CHƯƠNG 7

LỚP VÀ ĐỐI TƯỢNG

Lập trình có cấu trúc và lập trình hướng đối tượng
Lớp và đối tượng
Đối của phương thức - Con trỏ this
Hàm tạo (contructor)
Hàm hủy (destructor)
Các hàm trực tuyến (inline)

I. LẬP TRÌNH CÓ CẤU TRÚC VÀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

1. Phương pháp lập trình cấu trúc

- Lập trình cấu trúc là tổ chức chương trình thành các chương trình con. Trong một số ngôn ngữ như PASCAL có 2 kiểu chương trình con là thủ tục và hàm, còn trong C++ chỉ có một loại chương trình con là hàm.
- Hàm là một đơn vị chương trình độc lập dùng để thực hiện một phần việc nào đó như: Nhập số liệu, in kết quả hay thực hiện một số công việc tính toán. Hàm cần có đối và các biến, mảng cục bộ dùng riêng cho hàm.
- Việc trao đổi dữ liệu giữa các hàm thực hiện thông qua các đối và các biến toàn cục.
- Một chương trình cấu trúc gồm các cấu trúc dữ liệu (như biến, mảng, bản ghi) và các hàm, thủ tục.
- Nhiệm vụ chính của việc tổ chức thiết kế chương trình cấu trúc là tổ chức chương trình thành các hàm, thủ tục.

Ví dụ, ta xét yêu cầu sau: Viết chương trình nhập toạ độ (x,y) của một dãy điểm, sau đó tìm một cặp điểm cách xa nhau nhất.

Trên tư tưởng của lập trình cấu trúc có thể tổ chức chương trình như sau:

- Sử dụng 2 mảng thực toàn bộ x và y để chứa toạ độ dãy điểm.
- Xây dựng 2 hàm:

Hàm nhapsl dùng để nhập toạ độ n điểm, hàm này có một đối là biến nguyên n và được khai báo như sau:

```
void nhapsl(int n);
```

Hàm do_dai dùng để tính độ dài đoạn thẳng đi qua 2 điểm có chỉ số là i và j,

nó được khai báo như sau:

```
float do_dai(int i, int j);
```

Chương trình C của ví dụ trên được viết như sau:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
float x[100],y[100];
float do_dai(int i, int j)
{
    return sqrt(pow(x[i]-x[j],2)+pow(y[i]-y[j],2));
}
void nhapsl(int n)
{
    int i;
    for (i=1; i<=n; ++i)
    {
        printf("\n Nhap toa do x, y cua diem thu %d : ",i);
    scanf("%f%f",&x[i],&y[i]);
    }
}
void main()
{
    int n, i, j, imax,jmax;
    float d, dmax;
    printf("\n So diem N= ");
    scanf("%d", &n);
    nhapsl(n);
    dmax=do_dai(1,2);imax=1; jmax=2;
    for(i=1; i<=n-1; ++i)
    for (j=i+1; j<=n; ++j)
    {
        d=do_dai(i,j);
        if (d>dmax)
```

```
            {
                dmax=d;
                imax=i; jmax=j;
            }
        }
        printf("\nDoan thang lon nhat co do dai bang: %0.2f",dmax);
        printf("\n Di qua 2 diem co chi so la %d va %d",imax,jmax);
        getch();
}
```

2. Phương pháp lập trình hướng đối tượng

Là lập trình có cấu trúc + trừu tượng hóa dữ liệu. Có nghĩa chương trình tổ chức dưới dạng cấu trúc. Tuy nhiên việc thiết kế chương trình sẽ xoay quanh dữ liệu, lấy dữ liệu làm trung tâm. Nghĩa là trả lời câu hỏi: Chương trình làm việc với những đối tượng dữ liệu nào, trên các đối tượng dữ liệu này cần thao tác, thực hiện những gì. Từ đó gắn với mỗi đối tượng dữ liệu một số thao tác thực hiên cố định riêng của đối tượng dữ liệu đó, điều này sẽ qui định chặt chẽ hơn những thao tác nào được thực hiện trên đối tượng dữ liệu nào. Khác với lập trình cấu trúc thuần túy, trong đó dữ liệu được khai báo riêng rẽ, tách rời với thao tác xử lý, do đó việc xử lý dữ liệu thường không thống nhất khi chương trình được xây dựng từ nhiều lập trình viên khác nhau.

Từ đó lập trình hướng đối tượng được xây dựng dựa trên đặc trưng chính là khái niệm đóng gói. Đóng gói là khái niệm trung tâm của phương pháp lập trình hướng đối tượng, trong đó dữ liệu và các thao tác xử lý nó sẽ được qui định trước và "đóng" thành một "gói" thống nhất, riêng biệt với các dữ liệu khác tạo thành kiểu dữ liệu với tên gọi là các lớp. Như vậy một lớp không chỉ chứa dữ liệu bình thường như các kiểu dữ liệu khác mà còn chứa các thao tác để xử lý dữ liệu này. Các thao tác được khai báo trong gói dữ liệu nào chỉ xử lý dữ liệu trong gói đó và ngược lại dữ liệu trong một gói chỉ bị tác động, xử lý bởi thao tác đã khai báo trong gói đó. Điều này tạo tính tập trung cao khi lập trình, mọi đối tượng trong một lớp sẽ chứa cùng loại dữ liệu được chỉ định và cùng được xử lý bởi các thao tác như nhau. Mọi lập trình viên khi làm việc với dữ liệu trong một gói đều sử dụng các thao tác như nhau để xử lý dữ liệu trong gói đó. C++ cung cấp cách thức để tạo một cấu trúc dữ liệu mới thể hiện các gói nói trên, cấu trúc dữ liệu này được gọi là **lớp**.

Để minh hoạ các khái niệm vừa nêu về kiêu dữ liệu lớp ta trở lại xét bài toán tìm độ dài lớn nhất đi qua 2 điểm. Trong bài toán này ta gặp một thực thể là dãy điểm. Các thành phần dữ liệu của lớp dãy điểm gồm:

- Biến nguyên n là số điểm của dãy
- Con trỏ x kiểu thực trỏ đến vùng nhớ chứa dãy hoành độ

- Con trỏ y kiểu thực trỏ đến vùng nhớ chứa dãy tung độ

Các phương thức cần đưa vào theo yêu cầu bài toán gồm:

- Nhập toạ độ một điểm
- Tính độ dài đoạn thẳng đi qua 2 điểm

Dưới đây là chương trình viết theo thiết kế hướng đối tượng.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <alloc.h>
class daydiem
{
        int n;
        float *x,*y;
public:
        float do_dai(int i, int j)
        {
                return sqrt(pow(x[i]-x[j],2)+pow(y[i]-y[j],2));
        }
        void nhapsl(void);
};
void daydiem::nhapsl(void)
{
        int i;
        printf("\n So diem N= ");
        scanf("%d",&n);
        x = (float*)malloc((n+1)*sizeof(float));
        y = (float*)malloc((n+1)*sizeof(float));
        for (i=1; i<=n; ++i)
        {
                printf("\n Nhap toa do x, y cua diem thu %d : ",i);
                scanf("%f%f",&x[i],&y[i]);
        }
}
```

```
void main()
{
	clrscr();
	daydiem p;
	p.nhapsl();
	int n,i,j,imax,jmax;
	float d,dmax;
	n = p.n;
	dmax=p.do_dai(1,2);imax=1; jmax=2;
	for (i=1;i<=n-1;++i)
	for (j=i+1;j<=n;++j)
	{
		d=p.do_dai(i,j);
		if (d>dmax)
		{
			dmax=d;
			imax=i; jmax=j;
		}
	}
	printf("\n Doan thang lon nhat co do dai bang: %0.2f",dmax);
	printf("\n Di qua 2 diem co chi so la %d va %d" , imax,jmax);
	getch();
}
```

II. LỚP VÀ ĐỐI TƯỢNG

Trong lập trình hướng đối tượng, lớp (class) là một khái niệm rất quan trọng, nó cho phép giải quyết các vấn đề phức tạp của việc lập trình. Một lớp đơn (được định nghĩa như struct, union, hoặc class) bao gồm các hàm và dữ liệu có liên quan. Các hàm này là các hàm thành phần (member functon) hay còn là phương thức (method), thể hiện tác động của lớp có thể được thực hiện trên dữ liệu của chính lớp đó (data member).

Cũng giống như cấu trúc, lớp có thể xem như một kiểu dữ liệu. Vì vậy lớp còn gọi là kiểu đối tượng và lớp được dùng để khai báo các biến, mảng đối tượng (như thể dùng kiểu int để khai báo các biến mảng nguyên).

Như vậy từ một lớp có thể tạo ra (bằng cách khai báo) nhiều đối tượng (biến,

mảng) khác nhau. Mỗi đối tượng có vùng nhớ riêng của mình và vì vậy ta cũng có thể quan niệm lớp chính là tập hợp các đối tượng cùng kiểu.

1. Khai báo lớp

Để khai báo một lớp, ta sử dụng từ khoá class như sau:

```
class tên_lớp
{
    // Khai báo các thành phần dữ liệu (thuộc tính)
    // Khai báo các phương thức (hàm)
};
```

Chú ý: Việc khai báo một lớp không chiếm giữ bộ nhớ, chỉcác đối tượng của lớp mới thực sự chiếm giữ bộ nhớ.

Thuộc tính của lớp có thể là các biến, mảng, con trỏ có kiểu chuẩn (int, float, char, char*, long,...) hoặc kiểu ngoài chuẩn đã định nghĩa trước (cấu trúc, hợp, lớp,...). Thuộc tính của lớp không thể có kiểu của chính lớp đó, nhưng có thể là con trỏ của lớp này, ví dụ:

```
class A
{
    A x;          //Không cho phép, vì x có kiểu lớp A
    A* p ;        //Cho phép , vì p là con trỏ kiểu lớp A
} ;
```

2. Khai báo các thành phần của lớp (thuộc tính và phương thức)

a. Các từ khóa private và public

Khi khai báo các thành phần dữ liệu và phương thức có thể dùng các từ khoá private và public để quy định phạm vi sử dụng của các thành phần này. Trong đó từ khóa private qui định các thành phần (được khai báo với từ khóa này) chỉ được sử dụng bên trong lớp (trong thân các phương thức của lớp). Các hàm bên ngoài lớp (không phải là phương thức của lớp) không được phép sử dụng các thành phần này. Đặc trưng này thể hiện tính che giấu thông tin trong nội bộ của lớp, để đến được các thông tin này cần phải thông qua chính các thành phần hàm của lớp đó. Do vậy thông tin có tính toàn vẹn cao và việc xử lý thông tin (dữ liệu) này mang tính thống nhất hơn và hầu như dữ liệu trong các lớp đều được khai báo với từ khóa này. Ngược lại với private, các thành phần được khai báo với từ khóa public được phép sử dụng ở cả bên trong và bên ngoài lớp, điều này cho phép trong chương trình có thể sử dụng các hàm này để truy nhập đến dữ liệu của lớp. Hiển nhiên nếu các thành phần dữ liệu đã khai báo là privte thì các thành phần hàm phải có ít nhất một vài hàm được khai báo dạng public để chương trình có thể truy cập được, nếu không

toàn bộ lớp sẽ bị đóng kín và điều này không giúp gì cho chương trình. Do vậy cách khai báo lớp tương đối phổ biến là các thành phần dữ liệu được ở dạng private và thành phần hàm dưới dạng public. Nếu không quy định cụ thể (không dùng các từ khoá private và public) thì C++ hiểu đó là private.

b. Các thành phần dữ liệu (thuộc tính)

Được khai báo như khai báo các thành phần trong kiểu cấu trúc hay hợp. Bình thường các thành phần này được khai báo là private để bảo đảm tính giấu kín, bảo vệ an toàn dữ liệu của lớp không cho phép các hàm bên ngoài xâm nhập vào các dữ liệu này.

c. Các phương thức (hàm thành viên)

Thường khai báo là public để chúng có thể được gọi tới (sử dụng) từ các hàm khác trong chương trình.

