CHƯƠNG 4

ĐỊNH NGHĨA TOÁN TỬ TRÊN LỚP
(CLASS OPERATORS)

Mục đích chương này :

1. Cách định nghĩa các phép toán cho kiểu dữ liệu lớp và cấu trúc
2. Các toán tử chuyển kiểu áp dụng cho kiểu dữ liệu lớp

1. GIỚI THIỆU CHUNG

Thực ra, vấn đề định nghĩa chồng toán tử đã từng có trong C, ví dụ trong biểu thức:

```
a + b
```

ký hiệu + tuỳ theo kiểu của a và b có thể biểu thị:

3. phép cộng hai số nguyên,
4. phép cộng hai số thực độ chính xác đơn (**float**)
5. phép cộng hai số thực chính xác đôi (**double**)
6. phép cộng một số nguyên vào một con trỏ.

Trong C++, có thể định nghĩa chồng đối với hầu hết các phép toán (một ngôi hoặc hai ngôi) trên các lớp, nghĩa là một trong số các toán hạng tham gia phép toán là các đối tượng. Đây là một khả năng mạnh vì nó cho phép xây dựng trên các lớp các toán tử cần thiết, làm cho chương trình được viết ngắn gọn dễ đọc hơn và có ý nghĩa hơn. Chẳng hạn, khi định nghĩa một lớp `complex` để biểu diễn các số phức, có thể viết trong C++: `a+b, a-b, a*b, a/b` với `a,b` là các đối tượng `complex`.

Để có được điều này, ta định nghĩa chồng các phép toán `+, -, *` và `/` bằng cách định nghĩa hoạt động của từng phép toán giống như định nghĩa một hàm, chỉ khác là đây là hàm toán tử (operator function). Hàm toán tử có tên được ghép bởi từ khoá **operator** và ký hiệu của phép toán tương ứng. Bảng 4.1 đưa ra một số ví dụ về tên hàm toán tử.

Hàm toán tử có thể dùng như là một hàm thành phần của một lớp hoặc là hàm tự do; khi đó hàm toán tử phải được khai báo là bạn của các lớp có các đối tượng mà hàm thao tác.

Tên hàm	Dùng để
operator+	định nghĩa phép +
operator*	định nghĩa phép nhân *
operator/	định nghĩa phép chia /
operator+=	định nghĩa phép tự cộng +=
operator!=	định nghĩa phép so sánh khác nhau

Bảng 4.1 Một số tên hàm toán tử quen thuộc

2. VÍ DỤ TRÊN LỚP SỐ PHỨC

2.1 Hàm toán tử là hàm thành phần

Trong chương trình `complex1.cpp` toán tử + giữa hai đối tượng complex được định nghĩa như một hàm thành phần. Hàm toán tử thành phần có một tham số ngầm định là đối tượng gọi hàm nên chỉ có một tham số tường minh.

Ví dụ 4.1

```
/*complex1.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
  float real, image;
  public:
  complex(float r=0, float i =0) {
     real = r; image = i;
     }
  void display() {
     cout<<real<<(image>=0?'+':'-')<<"j*"<<fabs(image)<<endl ;
     }
```

```
 /*hàm operator+ định nghĩa phép toán + hai ngôi trên lớp số phức complex*/
  complex operator+(complex b) {
    complex c;
    c.real = a.real+b.real;
    c.image =a.image+b.image;
    return c;
    }
  };
void main() {
  clrscr();
  complex a(-2,5);
  complex b(3,4);
  cout<<"Hai so phuc:\n";
  a.display();
  b.display();
  cout<<"Tong hai so phuc:\n";
  complex c;
  c=a+b;//a.operator+(b)
  c.display();
  getch();
  }
```

```
Hai so phuc:
-2+j*5
3+j*4
Tong hai so phuc:
1+j*9
```

Chỉ thị

c = a+b;

trong ví dụ trên được chương trình dịch hiểu là:

c = a.operator+(b);

Nhận xét

7. Thực ra cách viết a+b chỉ là một quy ước của chương trình dịch cho phép người sử dụng viết gọn lại, nhờ đó cảm thấy tự nhiên hơn.
8. Hàm toán tử operator+ phải có thuộc tính **public** vì nếu không chương trình dịch không thể thực hiện được nó ở ngoài phạm vi lớp.
9. Trong lời gọi a.operator+(b), a đóng vai trò của tham số ngầm định của hàm thành phần và b là tham số tường minh. Số tham số tường minh cho hàm toán tử thành phần luôn ít hơn số ngôi của phép toán là 1 vì có một tham số ngầm định là đối tượng gọi hàm toán tử.
10. Chương trình dịch sẽ không thể hiểu được biểu thức 3+a vì cách viết tương ứng 3.operator(a) không có ý nghĩa. Để giải quyết tình huống này ta dùng hàm bạn để định nghĩa hàm toán tử.

2.2 Hàm toán tử là hàm bạn

Chương trình complex2.cpp được phát triển từ complex1.cpp bằng cách thêm hàm toán tử cộng thêm một số thực **float** vào phần thực của một đối tượng complex, được biểu thị bởi phép cộng với số thực **float** là toán hạng thứ nhất, còn đối tượng complex là toán hạng thứ hai. Trong trường hợp này không thể dùng phép cộng như hàm thành phần vì tham số thứ nhất của hàm toán tử không còn là một đối tượng.

Ví dụ 4.2

```
/*complex2.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
  public:
  complex(float r=0, float i =0) {
    real = r; image = i;
```

```
    }
  void display() {
    cout<<real<<(image>=0?'+':'-')<<"j*"<<fabs(image)<<endl ;
    }
 /*hàm thành phần operator+ định nghĩa phép toán + hai ngôi trên lớp số phức complex*/
  complex operator+(complex b) {
    cout<<"Goi toi complex::operator+(float, complex)\n";
    complex c;
    c.real = real+b.real;
    c.image =image+b.image;
    return c;
    }
/*hàm tự do operator+ định nghĩa phép toán + giữa một số thực và một đối tượng số phức*/
  friend complex operator+(float x, complex b);
  };
complex operator+(float x, complex b) {
  cout<<"Goi toi operator+(float, complex)\n";
  complex c;
  c.real = x+b.real;
  c.image = b.image;
  return c;
  }
void main() {
  clrscr();
  complex a(-2,5);
  complex b(3,4);
  cout<<"Hai so phuc:\n";
```

```
cout<<"a = ";
a.display();
cout<<"b = ";
b.display();
cout<<"Tong hai so phuc:\n";
complex c;
c=a+b;  //a.operator+(b);
cout<<"c = ";
c.display();
cout<<"Tang them phan thuc cua a 3 don vi\n";
complex d;
d=3+a; //operator+(3,a);
cout<<"d = ";
d.display();
getch();
}
```

```
Hai so phuc:
a = -2+j*5
b = 3+j*4
Tong hai so phuc:
Goi toi complex::operator+(complex)
c = 1+j*9
Tang them phan thuc cua a 3 don vi
Goi toi operator+(float, complex)
d = 1+j*5
```

Trong chương trình trên, biểu thức `a+b` được chương trình hiểu là lời gọi hàm thành phần `a.operator+(b)`, trong khi đó với biểu thức `3+a`, chương trình dịch sẽ thực hiện lời gọi hàm tự do `operator+(3,a)`.

Số tham số trong hàm toán tử tự do `operator+(...)` đúng bằng số ngôi của phép + mà nó định nghĩa. Trong định nghĩa của hàm toán tử tự do, tham số thứ nhất có thể có kiểu bất kỳ chứ không nhất thiết phải có kiểu lớp nào đó.

Với một hàm `operator+` nào đó chỉ có thể thực hiện được phép + tương ứng giữa hai toán hạng có kiểu như đã được mô tả trong tham số hình thức, nghĩa là muốn có được phép cộng "vạn năng" áp dụng cho mọi kiểu toán hạng ta phải định nghĩa rất nhiều hàm toán tử `operator+` (định nghĩa chồng các hàm toán tử).

