CHƯƠNG 3

ĐỐI TƯỢNG VÀ LỚP
(Object & Class)

Mục đích chương này:

1. Khái niệm về đóng gói dữ liệu.
2. Khai báo và sử dụng một lớp.
3. Khai báo và sử dụng đối tượng, con trỏ đối tượng, tham chiếu đối tượng.
4. Hàm thiết lập và hàm huỷ bỏ.
5. Khai báo và sử dụng hàm thiết lập sao chép.
6. Vai trò của hàm thiết lập ngầm định.

1. ĐỐI TƯỢNG

Đối tượng là một khái niệm trong lập trình hướng đối tượng biểu thị sự liên kết giữa dữ liệu và các thủ tục (gọi là các phương thức) thao tác trên dữ liệu đó. Ta có công thức sau:

ĐỐI TƯỢNG = DỮ LIỆU+PHƯƠNG THỨC

Ở đây chúng ta hiểu rằng đối tượng chính là công cụ hỗ trợ cho sự đóng gói. Sự đóng gói là cơ chế liên kết các lệnh thao tác và dữ liệu có liên quan, giúp cho cả hai được an toàn tránh được sự can thiệp từ bên ngoài và việc sử dụng sai. Nhìn chung định nghĩa một đối tượng phức tạp hơn so với định nghĩa các biến cấu trúc thông thường, bởi lẽ ngoài việc mô tả các thành phần dữ liệu, ta còn phải xác định được các thao tác tác động lên đối tượng đó. Hình 2.1 mô tả các đối tượng điểm trên mặt phẳng:

Mỗi đối tượng được xác định bởi hai thành phần toạ độ được biểu diễn bởi hai biến nguyên. Các thao tác tác động lên điểm bao gồm việc xác định toạ độ một điểm trên mặt phẳng toạ độ (thể hiện bằng việc gán giá trị cho hai thành phần toạ độ), thay đổi toạ độ và hiển thị kết quả lên trên mặt phẳng toạ độ (tương tự như việc chấm điểm trên mặt phẳng đó).

Lợi ích của việc đóng gói là khi nhìn từ bên ngoài, một đối tượng chỉ được biết tới bởi các mô tả về các phương thức của nó, cách thức cài đặt các

dữ liệu không quan trọng đối với người sử dụng. Với một đối tượng điểm, người ta chỉ quan tâm đến việc có thể thực hiện được thao tác gì trên nó mà không cần biết các thao tác đó được thực hiện như thế nào, cũng như điều gì xảy ra bên trong bản thân đối tượng đó. Ta thường nói đó là "sự trừu tượng hoá dữ liệu" (khi các chi tiết cài đặt cụ thể được giấu đi).

```
Mô tả đối tượng điểm {
//dữ liệu
int x,y;
//phương thức
void init(int ox,int oy);
void move(int dx,int dy);
void display();
};
```

HÌNH 3.1 MÔ TẢ CÁC ĐỐI TƯỢNG ĐIỂM

Đóng gói có nhiều lợi ích góp phần nâng cao chất lượng của chương trình. Nó làm cho công việc bảo trì chương trình thuận lợi hơn rất nhiều: một sự thay đổi cấu trúc của một đối tượng chỉ ảnh hưởng tới bản thân đối tượng; người sử dụng đối tượng không cần biết đến thay đổi này (với lập trình cấu trúc thì người lập trình phải tự quản lý sự thay đổi đó). Chẳng hạn có thể biểu diễn toạ độ một điểm dưới dạng số thực, khi đó chỉ có người thiết kế đối tượng phải quan tâm để sửa lại định nghĩa của đối tượng trong khi đó người sử dụng không cần hay biết về điều đó, miễn là những thay đổi đó không tác động đến việc sử dụng đối tượng điểm.

Tương tự như vậy, ta có thể bổ sung thêm thuộc tính màu và một số thao tác lên một đối tượng điểm, để có được một đối tượng điểm màu. Rõ ràng là đóng gói cho phép đơn giản hoá việc sử dụng một đối tượng.

Trong lập trình hướng đối tượng, đóng gói cho phép dữ liệu của đối tượng được che lấp khi nhìn từ bên ngoài, nghĩa là nếu người dùng muốn tác động lên dữ liệu của đối tượng thì phải gửi đến đối tượng các thông điệp(message). Ở đây các phương thức đóng vai trò là giao diện bắt buộc giữa các đối tượng và người sử dụng. Ta có nhận xét: "*Lời gọi đến một phương thức là truyền một thông báo đến cho đối tượng*".

Các thông điệp gửi tới đối tượng nào sẽ gắn chặt với đối tượng đó và chỉ đối tượng nào nhận được thông điệp mới phải thực hiện theo thông điệp đó; chẳng hạn các đối tượng điểm độc lập với nhau, vì vậy thông điệp thay đổi toạ độ đối tượng điểm p chỉ làm ảnh hưởng đến các thành phần toạ độ trong p chứ không thể thay đổi được nội dung của một đối tượng điểm q khác.

So với lập trình hướng đối tượng thuần tuý, các cài đặt cụ thể của đối tượng trong C++ linh động hơn một chút, bằng cách cho phép chỉ che dấu một bộ phận dữ liệu của đối tượng và mở rộng hơn khả năng truy nhập đến các thành phần riêng của đối tượng. Khái niệm lớp chính là cơ sở cho các linh động này.

Lớp là một mô tả trừu tượng của nhóm các đối tượng có cùng bản chất. Trong một lớp người ta đưa ra các mô tả về tính chất của các thành phần dữ liệu, cách thức thao tác trên các thành phần này(hành vi của các đối tượng), ngược lại mỗi một đối tượng là một thể hiện cụ thể cho những mô tả trừu tượng đó. Trong các ngôn ngữ lập trình, lớp đóng vai trò một kiểu dữ liệu được người dùng định nghĩa và việc tạo ra một đối tượng được ví như khai báo một biến có kiểu lớp.

2. LỚP

2.1 Khai báo lớp

Từ quan điểm của lập trình cấu trúc, lớp là một kiểu dữ liệu tự định nghĩa. Trong lập trình hướng đối tượng, chương trình nguồn được phân bố trong khai báo và định nghiã của các lớp.

Sau đây là một ví dụ điển hình về cú pháp khai báo lớp. Kinh nghiệm cho thấy mọi kiểu khai báo khác đều có thể chuẩn hoá để đưa về dạng này.

```
class <TÊN LỚP> {
  private:
   <KHAI BÁO CÁC THÀNH PHẦN RIÊNG TRONG TỪNG ĐỐI TƯỢNG>
  public:
    <KHAI BÁO CÁC THÀNH PHẦN CÔNG CỘNG CỦA TỪNG ĐỐI
TƯỢNG>
  };
<ĐỊNH NGHI· CỦA CÁC HÀM THÀNH PHẦN CHƯA ĐƯỢC ĐỊNH
NGHĨA BÊN TRONG KHAI BÁO LỚP>
...
```

Các chi tiết liên quan đến khai báo lớp sẽ lần lượt được đề cập đến trong các phần sau. Để dễ hình dung xét một ví dụ về khai báo lớp điểm trong mặt phẳng. Trong ví dụ này ta có đề cập đến một vài khía cạnh liên quan đến khai báo lớp, đối tượng và sử dụng chúng.

Ví dụ 3.1

```
/*point.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
  /*khai báo các thành phần dữ liệu riêng*/
  private:
    int x,y;
  /*khai báo các hàm thành phần công cộng*/
  public:
    void init(int ox, int oy);
    void move(int dx, int dy);
    void display();
    };
/*định nghĩa các hàm thành phần bên ngoài khai báo lớp*/
void point::init(int ox, int oy) {
  cout<<"Ham thanh phan init\n";
  x = ox; y = oy; /*x,y là các thành phần của đối tượng gọi hàm thành
phần*/
  }
void point::move(int dx, int dy) {
  cout<<"Ham thanh phan move\n";
  x += dx; y += dy;
  }
void point::display() {
  cout<<"Ham thanh phan display\n";
  cout<<"Toa do: "<<x<<" "<<y<<"\n";
  }
```

```
void main() {
  clrscr();
  point p;
  p.init(2,4); /*gọi hàm thành phần từ đối tượng*/
  p.display();
  p.move(1,2);
  p.display();
  getch();
  }
```

```
Ham thanh phan init
Ham thanh phan display
Toa do: 2 4
Ham thanh phan move
Ham thanh phan display
Toa do: 3 6
```

Nhận xét

7. Có thể khai báo trực tiếp các hàm thành phần bên trong khai báo lớp. Tuy vậy điều đó đôi khi làm mất mỹ quan của chương trình nguồn, do vậy người ta thường sử dụng cách khai báo các hàm thành phần ở bên ngoài khai báo lớp. Khi đó ta sử dụng cú pháp:

```
<tên kiểu giá trị trả lại> <tên lớp>::<tên hàm> (<danh sách tham số>) {
  <nội dung >
  }
```

8. Gọi hàm thành phần của lớp từ một đối tượng chính là truyền thông điệp cho hàm thành phần đó. Cú pháp như sau:

```
<TÊN ĐỐI TƯỢNG>.<TÊN HÀM THÀNH PHẦN>(<DANH SÁCH CÁC THAM SỐ NẾU CÓ>);
```

2.1.1 Tạo đối tượng

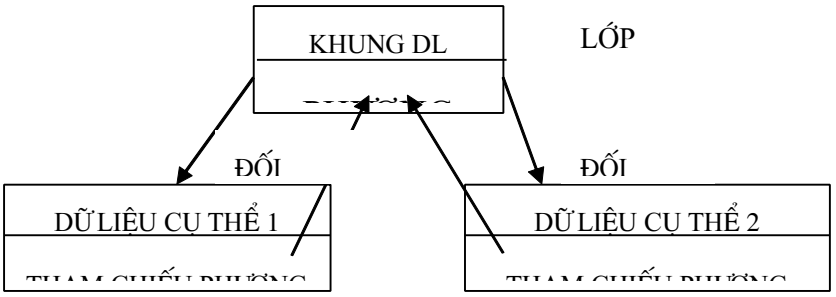
Trong C++, một đối tượng có thể được xác lập thông quan một biến/hằng có kiểu lớp.

```
<tên lớp>  <tên đối tượng>;
```

Do đó vùng nhớ được cấp phát cho một biến kiểu lớp sẽ cho ta một khung của đối tượng bao gồm dữ liệu là các thể hiện cụ thể của các mô tả dữ liệu trong khai báo lớp cùng với các thông điệp gửi tới các hàm thành phần.

Mỗi đối tượng sở hữu một tập các biến tương ứng với tên và kiểu của các thành phần dữ liệu định nghĩa trong lớp. Ta gọi chúng là các biến thể hiện của đối tượng. Tuy nhiên tất cả các đối tượng cùng một lớp chung nhau định nghĩa của các hàm thành phần.

Lớp là một kiểu dữ liệu vì vậy có thể khai báo con trỏ hay tham chiếu đến một đối tượng thuộc lớp và bằng cách ấy có thể truy nhập gián tiếp đến đối tượng. Nhưng chú ý là con trỏ và tham chiếu không phải là một thể hiện của lớp.



HÌNH 3.2 ĐỐI TƯỢNG LÀ MỘT THỂ

2.1.2 Các thành phần dữ liệu

Cú pháp khai báo các thành phần dữ liệu giống như khai báo biến:

```
<TÊN KIỂU> <TÊN THÀNH PHẦN>;
```

Một thành phần dữ liệu có thể là một biến kiểu cơ sở (**int**, **float**, **double**, **char, char***), kiểu trường bit, kiểu liệt kê (enum) hay các kiểu do người dùng định nghĩa. Thậm chí, thành phần dữ liệu còn có thể là một đối tượng thuộc lớp đã được khai báo trước đó. Tuy nhiên không thể dùng trực tiếp các lớp để khai báo kiểu thành phần dữ liệu thuộc vào bản thân lớp đang được định nghĩa. Muốn vậy, trong khai báo của một lớp có thể dùng các con trỏ hoặc tham chiếu đến các đối tượng của chính lớp đó.

Trong khai báo của các thành phần dữ liệu, có thể sử dụng từ khoá **static** nhưng không được sử dụng các từ khoá **auto**, **register**, **extern** trong khai báo các thành phần dữ liệu. Cũng không thể khai báo và khởi đầu giá trị cho các thành phần đó.

2.1.3 Các hàm thành phần

Hàm được khai báo trong định nghĩa của lớp được gọi là hàm thành phần hay phương thức của lớp (hàm thành phần là thuật ngữ của C++, còn phương thức là thuật ngữ trong lập trình hướng đối tượng nói chung). Các hàm thành phần có thể truy nhập đến các thành phần dữ liệu và các hàm thành phần khác trong lớp. Như trên đã nói, C++ cho phép hàm thành phần truy nhập tới các thành phần của các đối tượng cùng lớp, miễn là chúng được khai báo bên trong định nghĩa hàm (như là một đối tượng cục bộ hay một tham số hình thức của hàm thành phần). Phần tiếp sau sẽ có các ví dụ minh hoạ cho khả năng này.