Các phương thức có thể được khai báo và định nghĩa bên trong lớp hoặc chỉ khai báo bên trong còn định nghĩa cụ thể của phương thức có thể được viết bên ngoài. Thông thường, các phương thức ngắn được viết (định nghĩa) bên trong lớp, còn các phương thức dài thì viết bên ngoài lớp.

Một phương thức bất kỳ của một lớp, có thể sử dụng bất kỳ thành phần (thuộc tính và phương thức) nào của lớp đó và bất kỳ hàm nào khác trong chương trình (vì phạm vi sử dụng của hàm là toàn chương trình).

Giá trị trả về của phương thức có thể có kiểu bất kỳ (chuẩn và ngoài chuẩn)

Ví dụ sau sẽ minh hoạ các điều nói trên. Chúng ta sẽ định nghĩa lớp để mô tả và xử lý các điểm trên màn hình đồ hoạ. Lớp được đặt tên là DIEM.

- Các thuộc tính của lớp gồm:

```
int x ;          // Hoành độ (cột)
int y ;          // Tung độ (hàng)
int m ;          // Mầu
```

- Các phương thức:

  Nhập dữ liệu một điểm

  Hiển thị một điểm

  Ẩn một điểm

Lớp điểm được xây dựng như sau:

```
#include <iostream.h>
#include <graphics.h>
class DIEM
{
```

```
private:
        int x, y, m ;
public:
        void nhapsl() ;
        void hien() ;
        void an() { putpixel(x, y, getbkcolor());}
};

void DIEM::nhapsl()
{
        cout <<"\n Nhap hoanh do (cot) va tung do (hang) cua diem: ";
        cin >> x >> y ;
        cout << "\n Nhap ma mau cua diem: ";
        cin >> m ;
}

void DIEM::hien()
{
        int mau_ht ;
        mau_ht = getcolor();
        putpixel(x, y, m);
        setcolor(mau_ht);
}
```

Qua ví dụ trên có thể rút ra một số chú ý sau:

+ Trong cả 3 phương thức (dù viết trong hay viết ngoài định nghĩa lớp) đều được phép truy nhập đến các thuộc tính x, y và m của lớp.
+ Các phương thức viết bên trong định nghĩa lớp (như phương thức an()) được viết như một hàm thông thường.
+ Khi xây dựng các phương thức bên ngoài lớp, cần dùng thêm tên lớp và toán tử phạm vi :: đặt ngay trước tên phương phức để quy định rõ đây là phương thức của lớp nào.

3. Biến, mảng và con trỏ đối tượng

Như đã nói ở trên, một lớp (sau khi định nghĩa) có thể xem như một kiểu đối tượng và có thể dùng để khai báo các biến, mảng đối tượng. Cách khai báo biến, mảng đối tượng cũng giống như khai báo biến, mảng các kiểu khác (như int, float,

cấu trúc, hợp,...), theo mẫu sau:

Tên_lớp danh sách đối ;

Tên_lớp danh sách mảng ;

Ví dụ sử dụng DIEM ở trên, ta có thể khai báo các biến, mảng DIEM như sau:

```
DIEM d1, d2, d3 ;      // Khai báo 3 biến đối tượng d1, d2, d3
DIEM d[20] ;           // Khai báo mảng đối tượng d gồm 20 phần tử
```

Mỗi đối tượng sau khi khai báo sẽ được cấp phát một vùng nhớ riêng để chứa các thuộc tính của nó. Chú ý rằng sẽ không có vùng nhớ riêng để chứa các phương thức cho mỗi đối tượng, các phương thức sẽ được sử dụng chung cho tất cả các đối tượng cùng lớp. Như vậy về bộ nhớ được cấp phát thì đối tượng giống cấu trúc.

Trong trường hợp này:

```
sizeof(d1) = sizeof(d2) = sizeof(d3) = 3*sizeof(int) = 6
sizeof(d) = 20*6 = 120
```

a. Thuộc tính của đối tượng

Trong ví dụ trên, mỗi đối tượng d1, d2, d3 và mỗi phần tử d[i] đều có 3 thuộc tính là x, y, m. Chú ý là mỗi thuộc tính đều thuộc về một đối tượng, vì vậy không thể viết tên thuộc tính một cách riêng rẽ mà bao giờ cũng phải có tên đối tượng đi kèm, giống như cách viết trong cấu trúc của C. Nói cách khác, cách viết thuộc tính của đối tượng như sau:

tên_đối_tượng.Tên_thuộc_tính

Với các đối tượng d1, d2, d3 và mảng d, có thể viết như sau:

```
d1.x;          // Thuộc tính x của đối tượng d1
d2.x;          // Thuộc tính x của đối tượng d2
d3.y;          // Thuộc tính y của đối tượng d3
d[2].m;        // Thuộc tính m của phần tử d[2]
d1.x = 100;    // Gán 100 cho d1.x
d2.y =d1.x;    // Gán d1.x cho d2.y
```

b. Sử dụng các phương thức

Cũng giống như hàm, một phương thức được sử dụng thông qua lời gọi. Tuy nhiên trong lời gọi phương thức bao giờ cũng phải có tên đối tượng để chỉ rõ phương thức thực hiện trên các thuộc tính của đối tượng nào.

Ví dụ lời gọi sau sẽ thực hiện nhập số liệu vào các thành phần d1.x, d1.y và d1.m: **d1.nhapsl();** Câu lệnh sau sẽ thực hiện nhập số liệu vào các thành phần d[3].x, d[3].y và d[3].m: **d[3].nhapsl() ;**

Chúng ta sẽ minh họa các điều nói trên bằng một chương trình đơn giản sử

dụng lớp DIEM để nhập 3 điểm, hiện rồi ẩn các điểm vừa nhập. Trong chương trình đưa vào hàm kd_do_hoa() dùng để khởi động hệ đồ hoạ.

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

class DIEM
{
private:
        int x, y, m ;
public:
        void nhapsl();
        void an() { putpixel(x,y,getbkcolor());}
        void hien();
};

void DIEM::nhapsl()
{
        cout << "\n Nhap hoanh do (cot) va tung do (hang) cua DIEM: " ;
        cin>> x >> y ;
        cout << " \n Nhap ma tran cua diem: " ;
        cin >> m ;
}

void DIEM::hien()
{
        int mau_ht;
        mau_ht = getcolor() ;
        putpixel(x,y,m);
        setcolor(mau_ht);
}

void kd_do_hoa()
{
        int mh, mode ;
```

```
    mh=mode=0;
    initgraph(&mh, &mode, "C:\\TC\BGI");
}

void main()
{
    DIEMd1, d2, d3 ;
    d1.nhapsl(); d2.nhapsl(); d3.nhapsl();
    kd_do_hoa();
    setbkcolor(BLACK);
    d1.hien(); d2.hien(); d3.hien();
    getch();
    d1.an(); d2.an(); d3.an();
    getch();
    closegraph();
}
```

c. Con trỏ đối tượng

Con trỏ đối tượng dùng để chứa địa chỉ của biến, mảng đối tượng. Nó được khai báo như sau:

```
Tên_lớp *con trỏ;
```

Ví dụ dùng lớp DIEM có thể khai báo:

```
DIEM *p1, *p2, *p3 ;        // Khai báo 3 con trỏ p1, p2, p3
DIEM d1, d2 ;               // Khai báo 2 đối tượng d1, d2
DIEM d[20] ;                // Khai báo mảng đối tượng
```

và có thể thực hiện các câu lệnh:

```
p1= &d2 ;                   // p1 chứa địa chỉ của d2 , hay p1 trỏ tới d2
p2 = d ;                    // p2 trỏ tới đầu mảng d
p3 = new DIEM               // Tạo một đt và chứa địa chỉ của nó vào p3
```

Để sử dụng thuộc tính của đối tượng thông qua con trỏ, ta viết như sau:

Tên_con_trỏ → Tên_thuộc_tính

Chú ý: Nếu con trỏ chứa địa chỉ đầu của mảng, có thể dùng con trỏ như tên mảng.

Như vậy sau khi thực hiện các câu lệnh trên thì:

p1 → x và d2.x là như nhau

p2[i].y và d[i].y là như nhau

Từ đó ta có quy tắc sử dụng thuộc tính: Để sử dụng một thuộc tính của đối tượng ta phải dùng phép . hoặc phép →. Trong chương trình, không cho phép viết tên thuộc tính một cách đơn độc mà phải đi kèm tên đối tượng hoặc tên con trỏ theo các mẫu sau:

Tên_đối_tượng.Tên_thuộc_tính

Tên_con_trỏ → Tên_thuộc_tính

Tên_mảng_đối_ tượng[chỉ_số].Tên_thuộc_tính

Tên_con_trỏ[chỉ_số].Tên_thuộc_tính

Chương trình dưới đây cũng sử dụng lớp DIEM để nhập một dãy điểm, hiển thị và ẩn các điểm vừa nhập. Chương trình dùng một con trỏ kiểu DIEM và dùng toán tử new để tạo ta một dãy đối tượng

```
#include <conio.h>
#include <iostream.h>
#include <graphics.h>
class DIEM
{
private:
      int x, y, m ;
public:
      void nhapsl();
      void an() { putpixel(x,y,getbkcolor());}
      void hien();
};

void DIEM::nhapsl()
{
      cout <<"\n Nhap hoanh do (cot) va tung do (hang) cua diem:" ;
      cin >> x >> y ;
      cout << " \n Nhap ma mau cua DIEM: " ;
      cin >> m ;
}

void DIEM::hien()
{
```

```
        int mau_ht;
        mau_ht = getcolor() ;
        putpixel(x,y,m);
        setcolor(mau_ht);
}

void kd_do_hoa()
{
        int mh, mode ;
        mh=mode=0;
        initgraph(&mh, &mode, "C:\\TC\BGI");
}

void main()
{
        DIEM *p;
        int i, n;
        cout << "So diem: " ;
        cin >> n;
        p = new DIEM[n+1];
        for (i=1;i<=n;++i)
        p[i].nhapsl();
        kd_do_hoa();
        for (i=1;i<=n;++i) p[i].hien();
        getch();
        for (i=1;i<=n;++i) p[i].an();
        getch();
        closegraph();
}
```

III. ĐỐI CỦA PHƯƠNG THỨC, CON TRỎ THIS

1. Con trỏ this là đối thứ nhất của phương thức

Chúng ta hãy xem lại phương thức nhapsl của lớp DIEM

