giải phóng con trỏ
    delete myCar;
    return;
}
```

Chương trình 5.6 hiển thị ra thông báo là một lời giới thiệu xe;

Gioi thieu xe:

This is a Mercedes having a speed of 150km/h and its price is \$5000

5.5.2 Mảng các đối tượng

Mảng các đối tượng cũng có thể được khai báo và sử dụng như mảng của các biến có kiểu thông thường.

Khai báo mảng tĩnh các đối tượng

Mảng các đối tượng được khai báo theo cú pháp:

<Tên lớp> <Tên biến mảng>[<Số lượng đối tượng>];

Ví dụ:

Car cars[10];

Là khai báo một mảng có 10 đối tượng có cùng kiểu lớp Car.

Lưu ý:

- Có thể khai báo mảng tĩnh các đối tượng mà chưa cần khai báo độ dài mảng, cách này thường dùng khi chưa biết chính xác độ dài mảng:

Car cars[];

- Muốn khai báo được mảng tĩnh các đối tượng, lớp tương ứng phải có hàm khởi tạo không có tham số. Vì khi khai báo mảng, tương đương với khai báo một dãy các đối tượng với hàm khởi tạo không có tham số.

Khai báo mảng động với con trỏ

Một mảng các đối tượng cũng có thể được khai báo và cấp phát động thông qua con trỏ đối tượng như sau:

```
<Tên lớp> *<Tên biến mảng động> = new <Tên lớp>[<Độ dài mảng>];
```

Ví dụ:

```
Car *cars = new Car[10];
```

Sau khi được sử dụng, mảng động các đối tượng cũng cần phải giải phóng bộ nhớ:

```
delete [] <Tên biến mảng động>;
```

Ví dụ:

```
Car *cars = new Car[10];// Khai báo và cấp phát động  
... // Sử dụng biến mảng động  
delete [] cars; // Giải phóng bộ nhớ của mảng động
```

Sử dụng mảng đối tượng

Khi truy nhập vào các thành phần của một đối tượng có chỉ số xác định trong mảng đã khai báo, ta có thể sử dụng cú pháp:

```
<Tên biến mảng>[<Chỉ số đối tượng>].<Tên thành phần>(<Các đối số>);
```

Ví dụ:

```
Car cars[10]; cars[5].show();
```

sẽ thực hiện phương thức show() của đối tượng có chỉ số thứ 5 (tính từ chỉ số 0) trong mảng cars.

Chương trình 5.7 sẽ cài đặt một chương trình, trong đó nhập vào độ dài mảng, sau đó yêu cầu người dùng nhập thông tin về mảng các xe. Cuối cùng, chương trình sẽ tìm kiếm và hiển thị thông tin về chiếc xe có giá đắt nhất trong mảng.

Chương trình 5.7

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp Car */  
class Car{  
    private:  
        int speed; // Tốc độ  
        char mark[20]; // Nhãn hiệu  
        float price; // Giá xe  
    public:  
        void setSpeed(int); // Gán tốc độ cho xe
```

Chương 5: Lớp

```
int      getSpeed();           // Đọc tốc độ xe
void     setMark(char);       // Gán nhãn cho xe
char[]   getMark();          // Đọc nhãn xe void
         setPrice(float);    // Gán giá cho xe float
         getPrice();         // Đọc giá xe
         // Khởi tạo thông tin về xe
Car(int speedIn=0, char markIn[]="", float priceIn=0); void      show(); //
Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */ Car::Car(int speedIn, char
markIn[], float priceIn){
    speed = speedIn; strcpy(mark,
    markIn); price = priceIn;
}

void Car::setSpeed(int speedIn){           // Gán tốc độ cho xe speed
    = speedIn;
}
int Car::getSpeed(){                      // Đọc tốc độ xe return
    speed;
}
void Car::setMark(char markIn){           // Gán nhãn cho xe
    strcpy(mark, markIn);
}
char[] Car::getMark(){                   // Đọc nhãn xe
    return mark;
}
void Car::setPrice(float priceIn){        // Gán giá cho xe price
    = priceIn;
}
float Car::getPrice(){                  // Đọc giá xe
    return price;
}

void Car::show(){                       // Phương thức giới thiệu xe cout <<
    "This is a " << mark << " having a speed of "
    << speed << "km/h and its price is $" << price << endl; return;
}

// Hàm main, chương trình chính
```

```

void main(){
    clrscr();
    int length;                                // Chiều dài mảng
    float maxPrice = 0;                         // Giá đắt nhất
    int index = 0;                               // Chỉ số của xe đắt nhất
    Car *cars;                                  // Khai báo mảng đối tượng

    // Nhập số lượng xe, tức là chiều dài mảng cout << "So
    luong xe: ";
    cin >> length;

    // Cấp phát bộ nhớ động cho mảng cars =
    new Car[length];

    // Khởi tạo các đối tượng trong mảng for(int
    i=0;i<length; i++){
        int speed;                             // (Biến tạm) tốc độ
        char mark[20];                         // (Biến tạm) nhãn hiệu
        float price;                           // (Biến tạm) giá xe cout
        << "Xe thu " << i << ":" << endl;
        cout << "Toc do (km/h): ";
        cin >> speed; cars[i].setSpeed(speed);           // Nhập tốc độ
        cout << "Nhan hieu : ";
        cin >> mark; cars[i].setMark(mark);             // Nhập nhãn xe cout
        << "Gia ($): ";
        cin >> price; cars[i].setPrice(price);          // Nhập giá xe

        if(maxPrice < price){ maxPrice =
            price; index = i;
        }
    }

    // Tìm xe đắt nhất
    for(int i=0; i<length; i++) if(i == index){
        cars[i].show();                      // Giới thiệu xe đắt nhất break;
    }

    // Giải phóng bộ nhớ của mảng delete [] cars;
    return;
}

```

}

TỔNG KẾT CHƯƠNG 5

Nội dung chương 5 đã tập trung trình bày các vấn đề cơ bản về lớp đối tượng trong ngôn ngữ C++:

- Khai báo, định nghĩa lớp bằng từ khóa class
- Sử dụng biến lớp như những đối tượng cụ thể
- Khai báo, định nghĩa các thuộc tính và các phương thức của lớp: định nghĩa các phương thức trong hoặc ngoài phạm vi khai báo lớp
- Khai báo phạm vi truy nhập lớp bằng các từ khóa chỉ phạm vi: private, protected và public. Giới thiệu các kiểu hàm có thể truy nhập phạm vi bất quy tắc: hàm bạn và lớp bạn với từ khóa friend.
- Khai báo, định nghĩa tường minh hàm khởi tạo của lớp và sử dụng khi khai báo biến lớp
- Khai báo, định nghĩa tường minh hàm hủy bỏ của lớp, nhằm gọn nhẹ, giải phóng bộ nhớ trước khi đối tượng bị giải phóng khỏi bộ nhớ.
- Khai báo, cấp phát bộ nhớ, sử dụng và giải phóng bộ nhớ cho con trỏ đối tượng
- Khai báo, cấp phát bộ nhớ động, sử dụng và giải phóng vùng nhớ của mảng các đối tượng.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 5

1. Trong các khai báo lớp sau, những khai báo nào là đúng:
 - a. class MyClass;
 - b. Class MyClass;
 - c. class MyClass{...};
 - d. Class MyClass{...};
2. Giả sử ta đã định nghĩa lớp MyClass, bây giờ ta khai báo một đối tượng thuộc kiểu lớp này. Khai báo nào là đúng:
 - a. class MyClass me;
 - b. MyClass me;
 - c. MyClass me();
 - d. MyClass me{ };
3. Trong các khai báo thuộc tính ngay trong phạm vi của khai báo lớp như sau, những khai báo nào là đúng:
 - a. int myAge;
 - b. private int myAge;
 - c. public int myAge;
 - d. private: int myAge;

4. Trong các khai báo phương thức ngay trong phạm vi của khai báo lớp MyClass như sau, những khai báo nào là đúng:

- a. public: void show();
- b. void show();
- c. void show(){cout << "hello!";};
- d. void MyClass::show(){cout << "hello!";}

5. Xét đoạn chương trình sau:

```
class MyClass{  
    int age; public:  
        int getAge();  
};  
MyClass me;
```

Khi đó, trong các lệnh sau, lệnh nào có thể thêm vào cuối đoạn chương trình trên:

- a. cout << MyClass::age;
- b. cout << me.age;
- c. cout << me.getAge();
- d. cin >> me.age;

6. Trong các khai báo hàm khởi tạo cho lớp MyClass như sau, những khai báo nào là đúng:

- a. myClass();
- b. MyClass();
- c. MyClass(MyClass);
- d. void MyClas();
- e. MyClass MyClass();

7. Trong các khai báo hàm hủy bỏ cho lớp MyClass như sau, những khai báo nào là đúng:

- a. ~myClass();
- b. ~MyClass();
- c. ~MyClass(MyClass);
- d. void ~MyClas();
- e. MyClass ~MyClass();

8. Giả sử ta khai báo lớp MyClass có một thuộc tính age và một hàm khởi tạo:

```
class MyClass{  
    int age; public:  
        MyClass(MyClass me){  
            ...  
        };  
        int getAge(){return age};  
};
```

Trong các dòng lệnh sau, những dòng nào có thể xuất hiện trong hàm khởi tạo trên:

- a. age = MyClass.age;
- b. age = me.age;
- c. age = MyClass.getAge();
- d. age = me.getAge();
- e. age = 10;

9. Xét khai báo lớp như sau:

```
class MyClass{  
    public:  
        MyClass(MyClass);  
        MyClass(int);  
};
```

Khi đó, trong số các khai báo đối tượng sau, những khai báo nào là đúng:

- a. MyClass me;
- b. MyClass me();
- c. MyClass me(10);
- d. MyClass me1(10), me2(me1);

10. Xét khai báo lớp như sau:

```
class MyClass{  
    public:  
        MyClass();  
};
```

Khi đó, trong số các khai báo con trả đối tượng sau, những khai báo nào là đúng:

- a. MyClass *me;
- b. MyClass *me();
- c. MyClass *me = new me();
- d. MyClass *me = new MyClass();

11. Xét khai báo lớp như sau:

```
class MyClass{  
    public:  
        MyClass(int i=0, int j=0, float k=0);  
};
```

Khi đó, trong số các khai báo con trả đối tượng sau, những khai báo nào là đúng:

- a. MyClass *me;
- b. MyClass *me = new MyClass();
- c. MyClass *me = new MyClass(1, 20);
- d. MyClass *me = new MyClass(1, 20, 30);

12. Khai báo một lớp nhân viên, có tên là Employee, với ba thuộc tính có tính chất private:

- Tên nhân viên, có dạng một con trả kiểu char

- Tuổi nhân viên, có kiểu int
 - Lương nhân viên, có kiểu float.
13. Thêm vào lớp Employee trong bài 12 các phương thức có tính chất public:
- set/get giá trị thuộc tính tên nhân viên
 - set/get giá trị thuộc tính tuổi nhân viên
 - set/get giá trị thuộc tính lương nhân viên.
14. Viết một hàm khởi tạo không có tham số cho lớp Employee trong bài 13, các thuộc tính của lớp nhận giá trị mặc định:
- giá trị thuộc tính tên nhân viên, mặc định là chuỗi kí tự rỗng ""
 - giá trị thuộc tính tuổi nhân viên, mặc định là 18
 - giá trị thuộc tính lương nhân viên, mặc định là \$100.
15. Viết thêm một hàm khởi tạo với đầy đủ ba tham số tương ứng với ba thuộc tính của lớp Employee trong bài 14.
16. Thêm vào lớp Employee một phương thức show() để giới thiệu về tên, tuổi và lương của đối tượng nhân viên.
17. Viết một hàm hủy bỏ tường minh cho lớp Employee nhằm giải phóng vùng nhớ của con trỏ char, là kiểu của thuộc tính tên nhân viên.
18. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo không tham số
 - Dùng hàm show() để giới thiệu về đối tượng đó
19. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo không tham số
 - Nhập từ bàn phím giá trị các thuộc tính tên, tuổi, lương nhân viên.
 - Gán các giá trị này cho các thuộc tính của đối tượng đã khai báo, dùng các hàm set
 - Dùng hàm show() để giới thiệu về đối tượng đó
20. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một đối tượng kiểu Employee, dùng hàm khởi tạo với đủ 3 tham số
 - Dùng hàm show() để giới thiệu về đối tượng đó
21. Viết một chương trình sử dụng lớp Employee được xây dựng sau bài 17 với các thao tác sau:
- Khai báo một con trỏ đối tượng kiểu Employee
 - Cấp phát bộ nhớ, dùng hàm khởi tạo với đủ 3 tham số
 - Dùng hàm show() để giới thiệu về đối tượng mà con trỏ này đang trỏ tới
22. Viết một chương trình nhập dữ liệu cho một mảng động các đối tượng của lớp Employee trong bài 17. Chiều dài mảng động cũng được nhập từ bàn phím.

23. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên trẻ nhất và nhân viên già nhất trong mảng đó.
24. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên có lương cao nhất và nhân viên có lương thấp nhất trong mảng đó.
25. Viết một chương trình tìm kiếm trên mảng động đã được xây dựng trong bài 22: tìm kiếm và giới thiệu về nhân viên có tên xác định, do người dùng nhập từ bàn phím.

CHƯƠNG 6

TÍNH KẾ THỪA VÀ ĐA HÌNH

Nội dung chương này tập trung trình bày các vấn đề liên quan đến tính kế thừa và tương ứng bội (đa hình) trong ngôn ngữ C++:

- Khái niệm kế thừa, dẫn xuất và các kiểu dẫn xuất
- Khai báo, định nghĩa các hàm khởi tạo và hàm hủy bỏ trong lớp dẫn xuất
- Truy nhập tới các thành phần của lớp cơ sở và các lớp dẫn xuất
- Việc một lớp được kế thừa từ nhiều lớp cơ sở khác nhau
- Khai báo và sử dụng các lớp cơ sở trừu tượng trong kế thừa
- Tính đa hình trong kế thừa

6.1 KHÁI NIỆM KẾ THỪA

Lập trình hướng đối tượng có hai đặc trưng cơ bản:

- Đóng gói dữ liệu, được thể hiện bằng cách dùng khái niệm lớp để biểu diễn đối tượng với các thuộc tính private, chỉ cho phép bên ngoài truy nhập vào thông qua các phương thức get/set.
- Dùng lại mã, thể hiện bằng việc thừa kế giữa các lớp. Việc thừa kế cho phép các lớp thừa kế (gọi là lớp dẫn xuất) sử dụng lại các phương thức đã được định nghĩa trong các lớp gốc (gọi là lớp cơ sở).

6.1.1 Khai báo thừa kế

Cú pháp khai báo một lớp kế thừa từ một lớp khác như sau:

```
class <Tên lớp dẫn xuất>: <Từ khóa dẫn xuất> <Tên lớp cơ sở>{  
    ...      // Khai báo các thành phần lớp  
};
```

Trong đó:

- **Tên lớp dẫn xuất:** là tên lớp được cho kế thừa từ lớp khác. Tên lớp này tuân thủ theo quy tắc đặt tên biến trong C++.
- **Tên lớp cơ sở:** là tên lớp đã được định nghĩa trước đó để cho lớp khác kế thừa. Tên lớp này cũng tuân thủ theo quy tắc đặt tên biến của C++.
- **Từ khóa dẫn xuất:** là từ khóa quy định tính chất của sự kế thừa. Có ba từ khóa dẫn xuất là private, protected và public. Mục tiếp theo sẽ trình bày ý nghĩa của các từ khóa dẫn xuất này.