Trong chương trình `point.cpp`, trong khai báo của lớp `point` có chứa các khai báo các hàm thành phần của lớp. Các khai báo này cũng tuân theo cú pháp khai báo cho các hàm bình thường. Định nghĩa của các hàm thì có thể đặt ở bên trong hay bên ngoài khai báo lớp; Khi định nghĩa hàm thành phần đặt trong khai báo lớp (nếu hàm thành phần đơn giản, không chứa các cấu trúc lặp [1]) không có gì khác so với định nghĩa của hàm thông thường. Chương trình `point1.cpp` sau đây là một cách viết khác của `point.cpp` trong đó hàm thành phần `init()` được định nghĩa ngay bên trong khai báo lớp.

Ví dụ 3.2

```
/*point1.cpp*/
```

[1] Hàm thành phần định nghĩa trong khai báo lớp được chương trình dịch hiểu là hàm **inline,** nên không được quá phức tạp.

```
#include <iostream.h>
#include <conio.h>
class point {
    /*khai báo các thành phần dữ liệu private*/
  private:
    int x,y;
    /*khai báo các hàm thành phần public*/
  public:
    /*Định nghĩa hàm thành phần bên trong khai báo lớp*/
    void init(int ox, int oy){
      cout<<"Ham thanh phan init\n";
      x = ox; y = oy; /*x,y là các thành phần của đối tượng gọi hàm
thành phần*/
      }
    void move(int dx, int dy);
    void display();
    };
/*định nghĩa các hàm thành phần bên ngoài khai báo lớp*/
void point::move(int dx, int dy) {
  cout<<"Ham thanh phan move\n";
  x += dx; y += dy;
  }
void point::display() {
  cout<<"Ham thanh phan display\n";
  cout<<"Toa do: "<<x<<" "<<y<<"\n";
  }
void main() {
  clrscr();
```

```
point p;
p.init(2,4); /*gọi hàm thành phần từ đối tượng*/
p.display();
p.move(1,2);
p.display();
getch();
}
```

```
Ham thanh phan init
Ham thanh phan display
Toa do: 2 4
Ham thanh phan move
Ham thanh phan display
Toa do: 3 6
```

Khi định nghĩa hàm thành phần ở ngoài lớp, dòng tiêu đề của hàm thành phần phải chứa tên của lớp có hàm là thành viên tiếp theo là toán tử định phạm vi "::". Đó là cách để phân biệt hàm thành phần với các hàm tự do, đồng thời còn cho phép hai lớp khác nhau có thể có các hàm thành phần cùng tên.

Có thể đặt định nghĩa hàm thành phần trong cùng tập tin khai báo lớp hoặc trong một tập tin khác. Ví dụ sau đây sau đây là một cải biên khác từ `point.cpp`, trong đó ta đặt riêng khai báo lớp `point` trong một tệp tiêu đề. Tệp tiêu đề sẽ được tham chiếu tới trong tệp chương trình `point2.cpp` chứa định nghĩa các hàm thành phần của lớp `point`.

Ví dụ 3.3

Tệp tiêu đề

```
/*point.h*/
/* đây là tập tin tiêu đề khai báo lớp point được gộp vào tệp point2.cpp */
#ifndef point_h
#define point_h
#include <iostream.h>
```

```
class point {
    /*khai báo các thành phần dữ liệu private*/
  private:
    int x,y;
    /*khai báo các hàm thành phần public*/
  public:
  /*Định nghĩa hàm thành phần bên trong khai báo lớp*/
    void init(int ox, int oy);
    void move(int dx, int dy);
    void display();
    };
#endif
```

Tệp chương trình nguồn

```
/*point2.cpp*/
/*Tập tin chương trình, định nghĩa và sử dụng các hàm thành phần trong lớp
point được khai báo trong tập tin tiêu đề point.h */
#include "point.h"/*chèn định nghĩa lớp point vào chương trình*/
#include <conio.h>
/*định nghĩa các hàm thành phần bên ngoài khai báo lớp*/
void point::init(int ox, int oy) {
  cout<<"Ham thanh phan init\n";
  x = ox; y = oy; /*x,y là các thành phần của đối tượng gọi hàm thành
phần*/
  }
void point::move(int dx, int dy) {
  cout<<"Ham thanh phan move\n";
  x += dx; y += dy;
  }
```

```
void point::display() {
  cout<<"Ham thanh phan display\n";
  cout<<"Toa do: "<<x<<" "<<y<<"\n";
  }
void main() {
  clrscr();
  point p;
  p.init(2,4); /*gọi hàm thành phần từ đối tượng*/
  p.display();
  p.move(1,2);
  p.display();
  getch();
  }
```

```
Ham thanh phan init
Ham thanh phan display
Toa do: 2 4
Ham thanh phan move
Ham thanh phan display
Toa do: 3 6
```

2.1.4 Tham số ngầm định trong lời gọi hàm thành phần

Ở đây không nên nhầm lẫn khái niệm này với lời gọi hàm với tham số có giá trị ngầm định. Lời gọi hàm thành phần luôn có một và chỉ một tham số ngầm định là đối tượng thực hiện lời gọi hàm. Như thế các biến x, y trong định nghĩa của các hàm point::init(), point::display(), hay point::move() chính là các biến thể hiện của đối tượng dùng làm tham số ngầm định trong lời gọi hàm. Do vậy lời gọi hàm thành phần:

```
p.init(2,4)
```

sẽ gán 2 cho p.x còn p.y sẽ có giá trị 4.

Tất nhiên, theo nguyên tắc đóng gói, không gán trị cho các thành phần dữ liệu của đối tượng một cách trực tiếp.

```
p.x = 2;
p.y = 4;
```

Hơn nữa, không thể thực hiện lời gọi tới hàm thành phần nếu không chỉ rõ đối tượng được tham chiếu. Chỉ thị:

```
init(5,2);
```

trong hàm `main` sẽ có thể gây lỗi biên dịch nếu trong chương trình không có hàm tự do với tên `init`.

2.1.5 Phạm vi lớp

Phạm vi chỉ ra phần chương trình trong đó có thể truy xuất đến một đối tượng nào đó. Trong C có bốn kiểu phạm vi liên quan đến cách thức và vị trí khai báo biến: phạm vi khối lệnh, phạm vi tệp, phạm vi chương trình và phạm vi hàm nguyên mẫu, trong đó thường dùng nhất là phạm vi toàn cục (tệp, chương trình) và phạm vi cục bộ (khối lệnh, hàm). Mục đích của phạm vi là để kiểm soát việc truy xuất đến các biến/hằng/hàm.

Để kiểm soát truy nhập đến các thành phần (dữ liệu, hàm) của các lớp, C++ đưa ra khái niệm phạm vi lớp. Tất cả các thành phần của một lớp sẽ được coi là thuộc phạm vi lớp; trong định nghĩa hàm thành phần của lớp có thể tham chiếu đến bất kỳ một thành phần nào khác của cùng lớp đó. Tuân theo ý tưởng đóng gói, C++ coi tất cả các thành phần của một lớp có liên hệ với nhau. Ngoài ra, C++ còn cho phép mở rộng phạm vi lớp đến các lớp con cháu, bạn bè và họ hàng (Xem thêm chương 5 - Kế thừa và các mục tiếp sau để hiểu rõ hơn).

2.1.6 Từ khoá xác định thuộc tính truy xuất

Trong phần này ta nói tới vai trò của hai từ khoá **private** và **public** - dùng để xác định thuộc tính truy xuất của các thành phần lớp.

Trong định nghĩa của lớp ta có thể xác định khả năng truy xuất thành phần của một lớp nào đó từ bên ngoài phạm vi lớp. Trong lớp `point` có hai thành phần dữ liệu và ba thành phần hàm. Các thành phần dữ liệu được khai báo với nhãn là **private,** còn các hàm thành với nhãn **public**. **private** và **public** là các từ khoá xác định thuộc tính truy xuất. Mọi thành phần được liệt kê trong phần **public** đều có thể truy xuất trong bất kỳ hàm nào. Những

thành phần được liệt kê trong phần **private** chỉ được truy xuất bên trong phạm vi lớp, bởi chúng thuộc sở hữu riêng của lớp, trong khi đó các thành phần **public** thuộc sở hữu chung của mọi thành phần trong chương trình.

Với khai báo lớp `point` ta thấy rằng các thành phần **private** được tính từ chỗ nó xuất hiện cho đến trước nhãn **public**. Trong lớp có thể có nhiều nhãn **private** và **public**. Mỗi nhãn này có phạm vi ảnh hưởng cho đến khi gặp một nhãn kế tiếp hoặc hết khai báo lớp. Xem chương trình `tamgiac.cpp` sau đây:

Ví dụ 3.4

```
/*tamgiac.cpp*/
#include <iostream.h>
#include <math.h>
#include <conio.h>
/*khai báo lớp tam giác*/
class tamgiac{
  private:
     float a,b,c;/*độ dài ba cạnh*/
  public:
     void nhap();/*nhập vào độ dài ba cạnh*/
     void in();/*in ra các thông tin liên quan đến tam giác*/
  private:
     int loaitg();/*cho biết kiểu của tam giác: 1-d,2-vc,3-c,4-v,5-t*/
     float dientich();/*tính diện tích của tam giác*/
  };
/*định nghĩa hàm thành phần*/
void tamgiac::nhap() {
 /*nhập vào ba cạnh của tam giác, có kiểm tra điều kiện*/
  do {
     cout<<"Canh a : "; cin>>a;
     cout<<"Canh b : "; cin>>b;
     cout<<"Canh c : "; cin>>c;
     }while(a+b<=c||b+c<=a||c+a<=b);
  }
void tamgiac::in() {
  cout<<"Do dai ba canh :"<<a<<" "<<b<<" "<<c<<"\n";
 /* gọi hàm thành phần bên trong một hàm thành phần khác cùng lớp */
```

```
  cout<<"Dien tich tam giac : "<<dientich()<<"\n";
  switch(loaitg()) {
    case 1: cout<<"Tam giac deu\n";break;
    case 2: cout<<"Tam giac vuong can\n";break;
    case 3: cout<<"Tam giac can\n";break;
    case 4: cout<<"Tam giac vuong\n";break;
    default:cout<<"Tam giac thuong\n";break;
    }
  }
float tamgiac::dientich() {
  return (0.25*sqrt((a+b+c)*(a+b-c)*(a-b+c)*(-a+b+c)));
  }
int tamgiac::loaitg() {
  if (a==b||b==c||c==a)
    if (a==b && b==c)
      return 1;
    else if (a*a==b*b+c*c||b*b==a*a+c*c||c*c==a*a+b*b)
          return 2;
       else return 3;
  else if (a*a==b*b+c*c||b*b==a*a+c*c||c*c==a*a+b*b)
       return 4;
       else return 5;
  }
void main() {
  clrscr();
  tamgiac tg;
  tg.nhap();
  tg.in();
```

```
  getch();
  }
```

```
Canh a : 3
Canh b : 3
Canh c : 3
Do dai ba canh :3 3 3
Dien tich tam giac : 3.897114
Tam giac deu
Canh a : 3
Canh b : 4
Canh c : 5
Do dai ba canh :3 4 5
Dien tich tam giac : 6
Tam giac vuong
```

Các thành phần trong một lớp có thể được sắp xếp một cách hết sức tuỳ ý. Do đó có thể sắp xếp lại các khai báo hàm thành phần để cho các thành phần **private** ở trên, còn các thành phần **public** ở dưới trong khai báo lớp. Chẳng hạn có thể đưa ra một khai báo khác cho lớp `tamgiac` trong `tamgiac.cpp` như sau:

```
class tamgiac{
  private:
     float a,b,c;/*độ dài ba cạnh*/
     int loaitg();/*cho biết kiểu của tam giác: 1-d,2-vc,3-c,4-v,5-t*/
     float dientich();/*tính diện tích của tam giác*/
  public:
     void nhap();/*nhập vào độ dài ba cạnh*/
     void in();/*in ra các thông tin liên quan đến tam giác*/
};
```

Ngoài ra, còn có thể bỏ nhãn **private** đi vì C++ ngầm hiểu rằng các thành phần trước nhãn **public** đầu tiên là **private** (ở đây chúng ta tạm thời chưa bàn đến từ khoá **protected**). Tóm lại, khai báo "súc tích" nhất cho lớp tam giác như sau:

```
class tamgiac {
    float a,b,c;/*độ dài ba cạnh*/
    int loaitg();/*cho biết kiểu của tam giác: 1-d,2-vc,3-c,4-v,5-t*/
    float dientich();/*tính diện tích của tam giác*/
  public:
    void nhap();/*nhập vào độ dài ba cạnh*/
    void in();/*in ra các thông tin liên quan đến tam giác*/
   };
```

2.1.7 Gọi một hàm thành phần trong một hàm thành phần khác

Khi khai báo lớp, có thể gọi hàm thành phần từ một hàm thành phần khác trong cùng lớp đó. Khi muốn gọi một hàm tự do trùng tên và danh sách tham số ta phải sử dụng toán tử phạm vi "::". Bạn đọc có thể kiểm nghiệm điều này bằng cách định nghĩa một hàm tự do tên `loaitg` và gọi nó trong định nghĩa của hàm `tamgiac::in()`.