```
void DIEM::nhapsl()
```

```
{
    cout <<"\n Nhap hoanh do (cot) va tung do (hang) cua diem:" ;
    cin >> x >> y ;
    cout <<" \n Nhap ma mau cua diem: " ;
    cin >> m ;
}
```

Trong phương thức này chúng ta sử dụng tên các thuộc tính x, y và m một cách đơn độc. Điều này có vẻ như mâu thuẫn với quy tắc sử dụng thuộc tính nêu trong mục trước. Thực tế C++ đã ngầm định sử dụng một con trỏ đặc biệt với tên gọi this trong các phương thức trên. Các thuộc tính viết trong phương thức được hiểu là thuộc một đối tượng do con trỏ this trỏ tới. Do đó, nếu tường minh hơn, phương thức nhapsl() có thể được viết dưới dạng tương đương như sau:

```
void DIEM::nhapsl()
{
    cout << "\n Nhap hoanh do (cot) va tung do (hang) cua diem:" ;
    cin >> this → x >> this → y ;
    cout << "\n Nhap ma mau cua diem: " ;
    cin >>this → m;
}
```

Như vậy có thể kết luận rằng: Phương thức bao giờ cũng có ít nhất một đối là con trỏ this và nó luôn luôn là đối đầu tiên của phương thức.

2. Tham số ứng với đối con trỏ this

Xét một lời gọi tới phương thức nhapsl() :

```
DIEM d1;
d1.nhapsl() ;
```

Trong trường hợp này tham số truyền cho con trỏ this chính là địa chỉ của d1:

```
this = &d1
```

Do đó:

this → x chính là d1.x

this → y chính là d1.y

this → m chính là d1.m

Như vậy câu lệnh:**d1.nhapsl() ;**sẽ nhập dữ liệu cho các thuộc tính của đối tượng d1. Từ đó có thể rút ra kết luận sau:

Tham số truyền cho đối con trỏ this chính là địa chỉ của đối tượng đi kèm với

phương thức trong lời gọi phương thức.

3. Các đối khác của phương thức

Ngoài đối đặc biệt this (đối này không xuất hiện một cách tường minh), phương thức còn có các đối khác được khai báo thư trong các hàm. Đối của phương thức có thể cókiểu bất kỳ (chuẩn và ngoài chuẩn).

Ví dụ để xây dựng phương thức vẽ đường thẳng qua 2 điểm ta cần đưa vào 3 đối: Hai đối là 2 biến kiểu DIEM, đối thứ ba kiểu nguyên xác định mã mầu. Vì đã có đối ngầm định this là đối thứ nhất, nên chỉ cần khai báo thêm 2 đối. Phương thức có thể viết như sau:

```
void DIEM::doan_thang(DIEM d2, int mau)
{
    int mau_ht;
    mau_ht = getcolor();
    setcolor(mau);
    line(this → x, this → y,d2.x,d2.y);
    setcolor(mau_ht);
}
```

Chương trình sau minh hoạ các phương thức có nhiều đối. Ta vẫn dùng lớp DIEM nhưng có một số thay đổi:

- Bỏ thuộc tính m (mầu)
- Bỏ các phương thức ***hien*** và ***an***
- Đưa vào 4 phương thức mới:

  ve_doan_thang (Vẽ đoạn thẳng qua 2 điểm)

  ve_tam_giac (Vẽ tam giác qua 3 điểm)

  do_dai (Tính độ dài của đoạn thẳng qua 2 điểm)

  chu_vi (Tính chu vi tam giác qua 3 điểm)

Chương trình còn minh hoạ:

- Việc phương thức này sử dụng phương thức khác (phương thức ve_tam_giac sử dụng phương thức ve_doan_thang, phương thức chu_vi sử dụng phương thức do_dai)
- Sử dụng con trỏ this trong thân các phương thức ve_tam_giac và chu_vi

Nội dung chương trình là nhập 3 điểm, vẽ tam giác có đỉnh là 3 điểm vừa nhập sau đó tính chu vi tam giác.

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

```
#include <iostream.h>
#include <graphics.h>
#include <math.h>
#include <stdio.h>
class DIEM
{
private:
	int x, y ;
public:
	void nhapsl();
	void ve_doan_thang(DIEM d2, int mau) ;
	void ve_tam_giac(DIEM d2, DIEM d3,int mau) ;
	double do_dai(DIEM d2)
	{
		DIEM d1 = *this ;
		return sqrt(pow(d1.x - d2.x,2) + pow(d1.y - d2.y,2) ) ;
	}
	double chu_vi(DIEM d2, DIEM d3);
};

void DIEM::nhapsl()
{
	cout <<" \n Nhap hoanh do (cot) va tung do (hang) cua diem:" ;
	cin >> x >> y;
}

void kd_do_hoa()
{
	int mh, mode ;
	mh=mode=0;
	initgraph(&mh, &mode, "");
}

void DIEM::ve_doan_thang(DIEM d2, int mau)
{
```

```
        setcolor(mau);
        line(this → x,this → y,d2.x,d2.y);
}
void DIEM:: ve_tam_giac(DIEM d2, DIEM d3,int mau)
{
        (*this).ve_doan_thang(d2,mau);
        d2.ve_doan_thang(d3,mau);
        d3.ve_doan_thang(*this,mau);
}
double DIEM:: chu_vi(DIEM d2, DIEM d3)
{
        double s;
        s=(*this).do_dai(d2)+ d2.do_dai(d3) + d3.do_dai(*this) ;
        return s;
}
void main()
{
        DIEM d1, d2, d3;
        char tb_cv[20] ;
        d1.nhapsl();
        d2.nhapsl();
        d3.nhapsl();
        kd_do_hoa();
        d1.ve_tam_giac(d2,d3,15);
        double s = d1.chu_vi(d2,d3);
        sprintf(tb_cv, "chu_vi = %0.2f", s);
        outtextxy(10,10,tb_cv);
        getch();
        closegraph();
}
```

Một số nhận xét về đối của phương thức và lời gọi phương thức:

+ Quan sát nguyên mẫu phương thức:

```
void ve_doan_thang(DIEM d2, int mau) ;
```

sẽ thấy phương thức có 3 đối:

Đối thứ nhất là một đối tượng DIEM do this trỏ tới

Đối thứ hai là đối tượng DIEM d2

Đối thứ ba là biến nguyên mẫu

Nội dung phương thức là vẽ một đoạn thẳng đi qua các điểm *this và d2 theo mã mầu mau. Xem thân của phương sẽ thấy được nội dung này:

```
void DIEM::ve_doan_thang(DIEM d2, int mau) ;
{
    setcolor(mau);
    line(this → x,this → y,d2.x,d2.y);
}
```

Tuy nhiên trong trường hợp này, vai trò của this không cao lắm, vì nó được đưa vào chỉ cốt làm rõ đối thứ nhất. Trong thân phương thức có thể bỏ từ khóa this vẫn được.

+ Vai trò của this trở nên quan trọng trong phương thức ve_tam_giac:

```
voidve_tam_giac(DIEM d2, DIEM d3, int mau)
```

Phương thức này có 4 đối là:

this : trỏ tới một đối tượng kiểu DIEM

d2 : một đối tượng kiểu DIEM

d3 : một đối tượng kiểu DIEM

mau : một biến nguyên

Nội dung phương thức là vẽ 3 cạnh:

cạnh 1 đi qua *this và d2

cạnh 2 đi qua d2 và d3

cạnh 3 đi qua d3 và *this

Các cạnh trên đuợc vẽ nhờ sử dụng phương thức ve_doan_thang:

Vẽ cạnh 1 dùng lệnh: (*this).ve_doan_thang(d2,mau) ;

Vẽ cạnh 2 dùng lệnh: d2.ve_doan_thang(d3,mau);

Vẽ cạnh 3 dùng lệnh: d3.ve_doan_thang(*this,mau);

Trong trường này rõ ràng vai trò của this rất quan trọng. Nếu không dùng nó thì công việc trở nên khó khăn, dài dòng và khó hiểu hơn. Chúng ta hãy so sánh 2 phương án:

Phương án dùng this trong phương thức ve_tam_giac:

```
void DIEM::ve_tam_giac(DIEM d2, DIEM d3, int mau)
{
    (*this).ve_doan_thang(d2,mau);
    d2.ve_doan_thang(d3,mau); d3.ve_doan_thang(*this,mau);
}
```

phương án không dùng this trong phương thức ve_tam_giac:

```
void DIEM::ve_tam_giac(DIEM d2, DIEM d3, int mau)
{
    DIEM d1;
    d1.x = x; d1.y = y;
    d1.ve_doan_thang(d2,mau);
    d2.ve_doan_thang(d3,mau);
    d3.ve_doan_thang(d1,mau);
}
```

IV. HÀM TẠO (CONSTRUCTOR)

1. Hàm tạo (hàm thiết lập)

Hàm tạo cũng là một phương thức của lớp (nhưng là hàm đặc biệt) dùng để tạo dựng một đối tượng mới. Chương trình dịch sẽ cấp phát bộ nhớ cho đối tượng sau đó sẽ gọi đến hàm tạo. Hàm tạo sẽ khởi gán giá trị cho các thuộc tính của đối tượng và có thể thực hiện một số công việc khác nhằm chuẩn bị cho đối tượng mới.

a. Cách viết hàm tạo

i. Điểm khác của hàm tạo và các phương thức thông thường:

Khi viết hàm tạo cần để ý 3 sự khác biệt của hàm tạo so với các phương thức khác như sau:

- Tên của hàm tạo: Tên của hàm tạo bắt buộc phải trùng với tên của lớp.
- Không khai báo kiểu cho hàm tạo.
- Hàm tạo không có kết quả trả về.

ii. Sự giống nhau của hàm tạo và các phương thức thông thường

Ngoài 3 điểm khác biệt trên, hàm tạo được viết như các phương thức khác:

- Hàm tạo có thể được xây dựng bên trong hoặc bên ngoài định nghĩa lớp.

- Hàm tạo có thể có đối hoặc không có đối.
- Trong một lớp có thể có nhiều hàm tạo (cùng tên nhưng khác bộ đối).

Ví dụ sau định nghĩa lớp DIEM_DH (Điểm đồ họa) có 3 thuộc tính:

```
int x;          // hoành độ (cột) của điểm
int y;          // tung độ (hàng) của điểm
int m;          // mầu của điểm
```

và đưa vào 2 hàm tạo để khởi gán cho các thuộc tính của lớp:

```
// Hàm tạo không đối: Dùng các giá trị cố định để khởi gán cho x, y, m
DIEM_DH() ;
// Hàm tạo có đối: Dùng các đối x1, y1, m1 để khởi gán cho x, y, m
DIEM_DH(int x1, int y1, int m1 = 15) ;      // Đối m1 có giá trị mặc định 15
class DIEM_DH                               // (mầu trắng)
{
private:
    int x, y, m ;
public:
    // Hàm tạo không đối: khởi gán cho x = 0, y = 0, m = 1
    // Hàm này viết bên trong định nghĩa lớp
    DIEM_DH()
    {
        x = y = 0;
        m = 1;
    }
    // Hàm tạo này xây dựng bên ngoài định nghĩa lớp
    DIEM_DH(int x1, int y1, int m1 = 15) ;
    // Các phương thức khác
};

// Xây dựng hàm tạo bên ngoài định nghĩa lớp
DIEM_DH:: DIEM_DH(int x1, int y1, int m1) ;
{
    x = x1; y = y1; m = m1;
}
```

b. Dùng hàm tạo trong khai báo

+ Khi đã xây dựng các hàm tạo, ta có thể dùng chúng trong khai báo để tạo ra một đối tượng đồng thời khởi gán cho các thuộc tính của đối tượng được tạo. Dựa vào các tham số trong khai báo mà trình biên dịch sẽ biết cần gọi đến hàm tạo nào.
+ Khi khai báo một biến đối tượng có thể sử dụng các tham số để khởi gán cho các thuộc tính của biến đối tượng.
+ Khi khai báo mảng đối tượng không cho phép dùng các tham số để khởi gán.
+ Câu lệnh khai báo một biến đối tượng sẽ gọi tới hàm tạo 1 lần.
+ Câu lệnh khai báo một mảng n đối tượng sẽ gọi tới hàm tạo n lần.

Ví dụ:

```
DIEM_DH d;                      // Gọi tới hàm tạo không đối.
// Kết quả d.x = 0, d.y = 0, d.m = 1
DIEM_DH u(300, 100, 5);         // Gọi tới hàm tạo có đối.
// Kết quả u.x = 300, u.y = 100, d.m = 5
DIEM_DH v(400, 200);            // Gọi tới hàm tạo có đối.
// Kết quả v.x = 400, v.y = 200, d.m = 15
DIEM_DH p[20] ;                 // Gọi tới hàm tạo không đối 20 lần
```

Chú ý: Với các hàm có đối kiểu lớp, thì đối chỉ xem là các tham số hình thức, vì vậy khai báo đối (trong dòng đầu của hàm) sẽ không tạo ra đối tượng mới và do đó không gọi tới các hàm tạo.

c. Dùng hàm tạo trong cấp phát bộ nhớ

+ Khi cấp phát bộ nhớ cho một đối tượng có thể dùng các tham số để khởi gán cho các thuộc tính của đối tượng, ví dụ

```
DIEM_DH *q = new DIEM_DH(40, 20, 4);        // Gọi tới hàm tạo có đối
// Kết quả q → x = 40, q → y = 20, q → m = 4
DIEM_DH *r = new DIEM_DH ;                  //Gọi tới hàm tạo không đối
// Kết quả r → x = 0, r → y = 0, r → m = 1
```

+ Khi cấp phát bộ nhớ cho một dãy đối tượng không cho phép dùng tham số để khởi gán, ví dụ:

```
int n = 30;
DIEM_DH *s = new DIEM_DH[n] ;     // Gọi tới hàm tạo không đối 30 lần.
```

d. Dùng hàm tạo để biểu diễn các đối tượng hằng

+ Như đã biết, sau khi định nghĩa lớp DIEM_DH thì có thể xem lớp này như một kiểu dữ liệu như int, double, char, ...

Với kiểu int chúng ta có các hằng int, như 253.

Với kiểu double chúng ta có các hằng double, như 75.42

Khái niệm hằng kiểu int, hằng kiểu double có thể mở rộng cho hằng kiểu DIEM_DH

+ Để biểu diễn một hằng đối tượng (hay còn gọi: Đối tượng hằng) chúng ta phải dùng tới hàm tạo. Mẫu viết như sau:

Tên_lớp(danh sách tham số) ;

Ví dụ đối với lớp DIEM_DH nói trên, có thể viết như sau:

```
DIEM_DH(234, l 23, 4)      // Biểu thị một đối tượng kiểu DIEM_DH
                           // có các thuộc tính x = 234, y = 123, m = 4
```

Chú ý: Có thể sử dụng một hằng đối tượng như một đối tượng. Nói cách khác, có thể dùng hằng đối tượng để thực hiện một phương thức, ví dụ nếu viết:

```
DIEM_DH(234, l 23, 4).in();
```

thì có nghĩa là thực hiện phương thức in() đối với hằng đối tượng.

e. Ví dụ minh họa

Chương trình sau đây minh họa cách xây dựng hàm tạo và cách sử dùng hàm tạo trong khai báo, trong cấp phát bộ nhớ và trong việc biểu diễn các hằng đối tượng.

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
class DIEM_DH
{
private:
    int x, y, m;
public:
    // Hàm bạn dùng để in đối tượng DIEM_DH
    friend void in(DIEM_DH d)
    {
        cout <<"\n " << d.x << " "<< d.y<<" " << d.m ;
    }
    // Phương thức dùng để in đối tượng DIEM_DH
```

```
        void in()
        {
                cout <<"\n " << x << " "<< y<<" " << m ;
        }
        // Hàm tạo không đối
        DIEM_DH()
        {
                x = y = 0;
                m = 1;
        }
        // Hàm tạo có đối, đối m1 có giá trí mặc định là 15 (mầu trắng)
        DIEM_DH(int x1, int y1, int m1 = 15);
};

// Xây dựng hàm tạo
DIEM_DH::DIEM_DH(int x1, int y1, int m1)
{
        x = x1; y = y1; m = m1;
}

void main()
{
        DIEM_DH d1;                     // Gọi tới hàm tạo không đối
        DIEM_DH d2(200, 200, 10);       // Gọi tới hàm tạo có đối
        DIEM_DH*d;
        d = new DIEM_DH(300, 300);      // Gọi tới hàm tạo có đối
        clrscr();
        in(d1);                         //Gọi hàm bạn in()
        d2.in();                        //Gọi phương thức in()
        in(*d);                         // Gọi hàm bạn in()
        DIEM_DH(2, 2, 2).in();          // Gọi phương thức in()
        DIEM_DH t[3];                   // 3 lần gọi hàm tạo không đối
        DIEM_DH*q;                      // Gọi hàm tạo không đối
        int n;
```

```
        cout << "\n N = "; cin >> n;
        q = new DIEM_DH[n+1];          // (n+1) lần gọi hàm tạo không đối
        for (int i = 0; i< = n; ++i)
            q[i] = DIEM_DH(300+i, 200+i, 8); //(n+1) lần gọi hàm tạo có đối
        for (i = 0; i< = n; ++i)
            q[i].in();                          // Gọi phương thức in()
        for (i = 0; i< = n; ++i)
            DIEM_DH(300+i, 200+i, 8).in();      // Gọi phương thức in()
        getch();
}
```

2. Lớp không có hàm tạo và hàm tạo mặc định

a. Nếu lớp không có hàm tạo

Chương trình dịch sẽ cung cấp một hàm tạo mặc định không đối (default), hàm này thực chất không làm gì cả. Như vậy một đối tượng tạo ra chỉ được cấp phát bộ nhớ, còn các thuộc tính của nó chưa được xác định. Chúng ta có thể kiểm chứng điều này, bằng cách chạy chương trình sau:

```
// Hàm tạo mặc định
#include <conio.h>
#include <iostream.h>
class DIEM_DH
{
private:
    int x, y, m;
public:
    // Phương thức
    void in() { cout <<"\n " << x << " "<< y<<" " << m ; }
};

void main()
{
    DIEM_DH d;
    d.in();
    DIEM_DH *p;
```

```
    p = new DIEM_DH[10];
    clrscr();
    d.in();
    for (int i = 0; i<10; ++i) (p+i)->in();
    getch();
}
```

b. Nếu trong lớp đã có ít nhất một hàm tạo

Khi đó hàm tạo mặc định sẽ không được phát sinh nữa và mọi câu lệnh xây dựng đối tượng mới đều sẽ gọi đến một hàm tạo của lớp. Nếu không tìm thấy hàm tạo cần gọi thì chương trình dịch sẽ báo lỗi. Điều này thường xẩy ra khi chúng ta không xây dựng hàm tạo không đối, nhưng lại sử dụng các khai báo không tham số như ví dụ sau:

```
#include <conio.h>
#include <iostream.h>
class DIEM_DH
{
private:
    int x, y, m;
public:
    // Phương thức dùng để in đối tượng DIEM_DH
    void in()
    {
        cout <<"\n" << x << " "<< y<<" " << m ;
    }
    //Hàm tạo có đối
    DIEM_DH::DIEM_DH(int x1, int y1 , int m1)
    {
        x = x1; y = y1 ; m = m1;
    }
};

void main()
{
    DIEM_DH d1(200, 200, 10); // Gọi tới hàm tạo có đối
```

```
    DIEM_DH d2; // Gọi tới hàm tạo không đối
    d2 = DIEM_DH(300, 300, 8); // Gọi tới hàm tạo có đối
    d1.in();
    d2.in();
    getch();
}
```

Trong các câu lệnh trên, chỉ có câu lệnh thứ 2 trong hàm main() là bị báo lỗi. Câu lệnh này sẽ gọi tới hàm tạo không đối, mà hàm này chưa được xây dựng.

Giải pháp: có thể chọn một trong 2 giải pháp sau:

- Xây dựng thêm hàm tạo không đối.
- Gán giá trị mặc định cho tất cả các đối x1, y1 và m1 của hàm tạo đã xây dựng ở trên.

Theo phương án 2, chương trình có thể sửa như sau:

```
#include <conio.h>
#include <iostream.h>
class DIEM_DH
{
private:
    int x, y, m;
public:
    // Phương thức dùng để in đối tượng DIEM_DH
    void in() { cout <<"\n " << x << " "<< y<<" " << m ; }
    // Hàm tạo có đối , tất cả các đối đều có giá trị mặc định
    DIEM_DH::DIEM_DH(int x1 = 0, int y1 = 0, int m1 = 15)
    {
        x = x1; y = y1; m = m1;
    }
};

void main()
{
    // Gọi tới hàm tạo, không dùng tham số mặc định
    DIEM_DH d1(200, 200, 10);
    // Gọi tới hàm tạo, dùng 3 tham số mặc định
```

```
    DIEM_DH d2;
    // Gọi tới hàm tạo, dùng 1 tham số mặc định
    d2 = DIEM_DH(300, 300);
    d1.in();
    d2.in();
    getch();
}
```

3. Hàm tạo sao chép (Copy Constructor)

a. Hàm tạo sao chép mặc định

Giả sử đã định nghĩa một lớp nào đó, ví dụ lớp PS (phân số). Khi đó:

+ Ta có thể dùng câu lệnh khai báo hoặc cấp phát bộ nhớ để tạo các đối tượng mới, ví dụ:

```
PS p1, p2 ;
PS *p = new PS ;
```

+ Ta cũng có thể dùng lệnh khai báo để tạo một đối tượng mới từ một đối tượng đã tồn tại, ví dụ:

```
PS u;
PS v(u) ; // Tạo v theo u
```

ý nghĩa của câu lệnh này như sau:

– Nếu trong lớp PS chưa xây dựng hàm tạo sao chép, thì câu lệnh này sẽ gọi tới một hàm tạo sao chép mặc định (của C++). Hàm này sẽ sao chép nội dung từng bit của u vào các bit tương ứng của v. Như vậy các vùng nhớ của u và v sẽ có nội dung như nhau. Rõ ràng trong đa số các trường hợp, nếu lớp không có các thuộc tính kiểu con trỏ hay tham chiếu, thì việc dùng các hàm tạo sao chép mặc định (để tạo ra một đối tượng mới có nội dung như một đối tượng cho trước) là đủ và không cần xây dựng một hàm tạo sao chép mới.