Vấn đề bảo toàn các tính chất tự nhiên của các phép toán không được C++ đề cập, mà nó phụ thuộc vào cách cài đặt cụ thể trong chương trình dịch C++ hoặc bản thân người sử dụng khi định nghĩa các hàm toán tử. Chẳng hạn, phép gán:

```
c = a+b;
```

được chương trình dịch hiểu như là: `c = a.operator+(b);` trong khi đó với phép gán:

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

ngôn ngữ C++ không đưa ra diễn giải nghĩa duy nhất. Một số chương trình biên dịch sẽ tạo ra đối tượng trung gian t:

```
t=a.operator+(b);
```

và

```
d=t.operator+(c);
```

Chương trình `complex3.cpp` sau đây minh hoạ lý giải này:

Ví dụ 4.3

```
/*complex3.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
  public:
  complex(float r=0, float i =0) {
    cout<<"Tao doi tuong :"<<this<<endl;
    real = r; image = i;
    }
  void display() {
    cout<<real<<(image>=0?'+':'-')<<"j*"<<fabs(image)<<endl ;
    }
complex  operator+(complex b) {
    cout<<"Goi toi complex::operator+(complex)\n";
    cout<<this<<endl;
    complex c;
    c.real=real+b.real;
    c.image=image+b.image;
    return c;
    }
  friend complex operator+(float x, complex b);
  };
complex operator+(float x, complex b) {
  cout<<"Goi toi operator+(float, complex)\n";
  complex c;
```

```
  c.real = x+b.real;
  c.image = b.image;
  return c;
  }
void main() {
  clrscr();
  cout<<"so phuc a \n";
  complex a(-2,5);
  cout<<"so phuc b \n";
  complex b(3,4);
  cout<<"Hai so phuc:\n";
  cout<<"a = ";
  a.display();
  cout<<"b = ";
  b.display();
  complex c(2,3);
  cout<<"Cong a+b+c\n";
  cout<<"so phuc d \n";
  complex d;
  d = a+b+c;
  cout<<"a = ";a.display();
  cout<<"b = ";b.display();
  cout<<"c = ";c.display();
  cout<<"d = a+b+c : ";
  d.display();
  getch();
  }
```

```
so phuc a
```

```
Tao doi tuong :0xffee
so phuc b
Tao doi tuong :0xffe6
Hai so phuc:
a = -2+j*5
b = 3+j*4
Tao doi tuong :0xffde
Cong a+b+c
so phuc d
Tao doi tuong :0xffd6
Goi toi complex::operator+(complex)
0xffee
Tao doi tuong :0xffa0
Goi toi complex::operator+(complex)
0xffce
Tao doi tuong :0xffa8
a = -2+j*5
b = 3+j*4
c = 2+j*3
d = a+b+c : 3+j*12
```

Cũng có thể làm như sau: trong định nghĩa của hàm toán tử, ta trả về tham chiếu đến một trong hai đối tượng tham gia biểu thức (chẳng hạn a). Khi đó `a+b+c` được hiểu là `a.operator+(b)` và sau đó là `a.operator+(c)`. Tất nhiên trong trường hợp này nội dung của đối tượng `a` bị thay đổi sau mỗi phép cộng. Xét chương trình sau:

Ví dụ 4.4

```
/*complex4.cpp*/
#include <iostream.h>
#include <conio.h>
```

```
#include <math.h>
class complex {
    float real, image;
  public:
  complex(float r=0, float i =0) {
    cout<<"Tao doi tuong :"<<this<<endl;
    real = r; image = i;
    }
  void display() {
    cout<<real<<(image>=0?'+':'-')<<"j*"<<fabs(image)<<endl ;
    }
  complex & operator+(complex b) {
    cout<<"Goi toi complex::operator+(complex)\n";
    cout<<this<<endl;
    real+=b.real;
    image+=b.image;
    return *this;
    }
  friend complex operator+(float x, complex b);
  };
complex operator+(float x, complex b) {
  cout<<"Goi toi operator+(float, complex)\n";
  complex c;
  c.real = x+b.real;
  c.image = b.image;
  return c;
  }
void main() {
```

```
clrscr();
cout<<"so phuc a \n";
complex a(-2,5);
cout<<"so phuc b \n";
complex b(3,4);
cout<<"Hai so phuc:\n";
cout<<"a = ";
a.display();
cout<<"b = ";
b.display();
cout<<"so phuc c \n";
complex c;
c=a+b;   //a.operator+(b);
cout<<"c = a+b: ";
c.display();
cout<<"a = ";
a.display();
cout<<"Cong a+b+c\n";
cout<<"so phuc d \n";
complex d;
d = a+b+c;//a.operator+(b);a.operator+(c);
cout<<"a = ";a.display();
cout<<"b = ";b.display();
cout<<"c = ";c.display();
cout<<"d = a+b+c : ";
d.display();
getch();
}
```

```
so phuc a
Tao doi tuong :0xffee
so phuc b
Tao doi tuong :0xffe6
Hai so phuc:
a = -2+j*5
b = 3+j*4
so phuc c
Tao doi tuong :0xffde
Goi toi complex::operator+(complex)
0xffee
c = a+b: 1+j*9
a = 1+j*9
Cong a+b+c
so phuc d
Tao doi tuong :0xffd6
Goi toi complex::operator+(complex)
0xffee
Goi toi complex::operator+(complex)
0xffee
a = 5+j*22
b = 3+j*4
c = 1+j*9
d = a+b+c : 5+j*22
```

Trong hai ví dụ trên, việc truyền các đối số và giá trị trả về của hàm toán tử được thực hiện bằng giá trị. Với các đối tượng có kích thước lớn, người ta thường dùng tham chiếu để truyền đối cho hàm.

```
complex operator+(float , complex &);
```

Tuy nhiên việc dùng tham chiếu như là giá trị trả về của hàm toán tử, có nhiều điều đáng nói. Biểu thức nằm trong lệnh **return** bắt buộc phải tham chiếu đến một vùng nhớ tồn tại ngay cả khi thực hiện xong biểu thức tức là khi hàm toán tử kết thúc thực hiện. Vùng nhớ ấy có thể là một biến được cấp tĩnh **static** (các biến toàn cục hay biến cục bộ **static),** một biến thể hiện (một thành phần dữ liệu) của một đối tượng nào đó ở ngoài hàm. Bạn đọc có thể xem chương trình `vecmat3.cpp` trong chương 3 để hiểu rõ hơn. Vấn đề tương tự cũng được đề cập khi giá trị trả về của hàm toán tử là địa chỉ; trong trường hợp này, một đối tượng được tạo ra nhờ cấp phát động trong vùng nhớ heap dùng độc lập với vùng nhớ ngăn xếp dùng để cấp phát biến, đối tượng cục bộ trong chương trình, do vậy vẫn còn lưu lại khi hàm toán tử kết thúc công việc.

Hàm toán tử cũng có thể trả về kiểu `void` khi ảnh hưởng chỉ tác động lên một trong các toán hạng tham gia biểu thức. Xem định nghĩa của hàm đảo dấu số phức trong ví dụ sau:

Ví dụ 4.5

```
/*complex5.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
  public:
  complex(float r=0, float i =0) {
    real = r; image = i;
    }
  void display() {
    cout<<real<<(image>=0?'+':'-')<<"j*"<<fabs(image)<<endl ;
    }
  /*Hàm đảo dấu chỉ tác động lên toán hạng, không sử dụng được trong các
biểu thức */
  void operator-() {
```

```
    real = -real;
    image = -image;
    }
  complex  operator+(complex b) {
    complex c;
    c.real=real+b.real;
    c.image=image+b.image;
    return c;
    }
  friend complex operator+(float x, complex b);
  };
complex operator+(float x, complex b) {
  cout<<"Goi toi operator+(float, complex)\n";
  complex c;
  c.real = x+b.real;
  c.image = b.image;
  return c;
  }
void main() {
  clrscr();
  cout<<"so phuc a \n";
  complex a(-2,5);
  cout<<"so phuc b \n";
  complex b(3,4);
  cout<<"Hai so phuc:\n";
  cout<<"a = ";
  a.display();
  cout<<"b = ";
```

```
b.display();
complex c;
-a;
cout<<"a = ";a.display();
getch();
}
```

```
so phuc a
so phuc b
Hai so phuc:
a = -2+j*5
b = 3+j*4
a = 2-j*5
```

Chú ý:

Câu lệnh

```
complex c;
c=-a+b
```

sẽ gây lỗi vì `-a` có giá trị **void**.

