TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI

Đề cương bài giảng

Lập trình hướng đối tượng

Ngành : Công nghệ thông tin
Trình độ đào tạo: Đại học
Người biên soạn : Khoa CNTT

(Lưu hành nội bộ)

Hà Nội – 5 / 2021

Bài 4 : Phương thức toán tử

PHẦN 1. CÀI ĐẶT PHƯƠNG THỨC TOÁN TỬ

1.1. Định nghĩa hàm toán tử theo lập trình cấu trúc

a. Phân loại toán tử

Một biểu thức được tạo nên từ các toán tử (phép toán) và các toán hạng (số hạng). Ví dụ biểu thức Q = 2*x + b thì các toán tử * và + cùng với các toán hạng 2, x và b được sử dụng. Các toán tử có thể tạm chia làm hai loại:

- **Toán tử một ngôi:** Là những toán tử thực hiện trên một toán hạng. Thuộc loại này có phép phủ định (!), Phép tăng 1 đơn vị (++), giảm một đơn vị (--), phép đổi dấu…

- **Toán tử hai ngôi:** Là những toán tử thực hiện trên 2 toán hạng. Thuộc loại này bao gồm các toán tử cộng (+), trừ (-), nhân (*), chia (/)….

Trong lập trình, các toán tử cộng, trừ, nhân, chia,… trên các toán hạng thông thường đã được định nghĩa sẵn, ta chỉ việc sử dụng. Tuy nhiên, một số toán tử trên các toán hạng đặc biệt lại chưa được định nghĩa. Ví dụ: phép cộng , trừ, nhân, chia hai phân số, phép cộng, trừ, nhân, chia hai số phức .v.v..

Bài này nhằm giúp ta cách thức cài đặt các phép toán chưa được định nghĩa trong lập trình như vậy. Sau khi cài đặt, ta có thể sử dụng chúng như các toán tử thông thường.

b. Hàm toán tử trong lập trình cấu trúc.

Ta trở lại với phương pháp lập trình cấu trúc. Khi đó, một hàm toán tử có đặc điểm sau:

- Hàm toán tử được cài đặt tương tự hàm thông thường, chỉ khác ở tên hàm và cách sử dụng.

- Tên hàm: được viết theo dạng: **operator <Ký hiệu toán tử>**

- Cú pháp của hàm:

```
<Kiểu trả về> operator <Ký hiệu của toán tử> (các đối số)
{
    Thân hàm toán tử;
}
```

Ví dụ: Hàm toán tử cộng hai số thực bất kỳ được viết như sau:

```
float operator + (float x, float y)
{
    return x + y;
}
```

- **Cách sử dụng hàm toán tử:** Có hai cách gọi một hàm toán tử.

Cách 1: gọi như hàm thông thường. VD: để cộng hai số thực a, b ta có thể viết:

```
cout<< "Tong cua hai so a va b la" << operator+(a,b);
```

Cách 2: gọi như một toán tử: Ta có thể sử dụng hàm toán tử như một toán tử, tức là ta có thể viết:

```
cout<< "Tong cua hai so S1 va S2 la" << S1 + S2;
```

Phép cộng trên sẽ gọi tới hàm toán tử cộng đã định nghĩa.

VD: Một số phức có dạng: <Phần thực> + i * <Phần ảo>. Cho hai số phức **X = a + i*b** và **Y = c + i * d**. Khi đó **X + Y** sẽ cho số phức có dạng: **X+Y = (a+c) + i * (b + d)**. Hãy định nghĩa hàm toán tử để thực hiện cộng hai số phức bất kỳ.

```
typedef struct SP
{
  float Phanthuc;
  float Phanao;
};
//Dinh nghia ham toan tu cong hai so phuc
SP operator+(SP x, SP y)
{
     SP tg;
     tg.Phanthuc = x.Phanthuc + y.Phanthuc;
     tg.Phanao = x.Phanao + y.Phanao;
     return tg;
}
void main()
{
//Khai bao hai so phuc x va y va so phuc tong T
  SP x,y, T;
  x.Phanthuc = 2; x.Phanao = 3;
  y.Phanthuc= 3; y.Phanao = 5;
//Cong hai so phuc va in ket qua len man hinh
  T = operator+(x, y); //Co the viet T = x + y
  cout<<"Ket qua "<<T.Phanthuc<<"+ i * "<<T.Phanao;
  getch();
}
```

Chú ý: Thay bằng viết **`T = operator+(x, y);`** ta có thể viết: **`T = x + y;`** như cộng hai số thực thông thường do đã định nghĩa hàm toán tử cộng hai số phức ở trên.

