

# Chương 3

## Định nghĩa phép toán

# Nội dung

1. Mở đầu
2. Hàm phép toán
3. Chuyển kiểu
4. Gán và khởi động
5. Một số phép toán thông dụng

## 3.1 Mở đầu

- Trong C++, các kiểu dữ liệu nội tại (built-in data types): int, long, float, double, char... cùng với các phép toán +, -, \*, /... cung cấp một cài đặt cụ thể của khái niệm trong thế giới thực. Các phép toán như trên cho phép người sử dụng tương tác với chương trình theo một giao diện tự nhiên tiện lợi.
- Người sử dụng có thể có nhu cầu tạo các kiểu dữ liệu mới mà ngôn ngữ không cung cấp như ma trận, đa thức, số phức, vector...
- Lớp trong C++ cung cấp một phương tiện để quy định và biểu diễn các loại đối tượng như trên. Đồng thời tạo khả năng định nghĩa phép toán cho kiểu dữ liệu mới, nhờ đó người sử dụng có thể thao tác trên kiểu dữ liệu mới định nghĩa theo một giao diện thân thiện tương tự như kiểu có sẵn.

# Mở đầu

- Một phép toán là một ký hiệu mà nó thao tác trên dữ liệu, dữ liệu được thao tác được gọi là toán hạng, bản thân ký hiệu được gọi là phép toán.
- Phép toán có hai toán hạng được gọi là phép toán hai ngôi (nhị phân), chỉ có một toán hạng được gọi là phép toán một ngôi (đơn phân).
- Sau khi định nghĩa phép toán cho một kiểu dữ liệu mới, ta có thể sử dụng nó một cách thân thiện. Ví dụ:

```
SoPhuc z(1, 3), z1(2, 3.4), z2(5.1, 4);
```

```
z = z1 + z2;
```

```
z = z1 + z2*z1 + SoPhuc(3, 1);
```

## 3.2 Hàm phép toán

- Bản chất của phép toán là ánh xạ, vì vậy định nghĩa phép toán là *định nghĩa hàm*. Tất cả các phép toán có trong C++ đều có thể được định nghĩa.

+	-	*	/	%	^	&		~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
=	<<	>>	<<=	>>=	==	!=	<=	>=	&&
	++	--	->*	[]	()	new	delete		

- Ta định nghĩa phép toán bằng hàm có tên đặc biệt bắt đầu bằng từ khoá operator theo sau bởi ký hiệu phép toán cần định nghĩa.

## Ví dụ minh họa – Lớp PhanSo

```
typedef int bool;
typedef int Item;
const bool    false = 0, true = 1;
long USCLN(long x, long y)
{
    long r;
    x = abs(x); y = abs(y);
    if (x == 0 || y == 0) return 1;
    while ((r = x % y) != 0)
    {
        x = y;
        y = r;
    }
    return y;
}
```

# Ví dụ minh họa – Lớp PhanSo

```
class PhanSo
{
    long tu, mau;
    void UocLuoc();
public:
    PhanSo(long t, long m) {Set(t,m);}
    void Set(long t, long m);
    long LayTu() const {return tu;}
    long LayMau() const {return mau;}
    PhanSo Cong(PhanSo b) const;
    PhanSo operator + (PhanSo b) const;
    PhanSo operator - () const {return PhanSo(-tu,
        mau);}
    bool operator == (PhanSo b) const;
    bool operator != (PhanSo b) const;
    void Xuat() const;
};
```

## Ví dụ minh họa – Lớp PhanSo

```
void PhanSo::UocLuoc()  
{  
    long usc = USCLN(tu, mau);  
    tu /= usc; mau /= usc;  
    if (mau < 0)  
        mau = -mau, tu = -tu;  
    if (tu == 0) mau = 1;  
}
```



# Ví dụ minh họa – Lớp PhanSo

```
void PhanSo::Set(long t, long m)
{
    if (m)
    {
        tu = t;
        mau = m;
        UocLuoc();
    }
}

void PhanSo::Xuat() const
{
    cout << tu;
    if (tu != 0 && mau != 1)
        cout << "/" << mau;
}
```