3. KHẢ NĂNG VÀ GIỚI HẠN CỦA ĐỊNH NGHĨA CHỒNG TOÁN TỬ

Phần lớn toán tử trong C++ đều có thể định nghĩa chồng

Ký hiệu đứng sau từ khoá **operator** phải là một trong số các ký hiệu toán tử áp dụng cho các kiểu dữ liệu cơ sở, không thể dùng các ký hiệu mới. Một số toán tử không thể định nghĩa chồng (chẳng hạn toán tử truy nhập thành phần cấu trúc“.”, toán tử phạm vi “::”, toán tử điều kiện “? :”) và có một số toán tử ta phải tuân theo các ràng buộc sau:

(i) phép =, [] nhất định phải được định nghĩa như hàm thành phần của lớp.

(ii)phép << và >> dùng với **cout** và **cin** phải được định nghĩa như hàm bạn.

(iii) hai phép toán ++ và -- có thể sử dụng theo hai cách khác nhau ứng với dạng tiền tố ++a, --b và dạng hậu tố a++, b--. Điều này đòi hỏi hai hàm toán tử khác nhau.

Các toán tử được định nghĩa chồng phải bảo toàn số ngôi của chính toán tử đó theo cách hiểu thông thường, ví dụ: có thể định nghĩa toán tử "-" một ngôi và hai ngôi trên lớp tương ứng với phép đảo dấu (một ngôi) và phép trừ số học (hai ngôi), nhưng không thể định nghĩa toán tử gán một ngôi, còn ++ lại cho hai ngôi. Nếu làm vậy, chương trình dịch sẽ hiểu là tạo ra một ký hiệu phép toán mới.

Khi định nghĩa chồng toán tử, phải tuân theo nguyên tắc là ***Một trong số các toán hạng phải là đối tượng.*** Nói cách khác, hàm toán tử phải :

(iv) hoặc là hàm thành phần, khi đó, hàm đã có một tham số ngầm định có kiểu lớp chính là đối tượng gọi hàm. Tham số ngầm định này đóng vai trò toán hạng đầu tiên(đối với phép toán hai ngôi) hay toán hạng duy nhất (đối với phép toán một ngôi). Do vậy, nếu toán tử là một ngôi thì hàm toán tử thành phần sẽ không chứa một tham số nào khác. Ngược lại khi toán tử là hai ngôi, hàm sẽ có thêm một đối số tường minh.

(v) hoặc là một hàm tự do. Trong trường hợp này, ít nhất tham số thứ nhất hoặc tham số thứ hai (nếu có) phải có kiểu lớp.

Hơn nữa, mỗi hàm toán tử chỉ có thể áp dụng với kiểu toán hạng nhất định; cần chú ý rằng các tính chất vốn có, chẳng hạn tính giao hoán của toán tử không thể áp dụng một cách tuỳ tiện cho các toán tử được định nghĩa chồng. Ví dụ:

```
a+3.5
```

khác với

```
3.5+a
```

ở đây `a` là một đối tượng `complex` nào đó.

Cần lưu ý rằng không nên định nghĩa những hàm hàm toán tử khác nhau cùng làm những công việc giống nhau vì dễ xảy ra nhập nhằng. Chẳng hạn, đã có một hàm `operator+` là một hàm thành phần có tham số là đối tượng `complex` thì không được định nghĩa thêm một hàm `operator+` là một hàm tự do có hai tham số là đối tượng `complex`.

Trường hợp các toán tử ++ và --

Hàm cho dạng tiền tố	Hàm cho dạng hậu tố
operator++()	**operator++(int)**
operator--()	**operator--(int)**

Lưu ý rằng tham số int trong dạng hậu tố chỉ mang ý nghĩa tượng trưng (dump type)

Lựa chọn giữa hàm thành phần và hàm bạn

Phải tuân theo các quy tắc sau đây:

(vi) Lưu ý đến hạn chế của chương trình dịch, xem dạng nào được phép.

(vii) Nếu đối số đầu tiên là một đối tượng, có thể một trong hai dạng. Ngược lại phải dùng hàm bạn.

(viii) Trái lại, phải dùng hàm bạn.

4. CHIẾN LƯỢC SỬ DỤNG HÀM TOÁN TỬ

Về nguyên tắc, định nghĩa chồng một phép toán là khá đơn giản, nhưng việc sử dụng phép toán định nghĩa chồng lại không phải dễ dàng và đòi hỏi phải cân nhắc bởi lẽ nếu bị lạm dụng sẽ làm cho chương trình khó hiểu.

Phải làm sao để các phép toán vẫn giữ được ý nghĩa trực quan nguyên thuỷ của chúng. Chẳng hạn không thể định nghĩa cộng "+" như phép trừ "-" hai giá trị. Phải xác định trước ý nghĩa các phép toán trước khi viết định nghĩa của các hàm toán tử tương ứng.

Các phép toán một ngôi

Các phép toán một ngôi là:

`*, &, ~, !, ++, --,` **sizeof** (KIỂU)

Các hàm toán tử tương ứng chỉ có một đối số và phải trả về giá trị cùng kiểu với toán hạng, riêng **sizeof** có giá trị trả về kiểu nguyên không dấu và toán tử (kiểu) dùng để trả về một giá trị có kiểu như đã ghi trong dấu ngoặc.

Các phép toán hai ngôi

Các phép toán hai ngôi như:

`*,/,%,+,-,<<,>>,<,>,<=,>=,==,!=,&,|,^,&&,||`

Hai toán hạng tham gia các phép toán không nhất thiết phải cùng kiểu, mặc dù trong thực tế sử dụng thì thường là như vậy. Như vậy chỉ cần một trong hai đối số của hàm toán tử tương ứng là đối tượng là đủ.

Các phép gán

Các toán tử gán gồm có:

`=,+=,-=,*=,/=,%=,>>=,<<=,&=,^=,|=`

Do các toán tử gán được định nghĩa dưới dạng hàm thành phần, nên chỉ có một tham số tường minh và không có ràng buộc gì về kiểu đối số và kiểu giá trị trả về của các phép gán.

Toán tử truy nhập thành phần "->"

Phép toán này được dùng để truy xuất các thành phần của một cấu trúc hay một lớp và cần phân biệt với những cách sử dụng khác để tránh dẫn đến sự nhầm lẫn. Có thể định nghĩa phép toán lấy thành phần giống như đối với các phép toán một ngôi.

Toán tử truy nhập thành phần theo chỉ số

Toán tử lấy thành phần theo chỉ số được dùng để xác định một thành phần cụ thể trong một khối dữ liệu (cấp phát động hay tĩnh). Thông thường phép toán này được dùng với mảng, nhưng cũng có thể định nghĩa lại nó khi làm việc với các kiểu dữ liệu khác. Chẳng hạn với kiểu dữ liệu `vector` có thể định nghĩa phép lấy theo chỉ số để trả về một thành phần toạ độ nào đó `vector`. Và phải được định nghĩa như hàm thành phần có một đối số tường minh.

Toán tử gọi hàm

Đây là một phép toán thú vị nhưng nói chung rất khó đưa ra một ví dụ cụ thể.

5. MỘT SỐ VÍ DỤ TIÊU BIỂU

5.1 Định nghĩa chồng phép gán " ="

Việc định nghĩa chồng phép gán chỉ cần khi các đối tượng có các thành phần dữ liệu động (chương 3 đã đề cập vấn đề này). Chúng ta xét vấn đề này qua phân tích định nghĩa chồng phép gán "=" áp dụng cho lớp `vector`.

Điểm đầu tiên cần lưu ý là hàm `operator=` nhất thiết phải được định nghĩa như là hàm thành phần của lớp `vector`. Như vậy hàm `operator=` sẽ chỉ có một tham số tường minh (toán hạng bên phải dấu =).