1.2. Định nghĩa phương thức toán tử

Trong Lập trình Hướng đối tượng, khi muốn một phương thức là phương thức toán tử, ta cài đặt thế nào? Khi đã cài đặt chúng thì sử dụng thế nào? Để trả lời các câu hỏi trên ta xét.

a. Cài đặt phương thức toán tử một ngôi

Phương thức toán tử cũng tương tự như hàm toán tử. Ta xét một ví dụ:

VD: cài đặt lớp số phức bao gồm các thuộc tính phần thực và phần ảo, phương thức khởi tạo có đối khởi gán các giá trị cho phần thực và ảo và phương thức khởi tạo không đối. Phương thức toán tử (-) để đổi dấu một số phức. Phương thức xuat() để in số phức ra màn hình. Viết chương trình chính để tạo một số phức và in kết quả sau khi đã đổi dấu số phức ra màn hình.

```
class SoPhuc
{
    float Phanthuc, Phanao;
public:
    //Phuong thuc khoi tao tao khong doi
    SoPhuc()
    {
    }
    // Phuong thuc khoi tao co doi
    SoPhuc(float a, float b)
    {
       Phanthuc = a;
       Phanao   = b;
    }
    //Phuong thuc toan tu doi dau
    SoPhuc operator-()
    {
       SoPhuc tg;
       tg.Phanthuc = -Phanthuc;
       tg.Phanao   =  -Phanao;
       return tg;
    }

    void xuat()
    {
       cout<<" So phuc la "<<Phanthuc<<" + i *
       "<<Phanao;
    }
};

 void main()
 {
   SoPhuc x(2, 3);
   SoPhuc y = x.operator-(); //co the viet y = -x;
   y.xuat();
```

```
   getch();
  }
```

Ta nhận thấy:

- Phương thức toán tử một ngôi không có đối vào. Như vậy việc đổi dấu sẽ thực hiện trên số phức nào? Thực chất phương thức toán tử đổi dấu trên đã bao gồm một đối mặc định, đó là con trỏ this.

- Con trỏ this luôn là đối mặc định của các phương thức toán tử. Như vậy, hai cách viết sau là tương đương

`tg.Phanthuc = -Phanthuc;` `tg.Phanao = -Phanao;`	`tg.Phanthuc = -this -> Phanthuc;` `tg.Phanao = -this -> Phanao;`

- Khi sử dụng phương thức toán tử một ngôi ta cũng có 2 cách như với hàm toán tử. Như vậy, hai cách viết sau là tương đương:

`SoPhuc y = x.operator-();`	`SoPhuc y = -x;`

b. Cài đặt phương thức toán tử hai ngôi

Như đã biết, trong phương thức toán tử, con trỏ this luôn là một đối số mặc định. Như vậy, với phương thức toán tử hai ngôi, thay vì có hai đối vào, ta chỉ cần một đối, đối còn lại là con trỏ this.

VD: Cài đặt lớp số phức ở trên với phương thức toán tử hai ngôi cộng hai số phức bất kỳ.

```
class SoPhuc
{
     float Phanthuc, Phanao;
public:
     SoPhuc()
     {
     }
     SoPhuc(float a, float b)
     {
          Phanthuc = a;
          Phanao = b;
     }
     SoPhuc operator+(SoPhuc y)
     {
          SoPhuc tg;
          tg.Phanthuc = this -> Phanthuc + y.Phanthuc;
          tg.Phanao   = this -> Phanao + y.Phanao;
          return tg;
     }
```

```
    void xuat()
    {
        cout<<" So phuc la "<<Phanthuc<<" + i *
    "<<Phanao;
    }
};
  void main()
  {
      clrscr();
      //Khai bao hai so phuc x va y
      SoPhuc x(2, 3); x.xuat();
      SoPhuc y(5, 7); y.xuat();
      SoPhuc T = x.operator+(y);     //co the viet T =
    x+y;
      T.xuat();
      getch();
  }
```

Tương tự như phương thức toán tử một ngôi, ta nhận thấy:

- Phương thức toán tử hai ngôi có 1 đối vào. Đối vào còn lại chính là con trỏ this.
- Con trỏ this luôn là đối mặc định của các phương thức toán tử. Như vậy, hai cách viết sau là tương đương

tg.Phanthuc = Phanthuc + y.Phanthuc; **tg.Phanao = Phanao + y.Phanao;**	**tg.Phanthuc = this -> Phanthuc + y.Phanthuc;** **tg.Phanao = this -> Phanao + y.Phanao;**

- Khi sử dụng phương thức toán tử hai ngôi ta cũng có 2 cách như với hàm toán tử. Như vậy, hai cách viết sau là tương đương:

SoPhuc T = x.operator+(y);	**SoPhuc T = x + y**

PHẦN 2: TỔNG HỢP CÀI ĐẶT TOÁN TỬ

Ngoài việc nắm vững cách cài đặt các bài tập thông thường, ta cần bổ sung thêm một số kiến thức để cài đặt các lớp có tính chất đặc biệt. Các lớp có thêm phương thức toán tử là những lớp thuộc loại này.

Ở phần này chúng ta sẽ xem xét cách thức để cài đặt phương thức toán tử thông thường, hàm toán tử đặc biệt. Muốn vậy, sau phần này, cần đảm bảo bạn đã có thể trả lời đầy đủ các câu hỏi sau:

[1]. Tại sao phải cài đặt phương thức toán tử?

[2]. Cài đặt phương thức toán tử như thế nào? Tại sao phương thức toán tử 1 ngôi lại không có đối còn phương thức toán tử 2 ngôi lại chỉ có 1 đối?

[3]. Sử dụng các phương thức vừa cài đặt thế nào?

[4]. Hàm toán tử nhập, xuất làm nhiệm vụ gì? cài đặt và sử dụng thế nào?

Bây giờ ta sẽ xem xét lần lượt chúng.

2.1. Tại sao phải cài đặt phương thức toán tử?

Trước tiên, cần bàn đến toán tử. Như đã biết, toán tử (hay phép toán) được chia làm hai loại: toán tử 1 ngôi (thực hiện trên 1 toán hạng) và toán tử 2 ngôi (thực hiện trên 2 toán hạng).

Trong C++ đã có chứa sẵn rất nhiều toán tử, tức là người ta đã cài đặt sẵn các hàm toán tử để phục vụ việc tính toán. Một số toán tử phổ biến như:

- Toán tử 1 ngôi: phép đổi dấu (-), phép phủ định (!), phép tăng 1 đơn vị (++), phép giảm 1 đơn vị (--)...

- Toán tử 2 ngôi: Cộng (+), Trừ (-), Nhân (*), Chia (/), Đồng dư (%), So sánh (>, <, >=, <=, ==, !=), Logic (&&, | |), Phép gán (=)...

Nếu đã có sẵn, tại sao ta phải cài đặt chúng?

Hãy xét ví dụ sau: giả sử ta có 3 biến nguyên (int a, b, c) hoặc 3 biến thực (x, y, z). Khi đó, các câu lệnh sau là hoàn toàn hợp lệ:

c = a + b; và z = x + y;

Lý do thật đơn giản vì phép cộng hai số nguyên hoặc 2 số thực đã được cài đặt sẵn trong C++. Khi ta viết a + b, máy tính sẽ kiểm tra xem hai toán hạng a, b thuộc kiểu gì. Khi nó xác định được a, b nguyên, nó sẽ xem toán tử cộng hai

số nguyên đã được cài đặt hay chưa. Thật may mắn là toán tử này (+) đã có sẵn và câu lệnh là hợp lệ.

Tương tự các bạn có thể dùng các phép toán: -, *, /, %, ...

Nhưng giả sử ta có 3 biến không phải kiểu nguyên thủy mà là kiểu lớp (tức là 3 đối tượng thuộc lớp nào đó). Chẳng hạn 3 đối tượng thuộc lớp Sinhvien sau: **Sinhvien a, b, c;** Khi đó việc viết: c = a + b; sẽ không còn hợp lệ nữa. Lý do thật đơn giản vì phép cộng (+) trong C++ không định nghĩa cho việc cộng hai toán hạng có kiểu Sinhvien.

Tóm lại: Hầu hết các toán tử có sẵn trong C++ thường chỉ định nghĩa trên 1 loại toán hạng nhất định. Với các toán hạng đặc biệt, các toán tử này chưa được định nghĩa.