# Ví dụ minh họa – Lớp PhanSo

```
PhanSo PhanSo::Cong(PhanSo b) const
{
    return PhanSo(tu*b.mau + mau*b.tu, mau*b.mau);
}
PhanSo PhanSo::operator + (PhanSo b) const
{
    return PhanSo(tu*b.mau + mau*b.tu, mau*b.mau);
}
bool PhanSo::operator == (PhanSo b) const
{
    return tu*b.mau == mau*b.tu;
}
```

# Một số ràng buộc của phép toán

- Khi định nghĩa phép toán thì không được thay đổi các đặc tính mặc nhiên của phép toán như độ ưu tiên, số ngôi; không được sáng chế phép toán mới như mod, \*\*, ...
- Hầu hết các phép toán không ràng buộc ý nghĩa, chỉ một số trường hợp cá biệt như phép toán gán (operator =), lấy phần tử qua chỉ số (operator []), phép gọi hàm (operator ()), và phép lấy thành phần (operator ->\*) đòi hỏi phải được định nghĩa là hàm thành phần để toán hạng thứ nhất có thể là một đối tượng trái (lvalue) .
- Các phép toán có sẵn có cơ chế kết hợp được suy diễn từ các phép toán thành phần, ví dụ:

```
a += b;    // a = (a+(b)) ;
```

```
a *= b;    // a = (a*(b)) ;
```



# Một số ràng buộc của phép toán

- Điều trên không đúng đối phép toán định nghĩa cho các kiểu dữ liệu do người sử dụng định nghĩa. Nghĩa là ta phải chủ động định nghĩa phép toán  $+=$ ,  $-=$ ,  $*=$ ,  $>>=$ , ... dù đã định nghĩa phép gán và các phép toán  $+$ ,  $-$ ,  $*$ ,  $>>$ , ...
- Ràng buộc trên cho phép người sử dụng chủ động định nghĩa phép toán nào trước ( $+=$  trước hay  $+$  trước).

# Hàm thành phần và toàn cục

- Trong ví dụ trên, ta định nghĩa hàm thành phần có tên đặc biệt bắt đầu bằng từ khoá operator theo sau bởi tên phép toán cần định nghĩa. Sau khi định nghĩa phép toán, ta có thể dùng theo giao diện tự nhiên:

```
void main()  
{  
    PhanSo a(2,3), b(3,4), c(0,1), d(0,1);  
    c = a.Cong(b);  
    d = a + b;    // d = a.operator + (b);  
    cout << "c = "; c.Xuat(); cout << "\n";  
    cout << "d = "; d.Xuat(); cout << "\n";  
    cout << "c == d = " << (c == d) << "\n";  
    cout << "c != d = " << (c != d) << "\n";  
    (-a).Xuat(); // (a.operator -()).Xuat();  
}
```

# Hàm thành phần và hàm toàn cục

- Trong hầu hết các trường hợp, ta có thể định nghĩa phép toán bằng thành phần hoặc dùng hàm toàn cục.
- Khi định nghĩa phép toán bằng hàm thành phần, số tham số ít hơn số ngôi một vì đã có một tham số ngầm định là đối tượng gọi phép toán (toán hạng thứ nhất). Phép toán 2 ngôi cần 1 tham số và phép toán 1 ngôi không có tham số:

```
a - b; // a.operator -(b);
```

```
-a; // a.operator -();
```

- Khi định nghĩa phép toán bằng hàm toàn cục, số tham số bằng số ngôi, Phép toán 2 ngôi cần 2 tham số và phép toán một ngôi cần một tham số:

```
a - b; // operator -(a,b);
```

```
-a; // operator -(a);
```

# Hàm thành phần và hàm toàn cục

```
class PhanSo
{
    long tu, mau;
    void UocLuoc();
public:
    PhanSo(long t, long m) {Set(t,m);}
    void Set(long t, long m);
    long LayTu() const {return tu;}
    long LayMau() const {return mau;}
    PhanSo operator + (PhanSo b) const;
    friend PhanSo operator - (PhanSo a, PhanSo b);
    PhanSo operator -() const {return PhanSo(-tu,
                                                mau);}

    bool operator == (PhanSo b) const;
    bool operator != (PhanSo b) const;
    void Xuat() const;
```