Nhận xét

9. Nếu tất cả các thành phần của một lớp là **public**, lớp sẽ hoàn toàn tương đương với một cấu trúc, không có phạm vi lớp. C++ cũng cho phép khai báo các cấu trúc với các hàm thành phần. Hai khai báo sau là tương đương nhau:

```
struct point {
 int x, y;
 void init(int, int);
 void move(int, int);
 void display();
 };
```

```
class point {
public:
  int x, y;
  void init(int, int);
  void move(int, int);
  void display();
  };
```

10. Ngoài **public** và **private**, còn có từ khoá **protected** (được bảo vệ) dùng để chỉ định trạng thái của các thành phần trong một lớp. Trong phạm vi của lớp hiện tại một thành phần **protected** có tính chất giống như thành phần **private**.

2.2 Khả năng của các hàm thành phần

2.2.1 Định nghĩa chồng các hàm thành phần.

Các hàm thành phần có thể có trùng tên nhưng phải khác nhau ở kiểu giá trị trả về, danh sách kiểu các tham số. Hàm thành phần được phép gọi tới các hàm thành phần khác, thậm chí trùng tên. Chương trình `point3.cpp` sau đây là một cải biên mới của `point.cpp`:

Ví dụ 3.5

```
/*point3.cpp*/
```

```
#include <iostream.h>
#include <conio.h>
class point  {
      int x,y;
  public:
      /*định nghĩa chồng các hàm thành phần init và display*/
      void init();
      void init (int);
      void init (int,int);
      void display();
      void display(char *);
  };
void point::init()  {
  x=y=0;
  }
void point::init(int abs)  {
  x=abs;y=0;
  }
void point::int(int abs,int ord)  {
  x=abs;
  y=ord;
  }
void point::display()
  {
  cout<<"Toa do : "<<x<<" "<<y<<"\n";
  }
void point::display(char *mesg)  {
  cout<<mesg;
```

```
  display();
  }
void main()  {
  clrscr();
  point a; a.init();/*point::init()*/
  a.display();/*point::display()*/
  point b; b.init(5); /*point::init(int )*/
  b.display("point b - ");/*point::display(char *)*/
  point c; c.init(3,12); /*point::init(int, int)*/
  c.display("Hello ----");
  getch();
  }
```

```
Toa do : 0 0
point b - Toa do : 5 0
Hello ----Toa do : 3 12
```

2.2.2 Các tham số với giá trị ngầm định

Giống như các hàm thông thường, lời gọi hàm thành phần có thể sử dụng giá trị ngầm định cho các tham số. Giá trị ngầm định này sẽ được khai báo trong định nghĩa hàm thành phần hay trong khai báo (trong khai báo lớp) của nó. Chương trình `point4.cpp` sau đây được cải tiến từ `point3.cpp` ngắn gọn hơn nhưng vẫn giữ được tất cả các khả năng như trong `point3.cpp`

Ví dụ 3.6

```
/*point4.cpp*/
#include <iostream.h>
#include <conio.h>
class point  {
      int x,y;
```

```
  public:
      void init (int=0,int=0);
      void display(char *="");
  };
void point::int(int abs,int ord)  {
  x=abs;
  y=ord;
  }
void point::display(char *mesg)  {
  cout<<mesg;
  display();
  }
void main()  {
  clrscr();
  point a; a.init();/*a.init(0,0)*/
  a.display();/*a.display(""),*/
  point b; b.init(5);/*b.init(5,0)*/
  b.display("point b - ");
  point c; c.init(3,12);/*c.init(3,12)*/
  c.display("Hello ----");
  getch();
  }
```

```
Toa do : 0 0
point b - Toa do : 5 0
Hello ----Toa do : 3 12
```

2.2.3 Sử dụng đối tượng như tham số của hàm thành phần

Ở đây đề cập đến khả năng mở rộng phạm vi lớp đối với các đối tượng "họ hàng".

2.2.3.1 Truy nhập đến các thành phần private trong đối tượng

Hàm thành phần có quyền truy nhập đến các thành phần **private** của đối tượng gọi nó. Xem định nghĩa hàm thành phần `point::init()`:

```
void point::int(int abs,int ord)
  {
  x=abs;
  y=ord;
  }
```

2.2.3.2 Truy nhập đến các thành phần `private` trong các tham số là đối tượng truyền cho hàm thành phần.

Hàm thành phần có quyền truy nhập đến tất cả các thành phần **private** của các đối tượng, tham chiếu đối tượng hay con trỏ đối tượng có cùng kiểu lớp khi được dùng là tham số hình thức của nó.

```
class point {
   int x,y;
 public:
   ...
  /* Các đối tượng được truyền theo giá trị của chúng */
   int coincide(point pt)
      {return(x==pt.x && y==pt.y);}
  /* Các đối tượng được truyền bằng địa chỉ */
   int coincide(point *pt)
      {return(x==pt->x && y==pt->y);}
  /* Các đối tượng được  truyền bằng tham chiếu */
   int coincide(point &pt)
```

```
    {return(x==pt.x && y==pt.y);}
}
```

2.2.3.3 Dùng đối tượng như giá trị trả về của hàm thành phần hàm trong cùng lớp

Hàm thành phần có thể truy nhập đến các thành phần **private** của các đối tượng, con trỏ đối tượng, tham chiếu đối tượng định nghĩa bên trong nó.

```
class point
 {
   int x,y;
 public:
   ...
   point symetry()
      {
        point res;
        res.x=-x;res.y=-y;
        return res;
      }
  };
```

2.2.4 Con trỏ this

Từ khoá **this** trong định nghĩa của các hàm thành phần lớp dùng để xác định địa chỉ của đối tượng dùng làm tham số ngầm định cho hàm thành phần. Nói cách khác, con trỏ **this** tham chiếu đến đối tượng đang gọi hàm thành phần. Như vậy, có thể truy nhập đến các thành phần của đối tượng gọi hàm thành phần gián tiếp thông qua **this**. Sau đây là một cách viết khác cho định nghĩa của các hàm `point::coincide()` và `point::display()`:

```
int point::coincide(point pt)
   {return(this->x==pt.x && this->y==pt.y);}
void point::display()
```

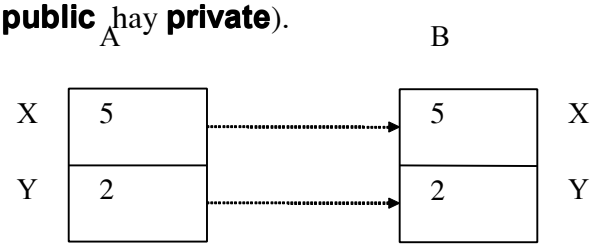
```
  {
 cout<<"Dia chi : "<<this<<"Toa do : "<<x<<" "<<y<<"\n";
}
```

3. PHÉP GÁN CÁC ĐỐI TƯỢNG

Có thể thực hiện phép gán giữa hai đối tượng cùng kiểu. Chẳng hạn, với lớp `point` khai báo ở trên:

```
point a, b;
a.init(5,2);
b=a;
```

Về thực chất đó là việc sao chép giá trị các thành phần dữ liệu (x, y) từ đối tượng `a` sang đối tượng `b` tương ứng từng đôi một (không kể đó là các thành phần **public** hay **private**).



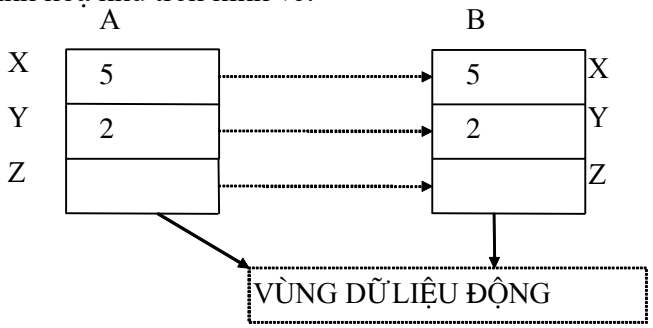
Chú ý

Khi các đối tượng trong phép gán chứa các thành phần dữ liệu động, việc sao chép lại không liên quan đến các vùng dữ liệu đó (Người ta nói rằng đó là sự "sao chép bề mặt"). Chẳng hạn, nếu hai đối tương `a` và `b` cùng kiểu, có các thành phần dữ liệu `x,y`(tĩnh) và `z` là một con trỏ chỉ đến một vùng nhớ được cấp phát động.

Phép gán

a = b;

được minh hoạ như trên hình vẽ:



Điều này có thể ít nhiều gây khó khăn cho việc quản lý cấp phát động. Thứ nhất, vùng nhớ động trước đây trong `a` (nếu có) bây giờ không thể kiểm soát được nữa. Thứ hai, vùng nhớ động của `b` bây giờ sẽ được truy nhập bởi các hàm thành phần của cả a và b và như vậy tính "riêng tư" dữ liệu của các đối tượng đã bị vi phạm.

4. HÀM THIẾT LẬP (constructor) VÀ HÀM HUỶ BỎ (destructor)

4.1 Hàm thiết lập

4.1.1 Chức năng của hàm thiết lập

Hàm thiết lập là một hàm thành phần đặc biệt không thể thiếu được trong một lớp. Nó được gọi tự động mỗi khi có một đối tượng được khai báo. Chức năng của hàm thiết lập là khởi tạo các giá trị thành phần dữ liệu của đối tượng, xin cấp phát bộ nhớ cho các thành phần dữ liệu động. Chương trình `point5.cpp` sau đây là một phiên bản mới của `point.cpp` trong đó thay thế hàm thành phần `init` bởi hàm thiết lập.

Ví dụ 3.7

```
/*point5.cpp*/
#include <iostream.h>
#include <conio.h>
/*định nghĩa lớp point* /
class point
{
  /*khai báo các thành phần dữ liệu*/
  int x;
  int y;
public:
  /*khai báo các thành phần hàm*/
  point(int ox,int oy) {x=ox;y=oy;}/*hàm thiết lập*/
  void move(int dx,int dy) ;
  void display();
  };
void point::move(int dx,int dy){
   x+=dx;
   y+=dy;
  }
void point::display(){
  cout<<"Toa do : "<<x<<" "<<y<<"\n";
  }
void main() {
clrscr();
point a(5,2); /*Sử dụng hàm thiết lập*/
a.display();
a.move(-2,4); a.display();
```

```
point b.init(1,-1);b.display();
clrscr();
}
```

```
Toa do : 5 2
Toa do : 3 6
Toa do : 1 -1
```

4.1.2 Một số đặc điểm quan trọng của hàm thiết lập

11. Hàm thiết lập có cùng tên với tên của lớp.
12. Hàm thiết lập phải có thuộc tính **public**.
13. Hàm thiết lập không có giá trị trả về. Và không cần khai báo void [2].
14. Có thể có nhiều hàm thiết lập trong cùng lớp (chồng các hàm thiết lập).
15. Khi một lớp có nhiều hàm thiết lập, việc tạo các đối tượng phải kèm theo các tham số phù hợp với một trong các hàm thiết lập đã khai báo. Ví dụ:

```
/*định nghĩa lại lớp point*/
class point {
 int x,y;
 public:
 point() {x=0;y=0;}
 point(int ox, int oy) {x=ox;y=oy;} /*hàm thiết lập có hai tham số*/
 void move(int,int);
 void display();
```

[2]Sự giới hạn này là không thể tránh được vì hàm thiết lập thường được gọi vào lúc định nghĩa một đối tượng mới, mà lúc đó thì không có cách nào để lấy lại hoặc xem xét giá trị trả về của hàm thiết lập cả. Điều này có thể trở thành một vấn đề khi hàm thiết lập cần phải trả về một trạng thái lỗi. Giải quyết vấn đề này người ta dùng đến khả năng kiểm soát lỗi sẽ được trình bày trong phụ lục 2.

```
};
point a(1); /* Lỗi vì tham số không phù hợp với hàm thiết lập */
point b;/*Đúng, tham số phù hợp với hàm thiết lập không tham số*/
point c(2,3);/*Đúng, tham số phù hợp với hàm thiết lập thứ hai, có hai tham số*/
```

16. Hàm thiết lập có thể được khai báo với các tham số có giá trị ngầm định. Xét ví dụ sau:

```
/*Định nghĩa lại lớp point*/
class point {
 int x,y;
 public:
 point(int ox, int oy = 0) {x=ox;y=oy;} /*hàm thiết lập có hai tham số*/
 void move(int,int);
 void display();
};
point a; /*Lỗi: không có hàm thiết lập ngầm định hoặc hàm thiết lập với các tham số có giá trị ngầm định*/
point b(1);//Đối số thứ hai nhận giá trị 0
point c(2,3);//Đúng
```

Nhận xét

Trong ví dụ trên, chỉ thị:

```
point b(1);
```

có thể được thay thế bằng cách viết khác như sau:

```
point b=1;
```

Cách viết thứ hai hàm ý rằng đã có chuyển kiểu ngầm định từ số nguyên 1 thành đối tượng kiểu `point`. Chúng ta sẽ đề cập vấn đề này một cách đầy đủ hơn trong chương 4.