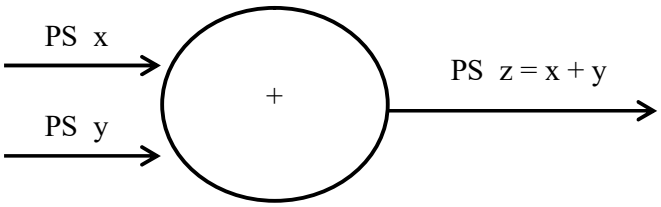
Các toán hạng đặc biệt thông thường là những biến kiểu lớp tức là các đối tượng thuộc 1 lớp nào đó. Một cách phổ biến là các đối tượng thuộc lớp: mảng, ma trận, phân số, số phức, tam thức, đa thức...

Nếu ta có 3 phân số: Phanso a, b, c; thì việc viết: c = a + b; là không hợp lệ. Nhưng trên thực tế ta có thể có nhu cầu cộng 2 phân số. Muốn vậy, trong lớp Phanso cần định nghĩa phương thức toán tử cộng hai phân số. Và khi đó, cách viết c = a + b; sẽ thực hiện việc cộng hai phân số như ta định nghĩa và cách viết đó là hợp lệ.

2.2. Cài đặt phương thức toán tử như thế nào? Tại sao phương thức toán tử 1 ngôi lại không có đối còn phương thức toán tử 2 ngôi lại chỉ có 1 đối?

Nếu ta muốn cộng hai Phân số, thì trong lớp Phân số ta cần định nghĩa phép cộng này; Nếu ta muốn cộng hai Tam thức, thì trong lớp Tam thức ta cần định nghĩa phép cộng này. Một cách tổng quát, nếu ta muốn cộng hai toán hạng là hai đối tượng thuộc lớp nào, thì trong lớp đó ta cần định nghĩa phép cộng này. Tương tự với các phép toán khác.

Bây giờ hãy xét phép cộng hai Phân số:



Hình 10. Phép cộng hai phân số x và y

Với x, y là hai đối tượng thuộc lớp PS, để thực hiện được phép cộng, trong lớp PS cần cài đặt phương thức toán tử cộng hai phân số.

Một phương thức toán tử được cài đặt theo mẫu sau:

```
<Kiểu trả về> operator <Ký hiệu phép toán>([danh sách các đối])
{
    //Thân phương thức
}
```

Trong đó:

<Kiểu trả về>: là kiểu của giá trị kết quả khi ta thực hiện phép toán. Trong ví dụ cộng hai phân số, kết quả thu được z, và z cũng là 1 phân số (tức z có kiểu PS). Do đó, phép cộng hai phân số có kiểu trả về là PS. Việc xác định kiểu trả về tương đối dễ dàng vì đa số (không phải tất cả) các phương thức toán tử đều trả về 1 kết quả (đầu ra) có kiểu trùng với kiểu của các toán hạng đầu vào.

<Ký hiệu phép toán>: là một trong các ký hiệu toán tử được dùng trong C++, chẳng hạn như: +, -, *, /, %, ++, --, ! ...Hiển nhiên ta có thể dùng một ký hiệu bất kỳ cho phép toán ta định nghĩa miễn là đảm bảo đúng về số ngôi. Ví dụ định nghĩa phép cộng (2 ngôi) thì ta có thể dùng 1 ký hiệu phép toán bất kỳ, miễn là nó 2 ngôi. Tuy nhiên, thật bất thường khi đang định nghĩa phép cộng ta lại dùng ký hiệu (-) hoặc (*)!

[danh sách các đối]: nếu có, thì nó chính là các giá trị đầu vào của phép toán. Nếu là phép toán 1 ngôi, ta có 1 đầu vào; nếu là phép toán 2 ngôi, ta có hai đầu vào (với ví dụ trên, hai đầu vào là 2 phân số x, y). **Vậy với phép toán 1 ngôi ta phải có 1 đối và phép toán 2 ngôi ta cần có 2 đối?**

Tất nhiên là như vậy. Nhưng khi định nghĩa lớp, ta luôn có 1 con trỏ đặc biệt trỏ tới 1 đối tượng ảo đại diện cho lớp gọi là con trỏ this. Con trỏ này luôn là "đối thứ nhất" của các phương thức toán tử trong lớp. Do vậy, với ví dụ trên thì this đóng vai trò như đối vào x, và ta chỉ cần 1 đối thứ 2 là y.

```
PS operator+(PS* this, PS y)
{
}
```