# Hàm thành phần và hàm toàn cục


```
PhanSo PhanSo::operator + (PhanSo b) const
{
    return PhanSo(tu*b.mau + mau*b.tu, mau*b.mau);
}
PhanSo operator - (PhanSo a, PhanSo b)
{
    return PhanSo(a.tu*b.mau - a.mau*b.tu,
                  a.mau*b.mau);
}
```



# Hàm thành phần và hàm toàn cục

```
void main()  
{  
    PhanSo a(2,3), b(3,4), c(0,1), d(0,1);  
    c = a + b;    // d = a.operator + (b);  
    d = a - b;    // d = operator - (a,b);  
    cout << "c = "; c.Xuat(); cout << "\n";  
    cout << "d = "; d.Xuat(); cout << "\n";  
}
```

# Hàm thành phần và toàn cục

- Khi có thể định nghĩa bằng hai cách, dùng hàm thành phần sẽ gọn hơn. Tuy nhiên chọn hàm thành phần hay hàm toàn cục hoàn toàn tùy theo sở thích của người sử dụng.
- Dùng hàm toàn cục thuận tiện hơn khi ta có nhu cầu chuyển kiểu ở toán hạng thứ nhất (Xem [3.3](#)).
- Các phép toán `=`, `[]`, `()`, `->*` như đã nói trên bắt buộc phải được định nghĩa là hàm thành phần vì toán hạng thứ nhất phải là `lvalue`.
- Khi định nghĩa phép toán có toán hạng thứ nhất thuộc lớp đang xét thì có thể dùng hàm thành phần hoặc hàm toàn cục.
- Tuy nhiên, nếu toán hạng thứ nhất không thuộc lớp đang xét thì phải định nghĩa bằng hàm toàn cục (Xem [ví dụ](#)). Trường hợp thông dụng là định nghĩa phép toán 

# Ví dụ sử dụng hàm toàn cục

```
class PhanSo
{
    long tu, mau;
public:
    PhanSo(long t, long m) {Set(t,m);}
    PhanSo operator + (PhanSo b) const;
    PhanSo operator + (long b) const
        {return PhanSo(tu + b*mau, mau);}
    void Xuat() const;
};

PhanSo a(2,3), b(4,1);
a + b; // a.operator + (b): Ok
a + 5; // a.operator + (5): Ok
3 + a; // 3.operator + (a): SAI
```

# Ví dụ sử dụng hàm toàn cục

```
class PhanSo
{
    long tu, mau;
public:
    PhanSo(long t, long m) {Set(t,m);}
    PhanSo operator + (PhanSo b) const;
    PhanSo operator + (long b) const;
        {return PhanSo(tu + b*mau, mau);}
    friend PhanSo operator + (long a, PhanSo b);
};
PhanSo operator + (long a, PhanSo b)
    { return PhanSo(a*b.mau+b.tu, b.mau); }
//...
PhanSo a(2,3), b(4,1), c(0,1);
c = a + b; // a.operator + (b): Ok
c = a + 5; // a.operator + (5): Ok
c = 3 + a; // operator + (3,a): Ok
```

## 3.3 Chuyển kiểu (type conversions)

- Về mặt khái niệm, ta có thể thực hiện trộn lẫn phân số và số nguyên trong các phép toán số học và quan hệ. Chẳng hạn có thể cộng phân số và phân số, phân số và số nguyên, số nguyên và phân số. Điều đó cũng đúng cho các phép toán khác như trừ, nhân, chia, so sánh. Nghĩa là ta có nhu cầu định nghĩa phép toán  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $<$ ,  $>$ ,  $==$ ,  $!=$ ,  $<=$ ,  $>=$  cho phân số và số nguyên.
- Sử dụng cách định nghĩa các hàm như trên cho phép toán  $+$  và làm tương tự cho các phép toán còn lại ta có thể thao tác trên phân số và số nguyên.
- Điều đó cũng áp dụng tương tự cho các kiểu dữ liệu khác do người sử dụng định nghĩa.