4.1.3 Hàm thiết lập ngầm định

Hàm thiết lập ngầm định do chương trình dịch cung cấp khi trong khai báo lớp không có định nghĩa hàm thiết lập nào. Lớp `point` định nghĩa trong chương trình `point.cpp` là một ví dụ trong đó chương trình biên dịch tự bổ sung một hàm thiết lập ngầm định cho khai báo lớp. Dĩ nhiên hàm thiết lập ngầm định đó không thực hiện bất cứ nhiệm vụ nào ngoài việc "lấp chỗ trống".

Đôi khi người ta cũng gọi hàm thiết lập không có tham số do người sử dụng định nghĩa là hàm thiết lập ngầm định.

Cần phải có hàm thiết lập ngầm định khi cần khai báo mảng các đối tượng. Ví dụ, trong khai báo:

```
X a[10];
```

bắt buộc trong lớp X phải có một hàm thiết lập ngầm định.

Ta minh hoạ nhận xét này bằng hai ví dụ sau:

a. Trong trường hợp thứ nhất không dùng hàm thiết lập không tham số:

Ví dụ 3.8

```
/*point6.cpp*/
#include <iostream.h>
/*định nghĩa lớp point* /
class point {
  /*khai báo các thành phần dữ liệu*/
  int x;
  int y;
public:
  /*khai báo các hàm thành phần */
  point(int ox,int oy) {x=ox;y=oy;}
  void move(int dx,int dy) ;
```

```
   void display();
   };
/*phân biệt các hàm thành phần  với các hàm thông thường nhờ tên lớp và toán tử ::*/
void point::move(int dx,int dy)  {
    x+=dx;
    y+=dy;
   }
void point::display()  {
    cout<<"Toa do : "<<x<<" "<<y<<"\n";
}
void main() {
point a(5,2); //OK
a.display();
a.move(-2,4); a.display();
point b[10];//lỗi vì không cung cấp thông số cần thiết cho hàm thiết lập
}
```

Trong chương trình point6.cpp, lỗi xảy ra vì ta muốn tạo ta mười đối tượng nhưng không cung cấp đủ các tham số cho hàm thiết lập có như đã định nghĩa (ở đây ta chưa đề cập đến hàm thiết lập sao chép ngầm định, nó sẽ được trình bày trong phần sau). Giải quyết tình huống này bằng hai cách: hoặc bỏ luôn hàm thiết lập hai tham số trong khai báo lớp nhưng khi đó, khai báo của đối tượng a sẽ không còn đúng nữa. Do vậy ta thường sử dụng giải pháp định nghĩa thêm một hàm thiết lập không tham số:

b. Định nghiã hàm thiết lập không tham số

Ví dụ 3.9

```
/*point7.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
```

```
   /*khai báo các thành phần dữ liệu*/
   int x;
   int y;
public:
   /*khai báo các  hàm thành phần*/
   point(int ox,int oy) {x=ox;y=oy;}
   /*định nghĩa thêm hàm thiết lập không tham số*/
   point() {x = 0; y = 0;}
   void move(int dx,int dy) ;
   void display();
   };
/*phân biệt các thành phần hàm với các hàm thông thường nhờ tên lớp và
toán tử .:*/
void point::move(int dx,int dy)  {
    x+=dx;
    y+=dy;
   }
void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<"\n";
}
void main() {
clrscr();
point a(5,2); //OK
a.display();
a.move(-2,4); a.display();
point b[10];/*Hết lỗi vì hàm thiết lập không tham số được gọi để tạo các
đối tượng
                  thành phần của  */
getch();
```

```
}
```

Còn một giải pháp khác không cần định nghĩa thêm hàm thiết lập không tham số. Khi đó cần khai báo giá trị ngầm định cho các tham số của hàm thiết lập hai tham số:

Ví dụ 3.10

```
/*point8.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
 /*khai báo các thành phần dữ liệu*/
  int x;
  int y;
public:
 /*khai báo các hàm thành phần */
  point(int ox = 1,int oy =0) {x=ox;y=oy;}
  void move(int dx,int dy) ;
  void display();
  };
void point::move(int dx,int dy){
  x+=dx;
  y+=dy;
  }
void point::display()  {
  cout<<"Toa do : "<<x<<" "<<y<<"\n";
  }
void main() {
clrscr();
point a(5,2); //OK
```

```
a.display();
a.move(-2,4); a.display();
point b[10];/*Trong trường hợp này các đối tượng thành phần của b được tạo ra nhờ
                hàm thiết lập được gọi với hai tham số có giá trị ngầm định là 1 và 0.*/
getch();
}
```

4.1.4 Con trỏ đối tượng

Con trỏ đối tượng được khai báo như sau:

```
point *ptr;
```

Con trỏ đối tượng có thể nhận giá trị là địa chỉ của các đối tượng có cùng kiểu lớp:

```
ptr = &a;
```

Khi đó có thể gọi các hàm thành phần của lớp `point` thông qua con trỏ như sau:

```
ptr->display();
ptr->move(-2,3);
```

Khi dùng toán tử **new** cấp phát một đối tượng động, hàm thiết lập cũng được gọi, do vậy cần cung cấp danh sách các tham số. Chẳng hạn, giả sử trong lớp `point` có một hàm thiết lập hai tham số, khi đó câu lệnh sau:

```
ptr = new point(3,2);
```

sẽ xin cấp phát một đối tượng động với hai thành phần x và y nhận giá trị tương ứng là 2 và 3. Kết quả này được minh chứng qua lời gọi hàm:

```
ptr->display();
```

```
Toa do : 2 3
```

Ta xét chương trình ví dụ sau:

Ví dụ 3.11

```
/*point9.cpp*/
#include <conio.h>
#include <iostream.h>
class point
{
 /*khai báo các thành phần dữ liệu*/
  int x;
  int y;
public:
 /*khai báo các hàm thành phần */
  point(int ox = 1,int oy =0) {x=ox;y=oy;}
  void move(int dx,int dy) ;
  void display();
  };
void point::move(int dx,int dy){
   x+=dx;
   y+=dy;
  }
void point::display() {
  cout<<"Toa do : "<<x<<" "<<y<<"\n";
  }
void main() {
clrscr();
point a(5,2);
a.display();
point *ptr = &a;
```

```
ptr->display();
a.move(-2,4);
ptr->display();
ptr->move(2,-4);
a.display();
ptr = new point;/*Gọi hàm thiết lập với hai tham số có giá trị ngầm định*/
ptr->display();
delete ptr;
ptr = new point(3);/*Tham số thứ hai của hàm thiết lập có giá trị ngầm định*/
ptr->display();
delete ptr;
ptr = new point (3,5);/*Hàm thiết lập được cung cấp hai tham số tường minh*/
ptr->display();
delete ptr;
point b[10];
getch();
}
```

```
Toa do : 5 2
Toa do : 5 2
Toa do : 3 6
Toa do : 5 2
Toa do : 1 0
Toa do : 3 0
Toa do : 3 5
```

4.1.5 Khai báo tham chiếu đối tượng

Khi đối tượng là nội dung một biến có kiểu lớp, ta có thể gán cho nó các "bí danh"; nghĩa là có thể khai báo các tham chiếu đến chúng. Một tham chiếu đối tượng chỉ có ý nghĩa khi tham chiếu tới một đối tượng nào đó đã được khai báo trước đó. Chẳng hạn:

```
point a(2,5);
point &ra=a;
a.display();
ra.display();
ra.move(2,3);
a.display();
```

```
Toa do : 2 5
Toa do : 2 5
Toa do : 4 8
```

4.2 Hàm huỷ bỏ

4.2.1 Chức năng của hàm huỷ bỏ

Ngược với hàm thiết lập, hàm huỷ bỏ được gọi khi đối tượng tương ứng bị xoá khỏi bộ nhớ. Ta xét chương trình ví dụ sau:

Ví dụ 3.12

```
/*test.cpp*/
#include <iostream.h>
#include <conio.h>
int line=1;
class test {
  public:
        int num;
        test(int);
        ~test();
```

```
  };
test::test(int n) {
  num = n;
  cout<<line++<<".";
  cout<<"++ Goi ham thiet lap voi num = "<<num<<"\n";
  }
test::~test() {
  cout<<line++<<".";
  cout<<"-- Goi ham huy bo voi num = "<<num<<"\n";
  }
void main() {
  clrscr();
  void fct(int);
  test a(1);
  for(int i=1; i<= 2; i++) fct(i);
  }
void fct(int p) {
  test x(2*p);
  }
```

```
1. ++ Goi ham thiet lap voi num = 1
2. ++ Goi ham thiet lap voi num = 2
3. -- Goi ham huy bo voi num = 2
4. ++ Goi ham thiet lap num = 4
5. -- Goi ham huy bo voi num = 4
6. -- Goi ham huy bo voi num = 1
```

Ta lý giải như sau: trong chương trình chính, dòng thứ nhất tạo ra đối tượng `a` có kiểu lớp test, do đó có dòng thông báo số 1. Vòng lặp **for** hai lần gọi tới hàm `fct()`. Mỗi lời gọi hàm `fct()` kéo theo việc khai báo một đối

tượng cục bộ `x` trong hàm. Vì là đối tượng cục bộ bên trong hàm `fct()` nên `x` bị xoá khỏi vùng bộ nhớ ngăn xếp (dùng để cấp phát cho các biến cục bộ khi gọi hàm) khi kết thúc thực hiện hàm. Do đó, mỗi lời gọi tới `fct()` sinh ra một cặp dòng thông báo, tương ứng với lời gọi hàm thiết lập, hàm huỷ bỏ(các dòng thông báo 2, 3, 4, 5 tương ứng). Cuối cùng, khi hàm `main()` kết thúc thực hiện, đối tượng `a` được giải phóng, hàm huỷ bỏ đối với a sẽ cho ra dòng thông báo thứ 6.

4.2.2 Một số qui định đối với hàm huỷ bỏ

17. Tên của hàm huỷ bỏ bắt đầu bằng dấu ~ theo sau là tên của lớp tương ứng. Chẳng hạn lớp `test` thì sẽ hàm huỷ bỏ tên là `~test`.
18. Hàm huỷ bỏ phải có thuộc tính **public**
19. Nói chung hàm huỷ bỏ không có tham số, mỗi lớp chỉ có một hàm huỷ bỏ (Trong khi đó có thể có nhiều các hàm thiết lập).
20. Khi không định nghĩa hàm huỷ bỏ, chương trình dịch tự động sản sinh một hàm như vậy (hàm huỷ bỏ ngầm định), hàm này không làm gì ngoài việc "lấp chỗ trống". Đối với các lớp không có khai báo các thành phần bộ nhớ động, có thể dùng hàm huỷ bỏ ngầm định. Trái lại, phải khai báo hàm huỷ bỏ tường minh để đảm bảo quản lý tốt việc giải phóng bộ nhớ động do các đối tượng chiếm giữ chiếm giữ khi chúng hết thời gian làm việc.
21. Giống như hàm thiết lập, hàm huỷ bỏ không có giá trị trả về.

4.3 Sự cần thiết của các hàm thiết lập và huỷ bỏ -lớp vector trong không gian n chiều

Trên thực tế, với các lớp không có các thành phần dữ liệu động chỉ cần sử dụng hàm thiết lập và huỷ bỏ ngầm định là đủ. Hàm thiết lập và huỷ bỏ do người lập trình tạo ra rất cần thiết khi các lớp chứa các thành phần dữ liệu động. Khi tạo đối tượng hàm thiết lập đã xin cấp phát một khối bộ nhớ động, do đó hàm huỷ bỏ phải giải phóng vùng nhớ đã được cấp phát trước đó. Ví dụ sau đây minh hoạ vai trò của hàm huỷ bỏ trong trường hợp lớp có các thành phần cấp phát động.