Giả sử `a` và `b` là hai đối tượng thuộc lớp `vector`, khi đó

```
a=b;
```

được hiểu là

```
a.operator=(b);
```

do đó b được truyền cho hàm dưới dạng tham trị hoặc tham chiếu. Việc truyền bằng tham trị đòi hỏi sự có mặt của hàm thiết lập sao chép, hơn thế nữa sẽ làm cho chương trình chạy chậm vì mất thời gian sao chép một lượng lớn dữ liệu. Vì vậy, b sẽ được truyền cho hàm operator= dưới dạng tham chiếu.

Giá trị trả về của hàm operator= phụ thuộc vào mục đích sử dụng của biểu thức gán. Chúng ta chọn giải pháp trả về tham chiếu của đối tượng đứng bên trái dấu bằng nhằm giữ hai tính chất quan trong của biểu thức gán: (i) trật tự kết hợp từ bên phải sang trái, (ii) có thể sử dụng kết quả biểu thức gán trong các biểu thức khác. Ngoài ra giải pháp này cũng hạn chế việc sao chép dữ liệu từ nơi này đi nơi khác trong bộ nhớ.

Chúng ta phân biệt hai trường hợp:

Trường hợp 1

```
a=a;
```

Với hai toán hạng là một. Trong trường hợp này hàm operator= không làm gì, ngoài việc trả về tham chiếu đến a.

Trường hợp 2

```
a=b;
```

khi hai đối tượng tham gia biểu thức gán hoàn toàn khác nhau, việc đầu tiên là phải giải phóng vùng nhớ động chiếm giữ trước đó trong a, trước khi xin cấp phát một vùng nhớ động khác bằng kích thước vùng nhớ động có trong b, cuối cùng sao chép nội dung từ vùng nhớ động trong b sang a. Và không quên "sao chép" giá trị của các thành phần "không động" còn lại.

Ta xét chương trình minh hoạ.

Ví dụ 4.6

```
/*vector4.cpp*/
#include <iostream.h>
#include <conio.h>
class vector{
    int n;   //số toạ độ của vector
    float *v;    //con trỏ tới vùng nhớ toạ độ
```

```
  public:
    vector(); //hàm thiết lập không tham số
    vector(int size); //hàm thiết lập 1 tham số
    vector(int size, float *a);
    vector(vector &);
    vector & operator=(vector & b);
    ~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;
  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++)
```

```
    v[i] = b.v[i];
  }
vector::~vector() {
   cout<<"Giai phong "<<v<<"cua doi tuong tai"<<this<<endl;
   delete v;
  }
vector & vector::operator=(vector & b) {
 cout<<"Goi operator=() cho "<<this<<" va "<<&b<<endl;
 if (this !=&b){
  /*xoá vùng nhớ động đã có trong đối tượng về trái */
  cout<<"xoa vung nho dong"<<v<<" trong "<<this<<endl;
  delete v;
  /*cấp phát vùng nhớ mới có kích thước như trong b*/
  v=new float [n=b.n];
  cout<<"cap phat vung nho dong moi"<<v<<" trong
"<<this<<endl;
  for(int i=0;i<n;i++) v[i]=b.v[i];
  }
  /*khi hai đối tượng giống nhau, không làm gì */
  else cout<<"Hai doi tuong la mot\n";
  return *this;
  }
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 = s1;//gọi hàm thiết lập sao chép
  s2.display();
  vector s3(0);
  s3=s1;//gọi hàm toán tử vector::operator=(...)
  s1=s1;
  getch();
  }
```

```
Tao doi tuong tai 0xfff2
So chieu :3
Xin cap phat vung bo nho 3 so thuc tai0x148c
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 sao chep
Tao doi tuong tai 0xffee
Xin cap phat vung bo nho 3 so thuc tai0x149c
Doi tuong tai :0xffee
So chieu :3
2 3 2
Su dung ham thiet lap 1 tham so
```

```
Tao doi tuong tai 0xffea
So chieu :0
Xin cap phat vung bo nho 0 so thuc tai0x14ac
Goi operator=() cho 0xffea va 0xfff2
xoa vung nho dong0x14ac trong 0xffea
cap phat vung nho dong moi0x14ac trong 0xffea
Goi operator=() cho 0xfff2 va 0xfff2
Hai doi tuong la mot
```

5.2 Định nghĩa chồng phép "[]"

Xét chương trình sau:

Ví dụ 4.7

```
/*vector5.cpp*/
#include <iostream.h>
#include <conio.h>
class vector{
    int n;  //số giá trị
    float *v;    //con trỏ tới vùng nhớ toạ độ
  public:
    vector(); //hàm thiết lập không tham số
    vector(vector &);
    int length() { return n;}
    vector & operator=(vector &);
    float & operator[](int i) {
      return v[i];
      }
    ~vector();
  };
vector::vector() {
```

```
  int i;
  cout<<"So chieu :";cin>>n;
  v= new float [n];
  }
vector::vector(vector &b) {
  int i;
  v= new float [n=b.n];
  for(i=0;i<n;i++)
    v[i] = b.v[i];
  }
vector::~vector() {
   delete v;
   }
vector & vector::operator=(vector & b){
  cout<<"Goi operator=() cho "<<this<<" va "<<&b<<endl;
  if (this !=&b) {
   /*xoá vùng nhớ động đã có trong đối tượng vế trái*/
    cout<<"xoa vung nho dong"<<v<<" trong "<<this<<endl;
    delete v;
     /*cấp phát vùng nhớ mới có kích thước như trong b*/
    v=new float [n=b.n];
    cout<<"cap phat vung nho dong moi"<<v<<" trong
"<<this<<endl;
   for(int i=0;i<n;i++) v[i]=b.v[i];
   }
   /*khi hai đối tượng giống nhau, không làm gì */
   else cout<<"Hai doi tuong la mot\n";
   return *this;
```

```
    }
void Enter_Vector(vector &s) {
  for (int i=0; i<s.length();i++) {
    cout<<"Toa do thu "<<i+1<<" : ";
    cin>>s[i];
    }
  }
void Display_Vector(vector &s) {
  cout<<"So chieu : "<<s.length()<<endl;
  for(int i=0; i<s.length(); i++)
    cout<<s[i]<<" ";
  cout<<endl;
  }
void main() {
  clrscr();
  cout<<"Tao doi tuong s1\n";
  vector s1;
  /*Nhập các toạ độ cho vector s1*/
  cout<<"Nhap cac toa do cua s1\n";
  Enter_Vector(s1);
  cout<<"Thong tin ve vector s1\n";
  Display_Vector(s1);
  vector s2 = s1;
  cout<<"Thong tin ve vector s2\n";
  Display_Vector(s2);
  getch();
  }
```

```
Tao doi tuong s1
```

```
So chieu :4
Nhap cac toa do cua s1
Toa do thu 1 : 2
Toa do thu 2 : 3
Toa do thu 3 : 2
Toa do thu 4 : 3
Thong tin ve vector s1
So chieu : 4
2 3 2 3
Thong tin ve vector s2
So chieu : 4
2 3 2 3
```

Nhận xét

11. Nhờ giá trị trả về của hàm **operator**`[]` là tham chiếu đến một thành phần toạ độ của vùng nhớ động nên ta có thể đọc/ghi các thành phần toạ độ của mỗi đối tượng vector. Như vậy có thể sử dụng các đối tượng vector giống như các biến mảng. Trong ví dụ trên chúng ta cũng không cần đến hàm thành phần `vector::display()` để in ra các thông tin của các đối tượng.
12. Có thể cải tiến hàm toán tử **operator**`[]` bằng cách bổ sung thêm phần kiểm tra tràn chỉ số.

5.3 Định nghĩa chồng << và >>

Có thể định nghĩa chồng hai toán tử vào/ra << và >> cho phép các đối tượng đứng bên phải chúng khi thực hiện các thao tác vào ra. Chương trình sau đưa ra một cách định nghĩa chồng hai toán tử này.