– Nếu trong lớp PS đã có hàm tạo sao chép (cách viết sẽ nói sau) thì câu lệnh: PS v(u); sẽ tạo ra đối tượng mới v, sau đó gọi tới hàm tạo sao chép để khởi gán v theo u.

Ví dụ sau sẽ minh họa cách dùng hàm tạo sao chép mặc định:

Trong chương trình đưa vào lớp PS (phân số):

+ Các thuộc tính gồm: t (tử số) và m (mẫu).

+ Trong lớp không có phương thức nào cả mà chỉ có 2 hàm bạn là các hàm toán tử nhập (>>) và xuất (<<).

+ Nội dung chương trình là: Dùng lệnh khai báo để tạo một đối tượng u (kiểu PS) có nội dung như đối tượng đã có d.

```
// Ham tao sao chep mac dinh
#include <conio.h>
#include <iostream.h>
class PS
{
private:
	int t, m ;
public:
	friend ostream& operator<< (ostream&os, const PS &p)
	{
		os << " = " << p.t << "/" << p.m;
		return os;
	}
	friend istream& operator>> (istream& is, PS &p)
	{
		cout << "\n Nhap tu va mau: " ;
		is >> p.t >> p.m ;
		return is;
	}
} ;

void main()
{
	PS d;
	cout << "\n Nhap PS d "; cin >> d;
	cout << "\n PS d " << d;
	PS u(d);
	cout << "\n PS u "<< u;
	getch();
}
```

b. Cách xây dựng hàm tạo sao chép

+ Hàm tạo sao chép sử dụng một đối kiểu tham chiếu đối tượng để khởi gán

cho đối tượng mới. Hàm tạo sao chép được viết theo mẫu:

```
Tên_lớp (const Tên_lớp & dt)
{
        // Các câu lệnh dùng các thuộc tính của đối tượng dt
        // để khởi gán cho các thuộc tính của đối tượng mới
}
```

+ Ví dụ có thể xây dựng hàm tạo sao chép cho lớp PS như sau:

```
class PS
{
private:
        int t, m ;
public:
        PS (const PS &p)
        {
                this->t = p.t ;
                this->m = p.m ;
        }
        ...
} ;
```

c. Khi nào cần xây dựng hàm tạo sao chép

+ Nhận xét: Hàm tạo sao chép trong ví dụ trên không khác gì hàm tạo sao chép mặc định.
+ Khi lớp không có các thuộc tính kiểu con trỏ hoặc tham chiếu, thì dùng hàm tạo sao chép mặc định là đủ.
+ Khi lớp có các thuộc tính con trỏ hoặc tham chiếu, thì hàm tạo sao chép mặc định chưa đáp ứng được yêu cầu.

Ví dụ:

```
class DT
{
private:
        int n; // Bac da thuc
        double *a; // Tro toi vung nho chua cac he so da thuc a0, a1, ...
public:
```

```
        DT() { this->n0; this->a = NULL; }
        DT(int n1)
        {
            this->n = n1;
            this->a = new double[n1+1];
        }
        friend ostream& operator << (ostream& os, const DT &d);
        friend istream& operator>> (istream& is, DT &d);
        ...
    } ;
```

Bây giờ chúng ta hãy theo dõi xem việc dùng hàm tạo mặc định trong đoạn chương trình sau sẽ dẫn đến sai lầm như thế nào:

```
        DT d ; // Tạo đối tượng d kiểu DT
        cin >> d ;
```

/* Nhập đối tượng d, gồm: nhập một số nguyên dương và gán cho d.n, cấp phát vùng nhớ cho d.a, nhập các hệ số của đa thức và chứa vào vùng nhớ được cấp phát */

```
        DT u(d);
```

/* Dùng hàm tạo mặc định để xây dựng đối tượng u theo d. Kết quả: u.n = d.n và u.a = d.a. Như vậy 2 con trỏ u.a và d.a cùng trỏ đến một vùng nhớ */

Nhận xét: Mục đích là tạo ra một đối tượng u giống như d, nhưng độc lập với d. Nghĩa là khi d thay đổi thì u không bị ảnh hưởng gì. Thế nhưng mục tiêu này không đạt được, vì u và d có chung một vùng nhớ chứa hệ số của đa thức, nên khi sửa đổi các hệ số của đa thức trong d thì các hệ số của đa thức trong u cũng thay đổi theo. Còn một trường hợp nữa cũng dẫn đến lỗi là khi một trong 2 đối tượng u và d bị giải phóng (thu hồi vùng nhớ chứa đa thức) thì đối tượng còn lại cũng sẽ không còn vùng nhớ nữa.

Ví dụ sau sẽ minh họa nhận xét trên: Khi d thay đổi thì u cũng thay đổi và ngược lại khi u thay đổi thì d cũng thay đổi theo.

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DT
{
private:
    int n; // Bac da thuc
```

```
        double *a; // Tro tơi vung nho chua cac he so da thuc a0, a1 , ...
public:
        DT() { this->n = 0; this->a = NULL; }
        DT(int n1)
        {
                this->n = n1 ;
                this->a = new double[n1+1];
        }
        friend ostream& operator<< (ostream& os, const DT &d);
        friend istream& operator>> (istream& is, DT &d);
} ;

ostream& operator<< (ostream& os, const DT &d)
{
        os << " Cac he so (tu ao): ";
        for (int i = 0 ; i< = d.n ; ++i)
        os << d.a[i] <<" " ;
        return os;
}

istream& operator >> (istream& is, DT &d)
{
        if (d.a! = NULL) delete d.a;
        cout << " \n Bac da thuc: " ;
        cin >> d.n;
        d.a = new double[d.n+1];
        cout << "Nhap cac he so da thuc:\n" ;
        for (int i = 0 ; i< = d.n ; ++i)
        {
                cout << "He so bac "<< i << " = " ;
                is >> d.a[i] ;
        }
        return is;
}
```

```
void main()
{
        DT d;
        clrscr();
        cout <<"\n Nhap da thuc d " ; cin >> d;
        DT u(d);
        cout << "\n Da thuc d "<< d ;
        cout << "\n Da thuc u " << u ;
        cout <<"\n Nhap da thuc d " ; cin >> d;
        cout << "\nDa thuc d " << d;
        cout <<"\n Da thuc u " << u ;
        cout <<"\n Nhap da thuc u " ; cin >> u;
        cout << "\n Da thuc d "<< d ;
        cout << "\n Da thuc u " << u ;
        getch();
}
```

d. Ví dụ về hàm tạo sao chép

Trong chương trình trên đã chỉ rõ: Hàm tạo sao chép mặc định là chưa thoả mãn đối với lớp DT. Vì vậy cần viết hàm tạo sao chép để xây dựng đối tượng mới (ví dụ u) từ một đối tượng đang tồn tại (ví dụ d) theo các yêu cầu sau:

+ Gán d.n cho u.n
+ Cấp phát một vùng nhớ cho u.a để có thể chứa được (d.n + 1) hệ số.
+ Gán các hệ số chứa trong vùng nhớ của d.a sang vùng nhớ của u.a

Như vậy chúng ta sẽ tạo được đối tượng u có nội dung ban đầu giống như d, nhưng độc lập với d.

Để đáp ứng các yêu cầu nêu trên, hàm tạo sao chép cần được xây dựng như sau:

```
DT::DT(const DT &d)
{
        this → n = d.n ;
        this → a = new double[d.n+1];
        for (int i = 0; i< = d.n; ++i)
        this → a[i] = d.a[i];
}
```

Chương trình sau sẽ minh họa điều này: Sự thay đổi của d không làm ảnh hưởng đến u và ngược lại sự thay đổi của u không làm ảnh hưởng đến d.

```
// Viết hàm tạo sao chép cho lớp DT
#include <conio.h>
#include <iostream.h>
#include <math.h>
class DT
{
private:
    int n; // Bac da thuc
    double *a; // Tro toi vung nho chua cac he so da thuc a0, a1 , ...
public:
    DT() { this → n = 0; this → a = NULL; }
    DT(int n1)
    {
        this → n = n1 ;
        this → a = new double[n1+1];
    }
    DT(const DT &d);
    friend ostream& operator<< (ostream& os, const DT&d);
    friend istream& operator>> (istream& is, DT&d);
};

DT::DT(const DT&d)
{
    this → n = d.n;
    this → a = new double[d.n+1];
    for (int i = 0; i< = d.n; ++i)
    this → a[i] = d.a[i];
}

ostream& operator<< (ostream& os, const DT &d)
{
    os << " Cac he so (tu ao): " ;
```

```
        for (int i = 0 ; i< = d.n ; ++i) os << d.a[ i] <<" " ;
        return os;
}

istream& operator>> (istream& is, DT &d)
{
        if (d.a! = NULL) delete d.a;
        cout << "\n Bac da thuc: " ;
        cin >> d.n;
        d.a = new double[d.n+1];
        cout << "Nhap cac he so da thuc:\n" ;
        for (int i = 0 ; i< = d.n ; ++i)
        {
                cout << "He so bac " << i << " = " ;
                is >> d.a[i] ;
        }
        return is;
}

void main()
{
        DT d;
        clrscr();
        cout <<"\n Nhap da thuc d " ; cin >> d;
        DT u(d);
        cout <<"\n Da thuc d " << d ;
        cout << "\n Da thuc u " << u ;
        cout <<"\n Nhap da thuc d " ; cin >> d;
        cout << "\n Da thuc d "<< d ;
        cout <<"\n Da thuc u " << u ;
        cout <<"\n Nhap da thuc u " ; cin >> u;
        cout << "\n Da thuc d " << d ;
        cout <<"\n Da thuc u " << u ;
        getch();
```