Nhưng con trỏ this có 1 đặc điểm rất đặc biệt là: khi ở trong lớp, nếu ta truy cập các thuộc tính, phương thức của lớp thì mặc định đó là truy cập các thuộc tính, phương thức của con trỏ this và con trỏ this không cần phải viết

tường minh. Tức là, con trỏ this có thể ẩn. Vậy phương thức toán tử trên có thể viết là:

```
PS operator+(PS y)
{
}
```

Và ta cần hiểu rằng phép cộng vẫn thực hiện trên 2 đầu vào là this và y. Chỉ có điều this ẩn đi mà thôi. Về hình thức thì phép toán 2 ngôi này có 1 đối vào (y).

Tương tự như vậy phương thức toán tử 1 ngôi có 1 đối vào và nó chính là this. Nhưng vì this ẩn nên rõ ràng về mặt hình thức phương thức toán tử 1 ngôi không có đối.

Để dễ hiểu về this, ta có thể hình dung như sau: khi ta viết a + b thì có 2 đối tượng a, b tham gia vào phép cộng này. Cả a và b đều có phép (+) nhưng ở đây chỉ dùng 1 phép (+) của 1 đối tượng (a chẳng hạn). Nếu dùng phép toán của đối tượng nào thì đối tượng ấy đóng vai trò this. Cách hiểu trên hơi ngây thơ nhưng phần nào lý giải được khái niệm con trỏ this. Để biết rõ hơn, xin nghe bài giảng.

Bây giờ cần quan tâm tới nội dung của phương thức toán tử. Với ví dụ trên, ta cần thực hiện phép cộng hai phân số **this + y** để thu được z và phương thức toán tử trả về z. Mỗi đối tượng đều có hai thuộc tính TS (tử số) và MS (mẫu số). Đối tượng z có thể hình dung như sau (chú ý ký hiệu → tương đương với dấu "."):

$$z = \frac{this \rightarrow TS}{this \rightarrow MS} + \frac{y.TS}{y.MS}$$

Vậy:

```
z.TS  =  this→TS*y.MS  +  this→MS*y.TS
z.MS  =  this→MS * y.MS
```

Vì this có thể ẩn nên ta có:

```
z.TS  =  TS * y.MS  +  MS * y.TS
z.MS  =  MS * y.MS
```

Vậy phương thức toán tử cộng hai phân số được viết như sau:

```
PS operator+(PS y)
{
    PS z;
    z.TS  =  TS*y.MS  +  MS*y.TS;
    z.MS  =  MS*y.MS;
    return z;
}
```

Thông thường một phương thức toán tử có dạng:

```
<Tên_lớp> operator <phép_toán>( [kiểu đối] [tên đối])
{
    Khai báo biến chứa kết quả (đầu ra) z
    Tính các thuộc tính của z
    return z
}
```

Với định nghĩa như vậy, phép trừ hai phân số như sau:

```
PS operator-(PS y)
{
    PS z;
    z.TS  =  TS*y.MS  -  MS*y.TS;
    z.MS  =  MS*y.MS;
    return z;
}
```

Và phép đổi dấu 1 phân số

```
PS operator-( )
{
    PS z;
    z.TS  =  - TS;
    z.MS  =   MS;
    return z;
}
```

Một cài đặt hoàn chỉnh lớp phân số với các phương thức toán tử +, - hai phân số, đổi dấu 1 phân số như trang sau:

```
class PS
{
      float TS, MS;
public:
      void nhap();
      void xuat();
      PS operator+(PS y) ;      //cộng hai ps
      PS operator-(PS y) ;      //trừ 2 ps
      PS operator-() ;          //đổi dấu 1 ps
};
```

```
void PS::nhap()
{
      cout<<"TS="; cin>>TS;
      cout<<"MS="; cin>>MS;
}
void PS ::xuat()
{
      cout<<TS<<"/"<<MS ;
}
```

```
PS PS ::operator+(PS y)
{
      PS z;
      z.TS = TS*y.MS + MS*y.TS;
      z.MS = MS*y.MS;
      return z;
}
PS PS ::operator-(PS y)
{
      PS z;
      z.TS = TS*y.MS - MS*y.TS;
      z.MS = MS*y.MS;
      return z;
}
PS PS ::operator-()
{
      PS z;
      z.TS = -TS;
      z.MS =  MS;
      return z;
}
```