Ví dụ:

```
class Bus: public Car{  
    ...      // Khai báo các thành phần  
};
```

là khai báo một lớp Bus (xe buýt) kế thừa từ lớp Car (xe ô tô) với tính chất kế thừa là public.

6.1.2 Tính chất dẫn xuất

Sự kế thừa cho phép trong lớp dẫn xuất có thể sử dụng lại một số mã nguồn của các phương thức và thuộc tính đã được định nghĩa trong lớp cơ sở. Nghĩa là lớp dẫn xuất có thể truy nhập trực tiếp đến một số thành phần của lớp cơ sở. Tuy nhiên, phạm vi truy nhập từ lớp dẫn xuất đến lớp cơ sở không phải bao giờ cũng giống nhau: chúng được quy định bởi các từ khóa dẫn xuất private, protected và public.

Dẫn xuất private

Dẫn xuất private quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần private của lớp dẫn xuất
- Các thành phần public của lớp cơ sở cũng trở thành các thành phần private của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Dẫn xuất protected

Dẫn xuất protected quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần protected của lớp dẫn xuất
- Các thành phần public của lớp cơ sở cũng trở thành các thành phần protected của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Dẫn xuất public

Dẫn xuất public quy định phạm vi truy nhập như sau:

- Các thành phần private của lớp cơ sở thì không thể truy nhập được từ lớp dẫn xuất.
- Các thành phần protected của lớp cơ sở trở thành các thành phần protected của lớp dẫn xuất.
- Các thành phần public của lớp cơ sở vẫn là các thành phần public của lớp dẫn xuất.
- Phạm vi truy nhập từ bên ngoài vào lớp dẫn xuất được tuân thủ như quy tắc phạm vi lớp thông thường.

Bảng 6.1 tóm tắt lại các quy tắc truy nhập được quy định bởi các từ khóa dẫn xuất.

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất
private	private	Không truy nhập được

	protected public	private private
protected	private protected public	Không truy nhập được protected protected
public	private protected public	Không truy nhập được protected public

6.2 HÀM KHỞI TẠO VÀ HỦY BỎ TRONG KẾ THỪA

6.2.1 Hàm khởi tạo trong kế thừa

Khi khai báo một đối tượng có kiểu lớp được dẫn xuất từ một lớp cơ sở khác. Chương trình sẽ tự động gọi tới hàm khởi tạo của lớp dẫn xuất. Tuy nhiên, thứ tự được gọi sẽ bắt đầu từ hàm khởi tạo tương ứng của lớp cơ sở, sau đó đến hàm khởi tạo của lớp dẫn xuất. Do đó, thông thường, trong hàm khởi tạo của lớp dẫn xuất phải có hàm khởi tạo của lớp cơ sở.

Cú pháp khai báo hàm khởi tạo như sau:

```
<Tên hàm khởi tạo dẫn xuất>(<Các tham số>):
    <Tên hàm khởi tạo cơ sở>(<Các đối số>){
        ...
        // Khởi tạo các thuộc tính mới bổ sung của lớp dẫn xuất
    };
```

Vì tên hàm khởi tạo là trùng với tên lớp, nên có thể viết lại thành:

```
<Tên lớp dẫn xuất>(<Các tham số>):
    <Tên lớp cơ sở>(<Các đối số>){
        ...
        // Khởi tạo các thuộc tính mới bổ sung của lớp dẫn xuất
    };
```

Ví dụ:

```
Bus():Car(){
    ...
    // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
}
```

là một định nghĩa một hàm khởi tạo của lớp Bus kế thừa từ lớp Car. Định nghĩa này được thực hiện trong phạm vi khai báo lớp Bus. Đây là một hàm khởi tạo không tham số, nó gọi tới hàm khởi tạo không tham số của lớp Car.

Lưu ý:

- Nếu định nghĩa hàm khởi tạo bên ngoài phạm vi lớp thì phải thêm tên lớp dẫn xuất và toán tử phạm vi “::” trước tên hàm khởi tạo.
- Giữa tên hàm khởi tạo của lớp dẫn xuất và hàm khởi tạo của lớp cơ sở, chỉ có một dấu hai chấm “::”, nếu là hai dấu “::” thì trở thành toán tử phạm vi lớp.

- Nếu không chỉ rõ hàm khởi tạo của lớp cơ sở sau dấu hai chấm “:” chương trình sẽ tự động gọi hàm khởi tạo ngầm định hoặc hàm khởi tạo không có tham số của lớp cơ sở nếu hàm đó được định nghĩa tường minh trong lớp cơ sở.

Ví dụ, định nghĩa hàm khởi tạo:

```
Bus():Car(){
    ...      // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
};
```

Có thể thay bằng:

```
Bus()          // Gọi hàm khởi tạo không tham số của lớp Car
    ...
};           // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
```

Chương trình 6.1 định nghĩa lớp Car có 3 thuộc tính với hai hàm khởi tạo, sau đó định nghĩa lớp Bus có thêm thuộc tính label là số hiệu của tuyến xe buýt. Lớp Bus sẽ được cài đặt hai hàm khởi tạo tường minh, gọi đến hai hàm khởi tạo tương ứng của lớp Car.

Chương trình 6.1

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    int speed;                      // Tốc độ
    char mark[20];                  // Nhãn hiệu
    float price;                    // Giá xe

    public:
        Car();                      // Khởi tạo không tham số
        Car(int, char[], float);    // Khởi tạo đủ tham số
};

Car::Car() {                                // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn) {
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
```

```

class Bus: public Car{
    int label;                                // Số hiệu tuyến xe
public:
    Bus();                                     // Khởi tạo không tham số
    Bus(int, char[], float, int); // Khởi tạo đủ tham số
};

Bus::Bus():Car() {                         // Khởi tạo không tham số
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

```

Trong hàm khởi tạo của lớp Bus, muốn khởi tạo các thuộc tính của lớp Car, ta phải khởi tạo gián tiếp thông qua hàm khởi tạo của lớp Car mà không thể gán giá trị trực tiếp cho các thuộc tính speed, mark và price. Lý do là các thuộc tính này có tính chất private, nên lớp dẫn xuất không thể truy nhập trực tiếp đến chúng.

6.2.2 Hàm hủy bỏ trong kế thừa

Khi một đối tượng lớp dẫn xuất bị giải phóng khỏi bộ nhớ, thứ tự gọi các hàm hủy bỏ ngược với thứ tự gọi hàm thiết lập: gọi hàm hủy bỏ của lớp dẫn xuất trước khi gọi hàm hủy bỏ của lớp cơ sở. Vì mỗi lớp chỉ có nhiều nhất là một hàm hủy bỏ, nên ta không cần phải chỉ ra hàm hủy bỏ nào của lớp cơ sở sẽ được gọi sau khi hủy bỏ lớp dẫn xuất. Do vậy, hàm hủy bỏ trong lớp dẫn xuất được khai báo và định nghĩa hoàn toàn giống với các lớp thông thường:

```

<Tên lớp>::~<Tên lớp>([<Các tham số>]){
    ... // giải phóng phần bộ nhớ cấp phát cho các thuộc tính bổ sung
}

```

Lưu ý:

- Hàm hủy bỏ của lớp dẫn xuất chỉ giải phóng phần bộ nhớ được cấp phát động cho các thuộc tính mới bổ sung trong lớp dẫn xuất, nếu có, mà không được giải phóng bộ nhớ được cấp cho các thuộc tính trong lớp cơ sở (phần này là do hàm hủy bỏ của lớp cơ sở đảm nhiệm).
- Không phải gọi tường minh hàm hủy bỏ của lớp cơ sở trong hàm hủy bỏ của lớp dẫn xuất.
- Ngay cả khi lớp dẫn xuất không định nghĩa tường minh hàm hủy bỏ (do không cần thiết) mà lớp cơ sở lại có định nghĩa tường minh. Chương trình vẫn gọi hàm hủy bỏ ngầm định của lớp dẫn xuất, sau đó vẫn gọi hàm hủy bỏ tường minh của lớp cơ sở.

Chương trình 6.2 cài đặt lớp Bus kế thừa từ lớp Car: lớp Car có một thuộc tính có dạng con trỏ nên cần giải phóng bằng hàm hủy bỏ tường minh. Lớp Bus có thêm một thuộc tính có dạng con

Chương 6: Tính kế thừa và đa hình

trở là danh sách các đường phố mà xe buýt đi qua (mảng động các chuỗi kí tự *char[]) nên cũng cần giải phóng bằng hàm hủy bỏ tường minh.

Chương trình 6.2

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    char *mark; // Nhãn hiệu xe
public:
    ~Car(); // Hủy bỏ tường minh
};

Car::~Car(){ // Hủy bỏ tường minh
    delete [] mark;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    char *voyage[]; // Hành trình tuyến xe
public:
    ~Bus(); // Hủy bỏ tường minh
};

Bus::~Bus(){ // Hủy bỏ tường minh
    delete [] voyage;
}
```

Trong hàm hủy bỏ của lớp Bus, ta chỉ được giải phóng vùng nhớ được cấp phát cho thuộc tính voyage (hành trình của xe buýt), là thuộc tính được bổ sung thêm của lớp Bus. Mà không được giải phóng vùng nhớ cấp phát cho thuộc tính mark (nhãn hiệu xe), việc này là thuộc trách nhiệm của hàm hủy bỏ của lớp Car vì thuộc tính mark được khai báo trong lớp Car.

6.3 TRUY NHẬP TỚI CÁC THÀNH PHẦN TRONG KẾ THỪA LỚP

6.3.1 Phạm vi truy nhập

Mỗi quan hệ giữa các thành phần của lớp cơ sở và lớp dẫn xuất được quy định bởi các từ khóa dẫn xuất, như đã trình bày trong mục 6.1.2, được tóm tắt trong bảng 6.2

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất
private	private protected	Không truy nhập được private

	public	private
protected	private protected public	Không truy nhập được protected protected
public	private protected public	Không truy nhập được protected public

Ta xét phạm vi truy nhập theo hai loại:

- Phạm vi truy nhập từ các hàm bạn, lớp bạn của lớp dẫn xuất
- Phạm vi truy nhập từ các đối tượng có kiểu lớp dẫn xuất

Truy nhập từ các hàm bạn và lớp bạn của lớp dẫn xuất

Nhìn vào bảng tổng kết 6.2, phạm vi truy nhập của hàm bạn, lớp bạn của lớp dẫn xuất vào lớp cơ sở như sau:

- Với dẫn xuất private, hàm bạn có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần private của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Với dẫn xuất protected, hàm bạn cũng có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần protected của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Với dẫn xuất public, hàm bạn cũng có thể truy nhập được các thành phần protected và public của lớp cơ sở vì chúng trở thành các thành phần protected và public của lớp dẫn xuất, có thể truy nhập được từ hàm bạn.
- Đối với cả ba loại dẫn xuất, hàm bạn đều không truy nhập được các thành phần private của lớp cơ sở, vì các thành phần này cũng không truy nhập được từ lớp dẫn xuất.

Truy nhập từ các đối tượng tạo bởi lớp dẫn xuất

Nhìn vào bảng tổng kết 6.2, phạm vi truy nhập của các đối tượng của lớp dẫn xuất vào lớp cơ sở như sau:

- Với dẫn xuất private, đối tượng của lớp dẫn xuất không truy nhập được bất cứ thành phần nào của lớp cơ sở vì chúng trở thành các thành phần private của lớp dẫn xuất, không truy nhập được từ bên ngoài.
- Với dẫn xuất protected, đối tượng của lớp dẫn xuất không truy nhập được bất cứ thành phần nào của lớp cơ sở vì chúng trở thành các thành phần protected của lớp dẫn xuất, không truy nhập được từ bên ngoài.
- Với dẫn xuất public, đối tượng của lớp dẫn xuất có thể truy nhập được các thành phần public của lớp cơ sở vì chúng trở thành các thành phần public của lớp dẫn xuất, có thể truy nhập được từ bên ngoài.

Bảng 6.3 tổng kết phạm vi truy nhập từ hàm bạn và đối tượng của lớp dẫn xuất vào các thành phần của lớp cơ sở, được quy định bởi các từ khóa dẫn xuất.

Kiểu dẫn xuất	Tính chất ở lớp cơ sở	Tính chất ở lớp dẫn xuất	Truy nhập từ hàm bạn của lớp dẫn xuất	Truy nhập từ đối tượng của lớp dẫn xuất
private	private	---	---	---
	protected	private	ok	---
	public	private	ok	---
protected	private	---	---	---
	protected	protected	ok	---
	public	protected	ok	---
public	private	---	---	---
	protected	protected	ok	---
	public	public	ok	ok

6.3.2 Sử dụng các thành phần của lớp cơ sở từ lớp dẫn xuất

Từ bảng tổng kết phạm vi truy nhập, ta thấy rằng chỉ có dẫn xuất theo kiểu public thì đối tượng của lớp dẫn xuất mới có thể truy nhập đến các thành phần (thuộc loại public) của lớp cơ sở. Khi đó, việc gọi đến các thành phần của lớp cơ sở cũng tương tự như gọi các thành phần lớp thông thường:

- Đối với biến đối tượng thông thường:
`<Tên đối tượng>.<Tên thành phần>([Các đối số]);`
- Đối với con trỏ đối tượng:
`<Tên đối tượng>-><Tên thành phần>([Các đối số]);`

Lưu ý:

- Cách gọi hàm thành phần này được áp dụng khi trong lớp dẫn xuất, ta không định nghĩa lại các hàm thành phần của lớp cơ sở. Trường hợp định nghĩa lại hàm thành phần của lớp cơ sở sẽ được trình bày trong mục 6.3.3.

Chương trình 6.3 minh họa việc sử dụng các thành phần lớp cơ sở từ đối tượng lớp dẫn xuất: lớp Bus kế thừa từ lớp Car. Lớp Bus có định nghĩa bổ sung một số phương thức và thuộc tính mới. Khi đó, đối tượng của lớp Bus có thể gọi các hàm public của lớp Bus cũng như của lớp Car.