```
}
```

V. HÀM HỦY (DESTRUCTOR)

Hàm hủy là một hàm thành viên của lớp (phương thức) có chức năng ngược với hàm tạo. Hàm hủy được gọi trước khi giải phóng (xoá bỏ) một đối tượng để thực hiện một số công việc có tính "dọn dẹp" trước khi đối tượng được hủy bỏ, ví dụ như giải phóng một vùng nhớ mà đối tượng đang quản lý, xoá đối tượng khỏi màn hình nếu như nó đang hiển thị, ...

Việc hủy bỏ một đối tượng thường xẩy ra trong 2 trường hợp sau:

+ Trong các toán tử và các hàm giải phóng bộ nhớ, như delete, free, ...
+ Giải phóng các biến, mảng cục bộ khi thoát khỏi hàm, phương thức.

1. Hàm hủy mặc định

Nếu trong lớp không định nghĩa hàm hủy, thì một hàm hủy mặc định không làm gì cả được phát sinh. Đối với nhiều lớp thì hàm hủy mặc định là đủ, và không cần đưa vào một hàm hủy mới.

2. Quy tắc viết hàm hủy

Mỗi lớp chỉ có một hàm hủy viết theo các quy tắc sau:

+ Kiểu của hàm: Hàm hủy cũng giống như hàm tạo là hàm không có kiểu, không có giá trị trả về.
+ Tên hàm: Tên của hàm hủy gồm một dấu ngã (đứng trước) và tên lớp:

  ~Tên_lớp
+ Đối: Hàm hủy không có đối

Ví dụ có thể xây dựng hàm hủy cho lớp DT (đa thức) như sau:

```
class DT
{
private:
    int n; // Bac da thua
    double *a; // Tro toi vung nho chua cac he so da thuc a0, a1 , ...
public:
    ~DT()
    {
        this → n = 0;
        delete this → a;
```

```
        }
    ...
};
```

3. Vai trò của hàm hủy trong lớp DT

Trong phần trước định nghĩa lớp DT (đa thức) khá đầy đủ gồm:

+ Các hàm tạo
+ Các toán tử nhập >>, xuất <<
+ Các hàm toán tử thực hiện các phép tính +, -, *, /

Tuy nhiên vẫn còn thiếu hàm hủy để giải phóng vùng nhớ mà đối tượng kiểu DT (cần hủy) đang quản lý.

Chúng ta hãy phân tích các khiếm khuyết của chương trình này:

+ Khi chương trình gọi tới một phương thức toán tử để thực hiện các phép tính cộng, trừ, nhân đa thức, thì một đối tượng trung gian được tạo ra. Một vùng nhớ được cấp phát và giao cho nó (đối tượng trung gian) quản lý.
+ Khi thực hiện xong phép tính sẽ ra khỏi phương thức. Đối tượng trung gian bị xoá, tuy nhiên chỉ vùng nhớ của các thuộc tính của đối tượng này được giải phóng. Còn vùng nhớ (chứa các hệ số của đa thức) mà đối tượng trung gian đang quản lý thì không hề bị giải phóng. Như vậy số vùng nhớ bị chiếm dụng vô ích sẽ tăng lên.

Nhược điểm trên dễ dàng khắc phục bằng cách đưa vào lớp DT hàm hủy trong mục 3 ở trên.

4. Ví dụ

Phần này chúng tôi trình bày một ví dụ tương đối hoàn chỉnh về lớp các hình tròn trong chế độ đồ họa. Chương trình gồm:

i. Lớp HT (hình tròn) với các thuộc tính:

```
int r;                  // Bán kính
int m;                  // Mầu hình tròn
int xhien, yhien;       // Vị trí hiển thị hình tròn trên màn hình
char *pht;              // Con trỏ trỏ tới vùng nhớ chứa ảnh hình tròn
int hienmh;             // Trạng thái hiện (hienmh = 1), ẩn (hienmh = 0)
```

ii. Các phương thức

+ Hàm tạo không đối thực hiện việc gán giá trị bằng 0 cho các thuộc tính của lớp. HT();
+ Hàm tạo có đối. HT(int n, int m1 = 15);

Thực hiện các việc:

– Gán r1 cho r, m1 cho m

– Cấp phát bộ nhớ cho pht

– Vẽ hình tròn và lưu ảnh hình tròn vào vùng nhớ của pht

+ Hàm hủy: ~HT();

Thực hiện các việc:

– Xoá hình tròn khỏi màn hình (nếu đang hiển thị)

– Giải phóng bộ nhớ đã cấp cho pht

+ Phương thức: void hien(int x, int y);

Có nhiệm vụ hiển thị hình tròn tại (x, y)

+ Phương thức : void an()

Có nhiệm vụ làm ẩn hình tròn

iii. Các hàm độc lập:

```
void ktdh();                    // Khởi tạo đồ họa
void ve_bau_troi();             // Vẽ bầu trời sao
void ht_di_dong_xuong();        // Vẽ một cặp 2 hình tròn di chuyển xuống
void ht_di_dong_len();          // Vẽ một cặp 2 hình tròn di chuyển lên trên
```

Nội dung chương trình là tạo ra các chuyển động xuống và lên của các hình tròn.

```
// Lop do hoa
// Ham huy
// Trong ham huy co the goi PT khac
#include <conio.h>
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <dos.h>
void ktdh();                    // Khởi tạo đồ họa
void ve_bau_troi();             // Vẽ bầu trời sao
void ht_di_dong_xuong();        // Vẽ một cặp 2 hình tròn di chuyển xuống
void ht_di_dong_len();          // Vẽ một cặp 2 hình tròn di chuyển lên trên
```

```
int xmax, ymax;
class HT
{
private:
        int r, m ;
        int xhien, yhien;
        char *pht;
        int hienmh;
public:
        HT();
        HT(int n, int m1 = 15);
        ~HT();
        void hien(int x, int y);
        void an();
};

HT:: HT()
{
        r = m = hienmh = 0;
        xhien = yhien = 0;
        pht = NULL;
}

HT::HT(int n, int m1)
{
        r = n; m = m1; hienmh = 0;
        xhien = yhien = 0;
        if (r<0) r = 0;
        if (r = = 0) pht = NULL;
        else
        {
                int size; char *pmh;
                size = imagesize(0, 0, r+r, r+r);
                pmh = new char[size];
```

```
            getimage(0, 0, r+r, r+r, pmh);
            setcolor(m);
            circle(r, r, r );
            setfillstyle(1, m);
            floodfill(r, r, m);
            pht = new char[size];
            getimage(0, 0, r+r, r+r, pht);
            putimage(0, 0, pmh, COPY_PUT);
            delete pmh;
            pmh = NULL;
        }
    }

    void HT::hien(int x, int y)
    {
        if (pmh! = NULL && !hienmh) // Chua hien
        {
            hienmh = 1;
            xhien = x; yhien = y;
            putimage(x, y, pht, XOR_PUT);
        }
    }

    void HT::an()
    {
        if (hienmh) // Dang hien
        {
            hienmh = 0;
            putimage(xhien, yhien, pht, XOR_PUT);
        }
    }

    HT::~HT()
    {
        an();
```

```
        if (pht! = NULL)
        {
                delete pht;
                pht = NULL;
        }
}

void ktdh()
{
        int mh = 0, mode = 0;
        initgraph(&mh, &mode, " ");
        xmax = getmaxx();
        ymax = getmaxy();
}

void ve_bau_troi()
{
        for (int i = 0; i<2000; ++i)
        putpixel(random(xmax), random(ymax), 1+random( 15));
}

void ht_di_dong_xuong()
{
        HT h(50, 4);
        HT u(60, 15);
        h.hien(0, 0);
        u.hien(40, 0);
        for (int x = 0; x< = 340; x+ = 10)
        {
                h.an();
                u.an();
                h.hien(x, x);
                delay(200);
                u.hien(x+40, x);
                delay(200);
```

```
        }
}

void ht_di_dong_len()
{
        HT h(50, 4);
        HT u(60, 15);
        h.hien(340, 340);
        u.hien(380, 340);
        for (int x = 340; x> = 0; x- = 10)
        {
                h.an();
                u.an();
                u.hien(x, x);
                delay(200);
                u.hien(x+40, x);
                delay(200);
        }
};

void main()
{
        ktdh();
        ve_bau_troi();
        ht_di_dong_xuong();
        ht_di_dong_len();
        getch();
        closegraph();
}
```

Các nhận xét:

+ Trong thân hàm hủy gọi tới phương thức an().
+ Điều gì xẩy ra khi bỏ đi hàm hủy:
  - Khi gọi hàm ht_di_dong_xuong() thì có 2 đối tượng kiểu HT được tạo ra. Trong thân hàm sử dụng các đối tượng này để vẽ các hình tròn di

chuyển xuống. Khi thoát khỏi hàm thì 2 đối tượng (tạo ra ở trên) được giải phóng. Vùng nhớ của các thuộc tính của chúng bị thu hồi, nhưng vùng nhớ cấp phát cho thuộc tính pht chưa được giải phóng và ảnh của 2 hình tròn (ở phía dưới màn hình) vẫn không được cất đi.

- Điều tương tự xẩy ra sau khi ra khỏi hàm ht_di_dong_len(): vùng nhớ cấp phát cho thuộc tính pht chưa được giải phóng và ảnh của 2 hình tròn (ở phía trên màn hình) vẫn không được thu dọn.

VI. CÁC HÀM TRỰC TUYẾN (INLINE)

Một số mở rộng của C++ đối với C đã được trình bày trong các chương trước như biến tham chiếu, định nghĩa chồng hàm, hàm với đối mặc định … Phần này ta xem một đặc trưng khác của C++ được gọi là hàm trực tuyến (inline).

1. Ưu nhược điểm của hàm

Việc tổ chức chương trình thành các hàm có 2 ưu điểm rõ rệt:

Thứ nhất là chia chương trình thành các đơn vị độc lập, làm cho chương trình được tổ chức một cách khoa học dễ kiểm soát, dễ phát hiện lỗi, dễ phát triển và mở rộng.

Thứ hai là giảm được kích thước chương trình, vì mỗi đoạn chương trình thực hiện nhiệm vụ của hàm được thay bằng một lời gọi hàm.

Tuy nhiên hàm cũng có nhược điểm là làm chậm tốc độ chương trình do phải thực hiện một số thao tác có tính thủ tục mỗi khi gọi hàm như: cấp phát vùng nhớ cho các đốivà biến cục bộ, truyền dữ liệu của các tham số cho các đối, giải phóng vùng nhớ trước khi thoát khỏi hàm.

Các hàm trực tuyến trong C++ có khả năng khắc phục được các nhược điểm nói trên.

2. Các hàm trực tuyến

Để biến một hàm thành trực tuyến ta viết thêm từ khoá **inline** vào trước khai báo nguyên mẫu hàm. Nếu không dùng nguyên mẫu thì viết từ khoá này trước dòng đầu tiên của định nghĩa hàm.

Ví dụ 1 :

```
inline float f(int n, float x);
float f(int n, float x)
{
      // Các câu lệnh trong thân hàm
}
```

hoặc

```
inline float f(int n, float x)
{
    // Các câu lệnh trong thân hàm
}
```

Chú ý: Trong mọi trường hợp, từ khoá inline phải xuất hiện trước các lời gọi hàm thì trình biên dịch mới biết cần xử lý hàm theo kiểu inline.

Ví dụ hàm f trong chương trình sau sẽ không phải là hàm trực tuyến vì từ khoá inline viết sau lời gọi hàm:

```
#include <conio.h>
#include <iostream.h>
void main()
{
    int s ;
    s = f(5,6);
    cout << s ;
    getch();
}
inline int f(int a, int b)
{
    return a*b;
}
```

Chú ý: Trong C^{++}, nếu hàm được xây dựng sau lời gọi hàm thì bắt buộc phải khai báo nguyên mẫu hàm trước lời gọi. Trong ví dụ trên, trình biên dịch C^{++} sẽ bắt lỗi vì thiếu khai báo nguyên ngẫu hàm f .

3. Cách biên dịch và dùng hàm trực tuyến

Chương trình dịch xử lý các hàm inline như các macro (được định nghĩa trong lệnh #define), nghĩa là nó sẽ thay mỗi lời gọi hàm bằng một đoạn chương trình thực hiện nhiệm vụ của hàm. Cách này làm cho chương trình dài ra, nhưng tốc độ chương trình tăng lên do không phải thực hiện các thao tác có tính thủ tục khi gọi hàm.

Phương án dùng hàm trực tuyến rút ngắn được thời gian chạy máy nhưng lại làm tăng khối lượng bộ nhớ chương trình (nhất là đối với các hàm trực tuyến có nhiều câu lệnh). Vì vậy chỉ nên dùng phương án trực tuyến đối với các hàm nhỏ.

4. Sự hạn chế của trình biên dịch

Không phải khi gặp từ khoá inline là trình biên dịch nhất thiết phải xử lý hàm theo kiểu trực tuyến.

Có một số hàm mà các trình biên dịch thường không xử lý theo cách inline như các hàm chứa biến static, hàm chứa các lệnh chu trình hoặc lệnh goto hoặc lệnh switch, hàm đệ quy. Trong trường hợp này từ khoá inline lẽ dĩ nhiên bị bỏ qua.

Thậm chí từ khoá inline vẫn bị bỏ qua ngay cả đối với các hàm không có những hạn chế nêu trên nếu như trình biên dịch thấy cần thiết (ví dụ đã có quá nhiều hàm inline làm cho bộ nhớ chương trình quá lớn)

Ví dụ 2 : Chương trình sau sử dụng hàm inline tính chu vi và diện tích của hình chữ nhật:

Cách 1: Không khai báo nguyên mẫu. Khi đó hàm dtcvhcn phải đặt trước hàm main.

```
#include <conio.h>
#include <iostream.h>
inline void dtcvhcn(int a, int b, int &dt, int &cv)
{
	dt=a*b;
	cv=2*(a+b);
}
void main()
{
	int a[20],b[20],cv[20],dt[20],n;
	cout << "\n So hinh chu nhat: " ;
	cin >> n;
	for (int i=1; i<=n; ++i)
	{
		cout <<"\n Nhap 2 canh cua hinh chu nhat thu " << i << ": ";
		cin >> a[i] >> b[i];
		dtcvhcn(a[i],b[i],dt[i], cv[i]);
	}
	clrscr();
	for (i=1; i<=n; ++i)
	{
		cout << "\n Hinh chu nhat thu "<< i << " : ";
```

```
            cout << "\n Do dai 2 canh= " << a[i] << " va " << b[i] ;
            cout <<"\n Dien tich= " << dt[i] ;
            cout << "\n Chu vi= " << cv[i] ;
        }
        getch();
    }
```

Cách 2:Sử dụng khai báo nguyên mẫu. Khi đó từ khoá inline đặt trước nguyên mẫu.

Chú ý: Không được đặt inline trước định nghĩa hàm. Trong chương trình dưới đây nếu đặt inline trước định nghĩa hàm thì hậu quả như sau: Chương trình vẫn dịch thông, nhưng khi chạy thì chương trình bị quẩn và không thoát đi được.

```
#include <conio.h>
#include <iostream.h>
inline void dtcvhcn(int a, int b, int &dt, int &cv);
void main()
{
    int a[20],b[20],cv[20],dt[20],n;
    cout << "\n So hinh chu nhat: " ;
    cin >> n;
    for (int i=1; i<=n; ++i)
    {
        cout <<"\n Nhap 2 canh cua hinh chu nhat thu " << i << ": ";
        cin >> a[i] >> b[i];
        dtcvhcn(a[i],b[i],dt[i], cv[i]);
    }
    clrscr();
    for (i=1; i<=n; ++i)
    {
        cout << "\n Hinh chu nhat thu "<< i << " : ";
        cout << "\n Do dai 2 canh= " << a[i] << " va " << b[i] ;
        cout <<"\n Dien tich= " << dt[i] ;
        cout << "\n Chu vi= " << cv[i] ;
    }
    getch();
```

```
}
void dtcvhcn(int a, int b, int&dt, int &cv)
{
	dt=a*b;
	cv=2*(a+b);
}
```

CHƯƠNG 8

HÀM BẠN, ĐỊNH NGHĨA PHÉP TOÁN CHO LỚP

Hàm bạn
Định nghĩa phép toán cho lớp

I. HÀM BẠN (FRIEND FUNCTION)

1. Hàm bạn

Để một hàm trở thành bạn của một lớp, có 2 cách viết:

Cách 1: Dùng từ khóa friend để khai báo hàm trong lớp và xây dựng hàm bên ngoài như các hàm thông thường (không dùng từ khóa friend). Mẫu viết như sau:

```
class A
{
private:
        // Khai báo các thuộc tính
public:
        ...
        // Khai báo các hàm bạn của lớp A
        friend void f1(...);
        friend double f2(...);
        friend A f3(...) ;
        ...
} ;
// Xây dựng các hàm f1, f2, f3
void f1(...)
{
        ...
}
double f2(...)
{
```

```
        ...
}
A f3(...)
{
        ...
}
```

Cách 2: Dùng từ khóa friend để xây dựng hàm trong định nghĩa lớp. Mẫu viết như sau:

```
class A
{
private:
        // Khai báo các thuộc tính
public:
        // Xây dựng các hàm bạn của lớp A
        void f1(...)
        {
                ...
        }
        double f2(...)
        {
                ...
        }
        A f3(...)
        {
                ...
        }
        ...
} ;
```

2. Tính chất của hàm bạn

Trong thân hàm bạn của một lớp có thể truy nhập tới các thuộc tính của các đối tượng thuộc lớp này. Đây là sự khác nhau duy nhất giữa hàm bạn và hàm thông thường.

Chú ý rằng hàm bạn không phải là phương thức của lớp. Phương thức có một

đối ẩn (ứng với con trỏ this) và lời gọi của phương thức phải gắn với một đối tượng nào đó (địa chỉ đối tượng này được truyền cho con trỏ this). Lời gọi của hàm bạn giống như lời gọi của hàm thông thường.

Ví dụ sau sẽ so sánh phương thức, hàm bạn và hàm thông thường.

Xét lớp SP (số phức), hãy so sánh 3 phương án để thực hiện việc cộng 2 số phức:

Phương án 1: Dùng phương thức

```
class SP
{
private:
    double a; // phần thực
    double b; // Phần ảo
public:
    SP cong(SP u2)
    {
        SP u:
        u.a = this → a + u2.a ;
        u.b = this → b + u2.b ;
        return u;
    }
};
```

Cách dùng:

```
SP u, u1, u2;
u = u1.cong(u2);
```

Phương án 2: Dùng hàm bạn

```
class SP
{
private:
    double a; // Phần thực
    double b; // Phần ảo
public:
    friend SP cong(SP u1 , SP u2)
```

```
    {
        SP u:
        u.a = u1.a + u2.a ;
        u.b = u1.b + u2.b ;
        return u;
    }
};
```

Cách dùng

```
SP u, u1, u2;
u = cong(u1, u2);
```

Phương án 3: Dùng hàm thông thường

```
class SP
{
private:
    double a; // phần thực
    double b; // Phần ảo
public:
    ...
};

SP cong(SP u1, SP u2)
{
    SP u:
    u.a = u1.a + u2.a ;
    u.b = u1.b + u2.b ;
    return u;
}
```

Phương án này không được chấp nhận, trình biên dịch sẽ báo lỗi trong thân hàm không được quyền truy xuất đến các thuộc tính riêng (private) a, b của các đối tượng u, u1 và u2 thuộc lớp SP.

3. Hàm bạn của nhiều lớp

Khi một hàm là bạn của nhiều lớp, thì nó có quyền truy nhập tới tất cả các thuộc tính của các đối tượng trong các lớp này.

Để làm cho hàm f trở thành bạn của các lớp A, B và C ta sử dụng mẫu viết như sau:

```
class A;          // Khai báo trước lớp A
class B;          // Khai báo trước lớp B
class C;          // Khai báo trước lớp C
// Định nghĩa lớp A
class A
{
      // Khai báo f là bạn của A
      friend void f(...) ;
} ;
// Định nghĩa lớp B
class B
{
      // Khai báo f là bạn của B
      friend void f(...) ;
} ;
// Định nghĩa lớp C
class C
{
      // Khai báo f là bạn của C
      friend void f(...) ;
} ;
// Xây dựng hàm f
void f(...)
{
      ...
}
```

Chương trình sau đây minh họa cách dùng hàm bạn (bạn của một lớp và bạn của nhiều lớp). Chương trình đưa vào 2 lớp VT (véc tơ), MT (ma trận) và 3 hàm bạn để thực hiện các thao tác trên 2 lớp này:

```
// Hàm bạn với lớp VT dùng để in một véc tơ
      friend void in(const VT &x);
// Hàm bạn với lớp MT dùng để in một ma trận
      friend void in(const MT &a);
```

```
// Hàm bạn với cả 2 lớp MT và VT dùng để nhân ma trận với véc tơ
    friend VT tich(const MT &a, const VT &x);
```

Nội dung chương trình là nhập một ma trận vuông cấp n và một véc tơ cấp n, sau đó thực hiện phép nhân ma trận với véc tơ vừa nhập.

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
class VT;
class MT;
class VT
{
private:
    int n;
    double x[20]; // Toa do cua diem
public:
    void nhapsl();
    friend void in(const VT &x);
    friend VT tich(const MT &a, const VT &x) ;
};

class MT
{
private:
    int n;
    double a[20][20];
public:
    friend VT tich(const MT &a, const VT &x);
    friend void in(const MT &a);
    void nhapsl();
} ;

void VT::nhapsl()
{
    cout << "\n Cap vec to = ";
```

```
        cin >> n ;
        for (int i = 1; i< = n ; ++i)
        {
                cout << "\n Phan tu thu " << i <<" = " ;
                cin >> x[i];
        }
}

void MT::nhapsl()
{
        cout <<"\n Cap ma tran = ";
        cin >> n ;
        for (int i = 1; i< = n ; ++i)
        for (int j = 1; j< = n; ++j)
        {
                cout << "\n Phan tu thu: "<<i<< " hang "<< i << " cot " << j << " = ";
                cin >> a[i][j];
        }
}

VT tich(const MT &a, const VT &x)
{
        VT y;
        int n = a.n;
        if (n! = x.n)
        return x;
        y.n = n;
        for (int i = 1; i< = n; ++i)
        {
                y.x[i] = 0;
                for (int j = 1; j< = n; ++j)
                y.x[i] = a.a[i][j]*x.x[j];
        }
        return y;
```

```
}
void in(const VT &x)
{
	cout << "\n";
	for (int i = 1; i< = x.n; ++i)
	cout << x.x[i] << " ";
}
void in(const MT &a)
{
	for (int i = 1; i< = a.n; ++i)
	{
		cout << "\n " ;
		for (int j = 1; j< = a.n; ++j)
		cout << a.a[i][j] << " ";
	}
}
void main()
{
	MT a; VT x, y;
	clrscr();
	a.nhapsl();
	x.nhapsl();
	y = tich(a, x);
	clrscr();
	cout << "\n Ma tran A:";
	in(a);
	cout << "\n Vec to x: " ;
	in(x);
	cout << "\n Vec to y = Ax: " ;
	in(y);
	getch();
}
```

II. ĐỊNH NGHĨA PHÉP TOÁN CHO LỚP

Đối với mỗi lớp ta có thể sử dụng lại các kí hiệu phép toán thông dụng (+, -, *, …) để định nghĩa cho các phép toán của lớp. Sau khi được định nghĩa các kí hiệu này sẽ được dùng như các phép toán của lớp theo cách viết thông thường. Cách định nghĩa này được gọi là phép chồng toán tử (như khái niệm chồng hàm trong các chương trước).