Ví dụ 3.13

```
/*vector.cpp*/
#include <iostream.h>
#include <conio.h>
class vector {
      int n;   //số chiều
      float *v;     //vùng nhớ toạ độ
  public:
      vector(); //Hàm thiết lập không tham số
      vector(int size); //Hàm thiết lập một tham số
      vector(int size, float *a);
      ~vector();//Hàm huỷ bỏ, luôn luôn không có tham số
      void display();
  };
vector::vector()  {
  int i;
  cout<<"Tao doi tuong tai "<<this<<endl;
  cout<<"So chieu :";cin>>n;
  v= new float [n];
  cout<<"Xin cap phat vung bo nho "<<n<<" so thuc
tai"<<v<<endl;
```

```
  for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
    }
  }
vector::vector(int size)  {
  int i;
  cout<<"Su dung ham thiet lap 1 tham so\n";
  cout<<"Tao doi tuong tai "<<this<<endl;
  n=size;
  cout<<"So chieu :"<<size<<endl;
  v= new float [n];
  cout<<"Xin cap phat vung bo nho "<<n<<" so thuc
tai"<<v<<endl;
  for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
    }
  }
vector::vector(int size,float *a )  {
  int i;
  cout<<"Su dung ham thiet lap 2 tham so\n";
  cout<<"Tao doi tuong tai "<<this<<endl;
  n=size;
  cout<<"So chieu :"<<n<<endl;
  v= new float [n];
  cout<<"Xin cap phat vung bo nho "<<n<<" so thuc
tai"<<v<<endl;
  for(i=0;i<n;i++)
```

```
    v[i] = a[i];
  }
vector::~vector() {
   cout<<"Giai phong "<<v<<"cua doi tuong tai"<<this<<endl;
   delete v;
   }
//Hiển thị kết quả
void vector::display()  {
  int i;
  cout<<"Doi tuong tai :"<<this<<endl;
  cout<<"So chieu :"<<n<<endl;
  for(i=0;i<n;i++) cout <<v[i] <<" ";
  cout <<"\n";
}
void main()  {
  clrscr();
  vector s1;
  s1.display();
  vector s2(4);
  s2.display();
  float a[3]={1,2,3};
  vector s3(3,a);
  s3.display();
  getch();
  }
```

```
Tao doi tuong tai 0xfff2
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x13cc
```

```
Toa do thu 1 : 2
Toa do thu 2 : 3
Toa do thu 3 : 2
Doi tuong tai :0xfff2
So chieu :3
2 3 2
Su dung ham thiet lap 1 tham so
Tao doi tuong tai 0xffee
So chieu :4
Xin cap phat vung bo nho 4 so thuc tai0x13dc
Toa do thu 1 : 3
Toa do thu 2 : 2
Toa do thu 3 : 3
Toa do thu 4 : 2
Doi tuong tai :0xffee
So chieu :4
3 2 3 2
Su dung ham thiet lap 2 tham so
Tao doi tuong tai 0xffea
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x13f0
Doi tuong tai :0xffea
So chieu :3
1 2 3
Doi tuong tai :0xfff2
So chieu :3
2 3 2
Giai phong 0x13f0cua doi tuong tai0xffea
```

```
Giai phong 0x13dccua doi tuong tai0xffee
Giai phong 0x13cccua doi tuong tai0xfff2
```

Chú ý

Không được lẫn lộn giữa cấp phát bộ nhớ động trong hàm thành phần của đối tượng (thông thường là hàm thiết lập) với việc cấp phát động cho một đối tượng.

4.4 Hàm thiết lập sao chép (COPY CONSTRUCTOR)

4.4.1 Các tình huống sử dụng hàm thiết lập sao chép

Xét các chỉ thị khai báo và khởi tạo giá trị cho một biến nguyên:

```
int p;
int x = p;
```

Chỉ thị thứ hai khai báo một biến nguyên x và gán cho nó giá trị của biến nguyên p. Tương tự, ta cũng có thể khai báo một đối tượng và gán cho nó nội dung của một đối tượng cùng lớp đã tồn tại trước đó. Chẳng hạn:

```
point p(2,3);/*giả thiết lớp point có hàm thiết lập hai tham số*/
point q =p;
```

Dĩ nhiên hai đối tượng, mới q và cũ p có cùng nội dung. Khi một đối tượng được tạo ra (khai báo) thì một hàm thiết lập của lớp tương ứng sẽ được gọi. Hàm thiết lập được gọi khi khai báo và khởi tạo nội dung một đối tượng thông qua một đối tượng khác, gọi là hàm thiết lập sao chép. Nhiệm vụ của hàm thiết lập sao chép là tạo ra một đối tượng giống hệt một đối tượng đã có. Thoạt nhìn hàm thiết lập sao chép có vẻ thực hiện các công việc giống như phép gán, nhưng nếu để ý sẽ thấy giữa chúng có chút ít khác biệt; phép gán thực hiện việc sao chép nội dung từ đối tượng này sang đối tượng khác, do vậy cả hai đối tượng trong phép gán đều đã tồn tại:

```
point p(2,3);//giả thiết lớp point có hàm thiết lập hai tham số
point q;//giả thiết lớp point có hàm thiết lập không tham số
q = p;
```

Ngược lại, hàm thiết lập thực hiện đồng thời hai nhiệm vụ: tạo đối tượng và sao chép nội dung từ một đối tượng đã có sang đối tượng mới tạo ra đó.

Ngoài tình huống trên đây, còn có hai trường hợp cần dùng hàm thiết lập sao chép: truyền đối tượng cho hàm bằng tham trị hoặc hàm trả về một đối tượng nhằm tạo một đối tượng giống hệt một đối tượng cùng lớp đã có trước đó. Trong phần sau chúng ta sẽ có ví dụ minh hoạ cho các trình bày này.

4.4.2 Hàm thiết lập sao chép ngầm định

Giống như hàm thiết lập ngầm định (hàm thiết lập không tham số), nếu không được mô tả tường minh, sẽ có một hàm thiết lập sao chép ngầm định do chương trình dịch cung cấp nhằm đảm bảo tính đúng đắn của chương trình trong các tình huống cần đến hàm thiết lập. Như vậy, trong khai báo của một lớp có ít nhất hai hàm thiết lập ngầm định: hàm thiết lập ngầm định và hàm thiết lập sao chép ngầm định.

Do là một hàm được tạo ra tự động nên hàm thiết lập sao chép ngầm định cũng chỉ thực hiện những thao tác tối thiểu ("ngầm định"): tạo giá trị của các thuộc tính trong đối tượng mới bằng các giá trị của các thuộc tính tương ứng trong đối tượng cũ. Bạn đọc có thể xem lại phần 3 của chương để hiểu rõ hơn. Nói chung, với các lớp không khai báo các thành phần dữ liệu động thì chỉ cần dùng hàm thiết lập sao chép ngầm định là đủ. Vấn đề sẽ khác đi khi cần đến các thao tác quản lý bộ nhớ động trong các đối tượng. Trong trường hợp này không được dùng hàm thiết lập sao chép ngầm định mà phải gọi hàm thiết lập sao chép tường minh.

4.4.3 Khai báo và định nghĩa hàm thiết lập sao chép tường minh

Dạng của hàm thiết lập sao chép

Xét các đối tượng thuộc lớp `point`. Câu lệnh

```
point q=p;
```

SẼ GỌI ĐẾN HÀM THIẾT LẬP SAO CHÉP.

Như nhận xét trong phần trên ta có thể viết theo cách khác như sau:

```
point q(p);
```

Từ cách viết trên có thể cho rằng dạng của hàm thiết lập sao chép cho lớp `point` có thể là:

```
point (point);
```

hoặc

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

Ta nhận thấy dạng thứ nhất không dùng được vì việc gọi nó đòi hỏi phải truyền cho hàm một đối tượng như một tham trị, do đó gây ra đệ quy vô hạn lần.

Với dạng thứ hai ta đã thiết lập một tham chiếu tới đối tượng như một tham số hình thức truyền cho hàm, nên có thể chấp nhận được.

Dạng khai báo của hàm thiết lập là:

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

trong đó từ khoá `const` trong khai báo tham số hình thức chỉ nhằm ngăn cấm mọi thay đổi nội dung của tham số truyền cho hàm.

Chương trình `point10.cpp` sau đây bổ sung thêm hàm thiết lập sao chép vào lớp `point`.

Ví dụ 3.14

```
/*point10.cpp*/
#include <conio.h>
#include <iostream.h>
/*®Þnh nghÜa líp point*/
class point {
  /*khai báo các thành phần dữ liệu*/
  int x;
  int y;
public:
 /*khai báo các thành phần hàm*/
  point(int ox = 1,int oy =0) {
     cout<<"Tao doi tuong : "<<this<<endl;
     cout<<"Dung ham thiet lap hai tham so\n";
```

```
     x=ox;y=oy;
     }
 /*Hàm thiết lập sao chép*/
  point(point &p) {
    cout<<"Tao doi tuong : "<<this<<endl;
    cout<<"Dung ham thiet lap sao chep\n";
    x = p.x; y = p.y;
    }
  void move(int dx, int dy) {
   x+=dx; y+=dy;
   }
  void display();
  };
void point::display()  {
   cout<<"Toa do : "<<x<<" "<<y<<"\n";
}
point fct(point a) {
  point b=a;
  b.move(2,3);
  return b;
  }
void main(){
clrscr();
point a(5,2);
a.display();
point b=fct(a);
b.display();
getch();
```

```
}
```

```
Tao doi tuong : 0xfff2
Dung ham thiet lap hai tham so
Toa do : 5 2
Tao doi tuong : 0xffea
Dung ham thiet lap sao chep
Tao doi tuong : 0xffde
Dung ham thiet lap sao chep
Tao doi tuong : 0xffee
Dung ham thiet lap sao chep
Toa do : 7 5
```

4.4.4 Hàm thiết lập sao chép cho lớp vector

Chương trình ví dụ sau giới thiệu cách định nghĩa hàm thiết lập khi đối tượng có các thành phần dữ liệu động.

Ví dụ 3.15

```
/*vector2.cpp*/
#include <iostream.h>
#include <conio.h>
class vector {
      int n;   //số chiều của vector
      float *v;     //vùng nhớ chứa các toạ độ
  public:
      vector();
      vector(int size);
      vector(int size, float *a);
      vector(vector &);//hàm thiết lập sao chép
      ~vector();
```

```
      void display();
  };
vector::vector()  {
  int i;
  cout<<"Tao doi tuong tai "<<this<<endl;
  cout<<"So chieu :";cin>>n;
  v= new float [n];
  cout<<"Xin cap phat vung bo nho "<<n<<" so thuc
tai"<<v<<endl;
  for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
    }
  }
vector::vector(int size) {
  int i;
  cout<<"Su dung ham thiet lap 1 tham so\n";
  cout<<"Tao doi tuong tai "<<this<<endl;
  n=size;
  cout<<"So chieu :"<<size<<endl;
  v= new float [n];
  cout<<"Xin cap phat vung bo nho "<<n<<" so thuc
tai"<<v<<endl;
  for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
    }
  }
vector::vector(int size,float *a ) {
```

```
  int i;
  cout<<"Su dung ham thiet lap 2 tham so\n";
  cout<<"Tao doi tuong tai "<<this<<endl;
  n=size;
  cout<<"So chieu :"<<n<<endl;
  v= new float [n];
  cout<<"Xin cap phat vung bo nho "<<n<<" so thuc
tai"<<v<<endl;
  for(i=0;i<n;i++)
    v[i] = a[i];
  }
vector::vector(vector &b) {
  int i;
  cout<<"Su dung ham thiet lap sao chep\n";
  cout<<"Tao doi tuong tai "<<this<<endl;
 /*xin cấp phát một vùng nhớ động bằng kích thước có trong đối tượng cũ*/
  v= new float [n=b.n];
  cout<<"Xin cap phat vung bo nho "<<n<<" so thuc
tai"<<v<<endl;
  for(i=0;i<n;i++)
 /*gán nội dung vùng nhớ động của đối tượng cũ sang đối tượng mới*/
    v[i] = b.v[i];
  }
vector::~vector() {
  cout<<"Giai phong "<<v<<"cua doi tuong tai"<<this<<endl;
   delete v;
   }
//hiển thị kết quả
void vector::display()  {
```

```
  int i;
  cout<<"Doi tuong tai :"<<this<<endl;
  cout<<"So chieu :"<<n<<endl;
  for(i=0;i<n;i++) cout <<v[i] <<" ";
  cout <<"\n";
}
void main() {
  clrscr();
  vector s1;//gọi hàm thiết lập không tham số
  s1.display();
  vector s2(4); //4 giá trị
  s2.display();
  float a[3]={1,2,3};
  vector s3(3,a);
  s3.display();
  vector s4 = s1;//hàm thiết lập sao chép
  s4.display();
  getch();
  }
```

```
Tao doi tuong tai 0xfff2
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x142c
Toa do thu 1 : 1
Toa do thu 2 : 2
Toa do thu 3 : 3
Doi tuong tai :0xfff2
So chieu :3
1 2 3
```

```
Su dung ham thiet lap 1 tham so
Tao doi tuong tai 0xffee
So chieu :4
Xin cap phat vung bo nho 4 so thuc tai0x143c
Toa do thu 1 :
Su dung ham thiet lap 1 tham so
Tao doi tuong tai 0xffee
So chieu :4
Xin cap phat vung bo nho 4 so thuc tai0x143c
Toa do thu 1 : 2
Toa do thu 2 : 3
Toa do thu 3 : 4
Toa do thu 4 : 5
Doi tuong tai :0xffee
So chieu :4
2 3 4 5
Su dung ham thiet lap 2 tham so
Tao doi tuong tai 0xffea
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x1450
Doi tuong tai :0xffea
So chieu :3
1 2 3
Su dung ham thiet lap sao chep
Tao doi tuong tai 0xffe6
Xin cap phat vung bo nho 3 so thuc tai0x1460
Doi tuong tai :0xffe6
So chieu :3
```

```
1 2 3
Giai phong 0x1460cua doi tuong tai0xffe6
Giai phong 0x1450cua doi tuong tai0xffea
Giai phong 0x143ccua doi tuong tai0xffee
Giai phong 0x142ccua doi tuong tai0xfff2
```