2.3. Sử dụng các phương thức vừa cài đặt thế nào?

Khi đã cài đặt phương thức toán tử nào đó cho 1 lớp thì các đối tượng thuộc lớp đó có thể sử dụng phương thức toán tử đó mà không hề bị lỗi toán tử. Tức là ta có thể thực hiện toán tử đó với các toán hạng là các đối tượng thuộc lớp này.

Với ví dụ trên, giả sử ta có hai phân số: PS x, y; ta muốn tính phân số z là tổng của x và y, ta có thể viết: PS z = x+y;

Phép cộng này bình thường sẽ bị lỗi do phép cộng có sẵn trong C++ không thể thực hiện trên 2 đối tượng có kiểu PS. Nhưng vì ta đã cài đặt phương thức toán tử + cho lớp PS nên việc viết như trên là hợp lệ.

Vậy hãy xem đoạn code có sử dụng các phương thức toán tử trên:

```
void main()
{
	PS x, y;
	x.nhap() ; y.nhap() ;
	PS z = x + y ;			//sử dụng phép cộng
	PS t = x – y ;			//sử dụng phép trừ
	z = -z ;			//sử dụng phép đổi dấu
	z.xuat() ; t.xuat() ;
}
```

PHẦN 3 – CÀI ĐẶT TOÁN TỬ NHẬP/XUẤT

Toán tử nhập, xuất làm nhiệm vụ gì? cài đặt và sử dụng thế nào?

Toán tử nhập (>>) và xuất (<<) thường được gọi là toán tử đặc biệt. Nhưng trước hết hãy xem nó là gì và dùng để làm gì.

Bây giờ, hãy hình dung ta có 1 biến nguyên a: (int a). Khi đó, việc viết cout<<a; hoặc cin>>a; là hoàn toàn bình thường và quen thuộc.

Nhưng thử hình dung a không phải kiểu int hay float, hay nói chung không phải kiểu nguyên thủy (int, float, char, double, single, long...) mà a là 1 đối tượng thuộc lớp nào đó (lớp PS chẳng hạn: PS a;) khi đó việc viết cout<<a; hoặc cin>>a; lại không hợp lệ! Nguyên nhân là vì các toán tử nhập (>>) và xuất (<<) dùng trong cin và cout không định nghĩa để nhập xuất một đối tượng.

Một cách tổng quát, nếu dùng cin>> và cout<< , thông thường ta không thể nhập, xuất cho 1 biến đối tượng (bất kể nó thuộc lớp nào mà ta định nghĩa). Nhưng nếu ta biến điều không thể đó thành có thể thì thật thuận tiện cho quá trình nhập xuất các đối tượng và có lẽ các phương thức Nhap() hay Xuat() sẽ mất vai trò của nó trong các lớp.

Để làm được điều này, một cách đơn giản, ta sẽ định nghĩa các toán tử nhập (>>) và xuất (<<) cho mỗi lớp. **Khi đã định nghĩa, ta hoàn toàn có thể dùng cin>> và cout<< để nhập, xuất cho các biến đối tượng thuộc lớp đó.**

Việc định nghĩa toán tử này về cơ bản giống như định nghĩa phương thức toán tử thông thường. Nhưng 5 nguyên tắc sau luôn phải chú ý khi định nghĩa toán tử nhập, xuất. Nếu vi phạm bất kỳ nguyên tắc nào, lỗi có thể phát sinh:

[1]. Toán tử nhập, xuất không bao giờ là phương thức của lớp mà chỉ là hàm bạn của lớp.

[2]. Toán tử nhập có kiểu trả về là istream& và toán tử xuất có kiểu trả về là ostream&.

[3]. Luôn có hai đối. Đối thứ nhất có kiểu istream& (với toán tử nhập) và ostream& (với toán tử xuất). Đối thứ hai có kiểu <Tên lớp>&

[4]. Với toán tử nhập, dùng đối thứ nhất thay cho cin để nhập cho đối thứ 2. Với toán tử xuất, dùng đối thứ nhất thay cho cout để xuất đối thứ 2.