# Chuyển kiểu

```
class PhanSo
{
    long tu, mau;
public:
    PhanSo(long t, long m) {Set(t,m);}
    void Set(long t, long m);
    PhanSo operator + (PhanSo b) const;
    PhanSo operator + (long b) const;
    friend PhanSo operator + (long a, PhanSo b);
    PhanSo operator - (PhanSo b) const;
    PhanSo operator - (long b) const;
    friend PhanSo operator - (long a, PhanSo b);
    PhanSo operator * (PhanSo b) const;
    PhanSo operator * (long b) const;
    friend PhanSo operator * (long a, PhanSo b);
    PhanSo operator / (PhanSo b) const;
    PhanSo operator / (long b) const;
    // con tiep trang sau
};
```

# Chuyển kiểu

```
// tiếp theo
friend PhanSo operator / (int a, PhanSo b);
PhanSo operator -() const;
bool operator == (PhanSo b) const;
bool operator == (long b) const;
friend bool operator == (long a, PhanSo b);
bool operator != (PhanSo b) const;
bool operator != (long b) const;
friend bool operator != (int a, PhanSo b);
bool operator < (PhanSo b) const;
bool operator < (long b) const;
friend bool operator < (int a, PhanSo b);
bool operator > (PhanSo b) const;
bool operator > (long b) const;
friend bool operator > (int a, PhanSo b);
bool operator <= (PhanSo b) const;
```



# Chuyển kiểu

- Với các khai báo như trên, ta có thể sử dụng phân số và số nguyên lẫn lộn trong một biểu thức:

```
void main()  
{  
    PhanSo a(2,3), b(1,4), c(3,1), d(2,5);  
    a = b * -c;  
    c = (b+2) * 2/a;  
    d = a/3 + (b*c-2)/5;  
}
```

- Tuy nhiên, viết các hàm tương tự nhau lặp đi lặp lại là cách tiếp gây mệt mỏi và dễ sai sót. Ta thể học theo cách chuyển kiểu ngầm định mà C++ áp dụng cho các kiểu dữ liệu có sẵn:

```
double r = 2;    // double x = double(2);  
double s = r + 3; // double s = r + double(3);  
cout << sqrt(9); // cout << sqrt(double(9));
```





## 3.3.1 Chuyển kiểu bằng PTTL

- Khi cần tính toán một biểu thức, nếu kiểu dữ liệu chưa hoàn toàn khớp, trình biên dịch sẽ tìm cách chuyển kiểu. Trong một biểu thức số học, nếu có sự tham gia của một toán hạng thực, các thành phần khác sẽ được chuyển sang số thực. Các trường hợp khác chuyển kiểu được thực hiện theo nguyên tắc nâng cấp (int sang long, float sang double ...). Ta có thể học theo cách chuyển kiểu từ số nguyên sang số thực để chuyển từ số nguyên sang phân số.
- Số nguyên có thể chuyển sang số thực một cách ngầm định khi cần vì có thể tạo được một số thực từ số nguyên.

```
double r = 2; // double r = double(2);
```

- Để có thể chuyển từ số nguyên sang phân số, ta cần dạy trình biên dịch cách tạo phân số từ số nguyên.

```
PhanSo a = 3; // PhanSo a = PhanSo(3);
```

```
// Hay PhanSo a(3);
```



# Chuyển kiểu bằng ph. thức thiết lập

- Việc tạo phân số từ số nguyên chính là phép gọi phương thức thiết lập. Nói cách khác ta cần xây dựng một phương thức thiết lập để tạo một phân số với tham số là số nguyên:

```
class PhanSo
```

```
{
```

```
    long tu, mau;
```

```
public:
```

```
    PhanSo(long t, long m) {Set(t,m);}
```

```
    PhanSo(long t) {Set(t,1);} // Co the chuyen  
                           kieu tu so nguyen sang phan so
```

```
    void Set(long t, long m);
```

```
    PhanSo operator + (PhanSo b) const;
```

```
    friend PhanSo operator + (int a, PhanSo b);
```

```
    PhanSo operator - (PhanSo b) const;
```

```
    friend PhanSo operator - (int a, PhanSo b);
```

```
    //
```