5. CÁC THÀNH PHẦN TĨNH (static)

5.1 Thành phần dữ liệu `static`.

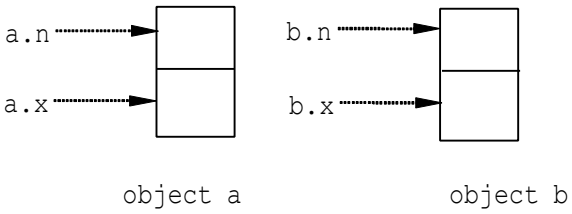
Thông thường, trong cùng một chương trình các đối tượng thuộc cùng một lớp chỉ sở hữu các thành phần dữ liệu của riêng nó. Ví dụ, nếu chúng ta định nghĩa lớp `exple1` bằng:

```
class exple1
  {
   int n;
   float x;
   ....
};
```

khai báo :

```
exple1 a,b;
```

sẽ tạo ra hai đối tượng `a,b` sở hữu riêng biệt hai vùng dữ liệu khác nhau như hình vẽ:



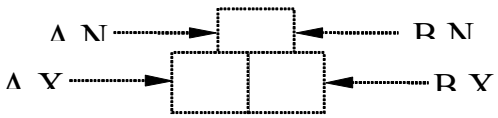
Có thể cho phép nhiều đối tượng cùng chia xẻ dữ liệu bằng cách đặt từ khoá **static** trước khai báo thành phần dữ liệu tương ứng. Ví dụ, nếu ta định nghĩa lớp `exple2` bằng:

```
class exple2
  {
   static int n;
   float x;
   ....
};
```

thì khai báo

```
exple2 a,b;
```

tạo ra hai đối tượng có chung thành phần n:



Như vậy, việc chỉ định **static** đối với một thành phần dữ liệu có ý nghĩa là trong toàn bộ lớp, chỉ có một thể hiện duy nhất của thành phần đó. Thành phần **static** được dùng chung cho tất cả các đối tượng của lớp đó và do đó vẫn chiếm giữ vùng nhớ ngay cả khi không khai báo bất kỳ đối tượng nào. Có thể nói rằng các thành phần dữ liệu tĩnh giống như các biến toàn cục trong phạm vi lớp. Các phần tiếp sau sẽ làm nổi bật nhận xét này.

5.2 Khởi tạo các thành phần dữ liệu tĩnh

Các thành phần dữ liệu **static** chỉ có một phiên bản trong tất cả các đối tượng. Như vậy không thể khởi tạo chúng bằng các hàm thiết lập của một lớp.

Cũng không thể khởi tạo lúc khai báo các thành phần dữ liệu **static** như trong ví dụ sau:

```
class exple2{
static int n=2;//lỗi
};
```

Một thành phần dữ liệu **static** phải được khởi tạo một cách tường minh bên ngoài khai báo lớp bằng một chỉ thị như sau:

```
int exple2::n=5;
```

Trong C++ việc khởi tạo giá trị như thế này không vi phạm tính riêng tư của các đối tượng. Chú ý rằng cần phải có tên lớp và toán tử phạm vi để chỉ định các thành phần của lớp được khởi tạo.

Ngoài ra, khác với các biến toàn cục thông thường, các thành phần dữ liệu **static** không được khởi tạo ngầm định là 0. Chương trình counter.cpp sau đây minh hoạ cách sử dụng và thao tác với thành phần dữ liệu **static,** dùng để đếm số đối tượng hiện đang được sử dụng:

Ví dụ 3.16

```
/*counter.cpp*/
#include <iostream.h>
#include <conio.h>
class counter  {
    static int count;   //đếm số đối tượng được tạo ra
  public :
    counter ();
    ~ counter ();

  };
int counter::count=0; //khởi tạo giá trị cho thành phần static
//hàm thiết lập
counter:: counter ()   {
  cout<<"++Tao : bay gio co "<<++count<<" doi tuong\n";
  }
counter:: ~counter ()  {
  cout<<"--Xoa : bay gio con "<<--count<<" doi tuong\n";
  }
void main()  {
```

```
  clrscr();
  void fct();
  counter a;
  fct();
  counter b;
  }
void fct()  {
  counter u,v;
  }
```

```
++Tao : bay gio co 1 doi tuong
++Tao : bay gio co 2 doi tuong
++Tao : bay gio co 3 doi tuong
--Xoa : bay gio con 2 doi tuong
--Xoa : bay gio con 1 doi tuong
++Tao : bay gio co 2 doi tuong
--Xoa : bay gio con 1 doi tuong
--Xoa : bay gio con 0 doi tuong
```

Nhận xét

22. Thành phần dữ liệu tĩnh có thể là **private** hay **public**.

23. Trong C thuật ngữ **static** có nghĩa là:"lớp lưu trữ cố định" hay có phạm vi giới hạn bởi file nguồn. Trong C++, các thành phần dữ liệu **static** còn có thêm ý nghĩa khác: "không phụ thuộc vào bất kỳ thể hiện nào của lớp".

Trong phần sau chúng ta sẽ đề cập đến các hàm thành phần **static**.

5.3 Các hàm thành phần static

Một hàm thành phần được khai báo bắt đầu với từ khoá **static** được gọi là hàm thành phần **static**, hàm thành phần **static** cũng độc lập với bất kỳ đối tượng nào của lớp. Nói cách khác hàm thành phần **static** không có tham số ngầm định. Vì không đòi hỏi đối tượng làm tham số ngầm định nên không thể sử dụng con trỏ **this** trong định nghĩa của hàm thành phần **static**.

Các hàm thành phần **static** của một lớp có thể được gọi, cho dù có khai báo các đối tượng của lớp đó hay không.

Cú pháp gọi hàm trong trường hợp này là:

<tên lớp>::<tên hàm thành phần>(<các tham số nếu có>)

Tất nhiên vẫn có thể gọi các hàm thành phần **static** thông qua các đối tượng. Tuy nhiên cách gọi thông qua tên lớp trực quan hơn vì phản ánh được bản chất của hàm thành phần **static.** [3]

Thông thường, các hàm thành phần **static** được dùng để xử lý chung trên tất cả các đối tượng của lớp, chẳng hạn để hiện thị các thông tin liên quan đến các thành phần dữ liệu **static**.

Chương trình counter1.cpp sau đây được cải tiến từ counter.cpp bằng cách thêm một hàm thành phần **static** trong lớp counter.

Ví dụ 3.17

```
/*counter1.cpp*/
#include <iostream.h>
#include <conio.h>
class counter {
    static int count;   //đếm số đối tượng được tạo ra
  public :
    counter ();
    ~ counter ();
    static void counter_display();
  };
int counter::count=0;  //khởi tạo giá trị cho thành phần static
void counter::counter_display() {
  cout<<"Hien dang co "<<count<<" doi tuong \n";
  }
counter:: counter () {
```

[3]Có một số phiên bản chương trình dịch C++ không chấp nhận cách gọi hàm thành phần static qua đối tượng.

```
  cout<<"++Tao : bay gio co "<<++count<<" doi tuong\n";
  }
counter:: ~counter () {
  cout<<"--Xoa : bay gio con "<<--count<<" doi tuong\n";
  }
void main() {
  clrscr();
  void fct();
  counter a;
  fct();
  counter::counter_display();
  counter b;
  }
void fct() {
  counter u;
  counter::counter_display();//gọi qua tên lớp
  counter v;
  v.counter_display();//gọi qua đối tượng
  }
```

```
++Tao : bay gio co 1 doi tuong
++Tao : bay gio co 2 doi tuong
Hien dang co 2 doi tuong
++Tao : bay gio co 3 doi tuong
Hien dang co 3 doi tuong
--Xoa : bay gio con 2 doi tuong
--Xoa : bay gio con 1 doi tuong
Hien dang co 1 doi tuong
++Tao : bay gio co 2 doi tuong
```

```
--Xoa : bay gio con 1 doi tuong
--Xoa : bay gio con 0 doi tuong
```

6. ĐỐI TƯỢNG HẰNG (CONSTANT)

6.1 Đối tượng hằng

Cũng như các phần tử dữ liệu khác, một đối tượng có thể được khai báo là hằng. Trừ khi có các chỉ định cụ thể, phương thức duy nhất sử dụngcác đối tượng hằng là các hàm thiết lập và hàm huỷ bỏ. Bởi lẽ các đối tượng hằng không thể thay đổi, mà chỉ được tạo ra hoặc huỷ bỏ đi. Tuy nhiên, ta có thể tạo ra các phương thức khác để xử lý các đối tượng hằng. Bất kỳ hàm thành phần nào có từ khoá **const** đứng ngay sau danh sách các tham số hình thức trong dòng khai báo (ta gọi là hàm thành phần **const**) đều có thể sử dụng các đối tượng hằng trong lớp. Nói cách khác, ngoài hàm thiết lập và huỷ bỏ, các đối tượng hằng chỉ có thể gọi được các hàm thành phần được khai báo với từ khoá **const**. Xét ví dụ sau đây:

```
class point {
   int x,y;
public:
   point(...);
   void display() const;
   void move(...);
};
```

6.2 Hàm thành phần *const*

Hàm thành phần của lớp được khai báo với từ khoá **const** đứng ngay sau danh sách các tham số hình thức được gọi là hàm thành phần **const**.

Hàm thành phần **const** thì không thể thay đổi nội dung một đối tượng. Một hàm thành phần **const** phải được mô tả cả trong khai báo và khi định nghĩa. Hàm thành phần **const** có thể được định nghĩa chồng bằng một hàm khác không phải **const**.

7. HÀM BẠN VÀ LỚP BẠN

7.1 Đặt vấn đề

Trong các phần trước ta thấy rằng nguyên tắc đóng gói dữ liệu trong C++ bị "vi phạm" tí chút; các thành phần **private** của đối tượng chỉ có thể truy nhập bởi các hàm thành phần của chính lớp đó. Ngoài ra, hàm thành

phần còn có thể truy nhập đến tất cả thành phần **private** của các đối tượng cùng lớp được khai báo cục bộ bên trong hàm thành phần đó hoặc được truyền như là tham số của hàm thành phần (có thể bằng tham trị , bằng tham chiếu hay bằng tham trỏ). Có thể thấy điều này trong định nghĩa của hàm thành phần `point::coincide()` và hàm thành phần `point::symetric()` đã trình bày ở trên. Những "vi phạm" trên đây đều chấp nhận được do làm tăng khả năng của ngôn ngữ và nâng cao tốc độ thực hiện chương trình.

Khi cho phép các hàm bạn, lớp bạn nguyên tắc đóng gói dữ liệu lại bị "vi phạm". Sự " vi phạm" đó hoàn toàn chấp nhận được và tỏ ra rất hiệu quả trong một số tình huống đặc biệt: giả sử chúng ta đã có định nghĩa lớp véc tơ vector, lớp ma trận matrix. Và muốn định nghĩa hàm thực hiện nhân một ma trận với một véc tơ. Với những kiến thức về C++ cho đến nay, ta không thể định nghĩa hàm này bởi nó không thể là hàm thành phần của lớp vector, cũng không thể là hàm thành phần của lớp matrix và càng không thể là một hàm tự do (có nghĩa là không của lớp nào). Tất nhiên, có thể khai báo tất cả các thành phần dữ liệu trong hai lớp vector và matrix là **public**, nhưng điều đó sẽ làm mất đi khả năng bảo vệ chúng. Một biện pháp khác là định nghĩa các hàm thành phần **public** cho phép truy nhập dữ liệu, tuy nhiên giải pháp này khá phức tạp và chi phí thời gian thực hiện không nhỏ. Khái niệm "hàm bạn" - `friend function` đưa ra một giải pháp tốt hơn nhiều cho vấn đề đặt ra ở trên. Khi định nghĩa một lớp, có thể khai báo rằng một hay nhiều hàm "bạn" (bên ngoài lớp) ; khai báo bạn bè như thế cho phép các hàm này truy xuất được tới các thành phần **private** của lớp giống như các hàm thành phần của lớp đó. Ưu điểm của phương pháp này là kiểm soát các truy nhập ở cấp độ lớp: không thể áp đặt hàm bạn cho một lớp nếu điều đó không được dự trù trước trong khai báo của lớp. Điều này có thể ví như việc cấp thẻ ra vào ở một số cơ quan; không phải ai muốn đều được cấp thẻ mà chỉ những người có quan hệ đặc biệt với cơ quan mới được cấp.

Có nhiều kiểu bạn bè:

24. Hàm tự do là bạn của một lớp.
25. Hàm thành phần của một lớp là bạn của một lớp khác.
26. Hàm bạn của nhiều lớp.
27. Tất cả các hàm thành phần của một lớp là bạn của một lớp khác.

Sau đây chúng ta sẽ xem xét cụ thể cách khai báo, định nghĩa và sử dụng một hàm bạn cùng các tình huống đã nêu ở trên.