[5]. Luôn return <Đối thứ nhất>

Bây giờ hãy xét từng nguyên tắc:

Vì toán tử nhập, xuất là hàm bạn của lớp [1] nên trong thân lớp, ta cần khai báo hàm bạn theo cú pháp: friend <Nguyên mẫu toán tử nhập, xuất>; Với lớp PS ở trên, khai báo này như sau:

friend istream& operator>>(istream& x, PS& y);

friend ostream& operator<<(ostream& x, PS& y);

Dễ thấy: [2] kiểu trả về của operator>> luôn là istream& và của operator<< luôn là ostream&. Và [3] vì nó là hai hàm bạn chứ không phải là hai phương thức toán tử của lớp nên không có khái niệm con trỏ this trong hai toán tử này. Do đó ta cần định nghĩa đầy đủ 2 đối cho nó. Đối thứ nhất có kiểu istream& cho toán tử nhập (istream& x) và kiểu ostream& cho toán tử xuất (ostream& x). Đối thứ 2 có kiểu <Tên lớp>& chính là tên của lớp ta đang định nghĩa, do đó nó phải có kiểu là PS&.

Các nguyên tắc [4] và [5] giúp ta định nghĩa nội dung toán tử. Xét toán tử nhập. Mục đích của nó là nhập các thuộc tính của đối tượng PS y (tức y.TS và y.MS). Do vậy ta chỉ cần cin>>y.TS; và cin>>y.MS. Tuy nhiên [4] ta cần dùng đối thứ nhất (x) thay cho cin. và như vậy ta cần viết: x>>y.TS; x>>y.MS. Sau khi nhập xong y, ta cần [5] return <Đối thứ nhất>; tức là return x;

```
class PS
{
      float TS, MS;
public:
     friend istream& operator>>(istream& x, PS& y);
     friend ostream& operator<<(ostream& x, PS& y);
};
istream& operator>>(istream& x, PS& y)
{
      cout<<"TS="; x>>y.TS;
      cout<<"MS="; x>>y.MS;
      return x ;
}
ostream& operator<<(ostream& x, PS& y)
{
      x<<y.TS<<"/"<<y.MS ;
      return x ;
}
```

Nói chung, với toán tử nhập, ta có thể dùng cout nhưng không thể dùng cin mà hãy dùng đối thứ nhất (x) thay cho cin; toán tử xuất thì ngược lại, ta luôn dùng x thay cho cout khi xuất các thành phần của y.

Sau khi định nghĩa xong, ta có thể sử dụng cin, cout để nhập xuất các đối tượng thuộc lớp mà ta vừa định nghĩa toán tử nhập xuất.

```
void main()
{
      PS x, y;
      cout<<"Nhập ps x :" ;
      cin>>x ;                //sử dụng toán tử nhập vừa định nghĩa
      cout<<"Nhập ps y :" ;
      cin>>y ;                //sử dụng toán tử nhập vừa định nghĩa
      cout<<"Phân số vừa nhập:" ;
      cout<<x<<endl<<y ;    //sử dụng toán tử xuất vừa định nghĩa
}
```

Ví dụ minh họa: Cài đặt và sử dụng toán tử nhập, xuất cho lớp Hàng gồm các thuộc tính: Mã hàng, Tên hàng, Số lượng, Đơn giá.

```
class Hang
{
      char MaH[30];
      char TenH[30];
      int SL;
      float DG;
public:
     friend istream& operator>>(istream& x, Hang& y);
     friend ostream& operator<<(ostream& x, Hang& y);
};
istream& operator>>(istream& x, Hang& y)
{
      cout<<"Mã Hang :";   x>>y.MaH;
      cout<<"Tên hàng :";  x>>y.TenH;
       cout<<"Số lượng :";  x>>SL;
      cout<<"Đơn giá :";   x>>DG;
      return x ;
}
ostream& operator<<(ostream& x, Hang& y)
{
      cout<<"Mã Hang :";    x<<y.MaH<<endl;
      cout<<"Tên hàng :";   x<<y.TenH<<endl;
      cout<<"Số lượng :";   x<<SL<<endl;
      cout<<"Đơn giá :";    x<<DG<<endl;
      return x ;
}
void main()
{
      Hang x;
      cout<<"Nhập x"; cin>>x;
      cout<<"Hàng vừa nhập:"<<x;
}
```