# Chuyển kiểu bằng ph. thức thiết lập

- Phương thức thiết lập với một tham số là số nguyên như trên hàm ý rằng một số nguyên *là một* phân số, có thể chuyển kiểu ngầm định từ số nguyên sang phân số.
- Khi đó ta có thể giảm bớt việc khai báo và định nghĩa phép toán + phân số và số nguyên, cơ chế chuyển kiểu tự động cho phép thực hiện thao tác cộng đó, nói cách khác có thể giảm việc định nghĩa 3 phép toán xuống còn 2:

```
//...
```

```
PhanSo a(2,3), b(4,1), c(0);
```

```
PhanSo d = 5; // PhanSo d = PhanSo(5);
```

```
// PhanSo d(5);
```

```
c = a + b; // c = a.operator + (b) : Ok
```

```
c = a + 5; // c = a.operator + (PhanSo(5)) : Ok
```

```
c = 3 + a; // c = operator + (3,a) : Ok
```



# Chuyển kiểu bằng ph. thức thiết lập

- Ta có thể giảm số phép toán cần định nghĩa từ 3 xuống 1 bằng cách dùng hàm toàn cục, khi đó có thể chuyển kiểu cả hai toán hạng.

```
class PhanSo
{
    long tu, mau;
public:
    PhanSo(long t, long m) {Set(t,m);}
    PhanSo(long t) {Set(t,1);} // Co the chuyen
                             kieu tu so nguyen sang phan so
    void Set(long t, long m);
    friend PhanSo operator + (PhanSo a, PhanSo b);
    friend PhanSo operator - (PhanSo a, PhanSo b);
    //...
};
```

# Chuyển kiểu bằng PTTL

- Khi đó cơ chế chuyển kiểu có thể được thực hiện cho cả hai toán hạng.

```
//...
```

```
PhanSo a(2,3), b(4,1), c(0);
```

```
PhanSo d = 5; // PhanSo d = PhanSo(5);
```

```
c = a + b; // c = operator + (a,b) : Ok
```

```
c = a + 5; // c = operator + (a,PhanSo(5)) : Ok
```

```
// Hay c = a + PhanSo(5);
```

```
c = 3 + a; // c = operator + (PhanSo(3),a) : Ok
```

```
// Hay c = PhanSo(3) + a
```

- (?) Nếu viết

```
c = 5 + 7;
```

Thì có thể chuyển kiểu cả hai toán hạng được không?

```
// c = PhanSo operator + (PhanSo(5), PhanSo(7));
```

# Hai cách chuyển kiểu bằng P TTL

- Chuyển kiểu bằng phương thức thiết lập được thực hiện theo nguyên tắc có thể tạo một đối tượng mới (phân số) từ một đối tượng đã có (số nguyên). Điều đó có thể được thực hiện theo cách nêu trên, hoặc dùng phương thức thiết lập với tham số có giá trị mặc nhiên.

```
class PhanSo
{
    long tu, mau;
public:
    PhanSo(long t, long m)
        {Set(t,m);}
    PhanSo(long t)
        {Set(t,1);}
    // ...
};
```

```
class PhanSo
{
    long tu, mau;
public:
    PhanSo(long t, long m
        = 1) {Set(t,m);}
    // ...
};
```

# Khi nào chuyển kiểu bằng PTTL

- Ta dùng chuyển kiểu bằng phương thức thiết lập khi thoả hai điều kiện sau:
  1. Chuyển từ kiểu đã có (số nguyên) sang kiểu đang định nghĩa (phân số).
  2. Có quan hệ là một từ kiểu đã có sang kiểu đang định nghĩa (một số nguyên là *một* phân số).
- Các ví dụ dùng chuyển kiểu bằng phương thức thiết lập bao gồm: Chuyển từ số thực sang số phức, char \* sang String, số thực sang điểm trong mặt phẳng.