Chương trình 6.3

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    private:
        int speed; // Tốc độ
}
```

Chương 6: Tính kế thừa và đa hình

```
char      mark[20];           // Nhãn hiệu
float     price;              // Giá xe
public:
    void    setSpeed(int);    // Gán tốc độ cho xe int
    getSpeed();               // Đọc tốc độ xe
    void    setMark(char);    // Gán nhãn cho xe
    char[]  getMark();        // Đọc nhãn xe void
    setPrice(float);          // Gán giá cho xe float
    getPrice();               // Đọc giá xe
// Khởi tạo thông tin về xe
Car(int speedIn=0, char markIn[]="", float priceIn=0); void show(); // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */ Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn; strcpy(mark,
    markIn); price = priceIn;
}

void Car::setSpeed(int speedIn){           // Gán tốc độ cho xe speed
    = speedIn;
}
int Car::getSpeed(){                      // Đọc tốc độ xe return
    speed;
}
void Car::setMark(char markIn){            // Gán nhãn cho xe
    strcpy(mark, markIn);
}
char[] Car::getMark(){                    // Đọc nhãn xe
    return mark;
}
void Car::setPrice(float priceIn){         // Gán giá cho xe price
    = priceIn;
}
float Car::getPrice(){                   // Đọc giá xe
    return price;
}

void Car::show(){                        // Phương thức giới thiệu xe cout <<
    "This is a " << mark << " having a speed of "
    << speed << "km/h and its price is $" << price << endl;
```

```
return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus{
public Car{
    int label; // Số hiệu tuyến xe public:
    // Khởi tạo đú tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0); void setLabel(int); //
    Gán số hiệu tuyến xe
    int getLabel(); // Đọc số hiệu tuyến xe
};

// Khởi tạo đú tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){ label = lIn;
}
void Bus::setLabel(int labelIn){ // Gán số hiệu tuyến xe label =
    labelIn;
}
int Bus::getLabel(){ // Đọc số hiệu tuyến xe return
    label;
}

// Chương trình chính void
main(){
    clrscr();
    Bus myBus; // Biến đối tượng của lớp Bus int
    speedIn, labelIn;
    float priceIn; char
    markIn[20];

    // Nhập giá trị cho các thuộc tính cout << "Toc
    do xe bus:";
    cin >> speedIn;
    cout << "Nhan hieu xe bus:"; cin >>
    markIn;
    cout << "Gia xe bus:"; cin >>
    priceIn;
    cout << "So hieu tuyen xe bus:"; cin >>
    labelIn;

    myBus.setSpeed(speedIn); // Phương thức của lớp Car
```

Chương 6: Tính kế thừa và đa hình

```
myBus.setMark(markIn);           // Phương thức của lớp Car
myBus.setPrice(priceIn);         // Phương thức của lớp Car
myBus.setLabel(labelIn);          // Phương thức của lớp Bus
myBus.show();                    // Phương thức của lớp Car
return;
}
```

Trong chương trình 6.3, đối tượng myBus có kiểu lớp Bus, là lớp dẫn xuất của lớp cơ sở Car, có thể sử dụng các phương thức của lớp Car và lớp Bus một cách bình đẳng. Khi đó, lệnh myBus.show() sẽ gọi đến phương thức show() của lớp Car, do vậy, chương trình trên sẽ in ra màn hình kết quả như sau (tùy theo dữ liệu nhập vào ở 4 dòng đầu):

```
Toc do xe bus: 80
Nhan hieu xe bus: Mercedes Gia xe
bus: 5000
So hieu tuyen xe bus: 27
This is a Mercedes having a speed of 80km/h and its price is $5000
```

Trong dòng giới thiệu xe bus (vì ta đang dùng đối tượng myBus của lớp Bus), không có giới thiệu số hiệu tuyến xe. Lý do là vì ta đang dùng hàm show của lớp Car. Muốn có thêm phần giới thiệu về số hiệu tuyến xe buýt, ta phải định nghĩa lại hàm show trong lớp Bus. Mục 6.3.3 sẽ trình bày nội dung này.

6.3.3 Định nghĩa chồng các phương thức của lớp cơ sở

Định nghĩa chồng phương thức của lớp cơ sở

Một phương thức của lớp cơ sở bị coi là nạp chồng nếu ở lớp dẫn xuất cũng định nghĩa một phương thức có cùng tên.

Ví dụ, trong lớp Car, đã có phương thức show(), bây giờ, trong lớp Bus kế thừa từ lớp Car, ta cũng định nghĩa lại phương thức show():

```
class Car{
    ... public:
        ...
        void show();           // Phương thức của lớp cơ sở
};

class Bus: public Car{
    ... public:
        ...
        void show();           // Phương thức nạp chồng
};
```

khi đó, phương thức show() của lớp Bus được coi là phương thức nạp chồng từ phương thức show() của lớp Car.

Sử dụng các phương thức nạp chồng

Từ một đối tượng của lớp dẫn xuất, việc truy nhập đến phương thức đã được định nghĩa lại trong lớp dẫn xuất được thực hiện như lời gọi một phương thức thông thường:

- Đối với biến đối tượng thông thường:
<Tên đối tượng>.<Tên thành phần>([Các đối số]);
- Đối với con trỏ đối tượng:
<Tên đối tượng>-><Tên thành phần>([Các đối số]);

Ví dụ:

```
Bus myBus; myBus.show();
```

sẽ gọi đến phương thức show() được định nghĩa trong lớp Bus.

Trong trường hợp, từ một đối tượng của lớp dẫn xuất, muốn truy nhập đến một phương thức của lớp cơ sở (đã bị định nghĩa lại ở lớp dẫn xuất) thì phải sử dụng chỉ thị phạm vi lớp trước phương thức được gọi:

- Đối với biến đối tượng thông thường:
<Tên đối tượng>.<Tên lớp cơ sở>::<Tên thành phần>([Các đối số]);
- Đối với con trỏ đối tượng:
<Tên đối tượng>-><Tên lớp cơ sở>::<Tên thành phần>([Các đối số]);

Ví dụ:

```
Bus myBus;  
myBus.Car::show();
```

sẽ gọi đến phương thức show() được định nghĩa trong lớp Car từ một đối tượng của lớp Bus.

Chương trình 6.4 minh họa việc định nghĩa chồng hàm show() trong lớp Bus và việc sử dụng hai phương thức show() của hai lớp từ một đối tượng của lớp dẫn xuất.

Chương trình 6.4

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp Car */  
class Car{  
    private:  
        int speed; // Tốc độ  
        char mark[20]; // Nhãn hiệu  
        float price; // Giá xe  
    public:  
        int getSpeed(); // Đọc tốc độ xe  
        char[] getMark(); // Đọc nhãn xe  
        float getPrice(); // Đọc giá xe  
        // Khởi tạo thông tin về xe
```

Chương 6: Tính kế thừa và đa hình

```
Car(int speedIn=0, char markIn[]="", float priceIn=0); void show(); // Giới
thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */ Car::Car(int speedIn, char
markIn[], float priceIn){
    speed = speedIn; strcpy(mark,
    markIn); price = priceIn;
}

void Car::setSpeed(int speedIn){           // Gán tốc độ cho xe speed
    = speedIn;
}

int Car::getSpeed(){                      // Đọc tốc độ xe return
    speed;
}

char[] Car::getMark(){                    // Đọc nhãn xe
    return mark;
}

float Car::getPrice(){                   // Đọc giá xe
    return price;
}

void Car::show(){                        // Phương thức giới thiệu xe cout <<
    "This is a " << mark << " having a speed of "
    << speed << "km/h and its price is $" << price << endl; return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */ class Bus: public
Car{
    int label;                           // Số hiệu tuyến xe public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0); void show(); // Giới
    thiệu xe bus
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){ label = lIn;
}
```

```
// Định nghĩa nạp chồng phương thức
void Bus::show() { // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getPrice()
        << "km/h and its price is $" << getSpeed() << endl;
    return;
}

// Chương trình chính
void main() {
    clrscr();
    Bus myBus(80, "Mercedes", 5000, 27); // Biến đối tượng của lớp Bus

    cout << "Gioi thieu xe:" << endl;
    myBus.Car::show(); // Phương thức của lớp Car
    cout << "Gioi thieu xe bus:" << endl;
    myBus.show(); // Phương thức của lớp Bus
    return;
}
```

Chương trình 6.4 sẽ hiển thị các thông báo như sau:

Gioi thieu xe:

This is a Mercedes having a speed of 80km/h and its price is \$5000 Gioi thieu xe bus:

This is a bus of type Mercedes, on the line 27, having a speed of 80km/h and its price is \$5000

Lưu ý:

- Trong phương thức show() của lớp Bus, ta phải dùng các hàm get để truy nhập đến các thuộc tính của lớp Car. Không được truy nhập trực tiếp đến tên các thuộc tính (speed, mark và price) vì chúng có dạng private của lớp Car.

6.3.4 Chuyển đổi kiểu giữa lớp cơ sở và lớp dẫn xuất

Về mặt dữ liệu, một lớp dẫn xuất bao giờ cũng chứa toàn bộ dữ liệu của lớp cơ sở: Ta luôn tìm thấy lớp cơ sở trong lớp dẫn xuất, nhưng không phải bao giờ cũng tìm thấy lớp dẫn xuất trong lớp cơ sở. Do vậy:

- Có thể gán một đối tượng lớp dẫn xuất cho một đối tượng lớp cơ sở:
 $\langle\text{Đối tượng lớp cơ sở}\rangle = \langle\text{Đối tượng lớp dẫn xuất}\rangle; // Đúng$
- Nhưng không thể gán một đối tượng lớp cơ sở cho một đối tượng lớp dẫn xuất:
 $\langle\text{Đối tượng lớp dẫn xuất}\rangle = \langle\text{Đối tượng lớp cơ sở}\rangle; // Không được$

Ví dụ, ta có lớp Bus kế thừa từ lớp Car và:

```
Bus myBus;
```

Car myCar;
khi đó, phép gán:
myCar = myBus; // đúng
thì chấp nhận được, nhưng phép gán:
myBus = myCar; // không được
thì không chấp nhận được.

Lưu ý:

- Nguyên tắc chuyển kiểu này cũng đúng với các phép gán con trỏ: một con trỏ đối tượng lớp cơ sở có thể trỏ đến địa chỉ của một đối tượng lớp dẫn xuất. Nhưng một con trỏ đối tượng lớp dẫn xuất không thể trỏ đến địa chỉ một đối tượng lớp cơ sở.
- Nguyên tắc chuyển kiểu này cũng đúng với truyền đối số cho hàm: có thể truyền một đối tượng lớp dẫn xuất vào vị trí của tham số có kiểu lớp cơ sở. Nhưng không thể truyền một đối tượng lớp cơ sở vào vị trí một tham số có kiểu lớp dẫn xuất.

Chương trình 6.5 minh họa việc chuyển kiểu giữa các đối tượng của lớp cơ sở và lớp dẫn xuất.

Chương trình 6.5

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    private:
        int speed; // Tốc độ
        char mark[20]; // Nhãn hiệu
        float price; // Giá xe
    public:
        int getSpeed(); // Đọc tốc độ xe
        char[] getMark(); // Đọc nhãn xe
        float getPrice(); // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[], float priceIn=0);
        void show(); // Giới thiệu xe
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn) {
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```

Chương 6: Tính kế thừa và đa hình

```
int Car::getSpeed() { // Đọc tốc độ xe
    return speed;
}

char[] Car::getMark() { // Đọc nhãn xe
    return mark;
}

float Car::getPrice() { // Đọc giá xe
    return price;
}

void Car::show() { // Phương thức giới thiệu xe cout <<
    "This is a " << mark << " having a speed of "
    << speed << "km/h and its price is $" << price << endl; return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus {
public Car{
    int label; // Số hiệu tuyến xe public:
    // Khởi tạo đú tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0); void show(); // Định nghĩa chòng phương thức
};

// Khởi tạo đú tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){ label = lIn;
}

// Định nghĩa nạp chòng phương thức
void Bus::show() { // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl; return;
}

// Chương trình chính void
main(){
    clrscr();
    Car myCar(100, "Ford", 3000); // Biến đói tượng lớp Car
    Bus myBus(80, "Mercedes", 5000, 27); // Biến đói tượng lớp Bus
```

```
cout << "Gioi thieu xe o to lan 1:" << endl;
myCar.show();
cout << "Gioi thieu xe o to lan 2:" << endl;
myCar = myBus;
myCar.show();
cout << "Gioi thieu xe bus:" << endl;
myBus.show();
return;
}
```

Chương trình 6.5 sẽ in ra kết quả thông báo như sau:

```
Goi thieu xe o to lan 1:
This is a Ford having a speed of 100km/h and its price is $3000 Gioi thieu xe lan o
to lan 2:
This is a Mercedes having a speed of 80km/h and its price is $5000 Gioi thieu xe bus:
This is a bus of type Mercedes, on the line 27, having a speed of 80km/h and its
price is $5000
```

Ở thông báo thứ nhất, đối tượng myCar gọi phương thức show() của lớp Car với các dữ liệu được khởi đầu cho myCar: (100, “Ford”, 3000). Ở thông báo thứ hai, myCar gọi phương thức show() của lớp Car, nhưng với dữ liệu vừa được gán từ đối tượng myBus: (80, “Mercedes”, 5000). Ở thông báo thứ ba, myBus gọi phương thức show() của lớp Bus với các dữ liệu của myBus: (80, “Mercedes”, 5000, 27).

6.4 ĐA KẾ THỪA

C++ cho phép đa kế thừa, tức là một lớp có thể được dẫn xuất từ nhiều lớp cơ sở khác nhau, với những kiểu dẫn xuất khác nhau.

6.4.1 Khai báo đa kế thừa

Đa kế thừa được khai báo theo cú pháp:

```
class <Tên lớp dẫn xuất>: <Từ khoá dẫn xuất> <Tên lớp cơ sở 1>,
<Từ khoá dẫn xuất> <Tên lớp cơ sở 2>,
...
<Từ khoá dẫn xuất> <Tên lớp cơ sở n>{
    ...
    // Khai báo thêm các thành phần lớp dẫn xuất
};
```

Ví dụ:

```
class Bus: public Car, public PublicTransport{
    ...
    // Khai báo các thành phần bổ sung
};
```

là khai báo lớp Bus (xe buýt) kế thừa từ hai lớp xe Car (ô tô) và PublicTransport (phương tiện giao thông công cộng) theo cùng một kiểu dẫn xuất là public.

Lưu ý:

- Trong đa kế thừa, mỗi lớp cơ sở được phân cách nhau bởi dấu phẩy “,”.
- Mỗi lớp cơ sở có thể có một kiểu dẫn xuất bởi một từ khoá dẫn xuất khác nhau.
- Nguyên tắc truy nhập vào các thành phần lớp cơ sở cũng hoàn toàn tương tự như trong kế thừa đơn.

6.4.2 Hàm khởi tạo và hàm huỷ bỏ trong đa kế thừa

Hàm khởi tạo trong đa kế thừa

Hàm khởi tạo trong đa kế thừa được khai báo tương tự như trong đơn kế thừa, ngoại trừ việc phải sắp xếp thứ tự gọi tới hàm khởi tạo của các lớp cơ sở: thông thường, thứ tự gọi đến hàm khởi tạo của các lớp cơ sở nên tuân theo thứ tự dẫn xuất từ các lớp cơ sở trong đa kế thừa.

Chương trình 6.6 minh họa việc định nghĩa hàm khởi tạo tường minh trong đa kế thừa: thứ tự gọi hàm khởi tạo của các lớp cơ sở trong hàm khởi tạo của lớp Bus là tương tự thứ tự dẫn xuất: hàm khởi tạo của lớp Car trước hàm khởi tạo của lớp PublicTransport.