1. Tên hàm toán tử

Gồm từ khoá operator và tên phép toán.

Ví dụ:

operator+(định nghĩa chồng phép +)

operator- (định nghĩa chồng phép -)

2. Các đối của hàm toán tử

– Với các phép toán có 2 toán hạng thì hàm toán tử cần có 2 đối. Đối thứ nhất ứng với toán hạng thứ nhất, đối thứ hai ứng với toán hạng thứ hai. Do vậy, với các phép toán không giao hoán (phép -) thì thứ tự đối là rất quan trọng.

Ví dụ: Các hàm toán tử cộng, trừ phân số được khai báo như sau:

```
struct PS
{
    int a;      //Tử số
    int b;      // Mẫu số
};
PS operator+(PS p1, PS p2);          // p1 + p2
PS operator-(PS p1 , PS p2);         // p1 - p2
PS operator*(PS p1, PS p2);          // p1 *p2
PS operator/(PS p1, PS p2);          // p1/p2
```

– Với các phép toán có một toán hạng, thì hàm toán tử có một đối. Ví dụ hàm toán tử đổi dấu ma trận (đổi dấu tất cả các phần tử của ma trận) được khai báo như sau:

```
struct MT
{
    double a[20][20] ;    // Mảng chứa các phần tử ma trận
    int m ;               // Số hàng ma trận
```

```
    int n ;                    // Số cột ma trận
};
MT operator-(MT x) ;
```

3. Thân của hàm toán tử

Viết như thân của hàm thông thường. Ví dụ hàm đổi dấu ma trận có thể được định nghĩa như sau:

```
struct MT
{
    double a[20][20] ;         // Mảng chứa các phần tử ma trận
    int m ;                    // Số hàng ma trận
    int n ;                    // Số cột ma trận
};
MT operator-(MT x)
{
    MT y;
    for (int i=1 ;i<= y.m ; ++i)
    for (int j =1 ;j<= y.n ; ++j)y.a[i][j] =- x.a[i][j];
    return y;
}
```

a. Cách dùng hàm toán tử

Có 2 cách dùng:

Cách 1: Dùng như một hàm thông thường bằng cách viết lời gọi

Ví dụ:

```
PS p, q, u, v ;
u = operator+(p, q) ;          // u = p + q
v = operator-(p, q) ;          // v= p - q
```

Cách 2: Dùng như phép toán của C++

Ví dụ:

```
PS p, q, u, v ;
u = p + q ;                    // u = p + q
v = p - q ;                    //v = p - q
```

Chú ý: Khi dùng các hàm toán tử như phép toán của C++ ta có thể kết hợp nhiều

phép toán để viết các công thức phức tạp. Cũng cho phép dùng dấu ngoặc tròn để quy định thứ tự thực hiện các phép tính. Thứ tự ưu tiên của các phép tính vẫn tuân theo các quy tắc ban đầu của C++. Chẳng hạn các phép * và / có thứ tự ưu tiên cao hơn so với các phép + và -

b. Các ví dụ về định nghĩa chồng toán tử

Ví dụ 1 : Trong ví dụ này ngoài việc sử dụng các hàm toán tử để thực hiện 4 phép tính trên phân số, còn định nghĩa chồng các phép toán << và >> để xuất và nhập phân số.

Hàm operator<< có 2 đối kiểu ostream& và PS (Phân số). Hàm trả về giá trị kiểu ostream& và được khai báo như sau:

```
ostream& operator<< (ostream& os, PS p);
```

Tượng tự hàm operator>> được khai báo như sau:

```
istream& operator>> (istream& is,PS &p);
```

Dưới đây sẽ chỉ ra cách xây dựng và sử dụng các hàm toán tử.

Chúng ta cũng sẽ thấy việc sử dụng các hàm toán tử rất tự nhiên, ngắn gọn và tiện lợi.

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
typedef struct
{
      int a,b;
} PS;
ostream& operator<< (ostream& os, PS p);
istream& operator>> (istream& is,PS &p);
int uscln(int x, int y);
PS rutgon(PS p);
PS operator+(PS p1, PS p2);
PS operator-(PS p1, PS p2);
PS operator*(PS p1, PS p2);
PS operator/(PS p1, PS p2);
ostream& operator<< (ostream& os, PS p)
{
      os << p.a << '/' << p.b ;
```

```
        return os;
}
istream& operator>> (istream& is,PS &p)
{
        cout << "\n Nhap tu va mau: " ;
        is >> p.a >> p.b ;
        return is;
}
int uscln(int x, int y)
{
        x=abs(x);y=abs(y);
        if (x*y==0) return 1;
        while (x!=y)
        {
                if (x>y) x-=y;
                else y-=x;
        }
        return x;
}
PS rutgon(PS p)
{
        PS q;
        int x;
        x=uscln(p.a,p.b);
        q.a = p.a / x ;
        q.b = p.b/ x ;
        return q;
}
PS operator+(PS p1, PS p2)
{
        PS q;
        q.a = p1.a*p2.b + p2.a*p1.b;
        q.b = p1 .b * p2.b ;
```

```
        return rutgon(q);
}
PS operator-(PS p1, PS p2)
{
        PS q;
        q.a = p1.a*p2.b - p2.a*p1 .b;
        q.b = p1.b * p2.b ;
        return rutgon(q);
}
PS operator*(PS p1, PS p2)
{
        PS q;
        q.a = p1.a * p2.a ;
        q.b = p1.b * p2.b ;
        return rutgon(q);
}
PS operator/(PS p1 , PS p2)
{
        PS q;
        q.a = p1.a * p2.b ;
        q.b = p1.b * p2.a ;
        return rutgon(q);
}
void main()
{
        PS p, q, z, u, v ;
        PS s;
        cout <<"\nNhap cac PS p, q, z, u, v: " ;
        cin >> p >> q >> z >> u >> v ;
        s = (p - q*z) / (u + v) ;
        cout << "\n Phan so s = " << s;
        getch();
}
```

Ví dụ 2 : Chương trình đưa vào các hàm toán tử:

operator- có một đối dùng để đảo dấu một đa thức

operator+ có 2 đối dùng để cộng 2 đa thức

operator- có 2 đối dùng để trừ 2 đa thức

operator* có 2 đối dùng để nhân 2 đa thức

operator^có 2 đối dùng để tính giá đa thức tại x

operator<< có 2 đối dùng để in đa thức

operator>> có 2 đối dùng để nhập đa thức

Chương trình sẽ nhập 4 đa thức: p, q, r, s. Sau đó tính đa thức: f = -(p+q)*(r-s)
Cuối cùng tính giá trị f(x), với x là một số thực nhập từ bàn phím.

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
struct DT
{
	double a[20];// Mang chua cac he so da thuc a0, a1,...
	int n ;// Bac da thuc
};
ostream& operator<< (ostream& os, DT d);
istream& operator>> (istream& is, DT &d);
DT operator-(const DT& d);
DT operator+(DT d1, DT d2);
DT operator-(DT d1, DT d2);
DT operator*(DT d1, DT d2);
double operator^(DT d, double x);// Tinh gia tri da thuc
ostream& operator<< (ostream& os, DT d)
{
	os << " Cac he so (tu ao): " ;
	for (int i=0 ;i<= d.n ;++i)
	os << d.a[i] <<" " ;
```

```
        return os;
}
istream& operator>> (istream& is, DT &d)
{
        cout << " Bac da thuc: " ;
        cin >> d.n;
        cout << "Nhap cac he so da thuc:" ;
        for (int i=0 ;i<=d.n ;++i)
        {
                cout << "\n He so bac " << i <<" = " ;
                is >> d.a[i] ;
        }
        return is;
}
DT operator-(const DT& d)
{
        DT p;
        p.n = d.n;
        for (int i=0 ;i<=d.n ;++i)
        p.a[i] = -d.a[i];
        return p;
}
DT operator+(DT d1, DT d2)
{
        DT d;
        int k,i;
        k = d1.n > d2.n ? d1.n : d2.n ;
        for (i=0;i<=k ;++i)
                if (i<=d1.n && i<=d2.n) d.a[i] = d1.a[i] + d2.a[i];
                else  if (i<=d1.n) d.a[i] = d1.a[i];
                        else d.a[i] = d2.a[i];
        i = k;
        while (i>0 && d.a[i]==0.0) --i;
```

```
        d.n=i;
        return d ;
}
DT operator-(DT d1, DT d2)
{
        return (d1 + (-d2));
}
DT operator*(DT d1 , DT d2)
{
        DT d;
        int k, i, j;
        k = d.n = d1.n + d2.n ;
        for (i=0;i<=k;++i) d.a[i] = 0;
        for (i=0 ;i<= d1 .n ;++i)
        for (j=0 ;j<= d2.n ;++j)
        d.a[i+j] += d1 .a[i]*d2.a[j];
        return d;
}
double operator^(DT d, double x)
{
        double s=0.0 , t=1.0;
        for (int i=0 ;i<= d.n ;++i)
        {
                s += d.a[i]*t;
                t *= x;
        }
        return s;
}
void main()
{
        DT p,q,r,s,f;
        double x,g;
        clrscr();
```

```
        cout <<"\n Nhap da thuc P " ;cin >> p;
        cout <<"\n Nhap da thuc Q " ;cin >> q;
        cout <<"\n Nhap da thuc R " ;cin >> r;
        cout <<"\n Nhap da thuc S " ;cin >> s;
        cout << "\n Nhap so thuc x: " ;cin >> x;
        f = -(p+q)*(r-s);
        g = f^x;
        cout << "\n Da thuc f "<< f ;
        cout << "\n x = " << x;
        cout << "\n f(x) = " << g;
        getch();
}
```