Ví dụ 4.8

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
```

```
    float real, image;
  friend ostream & operator<<(ostream &o, complex &b);
  friend istream & operator>>(istream &i, complex &b);
  };
ostream & operator<<(ostream &os, complex &b) {
  os<<b.real<<(b.image>=0?'+':'-
')<<"j*"<<fabs(b.image)<<endl;
  return os;
  }
istream & operator>>(istream &is, complex &b) {
  cout<<"Phan thuc : ";
  is>>b.real;
  cout<<"Phan ao : ";
  is>>b.image;
  return is;
  }
void main() {
  clrscr();
  cout<<"Tao so phuc a\n";
  complex a;
  cin>>a;
  cout<<"Tao so phuc b\n";
  complex b;
  cin >>b;
  cout<<"In hai so phuc\n";
  cout<<"a = "<<a;
  cout<<"b = "<<b;
  getch();
```

```
}
```

```
Tao so phuc a
Phan thuc : 3
Phan ao : 4
Tao so phuc b
Phan thuc : 5
Phan ao : 3
In hai so phuc
a = 3+j*4
b = 5+j*3
```

Nhận xét

13. Trong chương trình trên, ta không thấy các hàm thiết lập tường minh để gán giá trị cho các đối tượng. Thực tế, việc gán các giá trị cho các đối tượng được đảm nhiệm bởi hàm toán tử **operator**>>.

14. Việc hiển thị nội dung của các đối tượng số phức có trước đây do hàm thành phần `display()` đảm nhiệm thì nay đã có thể thay thế nhờ hàm toán tử `operator<<`.

15. Hai hàm operator<< và operator>> cho phép sử dụng cout và cin cùng lúc với nhiều đối tượng khác nhau: giá trị số nguyên, số thực, xâu ký tự, ký tự và các đối tượng của lớp complex. Có thể thử nghiệm các cách khác để thấy được rằng giải pháp đưa ra trong chương trình trên là tốt nhất.

5.4 Định nghĩa chồng các toán tử *new* và *delete*

Các toán tử **new** và **delete** được định nghĩa cho từng lớp và chúng chỉ có ảnh hưởng đối với các lớp liên quan, còn các lớp khác vẫn sử dụng các toán tử **new** và **delete** như bình thường.

Định nghĩa chồng toán tử **new** buộc phải sử dụng hàm thành phần và đáp ứng các ràng buộc sau:

(ix) có một tham số kiểu `size_t` (trong tệp tiêu đề `stddef.h`). Tham số này tương ứng với kích thước (tính theo byte) của đối tượng xin cấp phát. Lưu ý rằng đây là tham số giả (dump argument) vì nó sẽ không

được mô tả khi gọi tới toán tử **new,** mà do chương trình biên dịch tự động tính dựa trên kích thước của đối tượng liên đới.

(x) trả về một giá trị kiểu **void** * tương ứng với địa chỉ vùng nhớ động được cấp phát.

Khi định nghĩa chồng toán **delete** ta phải sử dụng hàm thành phần, tuân theo các quy tắc sau đây:

(xi) nhận một tham số kiểu con trỏ tới lớp tương ứng; con trỏ này mang địa chỉ vùng nhớ động đã được cấp phát cần giải phóng,

(xii) không có giá trị trả về (trả về **void**)

Nhận xét

Có thể gọi được các toán tử **new** và **delete** chuẩn (ngay cả khi chúng đã được định nghĩa chồng) thông qua toán tử phạm vi.

Các toán tử **new** và **delete** là các hàm thành phần **static** của các lớp bởi vì chúng không có tham số ngầm định.

Sau đây giới thiệu ví dụ định nghĩa chồng các toán tử **new** và **delete** trên lớp `point`. Ví dụ cũng chỉ ra cách gọi lại các toán tử **new** và **delete** truyền thống.

Ví dụ 4.9

```
/*newdelete.cpp*/
#include <iostream.h>
#include <stddef.h>
#include <conio.h>
class point {
    static int npt;/*số điểm tĩnh*/
    static int npt_dyn;/*số điểm động*/
    int x, y;
  public:
    point(int ox=0, int oy = 0) {
      x = ox; y = oy;
      npt++;
```

```
      cout<<"++Tong so diem : "<<npt<<endl;
      }
    ~point () {
      npt--;
      cout<<"--Tong so diem : "<<npt<<endl;
      }
    void * operator new (size_t sz) {
      npt_dyn++;
      cout<<"    Co "<<npt_dyn<<" diem dong "<<endl;
      return ::new char [sz];
      }
    void operator delete (void *dp) {
      npt_dyn--;
      cout<<"    Co "<<npt_dyn<<" diem dong "<<endl;
      ::delete (dp);
      }
};
int point::npt = 0;
int point::npt_dyn = 0;
void main() {
  clrscr();
  point * p1, *p2;
  point a(3,5);
  p1 = new point(1,3);
  point b;
  p2 = new point (2,0);
  delete p1;
  point c(2);
```

```
  delete p2;
  getch();
}
```

```
++Tong so diem : 1
    Co 1 diem dong
++Tong so diem : 2
++Tong so diem : 3
    Co 2 diem dong
++Tong so diem : 4
--Tong so diem : 3
    Co 1 diem dong
++Tong so diem : 4
--Tong so diem : 3
    Co 0 diem dong
```

Nhận xét

Dù cho **new** có được định nghĩa chồng hay không, lời gọi tới **new** luôn luôn cần đến các hàm thiết lập.

5.5 Phép nhân ma trận véc tơ

Chương trình `vectmat4.cpp` sau đây được cải tiến từ `vectmat3.cpp` trong đó hàm `prod()` được thay thế bởi **operator***.

Ví dụ 4.10

```
/*vectmat4.cpp*/
#include <iostream.h>
#include <conio.h>
class matrix;/*khai báo trước lớp matrix*/
/*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 operator*(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 qu
{
  for(int i=0;i<n;i++)
    cout <<v[i] <<" ";
  cout <<"\n";
}
/* lớp matrix*/
class matrix {
      static int n;   //số chiều của vector
      vector *m;     //vùng nhớ chứa các toạ độ
  public:
      matrix();
      matrix(matrix &);//hàm thiết lập sao chép
      ~matrix();
      void display();
      static int &Size() {return n;}
      friend vector operator*(matrix &, vector &);
  };
int matrix::n =0;
/*hàm thành phần của lớp matrix*/
matrix::matrix(){
  int i;
```

```
  m= new vector [n];
  }

matrix::matrix(matrix &b) {
  int i,j;
  m= new vector[n];
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      m[i].v[j]=b.m[i].v[j];
  }
matrix::~matrix() {
  delete m;
  }
void matrix::display()  {
for (int i=0; i<n; i++)
  m[i].display();
}
/*hàm toán tử*/
vector operator*(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 = m*v;/* opertaor*(m,v) */
  u.display();
  getch();
 }
```

```
Kich thuoc cua vector 4
Tao mot vector
Toa do thu 1 : 1
Toa do thu 2 : 2
Toa do thu 3 : 3
Toa do thu 4 : 4
 v=
1 2 3 4
Tao mot ma tran
```