## 3.3.2 Ch kiểu bằng ph. toán ch kiểu

- Sử dụng phương thức thiết lập để chuyển kiểu như trên tiện lợi trong một số trường hợp nhưng nó cũng có một số nhược điểm:
  1. Muốn chuyển từ kiểu đang định nghĩa sang một kiểu đã có, ta phải sửa đổi kiểu đã có.
  2. Không thể chuyển từ kiểu đang định nghĩa sang kiểu cơ bản có sẵn.
  3. Phương thức thiết lập với một tham số sẽ dẫn đến cơ chế chuyển kiểu tự động có thể không mong muốn.
- Các nhược điểm trên có thể được khắc phục bằng cách định nghĩa phép toán chuyển kiểu.
- Phép toán chuyển kiểu là hàm thành phần có dạng  
**X::operator T()**

Với phép toán trên, sẽ có cơ chế chuyển kiểu tự động từ kiểu đang được định nghĩa X sang kiểu đã có T.





# Dùng phép toán chuyển kiểu

- Ta dùng phép toán chuyển kiểu khi định nghĩa kiểu mới và muốn tận dụng các phép toán của kiểu đã có.

```
class String
{
    char *p;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s2) {p = strdup(s2.p);}
    ~String() {delete [] p;}
    String& operator = (const String& p2);
    int Length() const {return strlen(p);}
    void ToUpper() {strupr(p);}
    friend ostream& operator << (ostream &o, const
    String& s);
    operator const char *() const {return p;}
    operator char *() {return p;}
};
```



# Dùng phép toán chuyển kiểu

```
ostream & operator << (ostream &o, const String& s)
{
    return o << s.p;
}
```

```
void main()
{
    String s("Nguyen van A");
    cout << s.Length() << "\n";
    cout << strlen(s) << "\n";
    if (strcmp(s, "Nguyen van A") == 0)
        cout << "Hai chuoì bang nhau\n";
    else
        cout << "Hai chuoì khác nhau\n";
    strupr(s);
    cout << s << "\n";
}
```



# Ví dụ về phép toán chuyển kiểu

- Ví dụ sau minh họa rõ thêm nhu cầu chuyển kiểu. Một NumStr có thể chuyển sang số thực.

```
class NumStr
{
    char *s;
public:
    NumStr(char *p) {s = dupstr(p);}
    operator double() const {return atof(s);}
    friend ostream & operator << (ostream &o,
                                   NumStr &ns);
};

ostream & operator << (ostream &o, NumStr &ns)
{
    return o << ns.s;
}
```

# Ví dụ về phép toán chuyển kiểu

```
void main()
{
    NumStr s1("123.45"), s2("34.12");
    cout << "s1 = " << s1 << "\n";
    // Xuat 's1 = 123.45' ra cout
    cout << "s2 = " << s2 << "\n";
    // Xuat 's2 = 34.12' ra cout
    cout << "s1 + s2 = " << s1 + s2 << "\n";
    // Xuat 's1 + s2 = 157.57' ra cout
    cout << "s1 + 50 = " << s1 + 50 << "\n";
    // Xuat 's1 + 50 = 173.45' ra cout
    cout << "s1 * 2 = " << s1 * 2 << "\n";
    // Xuat 's1 * 2 = 246.9' ra cout
    cout << "s1 / 2 = " << s1 / 2 << "\n";
    // Xuat 's1 / 2 = 61.725' ra cout
}
```

# Dùng phép toán chuyển kiểu

- Phép toán chuyển kiểu cũng được dùng để biểu diễn quan hệ là một từ kiểu đang định nghĩa sang kiểu đã có.

```
class PhanSo
{
    long tu, mau;
    void UocLuoc();
public:
    PhanSo(long t = 0, long m = 1) {Set(t,m);}
    void Set(long t, long m);
    friend PhanSo operator + (PhanSo a, Phan So b);
    void Xuat() const;
    operator double() const {return double(tu)/mau;}
};
//...
PhanSo a(9,4);
cout << sqrt(a) << "\n";
    // cout << sqrt(a.operator double()) << "\n";
```

## 3.3.3 Sự nhập nhằng

- Nhập nhằng là hiện tượng xảy ra khi trình biên dịch tìm được ít nhất hai cách chuyển kiểu để thực hiện một việc tính toán nào đó.

```
int Sum(int a, int b)
{
```

```
    return a+b;
```

```
}
```

```
double Sum(double a, double b)
```

```
{
```

```
    return a+b;
```

```
}
```