Chương trình 6.6

```
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    int speed;                      // Tốc độ
    char mark[20];                  // Nhãn hiệu
    float price;                    // Giá xe

    public:
        Car();                      // Khởi tạo không tham số
        Car(int, char[], float);    // Khởi tạo đủ tham số
};

Car::Car() {                                // Khởi tạo không tham số
    speed = 0;
    strcpy(mark, "");
    price = 0;
}

// Khởi tạo đủ tham số
Car::Car(int speedIn, char markIn[], float priceIn) {
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}
```

Chương 6: Tính kế thừa và đa hình

```
{}

/* Định nghĩa lớp PublicTransport */ class
PublicTransport{
    float ticket;                                // Giá vé phương tiện public:
    PublicTransport();                            // Khởi tạo không tham số
    PublicTransport(float);                      // Khởi tạo đủ tham số
};

PublicTransport::PublicTransport(){           // Khởi tạo không tham số ticket =
    0;
}

// Khởi tạo đủ tham số PublicTransport::PublicTransport(float ticketIn){
    ticket = ticketIn;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */ class Bus: public
Car, public PublicTransport{ // Thứ tự khai báo
    int label;                                  // Số hiệu tuyến xe public:
    Bus();                                     // Khởi tạo không tham số Bus(int,
char[], float, float, int); // Khởi tạo đủ tham số
};

// Khởi tạo không tham số
Bus::Bus(): Car(), Transport(){           // Theo thứ tự dẫn xuất label =
    0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, float tIn, int lIn):
    Car(sIn, mIn, pIn), PublicTransport(tIn){ // Theo thứ tự dẫn xuất label = lIn;
}
```

Lưu ý:

- Trong trường hợp dùng hàm khởi tạo ngầm định hoặc không có tham số, ta có thể không cần gọi tường minh các hàm khởi tạo của các lớp cơ sở, trình biên dịch sẽ tự động gọi tới chúng theo đúng thứ tự dẫn xuất

Chương 6: Tính kế thừa và đa hình

Ví dụ, trong chương trình 6.6, hai cách định nghĩa hàm khởi tạo không tham số của lớp Bus sau là tương đương:

```
Bus::Bus(): Car(), Transport(){// Theo thứ tự dẫn xuất label = 0;  
}
```

là tương đương với:

```
Bus::Bus(){  
    // Theo thứ tự dẫn xuất ngầm định label = 0;  
}
```

Hàm huỷ bỏ trong đa kế thừa

Vì hàm huỷ bỏ là duy nhất của mỗi lớp, hòn nữa hàm huỷ bỏ của lớp cơ sở sẽ được tự động gọi đến khi giải phóng đối tượng của lớp dẫn xuất. Cho nên hàm huỷ bỏ trong đa kế thừa hoàn toàn tương tự hàm huỷ bỏ trong đơn kế thừa:

- Hàm huỷ bỏ của lớp dẫn xuất chỉ giải phóng bộ nhớ cho các thành phần bổ sung, nếu có, của lớp dẫn xuất.
- Hàm huỷ bỏ của lớp dẫn xuất sẽ được gọi đến sớm nhất. Sau đó các hàm huỷ bỏ của các lớp cơ sở sẽ được gọi đến.
- Quá trình này được trình biên dịch thực hiện tự động.

6.4.3 Truy nhập các thành phần lớp trong đa kế thừa

Việc truy nhập đến các thành phần của các lớp trong đa kế thừa được dựa trên các nguyên tắc sau:

- Việc truy nhập từ đối tượng lớp dẫn xuất đến các thành phần của mỗi lớp cơ sở được tuân theo quy tắc phạm vi tương tự như trong đơn kế thừa.
- Trong trường hợp các lớp cơ sở đều có các thành phần cùng tên, việc truy xuất đến thành phần của lớp nào phải được chỉ rõ bằng toán tử phạm vi: “<Tên lớp>::” đối với thành phần lớp cơ sở đó.

Ví dụ, ta định nghĩa lớp Bus kế thừa từ hai lớp cơ sở: Car và PublicTransport. Nhưng cả ba lớp này đều định nghĩa một phương thức show() để tự giới thiệu:

```
class Car{  
public:  
    void show();  
};  
class PublicTransport{ public:  
    void show();  
};  
class Bus: public Car, public PublicTransport{ public:  
    void show();  
};
```

Khi đó, khai báo:

```
Bus myBus;  
và lời gọi hàm:  
myBus.show(); // Gọi đến hàm của lớp Bus  
myBus.Car::show(); // Gọi đến hàm của lớp Car  
myBus.PublicTransport::show(); // Gọi đến hàm của lớp PublicTransport
```

Chương trình 6.7 minh họa việc truy nhập đến các thành phần trùng nhau trong các lớp cơ sở và được định nghĩa lại trong lớp dẫn xuất.

Chương trình 6.7

```
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
  
/* Định nghĩa lớp Car */  
class Car{  
    int speed; // Tốc độ  
    char mark[20]; // Nhãn hiệu  
    float price; // Giá xe  
  
public:  
    Car(); // Khởi tạo không tham số  
    Car(int, char[], float); // Khởi tạo đủ tham số  
    void show(); // Giới thiệu  
    float getSpeed(){return speed;};  
    char[] getMark(){return mark;};  
    float getPrice(){return price;};  
};  
  
Car::Car(){ // Khởi tạo không tham số  
    speed = 0;  
    strcpy(mark, "");  
    price = 0;  
}  
  
// Khởi tạo đủ tham số  
Car::Car(int speedIn, char markIn[], float priceIn){  
    speed = speedIn;  
    strcpy(mark, markIn);  
    price = priceIn;  
}
```

Chương 6: Tính kế thừa và đa hình

```
// Giới thiệu void
Car::show(){
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl; return;
}

/* Định nghĩa lớp PublicTransport */
class PublicTransport{
    float ticket; // Giá vé phương tiện public:
    PublicTransport(); // Khởi tạo không tham số
    PublicTransport(float); // Khởi tạo đủ tham số void
    show(); // Giới thiệu
    float getTicket(){return ticket;};
};

PublicTransport::PublicTransport() // Khởi tạo không tham số ticket =
{
    0;
}

// Khởi tạo đủ tham số PublicTransport::PublicTransport(float ticketIn){
    ticket = ticketIn;
}

// Giới thiệu
void PublicTransport::show(){
    cout << "This public transport had a ticket of $"
        << ticket << endl; return;
}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */
class Bus: public Car,
public PublicTransport{ // Thứ tự khai báo
    int label; // Số hiệu tuyến xe public:
    Bus(); // Khởi tạo không tham số Bus(int,
    char[], float, float, int); // Khởi tạo đủ tham số void show(); // Giới thiệu
};

// Khởi tạo không tham số
```

Chương 6: Tính kế thừa và đa hình

```
Bus::Bus(): Car(), Transport() { // Theo thứ tự dẫn xuất
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, float tIn, int lIn):
    Car(sIn, mIn, pIn), PublicTransport(tIn) { // Theo thứ tự dẫn xuất
    label = lIn;
}

// Giới thiệu
void Bus::show() {
    cout << "This is a bus on the line " << label
        << ", its speed is " << getSpeed()
        << "km/h, mark is " << getMark()
        << ", price is $" << getPrice()
        << " and ticket is " << getTicket() << endl;
    return;
}

// phương thức main
void main() {
    clrscr();
    Bus myBus(100, "Mercedes", 3000, 1.5, 27);

    myBus.Car::show(); // Hàm của lớp Car
    myBus.PublicTransport::show(); // Hàm của lớp PublicTransport
    myBus.show(); // Hàm của lớp Bus
    return;
}
```

Chương trình 6.7 sẽ in ra thông báo như sau:

This is a Mercedes having a speed of 100km/h and its price is \$3000 This public transport had a ticket of \$1.5

This is a bus on the line 27, its speed is 100km/h, mark is Mercedes, price is \$3000 and ticket is \$1.5

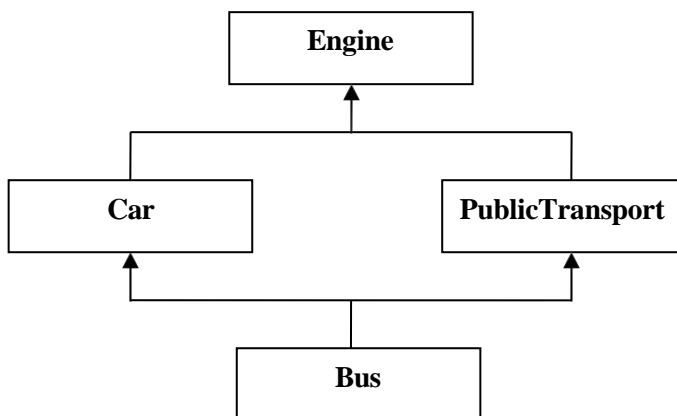
Dòng thứ nhất là kết quả của phương thức show() của lớp Car, dòng thứ hai, tương ứng là kết quả phương thức show() của lớp PublicTransport, dòng thứ ba là kết quả phương thức show() của lớp Bus.

6.5 LỚP CƠ SỞ TRÙU TƯỢNG

6.5.1 Đặt vấn đề

Sự cho phép đa kế thừa trong C++ dẫn đến một số hậu quả xấu, đó là sự đụng độ giữa các thành phần của các lớp cơ sở, khi có ít nhất hai lớp cơ sở lại cùng được kế thừa từ một lớp cơ sở khác. Xét trường hợp:

- Lớp Bus kế thừa từ lớp Car và lớp PublicTransport.
- Nhưng lớp Car và lớp PublicTransport lại cùng được thừa kế từ lớp Engine (động cơ). Lớp Engine có một thuộc tính là power (công suất của động cơ).



Khi đó, nảy sinh một số vấn đề như sau:

- Các thành phần dữ liệu của lớp Engine bị lặp lại trong lớp Bus hai lần: một lần do kế thừa theo đường Bus::Car::Engine, một lần theo đường Bus::PublicTransport::Engine. Điều này là không an toàn.
- Khi khai báo một đối tượng của lớp Bus, hàm khởi tạo của lớp Engine cũng được gọi hai lần: một lần do gọi truy hồi từ hàm khởi tạo lớp Car, một lần do gọi truy hồi từ hàm khởi tạo lớp PublicTransport.
- Khi giải phóng một đối tượng của lớp Bus, hàm huỷ bỏ của lớp Engine cũng sẽ bị gọi tới hai lần.

Để tránh các vấn đề này, C++ cung cấp một khái niệm là kế thừa từ *lớp cơ sở trừu tượng*. Khi đó, ta cho các lớp Car và PublicTransport kế thừa trừu tượng từ lớp Engine. Bằng cách này, các thành phần của lớp Engine chỉ xuất hiện trong lớp Bus đúng một lần. Lớp Engine được gọi là lớp cơ sở trừu tượng của các lớp Car và PublicTransport.

6.5.2 Khai báo lớp cơ sở trừu tượng

Việc chỉ ra một sự kế thừa trừu tượng được thực hiện bằng từ khoá **virtual** khi khai báo lớp cơ sở:

```

class <Tên lớp cơ sở>: <Từ khoá dẫn xuất> virtual <Tên lớp cơ sở>{
    ...
    // Khai báo các thành phần bổ sung
};
  
```

Ví dụ:

```

class Engine{
    ...
    // Các thành phần lớp Engine
  
```

```

};

class Car: public virtual Engine{
    ...      // Khai báo các thành phần bổ sung
};

```

là khai báo lớp Car, kế thừa từ lớp cơ sở trừu tượng Engine, theo kiểu dẫn xuất public.

Lưu ý:

- Từ khoá virtual được viết bằng chữ thường.
- Từ khoá virtual không ảnh hưởng đến phạm vi truy nhập thành phần lớp cơ sở, phạm vi này vẫn được quy định bởi từ khoá dẫn xuất như thông thường.
- Từ khoá virtual chỉ ra một lớp cơ sở là trừu tượng nhưng lại được viết trong khi khai báo lớp dẫn xuất.
- Một lớp dẫn xuất có thể được kế thừa từ nhiều lớp cơ sở trừu tượng

6.5.3 Hàm khởi tạo lớp cơ sở trừu tượng

Khác với các lớp cơ sở thông thường, khi có một lớp dẫn xuất từ một lớp cơ sở trừu tượng, lại được lấy làm cơ sở cho một lớp dẫn xuất khác thì trong hàm khởi tạo của lớp dẫn xuất cuối cùng, vẫn phải gọi hàm khởi tạo tường minh của lớp cơ sở trừu tượng. Hơn nữa, hàm khởi tạo của lớp cơ sở trừu tượng phải được gọi sớm nhất.

Ví dụ, khi lớp Car và lớp PublicTransport được kế thừa từ lớp cơ sở trừu tượng Engine. Sau đó, lớp Bus được kế thừa từ hai lớp Car và PublicTransport. Khi đó, hàm khởi tạo của lớp Bus cũng phải gọi tường minh hàm khởi tạo của lớp Engine, theo thứ tự sớm nhất, sau đó mới gọi đến hàm khởi tạo của các lớp Car và PublicTransport.

```

class Engine{
    public:
        Engine(){...};

};

class Car: public virtual Engine{                                //Lớp cơ sở virtual public:
    Car(): Engine(){...};

};

class PublicTransport: public virtual Engine{                   //Lớp cơ sở virtual public:
    PublicTransport():Engine(){...};

};

class Bus: public Car, public PublicTransport{ public:
    // Gọi hàm khởi tạo tường minh của lớp cơ sở trừu tượng Bus():Engine(),
    Car(), PublicTransport(){...};

};

```

Lưu ý:

- Trong trường hợp lớp Engine không phải là lớp cơ sở trừu tượng của các lớp Car và PublicTransport, thì trong hàm khởi tạo của lớp Bus không cần gọi hàm khởi tạo của lớp

Chương 6: Tính kế thừa và đa hình

Engine, mà chỉ cần gọi tới các hàm khởi tạo của các lớp cơ sở trực tiếp của lớp Bus là lớp Car và lớp PublicTransport.

Chương trình 6.8 minh họa việc khai báo và sử dụng lớp cơ sở trừu tượng: lớp Engine là lớp cơ sở trừu tượng của các lớp Car và lớp PublicTransport. Hai lớp này, sau đó, lại làm lớp cơ sở của lớp Bus.

Chương trình 6.8

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp Engine */
class Engine{
    int power; // Công suất
public:
    Engine() {power = 0;} // Khởi tạo không tham số
    Engine(int pIn) {power = pIn;} // Khởi tạo đủ tham số
    void show(); // Giới thiệu
    float getPower() {return power;}
};

// Giới thiệu
void Engine::show() {
    cout << "This is an engine having a power of "
        << power << "KWH" << endl;
    return;
}

/* Định nghĩa lớp Car dẫn xuất từ lớp cơ sở trừu tượng Engine*/
class Car: public virtual Engine{
    int speed; // Tốc độ
    char mark[20]; // Nhãn hiệu
    float price; // Giá xe
public:
    Car(); // Khởi tạo không tham số
    Car(int, int, char[], float); // Khởi tạo đủ tham số
    void show(); // Giới thiệu
    float getSpeed() {return speed;}
    char[] getMark() {return mark;}
    float getPrice() {return price;}
};
```

Chương 6: Tính kế thừa và đa hình

```
Car::Car(): Engine(){ // Khởi tạo không tham số speed =
    0;
    strcpy(mark, ""); price =
    0;
}

// Khởi tạo đủ tham số
Car::Car(int pwIn, int sIn, char mIn[], float prIn): Engine(pwIn){ speed = sIn;
    strcpy(mark, mIn); price =
    prIn;
}

// Giới thiệu void
Car::show(){
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h, its power is" << getPower()
        << "KWh and price is $" << price << endl; return;
}

/* Định nghĩa lớp PublicTransport dẫn xuất trùu tượng từ lớp Engine */
class PublicTransport{
public virtual Engine{
    float ticket; // Giá vé phương tiện public:
    PublicTransport(); // Khởi tạo không tham số
    PublicTransport(int, float); // Khởi tạo đủ tham số void
    show(); // Giới thiệu
    float getTicket(){return ticket;};
};

// Khởi tạo không tham số PublicTransport::PublicTransport():
Engine(){
    ticket = 0;
}

// Khởi tạo đủ tham số
PublicTransport::PublicTransport(int pwIn, float tIn): Engine(pwIn){ ticket = tIn;
}

// Giới thiệu
void PublicTransport::show(){
```

```

cout << "This is a public transport having a ticket of $"  

     << ticket << " and its power is " << getPower()  

     << "KWh" << endl; return;  

}

/* Định nghĩa lớp Bus kế thừa từ lớp Car và PublicTransport */ class Bus: public Car,  

public PublicTransport{ // Thứ tự khai báo  

    int label; // Số hiệu tuyến xe public:  

    Bus(); // Khởi tạo không tham số  

    Bus(int,int,char[],float,float,int); // Khởi tạo đủ tham số void show(); // Giới  

    thiệu  

};

// Khởi tạo không tham số  

Bus::Bus(): Engine(), Car(), Transport(){ // Theo thứ tự dẫn xuất label = 0;  