```
Toa do thu 1 : 3
Toa do thu 2 : 2
Toa do thu 3 : 1
Toa do thu 4 : 2
Toa do thu 1 : 3
Toa do thu 2 : 2
Toa do thu 3 : 3
Toa do thu 4 : 2
Toa do thu 1 : 3
Toa do thu 2 : 2
Toa do thu 3 : 3
Toa do thu 4 : 2
Toa do thu 1 : 2
Toa do thu 2 : 3
Toa do thu 3 : 2
Toa do thu 4 : 3
 m =
3 2 1 2
3 2 3 2
3 2 3 2
2 3 2 3
Tich m*v
18 24 24 26
```

6. CHUYỂN ĐỔI KIỂU

Với các kiểu dữ liệu chuẩn, ta có thể thực hiện các phép chuyển kiểu ngầm định, chẳng hạn có thể gán một giá trị **int** vào một biến **long**, hoặc cộng giá trị **long** vào một biến **float** . Thường có hai kiểu chuyển kiểu: chuyển kiểu ngầm định(tự động) và chuyển kiểu tường minh (ép kiểu). Phép chuyển kiểu ngầm định được thực hiện bởi chương trình biên dịch. Phép

chuyển kiểu tường minh xảy ra khi sử dụng phép ép kiểu bắt buộc. Phép ép kiểu thường được dùng trong các câu lệnh gọi hàm để gửi các tham số có kiểu khác với các tham số hình thức tương ứng.

Các kiểu lớp không thể thoải mái chuyển sang các kiểu khác được mà phải do người tự làm lấy. C++ cũng cung cấp cách thức định nghĩa phép chuyển kiểu ngầm định và tường minh. Phép chuyển kiểu ngầm định được định nghĩa bằng một hàm thiết lập chuyển kiểu (conversion constructor), còn phép chuyển kiểu tường minh được xác định thông qua toán tử chuyển kiểu hoặc ép kiểu (cast operator).

Phép chuyển kiểu ngầm định được định nghĩa thông qua một hàm thiết lập chuyển kiểu cho lớp. Với đối số có kiểu kiểu cần phải chuyển thành một đối tượng của lớp đó. Tham số này có thể có kiểu cơ sở hay là một đối tượng thuộc lớp khác. Hàm thiết lập một tham số trong lớp point trong các chương trình point?.cpp ở chương trước là ví dụ cho hàm thiết lập chuyển kiểu.

Trong chỉ thị

```
point p=2;
```

đã chuyển kiểu từ giá trị nguyên 2 sang một đối tượng `point`. Thực tế ở đây chương trình dịch gọi tới hàm thiết lập một tham số. Đây là sự chuyển kiểu một chiều, nhận giá trị hoặc đối tượng nào đó và chuyển nó thành đối tượng của lớp. Các hàm thiết lập chuyển kiểu không thể sử dụng để chuyển các đối tượng của lớp mình sang các kiểu khác và chúng chỉ có thể được sử dụng trong các phép gán và phép khởi tạo giá trị.

Tuy nhiên, các toán tử chuyển kiểu có thể được dùng để chuyển các đối tượng sang các kiểu khác và cũng có thể được dùng cho các mục đích khác ngoài phép gán và khởi tạo giá trị. C++ qui định rằng một hàm toán tử chuyển kiểu như thế buộc phải là hàm thành phần của lớp liên quan và không có tham số hoặc kiểu trả về. Tên của nó được cho theo dạng như sau:

```
operator type();
```

trong đó `type` là tên của kiểu dữ liệu mà một đối tượng sẽ được chuyển sang; có thể là kiểu dữ liệu cơ sở (khi đó ta phải chuyển kiểu từ đối tượng sang kiểu cơ sở) hay một kiểu lớp khác (khi đó ta phải chuyển kiểu từ đối tượng lớp này sang lớp khác).

6.1 Hàm toán tử chuyển kiểu ép buộc

Chương trình `complex6.cpp` sau đây minh hoạ cách cài đặt các hàm toán tử chuyển kiểu ngầm định và chuyển kiểu từ lớp `complex` sang một số thực. Vấn đề chuyển kiểu từ lớp này sang lớp khác sẽ được giới thiệu sau.

Ví dụ 4.11

```
/*complex6.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
  public:
    complex( ) {
      real = 0; image = 0;
      }
    /*hàm thiết lập đóng vai trò một hàm toán tử chuyển kiểu tự động*/
    complex(float r) {
      real = r; image = 0;
      }
    complex(float r, float i ) {
      real = r; image = i;
      }
  /*Hàm toán tử chuyển kiểu ép buộc*/
    operator float() {
      return real;
      }
    void display() {
      cout<<real<<(image>=0?'+':'-
')<<"j*"<<fabs(image)<<endl;
```

```
        }
  /*hàm operator+ định nghĩa phép toán + hai ngôi trên lớp số phức complex*/
     complex operator+(complex b) {
        complex c;
        c.real = real+b.real;
        c.image =image+b.image;
        return c;
        }
  };
void main() {
  clrscr();
  cout<<"a = ";
  complex a(-2,5);
  a.display();
  cout<<"b = ";
  complex b(3,4);
  b.display();
  cout<<"c = a+b : ";
  complex c;
  c=a+b;//a.operator+(b)
  c.display();
  cout<<"d = 3+c : ";
  complex d;
  d= complex(3)+c;
  d.display();
  cout<<"float x = a : ";
  float x = a;//float(complex)
```

```
  cout<<x<<endl;
  getch();
  }
```

```
a = -2+j*5
b = 3+j*4
c = a+b : 1+j*9
d = 3+c : 4+j*9
float x = a : -2
```

Chú ý:

16. Phép cộng `3+c` khác với `complex(3)+c` vì trong phép thứ nhất người ta thực hiện chuyển đổi `c` thành số thực và phép cộng đó thực hiện giữa hai số thực. Có thể thử nghiệm để kiểm tra lại kết quả sau:
17. Đoạn chương trình

```
d = 3 + c;
d.display()
```

cho ra kết quả

```
4+j*0
```

Các phép toán nguyên thuỷ có độ ưu tiên hơn so với các phép toán được định nghĩa chồng.

6.1.1 Hàm toán tử chuyển kiểu trong lời gọi hàm

Trong chương trình dưới đây ta định nghĩa hàm `fct()` với một tham số thực (**float**) và sẽ gọi hàm này hai lần: lần thứ nhất với một tham số thực, lần thứ hai với một tham số kiểu `complex`.

Trong lớp complex còn có một hàm thiết lập sao chép, không được gọi khi ta truyền đối tượng complex cho hàm `fct()` vì ở đây xảy ra sự chuyển đổi kiểu dữ liệu.

Ví dụ 4.12

```
/*complex7.cpp*/
```

```
#include <iostream.h>
#include <conio.h>
class complex {
    float real, image;
  public:
    complex(float r, float i ) {
      real = r; image = i;
      }
    complex(complex &b ) {
      cout<<"Ham thiet lap sao chep\n";
      real = b.r; image = b.i;
      }
    /*Hàm toán tử chuyển kiểu ép buộc*/
    operator float() {
      cout<<"Goi float() cho complex\n";
      return real;
      }
    void display() {
      cout<<real<<(image>=0?'+':'-
')<<"j*"<<fabs(image)<<endl;
      }
};
void fct(float n) {
  cout<<"Goi fct voi tham so : "<<n<<endl;
  }
void main() {
  clrscr();
  complex a(3,4);
```

```
  fct(6);//lời gọi hàm thông thường
  fct(a);//lời gọi hàm có xảy ra chuyển đổi kiểu dữ liệu
  getch();
  }
```

```
Goi fct voi tham so : 6
Goi float() cho complex
Goi fct voi tham so : 3
```

Trong chương trình này, lời gọi hàm

`fct(a)`

đã được chương trình dịch chuyển thành các thao tác:

(xiii) chuyển đổi đối tượng thành **float**,

(xiv) lời gọi hàm `fct()` với tham số là giá trị thu được sau chuyển đổi.

Sự chuyển đổi được thực hiện khi gọi hàm do đó không xảy ra việc sao chép lại đối tượng `a`.

6.1.2 Hàm toán tử chuyển kiểu trong biểu thức

Chương trình dưới đây cho ta biết biểu thức dạng `a+b` hoặc `a+3` được tính như thế nào với `a, b` là các đối tượng kiểu `complex`.

Ví dụ 4.13

```
/*complex8.cpp*/
#include <iostream.h>
#include <conio.h>
class complex {
    float real, image;
  public:
    complex(float r, float i ) {
       real = r; image = i;
       }
```

```
    /*Hàm toán tử chuyển kiểu ép buộc*/
    operator float() {
      cout<<"Goi float() cho complex\n";
      return real;
      }
    void display() {
      cout<<real<<(image>=0?'+':'-
')<<"j*"<<fabs(image)<<endl;
      }
  };
void main() {
  clrscr();
  complex a(3,4);
  complex b(5,7);
  float n1, n2;
  n1 = a+3; cout<<"n1 = "<<n1<<endl;
  n2 = a + b;cout<<"n2 = "<<n2<<endl;
  double z1, z2;
  z1 = a+3; cout<<"z1 = "<<z1<<endl;
  z2 = a + b;cout<<"z2 = "<<z2<<endl;
  getch();
  }
```