7.2 Hàm tự do bạn của một lớp

Trở lại ví dụ định nghĩa hàm `point::coincide()` kiểm tra sự trùng nhau của hai đối tượng kiểu `point`. Trước đây chúng ta định nghĩa nó như một hàm thành phần của lớp `point`:

```
class point {
int x,y;
public:
...
int coincide (point p); };
```

Ở đây còn có một cách khác định nghĩa `coincide` như một hàm tự do bạn của lớp `point`. Trước hết, cần phải đưa ra trong lớp `point` khai báo bạn bè:

```
friend int coincide(point , point);
```

Trong trường hợp hàm `coincide()` này là hàm tự do và không còn là hàm thành phần của lớp `point` nên chúng ta phải dự trù hai tham số kiểu `point` cho `coincide`. Việc định nghĩa hàm `coincide` giống như một hàm thông thường. Sau đây là một ví dụ của chương trình:

Ví dụ 3.18

```
/*friend1.cpp*/
#include <iostream.h>
class point {
int x, y;
public:
point(int abs =0, int ord =0) {
x = abs;y = ord;
}
friend int coincide (point,point);
};
```

```
int coincide (point p, point q) {
if ((p.x == q.x) && (p.y == q.y)) return 1;
else return 0;
}
void main() {
point a(1,0),b(1),c;
if (coincide (a,b)) cout <<"a trung voi b\n";
else cout<<"a va b khac nhau\n";
if (coincide(a,c)) cout<<"a trung voi c\n";
else cout<<"a va c khac nhau\n";
}
```

```
a trung voi b
a va c khac nhau
```

Nhận xét

28. Vị trí của khai báo "bạn bè" trong lớp `point` hoàn toàn tuỳ ý.

29. Trong hàm bạn, không còn tham số ngầm định **this** như trong hàm thành phần.

Cũng giống như các hàm thành phần khác danh sách "tham số " của hàm bạn gắn với định nghĩa chồng các toán tử. Hàm bạn của một lớp có thể có một hay nhiều tham số, hoặc có giá trị trả về thuộc kiểu lớp đó. Tuy rằng điều này không bắt buộc.

Có thể có các hàm truy xuất đến các thành phần riêng của các đối tượng cục bộ trong hàm. Khi hàm bạn của một lớp trả giá trị thuộc lớp này, thường đó là giá trị dữ liệu của đối tượng cục bộ bên trong hàm, việc truyền tham số phải thực hiện bằng tham trị, bở lẽ truyền bằng tham chiếu (hoặc bằng địa chỉ) hàm gọi sẽ nhận địa chỉ của một vùng nhớ bị giải phóng khi hàm kết thúc.

7.3 Các kiểu bạn bè khác

7.3.1 Hàm thành phần của lớp là bạn của lớp khác

Có thể xem đây như là một trường hợp đặc biệt của tình huống trên, chỉ khác ở cách mô tả hàm. Người ta sử dụng tên đầy đủ của hàm thành phần bao gồm tên lớp, toán tử phạm vi và tên hàm thành phần bạn bè.

Giả thiết có hai lớp A và B, trong B có một hàm thành phần f khai báo như sau:

```
int f(char , A);
```

Nếu f có nhu cầu truy xuất vào các thành phần riêng của A thì f cần phải được khai báo là bạn của A ở trong lớp A bằng câu lệnh:

```
friend int B::f(char , A);
```

Ta có các nhận xét quan trọng sau:

để biên dịch được các khai báo của lớp A có chứa khai báo bạn bè kiểu như:

```
friend int B::f(char, A);
```

chương trình dịch cần phải biết được nội dung của lớp B; nghĩa là khai báo của B (không nhất thiết định nghĩa của các hàm thành phần) phải được biên dịch trước khai báo của A.

Ngược lại, khi biên dịch khai báo:

```
int f(char, A) ;
```

bên trong lớp B, chương trình dịch không nhất thiết phải biết chi tiết nội dung của A, nó chỉ cần biết rằng là một lớp. Để có được điều này ta dùng chỉ thị sau:

```
class A;
```

trước khai báo lớp B. Việc biên dịch định nghĩa hàm f cần các thông tin đầy đủ về các thành phần của A và B; như vậy các khai báo của A và B phải có trước định nghĩa đầy đủ của f. Tóm lại, sơ đồ khai báo và định nghĩa phải như sau:

```
class A;
class B {
...
int f(char, A);
...
}; class A {
...
friend int B::f(char, A);
...
}; int B::f(char ...,A ...) {
...
}
```

Đề nghị bạn đọc thử nghiệm trường hợp "bạn bè chéo nhau", nghĩa là đồng thời hàm thành phần của lớp này là bạn của lớp kia và một hàm thành phần của lớp kia là bạn của lớp này.

7.3.2 Hàm bạn của nhiều lớp

Về nguyên tắc, mọi hàm (hàm tự do hay hàm thành phần) đều có thể là bạn của nhiều lớp khác nhau. Sau đây là một ví dụ một hàm là bạn của hai lớp A và B.

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

Nhận xét

Ở đây, khai báo của A có thể được biên dịch không cần khai báo của B nếu có chỉ thị `class B;` đứng trước.

Tương tự, khai báo của lớp B cũng được biên dịch mà không cần đến A nếu có chỉ thị `class A;` đứng trước.

Nếu ta muốn biên dịch cả hai khai báo của A và của B, thì cần phải sử dụng một trong hai chỉ thị đã chỉ ra ở trên. Còn định nghĩa của hàm f cần phải có đầy đủ cả hai khai báo của A và của B đứng trước. Sau đây là một ví dụ minh hoạ:

```
class B;
class A {
...
friend void f(A, B);
...
};
class B {
```

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

7.3.3 Tất cả các hàm của lớp là bạn của lớp khác

Đây là trường hợp tổng quát trong đó có thể khai báo lớp bạn bè với các hàm. Mọi vấn đề sẽ đơn giản hơn nếu ta đưa ra một khai báo tổng thể để nói rằng tất cả các hàm thành phần của lớp B là bạn của lớp A. Muốn vậy ta sẽ đặt trong khai báo lớp A chỉ thị:

```
friend class B;
```

Nhận xét: Trong trường hợp này, để biên dịch khai báo của lớp A, chỉ cần đặt trước nó chỉ thị: class B; kiểu khai báo lớp bạn cho phép không phải khai báo tiêu đề của các hàm có liên quan.

7.4 Bài toán nhân ma trận với vector

Trong phần này ta sẽ giải quyết bài toán xây dựng một hàm nhân ma trận (đối tượng thuộc lớp `matrix`) với vector (đối tượng thuộc lớp `vect`). Để đơn giản, ta giới hạn chỉ có các hàm thành phần:

(i) một `constructor` cho `vect` và `matrix`,

(ii)một hàm hiển thị (`display`) cho `vect`.

Ta trình bày hai giải pháp dựa trên việc định nghĩa `prod` có hai đối số, một là đối tượng `matrix` và một là đối tượng `vect`:

30. `prod` là hàm tự do và là bạn của hai lớp `vect` và `matrix`,

31. `prod` là hàm thành phần của `matrix` và là bạn của `vect`.

Giải pháp thứ nhất - prod là hàm bạn tự do

Ví dụ 3.19

```
/*vectmat1.cpp*/
```

```
#include <iostream.h>
class matrix;
//*****class vect
class vect {
double v[3]; //vector có ba thành phần
public:
vect (double v1=0, double v2=0, double v3=0)
{ v[0] = v1; v[1] = v2; v[2] = v3;
}
friend vect prod(matrix, vect);
void display () {
int i;
for (int i=0; i<3; i++) cout<<v[i]<<" ";
cout<<endl;
}
};
//*****class matrix
class matrix {
double mat[3][3];
public:
matrix(double t[3][3])
{ int i; int j;
for(i=0; i<3; i++)
for(j=0; j<3; j++)
mat[i][j] = t[i][j];
}
friend vect prod (matrix, vect);
};
```

```
//*****Hàm prod
vect prod (matrix m, vect x) {
int i,j;
double sum;
vect res; //kết quả
for(i=0; i<3; i++) {
for(j=0,sum=0; j<3; j++)
sum +=m.mat[i][j]*x.v[j];
res.v[i] = sum;
}
return res;
}
//**** chương trình kiểm tra
void main() {
vect w91,2,3);
vect res;
double tb[3][3] = {1,2,3,4,,5,6,7,8,9};
matrix a=tb;
res=prod(a,w); res.display();
}
```

```
14 32 50
```

Giải pháp thứ hai- prod là hàm thành phần của lớp matrix và là bạn của vect

Ví dụ 3.20

```
/*vectmat2.cpp*/
#include <iostream.h>
class vect;
```

```
//***class matrix
class matrix {
double mat[3][3];
public:
matrix(double t[3][3])
{ int i; int j;
for(i=0; i<3; i++)
for(j=0; j<3; j++)
mat[i][j] = t[i][j];
}
vect prod (vect);
};
//*** class vect
class vect {
double v[3];
public:
vect (double v1=0, double v2=0, double v3=0)
{ v[0] = v1; v[1] = v2; v[2] = v3;
}
friend vect matrix::prod(vect);
void display () {
int i;
for (int i=0; i<3; i++) cout<<v[i]<<" ";
cout<<endl;
}
};
//***Hàm matrix::prod
vect matrix::prod (vect x) {
```

```
int i,j;
double sum;
vect res;
for(i=0; i<3; i++) {
for(j=0,sum=0; j<3; j++)
sum +=mat[i][j]*x.v[j];
res.v[i] = sum;
}
return res;
}
//*** chương trình kiểm tra
void main() {
vect w91,2,3);
vect res;
double tb[3][3] = {1,2,3,4,,5,6,7,8,9};
matrix a=tb;
res=a.prod(w); res.display();
}
```

```
14 32 50
```

8. VÍ DỤ TỔNG HỢP

Chương trình `vectmat3.cpp` sau đây được phát triển dựa trên các chương trình `vector2.cpp` , `vectmat1.cpp` và `vectmat2.cpp`.

Ví dụ 3.21

```
/*vectmat2.cpp*/
#include <iostream.h>
#include <conio.h>
/*lớp vector*/
```

```
class matrix;/*khai báo trước lớp matrix*/
/*khai báo lớp vector*/
class vector{
      static int n;   //số chiều của vector
      float *v;     //vùng nhớ chứa các toạ độ
  public:
      vector();
      vector(float *);
      vector(vector &);//hàm thiết lập sao chép
      ~vector();
      void display();
      static int & Size() {return n;}
      friend vector prod(matrix &, vector &);
      friend class matrix;
  };
int vector::n = 0;
/*các hàm thành phần của lớp vector*/
vector::vector()
  {
  int i;
  v= new float [n];
  for(i=0;i<n;i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>v[i];
    }
  }
vector::vector(float *a) {
  for(int i =0; i<n; i++)
```

```
    v[i]=a[i];
  }
vector::vector(vector &b)
  {
  int i;
  for(i=0;i<n;i++)
    v[i] = b.v[i];
  }
vector::~vector(){
  delete v;
  }
void vector::display()   //hiển thị kết quả
{
  for(int i=0;i<n;i++)
    cout <<v[i] <<" ";
  cout <<"\n";
}
/*khai báo lớp matrix*/
class matrix{
      static int n;   //số chiều của vector
      vector *m;     //vùng nhớ chứa các toạ độ
  public:
      matrix();
      ~matrix();
      void display();
      static int &Size() {return n;}
      friend vector prod(matrix &, vector &);
  };
```

```
int matrix::n =0;
/*hàm thành phần của lớp matrix*/
matrix::matrix(){
  int i;
  m= new vector [n];
  }
matrix::~matrix() {
  delete m;
  }

void matrix::display()   //hiển thị kết quả
{
for (int i=0; i<n; i++)
  m[i].display();
}
/*hàm prod*/
vector prod(matrix &m,vector &v) {
  float *a = new float [vector::Size()];
  int i,j;
  for (i=0; i<matrix::Size(); i++) {
    a[i]=0;
    for(j=0; j<vector::Size(); j++)
      a[i]+=m.m[i].v[j]*v.v[j];
    }
  return vector(a);
  }
void main()
  {
```

```
 clrscr();
 int size;
 cout<<"Kich thuoc cua vector "; cin>>size;
 vector::Size() = size;
 matrix::Size() = size;
 cout<<"Tao mot vector \n";
 vector v;
 cout<<" v= \n";
 v.display();
 cout<<"Tao mot ma tran \n";
 matrix m;
 cout<<" m = \n";
 m.display();
 cout<<"Tich m*v \n";
 vector u = prod(m,v);
 u.display();
 getch();
}
```

```
Kich thuoc cua vector 3
Tao mot vector
Toa do thu 1 : 1
Toa do thu 2 : 1
Toa do thu 3 : 1
 v=
1 1 1
Tao mot ma tran
Toa do thu 1 : 2
Toa do thu 2 : 3
```

```
Toa do thu 3 : 2
Toa do thu 1 : 1
Toa do thu 2 : 2
Toa do thu 3 : 3
Toa do thu 1 : 2
Toa do thu 2 : 3
Toa do thu 3 : 2
 m =
2 3 2
1 2 3
2 3 2
Tich m*v
7 6 7
```

9. TÓM TẮT

9.1 Ghi nhớ

Trong C++, tên cấu trúc là một kiểu dữ liệu không cần phải kèm theo từ khoá **struct**.