}

// Khởi tạo đủ tham số  

Bus::Bus(int pwIn, int sIn, char mIn[], float prIn, float tIn, int lIn): Engine(pwIn), Car(sIn, mIn, prIn),  

    PublicTransport(tIn){  

    label = lIn;  

}

// Giới thiệu void  

Bus::show(){  

    cout << "This is a bus on the line " << label  

     << ", its speed is " << getSpeed()  

     << "km/h, power is" << Car::getPower()  

     << "KWh, mark is " << getMark()  

     << ", price is $" << getPrice()  

     << " and ticket is " << getTicket() << endl; return;  

}

// phương thức main void  

main(){  

    clrscr();  

    Bus myBus(250, 100, "Mercedes", 3000, 1.5, 27);  

    myBus.Car::Engine::show(); // Hàm của lớp Engine
}

```

Chương 6: Tính kế thừa và đa hình

```
myBus.PublicTransport::Engine::show() // Hàm của lớp Engine  
myBus.Car::show() // Hàm của lớp Car  
myBus.PublicTransport:: show() // Hàm của lớp PublicTransport  
myBus.show() // Hàm của lớp Bus  
return;  
}
```

Chương trình 6.8 sẽ in ra thông báo như sau:

This is an engine having a power of 250KWh This is an
engine having a power of 250KWh

This is a Mercedes having a speed of 100km/h, its power is 250KWh and price is \$3000

This is a public transport having a ticket of \$1.5 and its power is 250KWh

This is a bus on the line 27, its speed is 100km/h, power is 250KWh, mark is Mercedes,
price is \$3000 and ticket is \$1.5

Hai dòng đầu là kết quả của phương thức show() của lớp Engine: một lần gọi qua lớp Car, một lần
gọi qua lớp PublicTransport, chúng cho kết quả như nhau. Dòng thứ ba là kết quả phương thức
show() của lớp Car. Dòng thứ tư, tương ứng là kết quả phương thức show() của lớp
PublicTransport. Dòng thứ năm là kết quả phương thức show() của lớp Bus.

6.6 ĐA HÌNH

6.6.1 Đặt vấn đề

Sự kế thừa trong C++ cho phép có sự tương ứng giữa lớp cơ sở và các lớp dẫn xuất trong sơ đồ
thừa kế:

- Một con trỏ có kiểu lớp cơ sở luôn có thể trỏ đến địa chỉ của một đối tượng của lớp dẫn
xuất.
- Tuy nhiên, khi thực hiện lời gọi một phương thức của lớp, trình biên dịch sẽ quan tâm đến
kiểu của con trỏ chứ không phải đối tượng mà con trỏ đang trỏ tới: phương thức của lớp
mà con trỏ có kiểu được gọi chứ không phải phương thức của đối tượng mà con trỏ đang
trỏ tới được gọi.

Ví dụ, lớp Bus kế thừa từ lớp Car, cả hai lớp này đều định nghĩa phương thức show():

```
class Car{  
public:  
    void show();  
};  
class Bus: public Car{ public:  
    void show();  
};
```

khi đó, nếu ta khai báo một con trỏ lớp Bus, nhưng lại trỏ vào địa chỉ của một đối tượng lớp Car:

```
Bus myBus;  
Car *ptrCar = &myBus; // đúng
```

nhưng khi gọi:

```
ptrCar->show();
```

thì chương trình sẽ gọi đến phương thức show() của lớp Car (là kiểu của con trỏ ptrCar), mà không gọi tới phương thức show() của lớp Bus (là kiểu của đối tượng myBus mà con trỏ ptrCar đang trỏ tới).

Để giải quyết vấn đề này, C++ đưa ra một khái niệm là phương thức trùu tượng. Bằng cách sử dụng phương thức trùu tượng. Khi gọi một phương thức từ một con trỏ đối tượng, trình biên dịch sẽ xác định kiểu của đối tượng mà con trỏ đang trỏ đến, sau đó nó sẽ gọi phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới.

6.6.2 Khai báo phương thức trùu tượng

Phương thức trùu tượng (còn gọi là phương thức áo, hàm áo) được khai báo với từ khoá **virtual**:

- Nếu khai báo trong phạm vi lớp:

```
virtual <Kiểu trả về> <Tên phương thức>([<Các tham số>]);
```

- Nếu định nghĩa ngoài phạm vi lớp:

```
virtual <Kiểu trả về> <Tên lớp>::<Tên phương thức>([<Các tham số>]) {...}
```

Ví dụ:

```
class Car{  
public:  
    virtual void show();  
};
```

là khai báo phương thức trùu tượng show() của lớp Car: phương thức không có tham số và không cần giá trị trả về (void).

Lưu ý:

- Từ khoá virtual có thể đặt trước hay sau kiểu trả về của phương thức.
- Với cùng một phương thức được khai báo ở lớp cơ sở lẫn lớp dẫn xuất, chỉ cần dùng từ khoá virtual ở một trong hai lần định nghĩa phương thức đó là đủ: hoặc ở lớp cơ sở, hoặc ở lớp dẫn xuất.
- Trong trường hợp cây kế thừa có nhiều mức, cũng chỉ cần khai báo phương thức là trùu tượng (virtual) ở một mức bất kì. Khi đó, tất cả các phương thức trùng tên với phương thức đó ở tất cả các mức đều được coi là trùu tượng.
- Đôi khi không cần thiết phải định nghĩa chồng (trong lớp dẫn xuất) một phương thức đã được khai báo trùu tượng trong lớp cơ sở.

6.6.3 Sử dụng phương thức trùu tượng – đa hình

Một khi phương thức được khai báo là trùu tượng thì khi một con trỏ gọi đến phương thức đó, chương trình sẽ thực hiện phương thức tương ứng với đối tượng mà con trỏ đang trỏ tới, thay vì

Chương 6: Tính kế thừa và đa hình

thực hiện phương thức của lớp cùng kiểu với con trỏ. Đây được gọi là hiện tượng đa hình (tương ứng bội) trong C++.

Chương trình 6.9 minh họa việc sử dụng phương thức trùu tượng: lớp Bus kế thừa từ lớp Car, hai lớp này cùng định nghĩa phương thức trùu tượng show().

- Khi ta dùng một con trỏ có kiểu lớp Car trỏ vào địa chỉ của một đối tượng kiểu Car, nó sẽ gọi phương thức show() của lớp Car.
- Khi ta dùng cũng con trỏ đó, trỏ vào địa chỉ của một đối tượng kiểu Bus, nó sẽ gọi phương thức show() của lớp Bus.

Chương trình 6.9

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

/* Định nghĩa lớp Car */
class Car{
    private:
        int speed; // Tốc độ
        char mark[20]; // Nhãn hiệu
        float price; // Giá xe

    public:
        int getSpeed(){return speed;} // Đọc tốc độ xe
        char[] getMark(){return mark;} // Đọc nhãn xe
        float getPrice(){return price;} // Đọc giá xe
        // Khởi tạo thông tin về xe
        Car(int speedIn=0, char markIn[]="", float priceIn=0);
        virtual void show(); // Giới thiệu xe, trùu tượng
};

/* Khai báo phương thức bên ngoài lớp */
Car::Car(int speedIn, char markIn[], float priceIn){
    speed = speedIn;
    strcpy(mark, markIn);
    price = priceIn;
}

// Phương thức trùu tượng giới thiệu xe
virtual void Car::show(){
    cout << "This is a " << mark << " having a speed of "
        << speed << "km/h and its price is $" << price << endl;
    return;
}
```

Chương 6: Tính kế thừa và đa hình

```
/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label;           // Số hiệu tuyến xe
public:
    // Khởi tạo đủ tham số
    Bus(int sIn=0, char mIn[]="", float pIn=0, int lIn=0);
    void show();         // Giới thiệu xe
};

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}

// Định nghĩa nạp chồng phương thức trừu tượng
void Bus::show() {           // Giới thiệu xe bus
    cout << "This is a bus of type " << getMark() << ", on the line "
        << label << ", having a speed of " << getSpeed()
        << "km/h and its price is $" << getPrice() << endl;
    return;
}

// Chương trình chính
void main() {
    clrscr();
    Car *ptrCar, myCar(100, "Ford", 3000);
    Bus myBus(150, "Mercedes", 5000, 27); // Biến đối tượng của lớp Bus

    ptrCar = &myCar;           // Trỏ đến đối tượng lớp Car
    ptrCar->show();           // Phương thức của lớp Car

    ptrCar = &myBus;           // Trỏ đến đối tượng lớp Bus
    ptrCar->show();           // Phương thức của lớp Bus
    return;
}
```

Chương trình 6.9 hiển thị kết quả thông báo như sau:

This is a Ford having a speed of 100km/h and its price is \$3000

This is a bus of type Mercedes, on the line 27, having a speed of 150km/h and its price is \$5000

Dòng thứ nhất là kết quả khi con trỏ ptrCar trỏ đến địa chỉ của đối tượng myCar, thuộc lớp Car
nên sẽ gọi phương thức show() của lớp Car với các dữ liệu của đối tượng myCar: (100, Ford,

3000). Dòng thứ hai tương ứng là kết quả khi con trỏ ptrCar trỏ đến địa chỉ của đối tượng myBus, thuộc lớp Bus nên sẽ gọi phương thức show() của lớp Bus, cùng với các tham số của đối tượng myBus: (150, Mercedes, 5000, 27).

Lưu ý:

- Trong trường hợp ở lớp dẫn xuất không định nghĩa lại phương thức trừu tượng, thì chương trình sẽ gọi phương thức của lớp cơ sở, nhưng với dữ liệu của lớp dẫn xuất.

Ví dụ, nếu trong chương trình 6.9, lớp Bus không định nghĩa chồng phương thức trừu tượng show() thì kết quả hiển thị sẽ là hai dòng thông báo giống nhau, chỉ khác nhau ở dữ liệu của hai đối tượng khác nhau:

This is a Ford having a speed of 100km/h and its price is \$3000
This is a Mercedes having a speed of 150km/h and its price is \$5000

TỔNG KẾT CHƯƠNG 6

Nội dung chương 6 đã trình bày các vấn đề cơ bản liên quan đến thừa kế và tương ứng bội trong C++ như sau:

- Khai báo một lớp dẫn xuất kế thừa từ một lớp cơ sở bằng khai báo kế thừa “:” đi kèm với một từ khoá dẫn xuất.
- Có ba loại dẫn xuất khác nhau, được quy định bởi ba từ khoá dẫn xuất khác nhau: private, protected và public. Kiểu dẫn xuất phổ biến là dẫn xuất public.
- Sự kế thừa tạo ra mối quan hệ tương ứng giữa lớp cơ sở và lớp dẫn xuất: có thể chuyển kiểu ngầm định (trong phép gán, phép truyền đối số, phép trả địa chỉ) từ một đối tượng lớp cơ sở đến một đối tượng lớp dẫn xuất. Nhưng không thể chuyển kiểu ngược lại từ lớp dẫn xuất vào lớp cơ sở.
- Hàm khởi tạo của lớp dẫn xuất có thể gọi tường minh hoặc gọi ngầm định hàm khởi tạo của lớp cơ sở. Hàm khởi tạo lớp cơ sở bao giờ cũng được thực hiện trước hàm khởi tạo lớp dẫn xuất.
- Hàm huỷ bỏ của lớp dẫn xuất luôn gọi ngầm định hàm huỷ bỏ của lớp cơ sở. Trái với hàm khởi tạo, hàm huỷ bỏ lớp cơ sở luôn được thực hiện sau hàm huỷ bỏ của lớp dẫn xuất.
- Có thể truy nhập các phương thức của lớp cơ sở từ lớp dẫn xuất, phạm vi truy nhập là phụ thuộc vào kiểu dẫn xuất: private, protected hoặc public. Điều này cho phép sử dụng lại mã nguồn của lớp cơ sở, mà không cần định nghĩa lại ở lớp dẫn xuất.
- Trong lớp dẫn xuất, có thể định nghĩa chồng một số phương thức của lớp cơ sở. Khi có định nghĩa chồng, muốn truy nhập vào phương thức lớp cơ sở, phải dùng toán tử phạm vi lớp “<Tên lớp>::”.
- C++ còn cho phép một lớp có thể được dẫn xuất từ nhiều lớp cơ sở khác nhau, gọi là đa kế thừa. Trong đa kế thừa, quan hệ giữa lớp dẫn xuất với mỗi lớp cơ sở là tương tự như trong đơn kế thừa.
- Trong đa kế thừa, hàm khởi tạo lớp dẫn xuất sẽ gọi tường minh (hoặc ngầm định) hàm khởi tạo các lớp cơ sở, theo thứ tự khai báo kế thừa. Hàm huỷ bỏ lớp dẫn xuất lại gọi ngầm định các hàm huỷ bỏ của các lớp cơ sở.

- C++ cung cấp khái niệm kế thừa từ lớp cơ sở trừu tượng để tránh trường hợp trùng lặp dữ liệu ở lớp dẫn xuất, khi các lớp cơ sở lại cùng được dẫn xuất từ một lớp khác.
- C++ cũng cho phép cơ chế tương ứng bội (đa hình) bằng cách định nghĩa một phương thức là trừu tượng trong sơ đồ thừa kế. Khi đó, một con trỏ lớp cơ sở có thể trỏ đến địa chỉ của một đối tượng lớp dẫn xuất, và phương thức được thực hiện là tùy thuộc vào kiểu của đối tượng mà con trỏ đang trỏ tới.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 6

1. Trong các khai báo sau, khai báo nào là đúng cú pháp kế thừa lớp:
 - a. class A: public class B{...};
 - b. class A: public B{...};
 - c. class A: class B{...};
 - d. class A:: public B{...};
2. Trong các kiểu dẫn xuất sau, từ các phương thức lớp dẫn xuất, không thể truy nhập đến các thành phần private của lớp cơ sở:
 - a. private
 - b. protected
 - c. public
 - d. Cả ba kiểu trên
3. Trong các kiểu dẫn xuất sau, từ đối tượng của lớp dẫn xuất, có thể truy nhập đến các thành phần của lớp cơ sở:
 - a. private
 - b. protected
 - c. public
 - d. Cả ba kiểu trên
4. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử có các kiểu khai báo:
 - A myA, *ptrA;
 - B myB, *ptrB;

Khi đó, các lệnh nào sau đây là không có lỗi:

- a. myA = myB;
- b. myB = myA;
- c. ptrA = &myB;
- d. ptrB = &myA;
- e. ptrA = ptrB;
- f. ptrB = ptrA;

5. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử có các kiểu khai báo và nguyên mẫu hàm:

- A myA;
 - B myB;
- ```
void show(A, B);
```

Khi đó, các lệnh gọi hàm nào sau đây là không có lỗi:

- a. show(myA, myA);
- b. show(myA, myB);
- c. show(myB, myA);
- d. show(myB, myB);

6. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có một hàm khởi tạo:

B(int, float);

Khi đó, định nghĩa hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

- a. A::A() {...};
- b. A::A(): B() {...};
- c. A::A(int x, float y): B() {...};
- d. A::A(int x, float y): B(x, y) {...};

7. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có hai hàm khởi tạo:

B();

B(int, float);

Khi đó, những định nghĩa hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

- a. A::A() {...};
- b. A::A(): B() {...};
- c. A::A(int x, float y): B() {...};
- d. A::A(int x, float y): B(x, y) {...};

8. A là lớp dẫn xuất public từ lớp cơ sở B. Giả sử B có hàm huỷ bỏ tường minh:

~B();

Khi đó, những định nghĩa hàm huỷ bỏ nào sau đây của lớp A là chấp nhận được:

- a. A::~A() {...};
- b. A::~A(): ~B() {...};
- c. A::~A(int x) {...};
- d. A::~A(int x): ~B() {...};

9. Giả sử B là một lớp được khai báo:

```
class B{
 int x;
 public: int getx();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu private:

```
class A: private B{
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. myA.x;
- b. myA.getx();
- c. Cả hai lệnh trên.

d. Không lệnh nào cả.