```
Goi float() cho complex
n1 = 6
Goi float() cho complex
Goi float() cho complex
n2 = 8
Goi float() cho complex
```

```
z1 = 6
Goi float() cho complex
Goi float() cho complex
z2 = 8
```

Khi gặp biểu thức dạng `a+3` với phép toán + được định nghĩa với các toán hạng có kiểu lớp `complex` và số thực, chương trình dịch trước hết đi tìm xem đã có một toán tử + được định nghĩa chồng tương ứng với các kiểu dữ liệu của các toán hạng này hay chưa. Trong trường hợp này vì không có, nên chương trình dịch sẽ chuyển đổi kiểu dữ liệu của các toán hạng để phù hợp với một trong số các phép toán đã định nghĩa, cụ thể là chuyển đổi từ đối tượng `a` sang **float**.

6.2 Hàm toán tử chuyển đổi kiểu cơ sở sang kiểu lớp

Trở lại chương trình `complex6.cpp`, ta có thể thực hiện các chỉ thị kiểu như:

```
complex e=10;
```

hoặc

```
a=1;
```

Chỉ thị thứ nhất nhằm tạo một đối tượng tạm thời có kiểu `complex` tương ứng với phần thực bằng 10, phần ảo bằng 0 rồi sao chép sang đối tượng `e` mới được khai báo. Trong chỉ thị thứ hai, cũng có một đối tượng tạm thời kiểu `complex` được tạo ra và nội dung của nó (phần thực 1, phần ảo 0) được gán cho `a`. Như vậy, trong cả hai trường hợp đều phải gọi tới hàm thiết lập một tham số của lớp `complex`.

Tương tự, nếu có hàm `fct()` với khai báo:

```
fct(complex)
```

thì lời gọi

```
fct(4)
```

sẽ đòi hỏi phải chuyển đổi từ giá trị nguyên 4 thành một đối tượng tạm thời có kiểu `complex`, để truyền cho `fct()`. Sau đây là chương trình nhận được do sửa đổi từ `complex6.cpp`.

Ví dụ 4.14

```
/*complex9.cpp*/
```

```
#include <iostream.h>
#include <conio.h>
class complex {
    float real, image;
  public:
    complex(float r) {
    cout<<"Ham thiet lap dong vai tro cua ham toan tu
chuyen kieu ngam dinh\n";
      real = r; image = 0;
      }
    complex(float r, float i ) {
      cout<<"Ham thiet lap \n";
      real = r; image = i;
      }
    complex(complex &b) {
      cout<<"Ham thiet lap sao chep lai "<<&b<<" Sang
"<<this<<endl;
      real = b.real; image = b.image;
      }
  };
void fct(complex p) {
  cout<<"Goi fct \n";
  }
void main() {
  clrscr();
  complex a(3,4);
  a = complex(12);
  a = 12;
  fct(4);
```

```
  getch();
  }
```

```
Ham thiet lap
Ham thiet lap dong vai tro cua ham toan tu chuyen
kieu ngam dinh
Ham thiet lap dong vai tro cua ham toan tu chuyen
kieu ngam dinh
Ham thiet lap dong vai tro cua ham toan tu chuyen
kieu ngam dinh
Goi fct
```

6.2.1 Hàm thiết lập trong các chuyển đổi kiểu liên tiếp

Trong lớp số phức `complex` hàm thiết lập với một tham số cho phép thực hiện chuyển đổi **float** -->`complex` đồng thời cả chuyển đổi ngầm định

int --> **float**. tức là nó có thể cho phép một chuỗi các chuyển đổi:

int --> **float** -->`complex`

Chẳng hạn khi gặp phép gán kiểu như:

```
a = 2;
```

Cần chú ý khả năng chuyển đổi này phải dựa trên các quy tắc chuyển đổi thông thường như đã nói ở trên.

6.2.2 Lựa chọn giữa hàm thiết lập và phép toán gán

Với chỉ thị gán:

```
a=12;
```

trong chương trình `complex9.cpp` không có định nghĩa toán tử gán một số nguyên cho một đối tượng `complex`. Giả sử có một toán tử gán như vậy thì có thể sẽ xảy ra xung đột giữa:

(xv) chuyển đổi **float**-->`complex` bởi hàm thiết lập và phép gán `complex-->complex`.

(xvi) sử dụng trực tiếp toán tử gán **float**-->`complex`.

Để giải quyết vấn đề này ta tuân theo quy tắc : "Các chuyển đổi do người sử dụng định nghĩa chỉ được thực hiện khi cần thiết", nghĩa là với chỉ thị gán:

```
a = 12;
```

nếu như trong lớp `complex` có định nghĩa toán tử gán, nó sẽ được ưu tiên thực hiện. Chương trình sau đây minh hoạ nhận xét này:

Ví dụ 4.15

```
/*complex10.cpp*/
#include <iostream.h>
#include <conio.h>
class complex {
    float real, image;
  public:
    complex(float r) {
      cout<<"Ham thiet lap dong vai tro cua ham toan tu
chuyen kieu ngam dinh\n";
      real = r; image = 0;
      }
    complex(float r, float i ) {
      cout<<"Ham thiet lap \n";
      real = r; image = i;
      }
    complex & operator=(complex &p) {
      real = p.real;image = p.image;
      cout<<"gan complex -->complex tu "<<&p<<" sang
"<<this<<endl;
      return *this;
      }
    complex & operator=(float n) {
      real = n;image = 0;
```

```
      cout<<"gan float -->complex "<<endl;
      return *this;
      }
  };
void main() {
  clrscr();
  complex a(3,4);
  a = 12;
  getch();
  }
```

```
Ham thiet lap
gan float -->complex
```

6.2.3 Sử dụng hàm thiết lập để mở rộng ý nghĩa một phép toán

Ta xét lớp `complex` và hàm thiết lập một tham số của lớp được bổ sung thêm một hàm toán tử dưới dạng hàm bạn (trường hợp này không nên sử dụng hàm toán tử thành phần). Với các điều kiện này, khi a là một đối tượng kiểu `complex`, biểu thức kiểu như:

```
a + 3
```

sẽ có ý nghĩa. Thực vậy trong trường hợp này chương trình dịch sẽ thực hiện các thao tác sau:

(xvii) chuyển đổi từ số thực 3 sang số phức `complex`.

(xviii) cộng giữa đối tượng nhận được với a bằng cách gọi hàm toán tử `operator+`.

Kết quả sẽ là một đối tượng kiểu `complex`. Nói cách khác, biểu thức a + 3 tương đương với

```
operator+(a, point(3)).
```

Tương tự, biểu thức

```
5 + a
```

sẽ tương đương với:

`operator+(point(5),a)`.

Tuy nhiên trường hợp sau sẽ không còn đúng khi `operator+` là hàm toán tử thành phần. Sau đây là một chương trình minh hoạ các khả năng mà chúng ta vừa đề cập.

Ví dụ 4.16

```
/*complex11.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex {
    float real, image;
  public:
    complex(float r) {
      cout<<"Ham thiet lap dong vai tro cua ham toan tu
chuyen kieu ngam dinh\n";
      real = r; image = 0;
      }
    complex(float r, float i ) {
      cout<<"Ham thiet lap 2 tham so\n";
      real = r; image = i;
      }
    void display() {
      cout<<real<<(image>=0?"+j*":"-j*")<<fabs(image)<<endl;
      }
    friend complex operator+(complex , complex);
  };
complex operator+(complex a, complex b) {
  complex c(0,0);
  c.real = a.real + b.real;
```

```
  c.image = a.image + b.image;
  return c;
  }
void main() {
  clrscr();
  complex a(3,4),b(9,4);
  a = b + 5;a.display();
  a = 2 + b;a.display();
  getch();
  }
```