Lớp cho phép người lập trình mô tả các đối tượng thực tế với các thuộc tính và hành vi. Trong C++ thường sử dụng từ khoá **class** để khai báo một lớp. Tên lớp là một kiểu dữ liệu dùng khi khai báo các đối tượng thuộc lớp (các thể hiện cụ thể của lớp).

Thuộc tính của đối tượng trong một lớp được mô tả dưới dạng các biến thể hiện. Các hành vi là các hàm thành phần bên trong lớp.

Có hai cách định nghĩa các hàm thành phần của một lớp; khi định nghĩa hàm thành phần bên ngoài khai báo lớp phải đặt trước tên hàm thành phần tên của lớp và toán tử "::" để phân biệt với các hàm tự do cùng tên. Chỉ nên định nghĩa hàm thành phần bên trong lớp khi nó không quá phức tạp để cho chương trình dễ đọc.

Có thể khai báo và sử dụng các con trỏ đối tượng, tham chiếu đối tượng.

Hai từ khoá **public** và **private** dùng để chỉ định thuộc tính truy nhập cho các thành phần (dữ liệu/hàm) khai báo bên trong lớp.

Thành phần bên trong lớp được khai báo **public** có thể truy nhập từ mọi hàm khai báo một đối tượng thuộc lớp đó.

Thành phần **private** trong một đối tượng chỉ có thể truy nhập được bởi các hàm thành phần của đối tượng hoặc các hàm thành phần của lớp dùng để tạo đối tượng (ở đây tính cả trường hợp đối tượng là tham số của hàm thành phần).

Hai hàm thành phần đặc biệt của một lớp gọi là hàm thiết lập và hàm huỷ bỏ. Hàm thiết lập được gọi tự động (ngầm định) mỗi khi một đối tượng được tạo ra và hàm huỷ bỏ được gọi tự động khi đối tượng hết thời gian sử dụng.

Hàm thiết lập có thuộc tính **public**, cùng tên với tên lớp nhưng không có giá trị trả về.

Một lớp có ít nhất hai hàm thiết lập: hàm thiết lập sao chép ngầm định và hàm thiết lập do người lập trình thiết lập (nếu không được mô tả tường minh thì đó là hàm thiết lập ngầm định).

Hàm huỷ bỏ cũng có thuộc tính **public**, không tham số, không giá trị trả về và có tên bắt đầu bởi ~ theo sau là tên của lớp.

Bên trong phạm vi lớp (định nghĩa của các hàm thành phần), các thành phần của lớp được gọi theo tên. Trường hợp có một đối tượng toàn cục cùng tên, muốn xác định đối tượng ấy phải sử dụng toán tử "**::**"

Lớp có thể chứa các thành phần dữ liệu là các đối tượng có kiểu lớp khác. Các đối tượng này phải được khởi tạo trước đối tượng tương ứng của lớp bao.

Mỗi đối tượng có một con trỏ chỉ đến bản thân nó, ta gọi đó là con trỏ **this**. Con trỏ này có thể được sử dụng tường minh hoặc ngầm định để tham xác định các thành phần bên trong đối tượng. Thông thường người ta sử dụng this dưới dạng ngầm định.

Toán tử **new** tự động tạo ra một đối tượng với kích thước thích hợp và trả về con trỏ có kiểu lớp. Để giải phóng vùng nhớ cấp phát cho đối tượng này sử dụng toán tử **delete**.

Thành phần dữ liệu tĩnh biểu thị các thông tin dùng chung trong tất cả các đối tượng thuộc lớp. Khai báo của thành phần tĩnh bắt đầu bằng từ khoá **static**.

Có thể truy nhập tới các thành phần tĩnh thông qua các đối tượng kiểu lớp hoặc bằng tên lớp nhờ sử dụng toán tử phạm vi.

Hàm thành phần có thể được khai báo là tĩnh nếu nó chỉ truy nhập đến các thành phần dữ liệu tĩnh.

Hàm bạn của một lớp là hàm không thuộc lớp nhưng có quyền truy nhập tới các thành phần private của lớp.

Khai báo bạn bè có thể đặt bất kỳ nơi nào trong khai báo lớp.

9.2 Các lỗi thường gặp

Quên dấu ";" ở cuối khai báo lớp.

Khởi tạo giá trị cho các thành phần dữ liệu trong khai báo lớp.

Định nghĩa chồng một hàm thành phần bằng một hàm không thuộc lớp.

Truy nhập đến các thành phần riêng của lớp từ bên ngoài phạm vi lớp

Khai báo giá trị trả về cho hàm thiết lập và hàm huỷ bỏ.

Khai báo hàm huỷ bỏ có tham số, định nghĩa chồng hàm huỷ bỏ.

Gọi tường minh hàm thiết lập và hàm huỷ bỏ.

Gọi các hàm thành phần bên trong hàm thiết lập

Định nghĩa một hàm thành phần **const** thay đổi các thành phần dữ liệu của một đối tượng.

Định nghĩa một hàm thành phần const gọi tới một hàm thành phần không phải **const**.

Gọi các hàm thành phần không phải const từ các đối tượng **const**.

Thay đổi nội dung một đối tượng **const**.

Nhầm lẫn giữa **new** và **delete** với `malloc` và `free`.

Sử dụng this bên trong các hàm thành phần tĩnh.

9.3 Một số thói quen lập trình tốt

Nhóm tất cả các thành phần có cùng thuộc tính truy nhập ở một nơi trong khai báo lớp, nhờ vậy mỗi từ khoá mô tả truy nhập chỉ được xuất hiện một lần. Khai báo lớp vì vậy dễ đọc hơn. Theo kinh nghiệm, để các thành phần **private** trước tiên rồi đến các thành phần **protected**, cuối cùng là các thành phần **public**.

Định nghĩa tất cả các hàm thành phần bên ngoài khai báo lớp. Điều này nhằm phân biệt giữa hai phần giao diện và phần cài đặt của lớp.

Sử dụng các chỉ thị tiền xử lý `#ifndef`, `#define`, `#endif` để cho các tập tin tiêu đề chỉ xuất hiện một lần bên trong các chương trình nguồn.

Phải định nghĩa các hàm thiết lập để đảm bảo rằng các đối tượng đều được khởi tạo nội dung một cách đúng đắn.

Khai báo là **const** tất cả các hàm thành phần chỉ để sử dụng với các đối tượng **const**.

Nên sử dụng **new** và **delete** thay vì `malloc` và `free`.

10. BÀI TẬP

Bài tập 3.1

So sánh ý nghĩa của struct và class trong C++

Bài tập 3.2

Tạo một lớp gọi là Complex để thực hiện các thao tác số học với các số phức. Viết một chương trình để kiểm tra lớp này.

Số phức có dạng

<Phần thực> + <Phần ảo>*j

Sử dụng các biến thực để biểu diễn các thành phần dữ liệu riêng của lớp. Cung cấp một hàm thiết lập để tạo đối tượng. Hàm thiết lập sử dụng các tham số có giá trị ngầm định. Ngoài ra còn có các hàm thành phần **public** để thực hiện các công việc sau:

+ CỘNG HAI SỐ PHỨC: CÁC PHẦN THỰC ĐƯỢC CỘNG VỚI NHAU VÀ CÁC PHẦN ẢO ĐƯỢC CỘNG VỚI NHAU.
+ TRỪ HAI SỐ PHỨC: PHẦN THỰC CỦA SỐ PHỨC THỨ HAI ĐƯỢC TRỪ CHO PHẦN THỰC CỦA SỐ PHỨC THỨ NHẤT. TƯƠNG TỰ CHO PHẦN ẢO.
+ IN SỐ PHỨC RA MÀN HÌNH DƯỚI DẠNG (A, B) TRONG ĐÓ A LÀ PHẦN THỰC VÀ B LÀ PHẦN ẢO.

Bài tập 3.3

Tạo một lớp gọi là PS để thực hiện các thao tác số học với phân số. Viết chương trình để kiểm tra lớp vừa tạo ra.

Sử dụng các biến nguyên để biểu diễn các thành phần dữ liệu của lớp-tử số và mẫu số. Viết định nghĩa hàm thiết lập để tạo đối tượng sao cho phần ảo phải là số nguyên dương. Ngoài ra còn có các hàm thành phần khác thực hiện các công việc cụ thể:

+ CỘNG HAI PHÂN SỐ. KẾT QUẢ PHẢI ĐƯỢC TỐI GIẢN.
+ TRỪ HAI PHÂN SỐ. KẾT QUẢ PHẢI ĐƯỢC TỐI GIẢN.
+ NHẬN HAI PHÂN SỐ. KẾT QUẢ DƯỚI DẠNG TỐI GIẢN.
+ CHIA HAI PHÂN SỐ. KẾT QUẢ DƯỚI DẠNG TỐI GIẢN.
+ IN RA MÀN HÌNH PHÂN SỐ DƯỚI DẠNG A/B TRONG ĐÓ A LÀ TỬ SỐ, CÒN B LÀ MẪU SỐ.
+ IN PHÂN SỐ DƯỚI DẠNG SỐ THẬP PHÂN.

Bài tập 3.4

Khai báo, định nghĩa và sử dụng lớp time mô tả các thông tin về giờ, phút và giây với các yêu cầu như sau:

Tạo tập tin tiêu đề TIME.H chứa khai báo của lớp time với các thành phần dữ liệu mô tả giờ, phút và giây: hour, minute,second.

Trong lớp time khai báo :

+ MỘT HÀM THIẾT LẬP NGẦM ĐỊNH, DÙNG ĐỂ GÁN CHO CÁC THÀNH PHẦN DỮ LIỆU GIÁ TRỊ 0.
+ HÀM THÀNH PHẦN set(int, int,int) VỚI BA THAM SỐ TƯƠNG ỨNG MANG GIÁ TRỊ CỦA BA THÀNH PHẦN DỮ LIỆU.
+ HÀM HIỂN THỊ TRONG ĐÓ GIỜ ĐƯỢC HIỂN THỊ VỚI GIÁ TRỊ 0 TỚI 24.
+ HÀM HIỂN THỊ CHUẨN CÓ PHÂN BIỆT GIỜ TRƯỚC VÀ SAU BUỔI TRƯA.

Tạo tập tin chương trình TIME.CPP chứa định nghĩa của các hàm thành phần trong lớp time, và chương trình minh hoạ cách sử dụng lớp time.

Bài tập 3.5

Tương tự như bài 3.4 nhưng ở đây hàm thiết lập có ba tham số có giá trị ngầm định bằng 0.

Bài tập 3.6

Vẫn dựa trên lớp `time`, nhưng ở đây ta bổ sung thêm các hàm thành phần để thiết lập riêng rẽ giờ, phút, giây:

+ `void setHour(int)`
+ `void setMiniute(int)`
+ `void setSecond(int)`

Đồng thời có các hàm để lấy từng giá trị đó của từng đối tượng:

+ `int getHour()`
+ `int getMiniute();`
+ `int getSecond()`

Bài tập 3.7

Thêm một hàm thành phần `tick()` vào lớp `time` để tăng thời gian trong một đối tượng `time` mỗi lần một giây. Lưu ý các trường hợp, tăng sang phút tiếp theo, tăng sang giờ tiếp theo, tăng sang ngày tiếp theo.

Bài tập 3.8

Khai báo lớp `date` mô tả thông tin về ngày tháng năm: `month, day,year`.

Lớp date có hai hàm thành phần:

+ HÀM THIẾT LẬP VỚI BA THAM SỐ CÓ GIÁ TRỊ NGẦM ĐỊNH.
+ HÀM `print()` IN THÔNG TIN VỀ NGÀY THÁNG DƯỚI DẠNG QUEN THUỘC `mm-dd-yy`.

Viết chương trình kiểm ta việc sử dụng phép gán cho các đối tượng thuộc lớp date.

Bài tập 3.9

Dựa trên lớp `date` của bài 3.8 người ta thực hiện một số thay đổi để kiểm soát lỗi trên giá trị các tham số của hàm thiết lập. Đồng thời bổ sung thêm hàm thành phần `nextDay`() để tăng `date` từng ngày một.

Bài tập 3.10

Kết hợp lớp `time` trong bài 3.7 và lớp `date` trong bài 3.9 để tạo nên một lớp `date_time` mô tả đồng thời thông tin về ngày, giờ. Thay đổi hàm thành phần `tick`() để gọi tới hàm tăng ngày mỗi khi cần thiết. Thêm các hàm hiển thị thông tin về ngày giờ. Hoàn thiện chương trình để kiểm tra hoạt động của lớp.

Bài tập 3.11

Viết chương trình khai báo lớp mô tả hoạt động của một ngăn xếp hoặc hàng đợi chứa các số nguyên.