10. Giả sử B là một lớp được khai báo:

```
class B{
 int x;
 public: int getx();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu protected:

```
class A: protected B{
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. myA.x;
- b. myA.getx();
- c. Cả hai lệnh trên.
- d. Không lệnh nào cả.

11. Giả sử B là một lớp được khai báo:

```
class B{
 int x;
 public: int getx();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu public:

```
class A: public B{
};
```

khi đó, nếu myA là một đối tượng lớp A, lệnh nào sau đây là chấp nhận được:

- a. myA.x;
- b. myA.getx();
- c. Cả hai lệnh trên.
- d. Không lệnh nào cả.

12. Giả sử B là một lớp được khai báo:

```
class B{
 public: void show();
};
```

và A là một lớp dẫn xuất từ lớp B theo kiểu public, có định nghĩa chòng hàm show():

```
class A: public B{
 public: void show();
};
```

khi đó, nếu myA là một đối tượng lớp A, muốn thực hiện phương thức show() của lớp B thì lệnh nào sau đây là chấp nhận được:

- a. myA.show();
- b. myA.B::show();
- c. B::myA.show();

d. A::B::show();

13. Muốn khai báo một lớp A kế thừa từ hai lớp cơ sở B và C, những lệnh nào là đúng:

- a. class A: B, C{...};
- b. class A: public B, C{...};
- c. class A: public B, protected C{...};
- d. class A: public B, public C{...};

14. B là một lớp có hai hàm khởi tạo:

B();

B(int);

C cũng là một lớp có hai hàm khởi tạo:

C();

C(int, int);

Và A là một lớp kế thừa từ B và C:

class A: public B, public C{...};

Khi đó, hàm khởi tạo nào sau đây của lớp A là chấp nhận được:

- a. A::A(){...};
- b. A::A():B(),C(){...};
- c. A::A(int x, int y): C(x, y){...};
- d. A::A(int x, int y, int z): B(x), C(y, z){...};
- e. A::A(int x, int y, int z): C(x, y), B(z){...};

15. Muốn khai báo lớp A kế thừa từ lớp cơ sở trùu tượng B, những khai báo nào sau đây là đúng:

- a. virtual class A: public B{...};
- b. class virtual A: public B{...};
- c. class A: virtual public B{...};
- d. class A: public virtual B{...};

16. Lớp A là một lớp dẫn xuất, được kế thừa từ lớp cơ sở B. Hai lớp này đều định nghĩa hàm show(). Muốn hàm này trở thành trùu tượng thì những định nghĩa nào sau đây là đúng:

- a. void A::show(){...} và void B::show(){...}
- b. virtual void A::show(){...} và void B::show(){...}
- c. void A::show(){...} và virtual void B::show(){...}
- d. virtual void A::show(){...} và virtual void B::show(){...}

17. Khai báo lớp người (Human) bao gồm các thuộc tính sau:

- Tên người (name)
- Tuổi của người đó (age)
- Giới tính của người đó (sex)

Sau đó khai báo lớp Cá nhân (Person) kế thừa từ lớp Human vừa được định nghĩa ở trên.

- 
18. Bổ sung các phương thức truy nhập các thuộc tính của lớp Human, các phương thức này có tính chất public.
  19. Bổ sung thêm các thuộc tính của lớp Person: địa chỉ và số điện thoại. Thêm các phương thức truy nhập các thuộc tính này trong lớp Person.
  20. Xây dựng hai hàm khởi tạo cho lớp Human: một hàm không tham số, một hàm với đủ ba tham số tương ứng với ba thuộc tính của nó. Sau đó, xây dựng hai hàm khởi tạo cho lớp Person có sử dụng các hàm khởi tạo của lớp Human: một hàm không tham số, một hàm đủ năm tham số (ứng với hai thuộc tính của lớp Person và ba thuộc tính của lớp Human).
  21. Xây dựng một hàm main, trong đó có yêu cầu nhập các thuộc tính để tạo một đối tượng có kiểu Human và một đối tượng có kiểu Person, thông qua các hàm set thuộc tính đã xây dựng.
  22. Xây dựng hàm show() cho hai lớp Human và Person. Thay đổi hàm main: dùng một đối tượng có kiểu lớp Person, gọi hàm show() của lớp Person, sau đó lại gọi hàm show() của lớp Human từ chính đối tượng đó.
  23. Khai báo thêm một lớp người lao động (Worker), kế thừa từ lớp Human, có thêm thuộc tính là số giờ làm việc trong một tháng (hour) và tiền lương của người đó (salary). Sau đó, khai báo thêm một lớp nhân viên (Employee) kế thừa đồng thời từ hai lớp: Person và Worker. Lớp Employee có bổ sung thêm một thuộc tính là chức vụ (position).
  24. Chuyển lớp Human thành lớp cơ sở trùu tượng của hai lớp Person và Worker. Xây dựng thêm hai hàm show() của lớp Worker và lớp Employee. Trong hàm main, khai báo một đối tượng lớp Employee, sau đó gọi đến các hàm show() của các lớp Employee, Person, Worker và Human.
  25. Chuyển hàm show() trong các lớp trên thành phương thức trùu tượng. Trong hàm main, khai báo một con trỏ kiểu Human, sau đó, cho nó trỏ đến lần lượt các đối tượng của các lớp Human, Person, Worker và Employee, mỗi lần đều gọi phương thức show() để hiển thị thông báo ra màn hình.

# CHƯƠNG 7

## MỘT SỐ LỚP QUAN TRỌNG

Nội dung chương này tập trung trình bày một số lớp cơ bản trong C++:

- Lớp vật chứa (Container)
- Lớp tập hợp (Set)
- Lớp chuỗi kí tự (String)
- Lớp ngăn xếp (Stack) và hàng đợi (Queue)
- Lớp danh sách liên kết (Lists)

### 7.1 LỚP VẬT CHỨA

Lớp vật chứa (Container) bao gồm nhiều lớp cơ bản của C++: lớp Vector, lớp danh sách (List) và các kiểu hàng đợi (Stack và Queue), lớp tập hợp (Set) và lớp ánh xạ (Map). Trong chương này sẽ trình bày một số lớp cơ bản của Container là: Set, Stack, Queue và List

#### 7.1.1 Giao diện của lớp Container

Các lớp cơ bản của Container có một số toán tử và phương thức có chức năng giống nhau, bao gồm:

- **==:** Toán tử so sánh bằng
- **<:** Toán tử so sánh nhỏ hơn
- **begin():** Giá trị khởi đầu của con chạy iterator
- **end():** Giá trị kết thúc của con chạy iterator
- **size():** Số lượng phần tử đối tượng của vật chứa
- **empty():** Vật chứa là rỗng
- **front():** Phần tử thứ nhất của vật chứa
- **back():** Phần tử cuối của vật chứa
- **[]:** Toán tử truy nhập đến phần tử của vật chứa
- **insert():** Thêm vào vật chứa một (hoặc một số) phần tử
- **push\_back():** Thêm một phần tử vào cuối vật chứa
- **push\_front():** Thêm một phần tử vào đầu vật chứa
- **erase():** Loại bỏ một (hoặc một số) phần tử khỏi vật chứa
- **pop\_back():** Loại bỏ phần tử cuối của vật chứa
- **pop\_front():** Loại bỏ phần tử đầu của vật chứa.

Ngoài ra, tuỳ vào các lớp cụ thể mà có một số toán tử và phương thức đặc trưng của lớp đó. Các toán tử và phương thức này sẽ được trình bày chi tiết trong nội dung từng lớp tiếp theo.

## 7.1.2 Con chạy Iterator

Iterator là một con trỏ trỏ đến các phần tử của vật chứa. Nó đóng vai trò là một con chạy cho phép người dùng di chuyển qua từng phần tử có mặt trong vật chứa. Mỗi phép tăng giảm một đơn vị của con chạy này tương ứng với một phép dịch đến phần tử tiếp theo hay phần tử trước của phần tử hiện tại mà con chạy đang trỏ tới.

### Khai báo con chạy

Cú pháp chung để khai báo một biến con chạy iterator như sau:

```
Tên_lớp<T>::iterator Tên_con_chạy;
```

Trong đó:

- **Tên lớp:** là tên của lớp cơ bản ta đang dùng, ví dụ lớp Set, lớp List...
- **T:** là tên kiểu lớp của các phần tử chứa trong vật chứa. Kiểu có thể là các kiểu cơ bản trong C++, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.
- **Tên con chạy:** là tên biến sẽ được sử dụng làm biến chạy trong vật chứa.

Ví dụ:

```
Set<int>::iterator iter;
```

là khai báo một biến con chạy iter cho lớp tập hợp (Set), trong đó, các phần tử của lớp vật chứa có kiểu cơ bản int. Hoặc:

```
List<Person>::iterator iter;
```

là khai báo một biến con chạy iter cho lớp danh sách (List), trong đó, các phần tử có kiểu lớp do người dùng tự định nghĩa là Person.

### Sử dụng con chạy

Con chạy được sử dụng khi cần duyệt lần lượt các phần tử có mặt trong vật chứa. Ví dụ sau sẽ in ra các phần tử có kiểu int của một tập hợp:

```
Set<int> mySet; // Khai báo một đối tượng của lớp Set,
// các phần tử có kiểu int Set<int>::iterator i;
// Khai báo con chạy của lớp Set,
// các phần tử có kiểu int
... // Thêm phần tử vào
for(i=mySet.begin(); i<mySet.end(); i++) cout << mySet[i]
<< " ";
```

### Lưu ý:

- Biến con chạy phải được khởi đầu bằng phương thức begin() và kết thúc bằng phương thức end() của đối tượng cần duyệt tương ứng.
- Một con chạy có thể được sử dụng nhiều lần cho nhiều đối tượng nếu chúng có cùng kiểu lớp vật chứa và các phần tử của chúng cũng cùng kiểu.

## 7.2 LỚP TẬP HỢP

Lớp tập hợp (Set) chứa các phần tử có cùng kiểu, không phân biệt thứ tự giữa các phần tử nhưng lại phân biệt giữa các phần tử: các phần tử là khác nhau từng đôi một. Muốn sử dụng lớp tập hợp, phải có chỉ thị đầu tệp:

```
#include<set.h> // Thư viện riêng của lớp tập hợp
hoặc:
#include<stl.h> // Thư viện chung cho các lớp vật chứa
```

### 7.2.1 Hàm khởi tạo

Lớp tập hợp có ba kiểu khởi tạo chính:

- Khởi tạo không tham số:  
    Set<T> Tên\_đối\_tượng;
- Khởi tạo bằng một mảng các đối tượng phần tử:  
    Set<T> Tên\_đối\_tượng(T\*, chiều\_dài\_mảng);
- Khởi tạo bằng một đối tượng thuộc lớp tập hợp khác:  
    Set<T> Tên\_đối\_tượng(Set<T>);

Trong đó:

- **T**: là tên kiểu của các phần tử của tập hợp. Kiểu này có thể là kiểu cơ bản của C++, cũng có thể là các kiểu cấu trúc (struct) hoặc lớp (class) do người dùng tự định nghĩa.

Ví dụ:

```
Set<int> mySet;
```

là khai báo một đối tượng mySet của lớp tập hợp, mySet khởi đầu chưa có phần tử nào, các phần tử của đối tượng này có kiểu cơ bản int. Hoặc:

```
Person *myPerson = new Person[10]; // Khởi tạo mảng đối tượng
Set<Person> mySet(myPerson, 10); // Khai báo tập hợp
```

là khai báo một đối tượng mySet của lớp tập hợp, có 10 phần tử tương ứng với các phần tử trong mảng động myPerson, các phần tử có kiểu lớp Person.

### 7.2.2 Toán tử

Các toán tử trên lớp tập hợp là tương ứng với các toán tử số học trên tập hợp thông thường: OR “|”, AND “&”, XOR “^” và phép trừ “-”.

#### Phép toán “|” và “|=”

Phép toán này trả về phép hợp (OR) của hai đối tượng của lớp tập hợp, kết quả cũng là một đối tượng của lớp tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> | <Đối_tượng_3>;
<Đối_tượng_1> |= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 | mySet2;
thì mySet3 sẽ chứa các phần tử kiểu int là {1,2,3,4}.
```

### Phép toán “&” và “&=”

Phép toán này trả về phép giao (AND) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> & <Đối_tượng_3>;
<Đối_tượng_1> &= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 & mySet2;
thì mySet3 sẽ chứa các phần tử có kiểu int là {2}.
```

### Phép toán “^” và “^=”

Phép toán này trả về phép hợp ngoại (XOR) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> ^ <Đối_tượng_3>;
<Đối_tượng_1> ^= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 ^ mySet2;
thì mySet3 sẽ chứa các phần tử có kiểu int là {1,3,4}.
```

### Phép toán “-” và “-=”

Phép toán này trả về phép hiệu (loại trừ) giữa hai đối tượng tập hợp, kết quả cũng là một đối tượng tập hợp:

```
<Đối_tượng_1> = <Đối_tượng_2> - <Đối_tượng_3>;
<Đối_tượng_1> -= <Đối_tượng_2>;
```

Ví dụ, mySet1 chứa hai phần tử kiểu int {1,2}, mySet2 chứa ba phần tử kiểu int {2,3,4}, và:

```
mySet3 = mySet1 - mySet2;
thì mySet3 sẽ chứa các phần tử có kiểu int là {1}.
```

### 7.2.3 Phương thức

Lớp tập hợp có một số phương thức cơ bản sau:

- Thêm một phần tử vào tập hợp
- Loại một phần tử khỏi tập hợp
- Tìm kiếm một phần tử trong tập hợp

#### Thêm một phần tử vào tập hợp

Có hai cú pháp để thêm một phần tử vào tập hợp:

```
pair<iterator, bool> insert(T&); iterator insert(<Vị trí
con chạy>, T&);
```

Trong đó:

## Chương 7: Một số lớp quan trọng

- Phương thức thứ nhất thêm một phần tử vào tập hợp, nếu phần tử đã có mặt trong tập hợp, trả về false và vị trí con chạy của phần tử đó. Nếu phần tử chưa tồn tại, trả về true và vị trí con chạy của phần tử mới thêm vào.
- Phương thức thứ hai cũng thêm vào một phần tử, nhưng chỉ kiểm tra xem phần tử đã tồn tại hay chưa bắt đầu tự vị trí con chạy được chỉ ra trong biến <vị trí con chạy>. Phương thức này trả về vị trí con chạy của phần tử mới thêm vào (nếu thành công) hoặc vị trí của phần tử đã có mặt.

Ví dụ, mySet là một tập hợp kiểu int:

```
mySet.insert(10);
```

sẽ thêm một phần tử có giá trị 10 vào tập hợp.

### Lưu ý:

- Do tuân thủ theo lí thuyết tập hợp, nên khi chèn vào tập hợp nhiều lần với cùng một giá trị phần tử. Trong tập hợp chỉ tồn tại duy nhất một giá trị phần tử đó, không được trùng lặp.