```
Ham thiet lap 2 tham so
Ham thiet lap 2 tham so
Ham thiet lap dong vai tro cua ham toan tu chuyen kieu ngam dinh
Ham thiet lap 2 tham so
14+j*4
Ham thiet lap dong vai tro cua ham toan tu chuyen kieu ngam dinh
Ham thiet lap 2 tham so
11+j*4
```

Nhận xét

Hàm thiết lập làm nhiệm vụ của hàm toán tử chuyển đổi kiểu cơ sở sang kiểu lớp không nhất thiết chỉ có một tham số hình thức. Trong trường hợp hàm thiết lập có nhiều tham số hơn, các tham số tính từ tham số thứ hai phải có giá trị ngầm định.

6.3 Chuyển đổi kiểu từ lớp này sang một lớp khác

Khả năng chuyển đổi qua lại giữa kiểu cơ sở và một kiểu lớp có thể được mở rộng cho hai kiểu lớp khác nhau:

(xix) Trong lớp A, ta có thể định nghĩa một hàm toán tử để thực hiện chuyển đổi từ kiểu A sang kiểu B (cast operator).

(xx) Hàm thiết lập của lớp A chỉ với một tham số kiểu B sẽ thực hiện chuyển đổi kiểu từ B sang A.

6.3.1 Hàm toán tử chuyển kiểu bắt buộc

Ví dụ sau đây minh hoạ khả năng dùng hàm toán tử `complex` của lớp `point` cho phép thực hiện chuyển đổi một đối tượng kiểu `point` thành một đối tượng kiểu `complex`.

Ví dụ 4.17

```
/*pointcomplex1.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class complex;//khai báo trước lớp complex
class point {
    int x, y;
  public:
    point(int ox = 0, int oy =0) {x = ox; y = oy;}
    operator complex();//chuyển đổi point-->complex
  };
class complex {
    float real, image;
  public:
    complex(float r=0, float i=0 ) {
      real = r; image = i;
      }
    friend point::operator complex();
    void display() {
      cout<<real<<(image>=0?"+j*":"-j*")<<fabs(image)<<endl;
      }
  };
point::operator complex() {
  complex r(x,y);
  cout<<"Chuyen doi "<<x<<" "<<y
      <<" thanh "<<r.real<<" + "<< r.image<<endl;
  return r;
```

```
  }
void main() {
  clrscr();
  point a(2,5); complex c;
  c = (complex) a; c.display();
  point b(9,12);
  c = b; c.display();
  getch();
  }
```

```
Chuyen doi 2 5 thanh 2 + 5
2+j*5
Chuyen doi 9 12 thanh 9 + 12
9+j*12
```

6.3.2 Hàm thiết lập dùng làm hàm toán tử

Chương trình sau đây minh hoạ khả năng dùng hàm thiết lập `complex(point)` biểu để thực hiện chuyển đổi một đối tượng kiểu `point` thành một đối tượng kiểu `complex`.

Ví dụ 4.18

```
/*pointcomplex2.cpp*/
#include <iostream.h>
#include <conio.h>
#include <math.h>
class point;//khai báo trước lớp complex
class complex {
    float real, image;
  public:
    complex(float r=0, float i=0 ) {
       real = r; image = i;
```

```
    }
  complex(point);
  void display() {
    cout<<real<<(image>=0?"+j*":"-j*")<<fabs(image)<<endl;
    }
};
class point {
  int x, y;
public:
  point(int ox = 0, int oy =0) {x = ox; y = oy;}
  friend complex::complex(point);
};
complex::complex(point p) {
  real= p.x; image = p.y;
  }
void main() {
  clrscr();
  point a(3,5);
  complex c = a; c.display();
  getch();
  }
```

```
3+j*5
```

7. TÓM TẮT

7.1 Ghi nhớ

Toán tử được định nghĩa chồng bằng cách định nghĩa một hàm toán tử. Tên hàm toán tử bao gồm từ khoá **operator** theo sau là ký hiệu của toán tử được định nghĩa chồng.

Hầu hết các toán tử của C++ đều có thể định nghĩa chồng. Không thể tạo ra các ký hiệu phép toán mới.

Phải đảm bảo các đặc tính nguyên thuỷ của toán tử được định nghĩa chồng, chẳng hạn: độ ưu tiên, trật tự kết hợp, số ngôi .

Không sử dụng tham số có giá trị ngầm định để định nghĩa chồng toán tử.

Các toán tử `()`, `[]`, `->`, = yêu cầu hàm toán tử phải là hàm thành phần của lớp.

Hàm toán tử có thể là hàm thành phần hoặc hàm bạn của lớp.

Khi hàm toán tử là hàm thành phần, toán hạng bên trái luôn là đối tượng thuộc lớp.

Nếu toán hạng bên trái là đối tượng của lớp khác thì hàm toán tử tương ứng phải là hàm bạn.

Chương trình dịch không tự biết cách chuyển kiểu giữa kiểu dữ liệu chuẩn và kiểu dữ liệu tự định nghĩa. Vì vậy người lập trình cần phải mô tả tường minh các chuyển đổi này dưới dạng hàm thiết lập chuyển kiểu hay hàm toán tử chuyển kiểu.

Một hàm toán tử chuyển kiểu thực hiện chuyển đổi từ một đối tượng thuộc lớp sang đối tượng thuộc lớp khác hoặc một đối tượng có kiểu được định nghĩa trước.

Hàm thiết lập chuyển kiểu có một tham số và thực hiện chuyển đổi từ một giá trị sang đối tượng kiểu lớp.

Toán tử gán là toán tử hay được định nghĩa chồng nhất, đặc biệt khi lớp có các thành phần dữ liệu động.

Để định nghĩa chồng toán tử tăng, giảm một ngôi, phải phân biệt hai hàm toán tử tương ứng cho dạng tiền tố và dạng hậu tố.

7.2 Các lỗi thường gặp

Tạo một toán tử mới.

Thay đổi định nghĩa của các toán tử trên các kiểu được định nghĩa trước.

Cho rằng việc định nghĩa chồng một toán tử sẽ tự đông kéo theo định nghĩa chồng của các toán tử liên quan.

Quên định nghĩa chồng toán tử gán và hàm thiết lập sao chép cho các lớp có các thành phần dữ liệu động.

7.3 Một số thói quen lập trình tốt

Sử dụng toán tử định nghĩa chồng khi điều đó làm cho chương trình trong sáng hơn.

Tránh lạm dụng định nghĩa chồng toán tử vì điều đó dẫn đến khó kiểm soát chương trình.

Chú ý đến các tính chất nguyên thuỷ của toán tử được định nghĩa chồng.

Hàm thiết lập, toán tử gán, hàm thiết lập sao chép của một lớp thường đi cùng nhau.

8. BÀI TẬP

Bài tập 4.1.

Bổ sung thêm một số toán tử trên lớp số phức `complex`.

Định nghĩa hai phép toán vào ra trên lớp `vector`.

Bài tập 4.2.

Một ma trận được hiểu là một vector mà mỗi thành phần của nó lại là một vector. Theo tinh thần đó hãy định nghĩa lớp `matrix` dựa trên `vector`. Tìm cách để cho chương trình dịch hiểu được phép truy nhập

`m[i][j]` trong đó m là một đối tượng thuộc lớp `matrix`.

Bài tập 4.3.

Định nghĩa các phép nhân, cộng, trừ trên các ma trận vuông.

Bài tập 4.4.

Dựa trên định nghĩa của lớp `complex` , lớp `vector` và lớp `matrix` để xây dựng chương trình mô phỏng các thao tác trên ma trận và vector phức.

Bài tập 4.5.

Thay thế hàm thành phần `tick()` trong lớp `date_time` bài tập 3.10 bởi hàm toán tử tương ứng với phép toán ++.

Bài tập 4.6.

Tạo lớp phân số `PS` với các khả năng sau:

(xxi) Tạo một hàm thiết lập với mẫu số dương, ở dạng tối giản hoặc rút gọn về dạng tối giản.

(xxii) Định nghĩa chồng các toán tử cộng, trừ, nhân, chia cho lớp này.

(xxiii) Định nghĩa chồng các toán tử quan hệ trên lớp số phức.