### Loại một phần tử khỏi tập hợp

Có ba cú pháp để loại bỏ phần tử khỏi tập hợp:

```
int erase(T&);
void erase(<vị trí con chạy>);
void erase(<vị trí bắt đầu>, <vị trí kết thúc>);
```

Trong đó:

- T: là tên kiểu các phần tử của tập hợp.
- Phương thức thứ nhất xoá phần tử có giá trị được chỉ rõ trong tham số đầu vào.
- Phương thức thứ hai loại bỏ phần tử ở vị trí của con chạy được xác định bởi tham số đầu vào.
- Phương thức thứ ba loại bỏ một số phần tử nằm trong phạm vi từ <vị trí bắt đầu> cho đến <vị trí kết thúc> của con chạy.

Ví dụ, mySet là một tập hợp các phần tử có kiểu int:

```
mySet.erase(10);
```

sẽ xoá phần tử có giá trị 10, hoặc:

```
mySet.erase((Set<int>::iterator)10);
```

sẽ xoá phần tử ở vị trí thứ 10 trong tập hợp, hoặc:

```
mySet.erase(mySet.begin(), mySet.end());
```

sẽ xoá toàn bộ các phần tử hiện có của tập hợp mySet.

### Tìm kiếm một phần tử trong tập hợp

Có hai cú pháp để tìm kiếm một phần tử trong tập hợp:

```
iterator find(T&); int
count(T&);
```

Trong đó:

- Phương thức thứ nhất tìm phần tử có giá trị xác định bởi tham số đầu vào, kết quả là vị trí con chạy của phần tử đó.
- Phương thức thứ hai chỉ để kiểm tra xem phần tử có xuất hiện trong tập hợp hay không: trả về 1 nếu có mặt, trả về 0 nếu không có mặt.

Ví dụ, mySet là một tập hợp các phần tử có kiểu int:

```
Set<int>::iterator index = mySet.find(10); cout << index;
sẽ hiển thị vị trí con chạy của phần tử có giá trị 10 trong tập hợp mySet.
```

#### 7.2.4 Áp dụng

Chương trình 7.1 minh họa một số thao tác trên một tập hợp các phần tử có kiểu char: thêm phần tử, loại bỏ phần tử, duyệt các phần tử.

##### Chương trình 7.1

```
#include<stdio.h>
#include<conio.h>
#include<set.h>
void main(){
 clrscr();
 Set<char> mySet;
 int function;
 do{
 clrscr();
 cout << "CAC CHUC NANG:" << endl;
 cout << "1: Them mot phan tu vao tap hop" << endl;
 cout << "2: Loai bo mot phan tu khoi tap hop" << endl;
 cout << "3: Xem tat ca cac phan tu cua tap hop" << endl;
 cout << "5: Thoat!" << endl;
 cout << "===== " << endl;
 cout << "Chon chuc nang: " << endl;
 cin >> function;
 switch(function){
 case '1': // Them vao
 char phantu;
 cout << "Ki tu them vao: ";
 cin >> phantu;
 mySet.insert(phantu);
 break;
 case '2': // Loai ra
 char phantu;
 cout << "Loai bo ki tu: " << endl;
 cin >> phantu;
```

```
 mySet.erase(phantu);
 break;
 case '3': // Duyệt
 cout << "Cac phan tu cua tap hop la:" << endl;
 Set<char>::iterator i;
 for(i=mySet.begin(); i<mySet.end(); i++)
 cout << mySet[i] << " ";
 break;
 }while(function != '5');
return;
}
```

## 7.3 LỚP CHUỖI

Lớp chuỗi (String) cũng là một loại lớp chúa, nó chứa một tập các phần tử là một dãy các kí tự có phân biệt thứ tự, các phần tử không nhất thiết phải phân biệt nhau. Muốn sử dụng lớp String, cần thêm vào chỉ thị đầu tệp:

```
#include <string.h>
```

### 7.3.1 Hàm khởi tạo

Lớp String có ba hàm khởi tạo chính:

- Hàm khởi tạo không tham số:

```
String <tên biến>;
```

- Hàm khởi tạo từ một string khác:

```
String <tên biến>(const String &);
```

- Hàm khởi tạo từ một mảng các kí tự:

```
String <tên biến>(char*, <chiều dài mảng>);
```

Ví dụ:

```
String myStr;
```

là khai báo một chuỗi myStr chưa có phần tử nào (rỗng). Hoặc:

```
String myStr("hello!");
```

là khai báo một chuỗi myStr có các phần tử theo thứ tự là {'h', 'e', 'l', 'l', 'o', '!'}.

```
char* myChar = new char[10]; String
```

```
myStr(myChar, 10);
```

là khai báo một chuỗi myStr có 10 phần tử tương ứng với các kí tự trong mảng myChar.

**Lưu ý:**

- Trong trường hợp khởi tạo bằng một chuỗi khác, ta có thể truyền vào một đối số có kiểu cơ bản khác, trình biên dịch sẽ tự động chuyển đổi số đó sang dạng string.

Ví dụ:

```
String myStr(12);
```

## Chương 7: Một số lớp quan trọng

thì chuỗi myStr sẽ chứa hai phần tử kí tự {'1', '2'}. Hoặc:

```
String myStr(15.55);
```

thì chuỗi myStr sẽ chứa năm phần tử kí tự {'1', '5', '!', '5', '5'}.

### 7.3.2 Toán tử

Lớp String có các toán tử cơ bản là:

- Phép gán chuỗi
- Phép cộng chuỗi
- Phép so sánh chuỗi
- Phép vào/ra

#### Phép gán chuỗi “=”

Cú pháp phép gán chuỗi là tương tự cú pháp gán các đối tượng cơ bản:

```
<Tên biến 1> = <Tên biến 2>;
```

Ví dụ:

```
String s1(12), s2; s2 = s1;
```

thì chuỗi s2 cũng chứa hai phần tử như s1 {'1', '2'}.

#### Lưu ý:

- Có thể gán trực tiếp các đối tượng cơ bản cho chuỗi:

```
String myStr = 12; // myStr có hai phần tử {'1', '2'}
```

- Nhưng phép gán lại có độ ưu tiên thấp hơn phép toán học:

```
String myStr = 12+1.5; // tương đương myStr = 13.5
```

#### Phép cộng chuỗi “+” và “+=”

Phép cộng chuỗi sẽ nối chuỗi thứ hai vào sau chuỗi thứ nhất, kết quả cũng là một chuỗi:

```
<Tên biến 1> = <Tên biến 2> + <Tên biến 3>;
```

```
<Tên biến 1> += <Tên biến 2>;
```

Ví dụ:

```
String s1(12), s2(3); s1+= s2;
```

thì s1 sẽ có ba phần tử {'1', '2', '3'}.

#### Phép so sánh chuỗi

Các phép so sánh chuỗi đều là các phép toán hai ngôi, trả về kết quả ở dạng bool (true/false):

- Phép so sánh lớn hơn “>”: chuỗi\_1 > chuỗi\_2;
- Phép so sánh lớn hơn hoặc bằng “>=”: chuỗi\_1 >= chuỗi\_2;
- Phép so sánh nhỏ hơn “<”: chuỗi\_1 < chuỗi\_2;
- Phép so sánh nhỏ hơn hoặc bằng “<=”: chuỗi\_1 <= chuỗi\_2;

- Phép so sánh bằng “==”: chuỗi\_1 == chuỗi\_2;
- Phép so sánh khác (không bằng) “!=”: chuỗi\_1 != chuỗi\_2;

Lưu ý:

- Phép so sánh chuỗi thực hiện so sánh mã ASCII của từng kí tự ở hai chuỗi theo thứ tự tương ứng cho đến khi có sự khác nhau đầu tiên giữa hai kí tự.
- Phép so sánh là phép so sánh dựa trên từ điển, có phân biệt chữ hoa và chữ thường.

Ví dụ:

```
“12” < “a”; // có giá trị đúng “a” <=
“A”; // có giá trị sai
```

### Phép vào/ra

- Phép xuất ra “<<”: cout << biến\_chuỗi;
- Phép nhập vào “>>”: cin >> biến\_chuỗi;

Ví dụ:

```
String s("hello!"); cout << s;
sẽ in ra màn hình dòng chữ “hello!”.
```

### 7.3.3 Phương thức

Lớp chuỗi có một số phương thức cơ bản:

- Lấy chiều dài chuỗi
- Tìm một chuỗi con
- Thêm một chuỗi con
- Xoá một chuỗi con
- Chuyển kiểu kí tự

#### Lấy chiều dài chuỗi

Cú pháp:

```
<biến_chuỗi>.length();
trả về chiều dài của chuỗi (số lượng phần tử kí tự trong chuỗi).
```

Ví dụ:

```
String s("hello!"); cout <<
s.length();
sẽ in ra màn hình độ dài chuỗi s là 6.
```

#### Tìm một chuỗi con

Cú pháp:

```
<biến_chuỗi>.find(<Chuỗi con>, <Vị trí bắt đầu>, <Kiểu so khớp>);
Trong đó:
```

- Tham số thứ nhất là chuỗi con cần tìm.
- Tham số thứ hai là vị trí để bắt đầu tìm, mặc định là bắt đầu tìm từ phần tử có chỉ số 0.
- Tham số thứ ba chỉ ra cách so khớp có phân biệt chữ hoa với chữ thường: SM\_IGNORE là không phân biệt, SM\_SENSITIVE là có phân biệt.
- Phương thức này trả về kết quả dạng bool, tương ứng là có tìm thấy hay không.

Ví dụ:

```
s.find("12", 0, SM_IGNORE);
```

sẽ tìm trong chuỗi s xem có sự xuất hiện của chuỗi “12” hay không, vị trí bắt đầu tìm là 0, với cách tìm không phân biệt chữ hoa chữ thường.

### **Thêm một chuỗi con**

Cú pháp:

```
<biến_chuỗi>.insert(<vị trí chèn>, <chuỗi con>);
```

Trong đó:

- Tham số thứ nhất là vị trí chỉ số mà tại đó, chuỗi con sẽ được chèn vào
- Tham số thứ hai là chuỗi con cần chèn, chuỗi con này cũng có thể là một kí tự.

Ví dụ:

```
s.insert(0, "12");
```

sẽ chèn vào đầu chuỗi s một chuỗi con có hai phần tử “12”.

### **Xoá một chuỗi con**

Cú pháp:

```
<biến_chuỗi>.delete(<vị trí bắt đầu>, <độ dài chuỗi xoá>);
```

Trong đó:

- Tham số thứ nhất là vị trí bắt đầu xoá chuỗi con
- Tham số thứ hai là độ dài chuỗi con bị xoá, giá trị mặc định là 1.

Ví dụ:

```
s.delete(0, 2);
```

sẽ xoá hai kí tự đầu của chuỗi s.

### **Chuyển kiểu kí tự**

- Đổi chuỗi thành các kí tự hoa:              `<biến_chuỗi>.toUpperCase();`
- Đổi chuỗi thành các kí tự thường:          `<biến_chuỗi>.toLowerCase();`

Ví dụ:

```
s.toUpperCase(); // chuyển chuỗi s thành kí tự hoa s.toLowerCase();
// chuyển chuỗi s thành kí tự thường
```

### 7.3.4 Áp dụng

Chương trình 7.2 minh họa một số thao tác cơ bản trên lớp chuỗi, có sử dụng thư viện lớp chuỗi chuẩn của C++: cộng thêm một chuỗi, chèn thêm một chuỗi con, xoá một chuỗi con, tìm một chuỗi con...

#### Chương trình 7.2

```
#include<stdio.h>
#include<conio.h>
#include<string.h> void
main(){
 clrscr(); String
 myStr; int function;
 do{
 clrscr();
 cout << "CAC CHUC NANG:" << endl;
 cout << "1: Cong them mot chuoi" << endl; cout << "2:
 Chen them mot chuoi" << endl; cout << "3: Xoa di
 mot chuoi" << endl; cout << "4: Tim mot chuoi con"
 << endl; cout << "5: Chuyen thanh chu hoa" << endl;
 cout << "6: Chuyen thanh chu thuong" << endl; cout <<
 "7: Xem noi dung chuoi" << endl; cout << "8: Xem
 chieu dai chuoi" << endl; cout << "9: Thoat!" << endl;
 cout << "===== " << endl;
 cout << "Chon chuc nang: " << endl; cin >>
 function;
 switch(function){
 case '1':// Thêm vào cuối String subStr;
 cout << "Chuoi them vao: "; cin >>
 subStr;
 myStr += subStr; break;
 case '2':// Chèn vào chuỗi String subStr;
 int position;
 cout << "Chuoi them vao: "; cin >>
 subStr;
 cout << "Vi tri chen: "; cin >>
 position;
```

```
 myStr.insert(position, subStr); break;
 case '3':// Xoá đi một chuỗi con int position, count;
 cout << "Vi tri bat dau xoa:"; cin >>
 position;
 cout << "Do dai xoa:"; cin >>
 count;
 myStr.delete(position, count); break;
 case '4':// Tìm chuỗi con String subStr;
 int position;
 cout << "Chuoi con can tim:"; cin >>
 subStr;
 cout << "Vi tri bat dau tim:"; cin >>
 position; if(myStr.find(position, subStr))
 cout << "Co xuat hien!" << endl;
 else
 cout << "Khong xuat hien!" << endl;
 break;
 case '5':// Chuyển thành chữ hoa myStr.toUpperCase();
 cout << myStr << endl; break;
 case '6':// Chuyển thành chữ thường myStr.toLowerCase();
 cout << myStr << endl; break;
 case '7': // Duyệt
 cout << "Noi dung chuoi:" << endl; cout <<
 myStr << endl;
 break;
 case '8':// Duyệt cout << "Chieu dai
 chuoi:"
 << myStr.length() << endl; break;
}while(function != '9'); return;
}
```

## 7.4 LỚP NGĂN XẾP VÀ HÀNG ĐỢI

### 7.4.1 Lớp ngăn xếp

Lớp ngăn xếp (stack) cũng là một loại lớp vật chứa, nó chứa các phần tử cùng kiểu, không bắt buộc phải phân biệt nhau nhưng có phân biệt về thứ tự: các thao tác thêm phần tử và lấy phần tử ra đều được thực hiện ở một đầu ngăn xếp. Phần tử nào được thêm vào trước thì sẽ bị lấy ra sau.

Muốn dùng lớp Stack phải dùng chỉ thị đầu tệp:

```
#include<stack.h>
```

#### Hàm khởi tạo

Lớp Stack có hai cách khởi tạo:

- Khởi tạo không tham số:  
`Stack<T> biến_ngăn_xếp;`
- Khởi tạo bằng một ngăn xếp khác, có cùng kiểu phần tử:  
`Stack<T> biến_ngăn_xếp(Stack<T>);`

Trong đó:

- T: là kiểu của các phần tử chứa trong ngăn xếp. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
Stack<int> myStack;
```

là khai báo một biến myStack, chứa các phần tử có kiểu cơ bản int.

#### Toán tử

Lớp Stack chỉ dùng đến các toán tử gán “=” và toán tử so sánh bằng “==”:

- Phép gán “=”:  
`<ngăn xếp 1> = <ngăn xếp 2>;`  
Dùng để gán hai đối tượng ngăn xếp.
- Phép so sánh bằng “==”:  
`<ngăn xếp 1> == <ngăn xếp 2>;`  
Dùng để kiểm tra xem hai đối tượng ngăn xếp có bằng nhau hay không. Kết quả trả về có kiểu bool (true/false).

#### Phương thức

- Thêm một phần tử:  
`<biến ngăn xếp>.push(T);`  
sẽ thêm một phần tử có kiểu T vào đỉnh ngăn xếp.
- Loại một phần tử:  
`<biến ngăn xếp>.pop();`  
sẽ trả về phần tử đang nằm ở đỉnh ngăn xếp.
- Kiểm tra tính rỗng:

<biến ngăn xếp>.empty();  
sẽ trả về kết quả kiểu bool, tương ứng với trạng thái của ngăn xếp có rỗng hay không.

- Kích thước ngăn xếp:

<biến ngăn xếp>.size();

sẽ trả về số lượng các phần tử hiện đang có mặt trong ngăn xếp.

### Áp dụng

Trong phần này, ta sẽ viết lại chương trình 3.4c đã được minh họa trong phần viết về cấu trúc ngăn xếp (chương 3). Nhưng thay vì phải định nghĩa cấu trúc ngăn xếp, ta dùng lớp ngăn xếp của thư viện C++: đảo ngược một xâu kí tự được nhập vào từ bàn phím.

#### Chương trình 7.3

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stack.h>

void main() {
 clrscr();
 Stack<char> myStack;
 char strIn[250];
 cout << "Nhập chuỗi: ";
 cin >> strIn; // Nhập chuỗi kí tự từ bàn phím
 for(int i=0; i<strlen(strIn); i++) // Đặt vào ngăn xếp
 myStack.push(strIn[i]);
 while(!myStack.empty()) // Lấy ra từ ngăn xếp
 cout << myStack.pop();
 return;
}
```

#### 7.4.2 Lớp hàng đợi

Lớp hàng đợi (queue) cũng là một loại lớp vật chứa, nó chứa các phần tử cùng kiểu, không bắt buộc phải phân biệt nhau nhưng có phân biệt về thứ tự: các thao tác thêm phần tử được thực hiện ở một đầu, các thao tác lấy phần tử ra được thực hiện ở một đầu còn lại của hàng đợi. Phần tử nào được thêm vào trước thì sẽ bị lấy ra trước.

Muốn dùng lớp Queue phải dùng chỉ thị đầu tệp:

```
#include<queue.h>
```

#### Hàm khởi tạo

Lớp Queue có hai cách khởi tạo:

- Khởi tạo không tham số:  
`Queue<T> biến_hàng_đợi;`
- Khởi tạo bằng một hàng đợi khác, có cùng kiểu phần tử:  
`Queue<T> biến_hàng_đợi(Queue<T>);`

Trong đó:

- `T`: là kiểu của các phần tử chứa trong hàng đợi. `T` có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
Queue<int> myQueue;
là khai báo một biến myQueue, chứa các phần tử có kiểu cơ bản int.
```

### Toán tử

Lớp Queue chỉ dùng đến các toán tử gán “=” và toán tử so sánh bằng “==”:

- Phép gán “=”:  
`<hàng đợi 1> = <hàng đợi 2>;`

Dùng để gán hai đối tượng hàng đợi.

- Phép so sánh bằng “==”:  
`<hàng đợi 1> == <hàng đợi 2>;`

Dùng để kiểm tra xem hai đối tượng hàng đợi có bằng nhau hay không. Kết quả trả về có kiểu `bool` (`true/false`).

### Phương thức

- Thêm một phần tử:

```
<biến hàng đợi>.push(T);
```

sẽ thêm một phần tử có kiểu `T` vào cuối hàng đợi.

- Loại một phần tử:

```
<biến hàng đợi>.pop();
```

sẽ trả về phần tử đang nằm ở đỉnh đầu hàng đợi.

- Kiểm tra tính rỗng:

```
<biến hàng đợi>.empty();
```

sẽ trả về kết quả kiểu `bool`, tương ứng với trạng thái của hàng đợi có rỗng hay không.

- Kích thước hàng đợi:

```
<biến hàng đợi>.size();
```

sẽ trả về số lượng các phần tử hiện đang có mặt trong hàng đợi.

### Áp dụng

Trong phần này, ta sẽ cài đặt lại chương trình trong phần cấu trúc hàng đợi (chương 3). Nhưng thay vì phải tự định nghĩa cấu trúc hàng đợi, ta dùng lớp `Queue` của thư viện C++ để mô phỏng chương trình quản lý tiến trình của hệ điều hành.

#### Chương trình 7.4

```
#include<stdio.h>
#include<conio.h>
#include<queue.h>

void main(){
 clrscr(); Queue<int>
 myQueue; int function;
 do{
 clrscr();
 cout << "CAC CHUC NANG:" << endl;
 cout << "1: Them mot tien trinh vao hang doi" << endl;
 cout << "2: Dua mot tien trinh trinh vao thuc hien" << endl; cout << "3: Xem
tat ca cac tien trinh trong hang doi" << endl; cout << "5: Thoat!" << endl;
 cout << "===== " << endl;
 cout << "Chon chuc nang: " << endl; cin >>
 function;
 switch(function){
 case '1':// Thêm vào hàng đợi int maso;
 cout << "Ma so tien trinh vao hang doi: "; cin >> maso;
 myQueue.push(maso); break;
 case '2': // Lấy ra khỏi hàng đợi
 cout << "Tien trinh duoc thuc hien: " << myQueue.pop() << endl;
 break;
 case '3':// Duyệt hàng đợi Queue<int>::iterator i;
 for(i=myQueue.begin(); i<myQueue.end(); i++)
 cout << myQueue[i] << " "; break;
 }while(function != '5'); return;
}
```

## 7.5 LỚP DANH SÁCH LIÊN KẾT

Lớp danh sách liên kết (List) cũng là một kiểu lớp vật chúa, nó chứa các phần tử cùng kiểu, có tính đến thứ tự. Muốn sử dụng lớp List của thư viện C++, phải khai báo chỉ thị đầu tệp:

```
#include<list.h> // Dành riêng cho lớp List
```

hoặc:

```
#include<stl.h> // Dùng chung cho các lớp vật chúa
```

### 7.5.1 Hàm khởi tạo

Lớp List có ba kiểu khởi tạo:

- Khởi tạo không tham số:

```
List<T> biến_danh_sách;
```

- Khởi tạo từ một danh sách cùng kiểu:

```
List<T> biến_danh_sách(List<T>);
```

- Khởi tạo từ một mảng các phần tử:

```
List<T> biến_danh_sách(T* <Mảng phần tử>, int <chiều dài mảng>);
```

Trong đó:

- T: là kiểu của các phần tử chứa trong danh sách liên kết. T có thể là các kiểu cơ bản, cũng có thể là các kiểu phức tạp do người dùng tự định nghĩa.

Ví dụ:

```
List<int> myList;
```

sẽ khai báo một danh sách liên kết myList, các phần tử của nó có kiểu cơ bản int.

### 7.5.2 Toán tử

#### Toán tử gán “=”

Cú pháp:

```
<danh sách 1> = <danh sách 2>;
```

sẽ copy toàn bộ các phần tử của <danh sách 2> vào <danh sách 1>.

#### Toán tử so sánh bằng “==”

Cú pháp:

```
<danh sách 1> = <danh sách 2>;
```

sẽ trả về một giá trị kiểu bool, tương ứng với việc hai danh sách này có bằng nhau hay không. Việc so sánh được tiến hành trên từng phần tử ở vị trí tương ứng nhau.

#### Lưu ý:

- Ngoài ra còn có các phép toán so sánh khác cũng có thể thực hiện trên danh sách: <, >, <=, >=, !=.

### 7.5.3 Phương thức

#### **Thêm một phần tử vào danh sách**

Cú pháp:

```
<biến danh sách>.insert(<vị trí chèn>, <phần tử>);
<biến danh sách>.push_front(<phần tử>);
<biến danh sách>.push_back(<phần tử>);
```

Trong đó:

- Phương thức thứ nhất chèn một phần tử vào một vị trí bất kỳ của danh sách, vị trí chèn được chỉ ra bởi tham số thứ nhất.
- Phương thức thứ hai chèn một phần tử vào đầu danh sách
- Phương thức thứ ba chèn một phần tử vào cuối danh sách.

Ví dụ:

```
List<int> myList; myList.push_front(7);
```

sẽ chèn vào đầu danh sách myList một phần tử có giá trị là 7.

#### **Xoá đi một phần tử**

Cú pháp:

```
<biến danh sách>.erase(<vị trí xoá>);
<biến danh sách>.pop_front();
<biến danh sách>.pop_back();
```

Trong đó:

- Phương thức thứ nhất xóa một phần tử ở một vị trí bất kỳ của danh sách, vị trí xoá được chỉ ra bởi tham số thứ nhất.
- Phương thức thứ hai xoá một phần tử ở đầu danh sách
- Phương thức thứ ba xoá một phần tử ở cuối danh sách.

Ví dụ:

```
List<int> myList;
cout << myList.pop_front();
```

sẽ in ra màn hình phần tử đầu của danh sách myList.

#### **Kiểm tra tính rỗng của danh sách**

Cú pháp:

```
<biến danh sách>.empty();
```

trả về giá trị bool, tương ứng với trạng thái hiện tại của biến danh sách là rỗng hay không.

#### **Xem kích thước danh sách**

Cú pháp:

```
<biến danh sách>.size();
```

Ví dụ:

```
cout << myList.size();
sẽ in ra màn hình kích cỡ (số lượng các phần tử) của danh sách.
```

### Xem nội dung một phần tử

Cú pháp:

```
<biên danh sách>.get();
<biên danh sách>.next();
```

Trong đó:

- Phương thức thứ nhất trả về phần tử ở vị trí hiện tại của con chạy
- Phương thức thứ hai trả về phần tử tiếp theo, và con chạy cũng di chuyển sang phần tử đó.

Ví dụ:

```
List<int> myList; cout <<
myList.get();
sẽ in ra màn hình nội dung phần tử thứ nhất của danh sách myList.
```

### 7.5.4 Áp dụng

Trong phần này, ta cài đặt lại chương trình 3.6c đã được trình bày trong phần cấu trúc danh sách liên kết (chương 3). Nhưng thay vì tự tạo danh sách liên kết, ta dùng lớp List trong thư viện của C++ để quản lý nhân viên văn phòng.

#### **Chương trình 7.5**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<list.h>

typedef struct{
 char name[25]; // Tên nhân viên
 int age; // Tuổi nhân viên
 float salary; // Lương nhân viên
} Employee;

void main(){
 clrscr();
 List<Employee> myList;
 int function;
 do{
 clrscr();
 cout << "CAC CHUC NANG:" << endl;
 cout << "1: Them mot nhan vien" << endl;
 cout << "2: Xoa mot nhan vien" << endl;
 cout << "3: Xem tat ca cac nhan vien trong phong" << endl;
```

```
cout << "5: Thoat!" << endl;
cout << "===== " << endl;
cout << "Chon chuc nang: " << endl; cin >>
function;
switch(function){
 case '1':// Thêm vào ds int position;
 Employee employee;
 cout << "Vi tri can chen: "; cin >>
 position;
 cout << "Ten nhan vien: "; cin >>
 employee.name;
 cout << "Tuoi nhan vien: "; cin >>
 employee.age;
 cout << "Luong nhan vien: "; cin >>
 employee.salary;
 myList.insert(position, employee); break;
 case '2':// Lấy ra khỏi ds int position;
 cout << "Vi tri can xoa: "; cin >>
 position;
 Employee result = myList.erase(position); if(result != NULL){
 cout << "Nhan vien bi loai: " endl; cout << "Ten:
" << result.name << endl; cout << "Tuoi: " <<
result.age << endl;
 cout << "Luong: " << result.salary << endl;
 }
 break;
 case '3': // Duyệt ds
 cout << "Cac nhan vien cua phong:" << endl; Employee result =
myList.front();
 do{
 cout << result.name << " "
 << result.age << " "
 << result.salary << endl; result =
myList.next();
 }while(result != NULL); break;
}while(function != '5'); return;
}
```

## TỔNG KẾT CHƯƠNG 7

Nội dung chương 7 đã trình bày một số lớp cơ bản của lớp vật chứa (Container) trong thư viện của C++:

- Lớp tập hợp (Set)
- Lớp chuỗi (String)
- Lớp ngăn xếp (Stack)
- Lớp hàng đợi (Queue)
- Lớp danh sách liên kết (List)

Hầu hết các lớp vật chứa này đều có thể chứa các phần tử với kiểu bất kỳ: có thể là kiểu cơ bản, cũng có thể là kiểu phức tạp do người dùng tự định nghĩa. Ngoại trừ lớp chuỗi, chỉ chứa các phần tử có kiểu char.

Các lớp vật chứa này có một số phương thức tương tự nhau, nhưng cách thực hiện lại khác nhau:

- Thêm vào một phần tử
- Lấy ra một phần tử
- Kiểm tra tính rỗng
- Kiểm tra số lượng phần tử.

Người dùng có thể áp dụng các lớp có sẵn trong thư viện này của C++ để giải quyết các bài toán khác nhau mà không phải tự định nghĩa lại các lớp.

## CÂU HỎI VÀ BÀI TẬP CHƯƠNG 7

1. Dùng thư viện lớp stack để kiểm tra một số tự nhiên nhập vào từ bàn phím có phải là một số parlindom hay không: Một số tự nhiên là parlindom nếu đổi ngược thứ tự các chữ số, ta vẫn thu được chính số đó, ví dụ, 87278 là một số như vậy.
2. Viết lại chương trình 7.1 dùng lớp tập hợp, nhưng dùng kiểu của các phần tử là lớp Car đã được định nghĩa trong chương 5.
3. Dùng thư viện lớp String để viết chương trình chia nhỏ một chuỗi ban đầu thành một số chuỗi con, ranh giới phân chia là một chuỗi con được nhập vào từ bàn phím. Ví dụ “abc acb”, mà chia nhỏ theo chuỗi con “c” sẽ thu được các chuỗi: “ab”, “ca”, “cb”.
4. Dùng thư viện lớp String để viết chương trình thay thế các chuỗi con của một chuỗi ban đầu bằng một chuỗi con khác. Các chuỗi con được nhập từ bàn phím.
5. Viết lại chương trình 7.5 , vẫn dùng thư viện lớp List, nhưng thay thế kiểu phần tử bằng kiểu lớp Car đã được định nghĩa trong chương 5.

## TÀI LIỆU THAM KHẢO

### *Tài liệu tiếng Anh*

- [1] James P. Cohoon and Jack W. Davidson, *C++ Program Design – An Introduction to Programming and Object-Oriented Design*, 2<sup>nd</sup> edition, WCB McGraw-Hill, 1999.
- [2] Nell Dale, Chip Weems and Mark Headington, *Programming and Problem Solving with C++*, John & Barlett Publisher, 1996.
- [3] Michael T. Goodrich, Roberto Tamassia and David Mount, *Data Structures and Algorithms in C++*, John Wiley & Sons Inc, 2004.
- [4] Scott R.Ladd, *C++ Components and Algorithms*. 2<sup>nd</sup> edition. . M&T Books, 1994.
- [5] Scott R.Ladd, *C++ Templates and Tools*, M&T Books, 1994
- [6] Robert Lafore, *Object – Oriented Programming in C++*, Fourth edition, SAMS, 2001.

### *Tài liệu tiếng Việt*

- [1] Lê Đ. Hưng, Tạ T. Anh, Nguyễn H. Đức và Nguyễn T. Thuỷ, *Lập trình hướng đối tượng với C++*, NXB Khoa học và Kỹ thuật, 2005.
- [2] Nguyễn T. Thuỷ, Tạ T. Anh, Nguyễn Q. Huy và Nguyễn H. Đức, *Bài tập lập trình hướng đối tượng với C++*, NXB Khoa học và Kỹ thuật, 2004.

### *Các địa chỉ web*

1. <http://www.angelfire.com/country/aldev0/cpphowto/>
2. <http://www.gnacademy.org/text/cc/Tutorial/tutorial.html>
3. <http://sophia.dtp.fmph.uniba.sk/cpptut/tutorial.us.html>
4. <http://www.brpreiss.com/books/opus4/html/book.html>
5. <http://www.fredosaurus.com/notes-cpp/index.html>