# Sự nhập nhằng

- Lưu ý rằng hiện tượng nhập nhằng không xảy ra khi thực hiện phép toán số học mà ngôn ngữ cung cấp

```
void main()
{
    int a = 3, b = 7;
    double r = 3.2, s = 6.3;
    cout << a+b << "\n";           // Ok
    cout << r+s << "\n";           // Ok
    cout << a+r << "\n";           // Ok: double(a)+r
    cout << Sum(a,b) << "\n";      // Ok Sum(int, int)
    cout << Sum(r,s) << "\n";      // Ok Sum(double, double)
    cout << Sum(a,r) << "\n";
    // Nhập nhằng, Sum(int, int) hay Sum(double, double)
}
```

# Sự nhập nhằng

- Hiện tượng nhập nhằng thường xảy ra người sử dụng định nghĩa lớp và qui định cơ chế chuyển kiểu bằng phương thức thiết lập và/hay phép toán chuyển kiểu.

```
class PhanSo
{
    long tu, mau;
    void UocLuoc();
    int SoSanh(PhanSo b);
public:
    PhanSo(long t = 0, long m = 1) {Set(t,m);}
    void Set(long t, long m);
    friend PhanSo operator + (PhanSo a, PhanSo b);
    friend PhanSo operator - (PhanSo a, PhanSo b);
    friend PhanSo operator * (PhanSo a, PhanSo b);
    friend PhanSo operator / (PhanSo a, PhanSo b);
    operator double() const {return double(tu)/mau;}
```



# Sự nhập nhằng

- Lớp phân số có hai cơ chế chuyển kiểu, từ số nguyên sang phân số nhờ phương thức thiết lập và từ phân số sang số thực nhờ phép toán chuyển kiểu.
- Tuy nhiên hiện tượng nhập nhằng xảy ra khi ta thực hiện phép cộng phân số và số nguyên hoặc phân số với số thực.

```
void main()
{
    PhanSo a(2,3), b(3,4), c;
    cout << sqrt(a) << "\n"; // Ok
    c = a + b; // Ok
    c = a + 2; // Nhập nhằng
    c = 2 + a; // Nhập nhằng
    double r = 2.5 + a; // Nhập nhằng
    r = a + 2.5; // Nhập nhằng
}
```

# Sự nhập nhằng

- Để tránh hiện tượng nhập nhằng như trên, ta chuyển kiểu một cách tường minh.

```
void main()
{
    PhanSo a(2, 3), b(3, 4), c;
    C = a + b;    // Ok
    c = a + 2;    // Nhập nhằng
    c = 2 + a;    // Nhập nhằng
    c = 2.5 + a;  // Nhập nhằng
    c = a + 2.5;  // Nhập nhằng
    c = a + PhanSo(2); // Ok
    c = PhanSo(2) + a; // Ok
    cout << double(a) + 2.5 << "\n"; // Ok
    cout << 2.5 + double(a) << "\n"; // Ok
}
```

# Sự nhập nhằng

- Tuy nhiên việc chuyển kiểu tường minh làm mất đi sự tiện lợi của cơ chế chuyển kiểu tự động. Thông thường ta phải chịu hy sinh. Trong lớp phân số ta loại bỏ phép toán chuyển kiểu.
- Sự nhập nhằng còn xảy ra nếu việc chuyển kiểu đòi hỏi được thực hiện qua hai cấp.

## 3.4 Gán và khởi động

- Đối với lớp với đối tượng có nhu cầu cấp phát tài nguyên, việc khởi động đối tượng đòi hỏi phải có phương thức thiết lập sao chép để tránh hiện tượng các đối tượng chia sẻ tài nguyên dẫn đến một vùng tài nguyên bị giải phóng nhiều lần khi các đối tượng bị huỷ bỏ. Việc sao chép có thể là sâu hoặc nông (Xem [chương 2](#)).
- Khi thực hiện phép gán trên các đối tượng cùng kiểu, cơ chế gán mặc nhiên là gán từng thành phần. Điều này làm cho đối tượng bên trái của phép gán “bỏ rơi” tài nguyên cũ và chia sẻ tài nguyên với đối tượng ở vế phải. Xét lớp String sau đây:

# Gán và khởi động

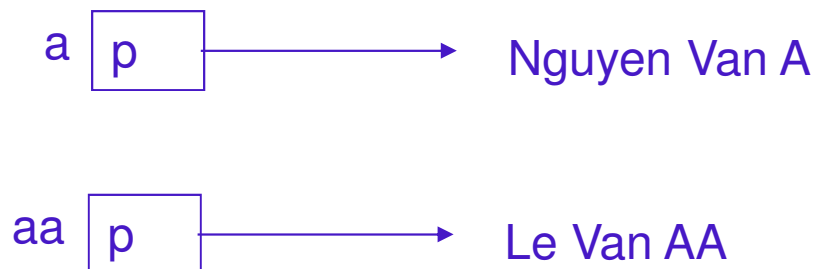
```
class String
{
    char *p;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s) {p = strdup(s.p);}
    ~String() {cout << "delete " << (void *)p <<
                "\n"; delete [] p;}
    void Output() const {cout << p;}
};

void main()
{
    clrscr();
    String a("Nguyen Van A");
    String b = a; // Khởi động
    String aa = "La van AA";
    cout << "aa = "; aa.Output(); cout << "\n";
    aa = a;          // Gán
    cout << "aa = "; aa.Output(); cout << "\n";
}
```

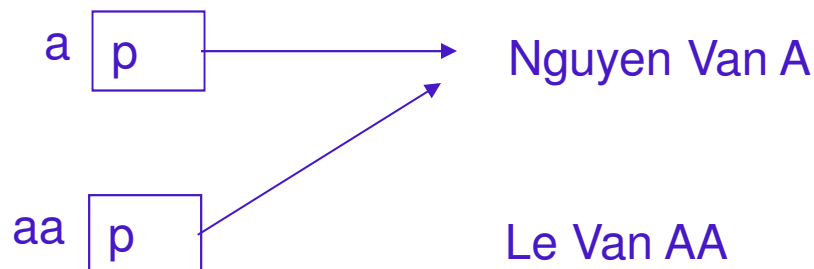


# Gán và khởi động

## ■ Trước khi gán



## ■ Sau khi gán



## ■ Khi thực hiện đoạn chương trình trên ta được xuất liệu sau:

`aa = La van AA`

`aa = Nguyen Van A`

`delete 0x0d36`

`delete 0x0d48`

`delete 0x0d36`

`Null pointer assignment`

## ■ Phần tài nguyên (cũ) của aa bị mất dấu không thể giải phóng, phần tài nguyên của a bị chia sẻ với aa (mới).

# Gán và khởi động

- Lỗi sai trên được khắc phục bằng cách định nghĩa phép gán cho lớp String.

```
class String
{
    char *p;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s) {p = strdup(s.p);}
    ~String() {cout << "delete "<< (void *)p <<
"\n"; delete [] p;}
    String & operator = (const String &s);
    void Output() const {cout << p;}
};
```

# Gán và khởi động

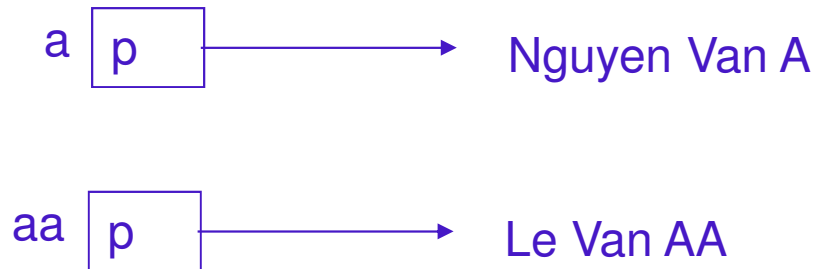
```
String & String::operator = (const String &s)
{
    if (this != &s)
    {
        delete [] p;
        p = strdup(s.p);
    }
    return *this;
}
```

- Phép gán thực hiện hai thao tác chính là dọn dẹp tài nguyên cũ và sao chép mới. Thao tác dọn dẹp tương đương phương thức huỷ bỏ và thao tác sao chép tương đương phương thức thiết lập sao chép.
- Khi có phép gán được định nghĩa như trên, đoạn chương trình kể trên cho xuất liệu:

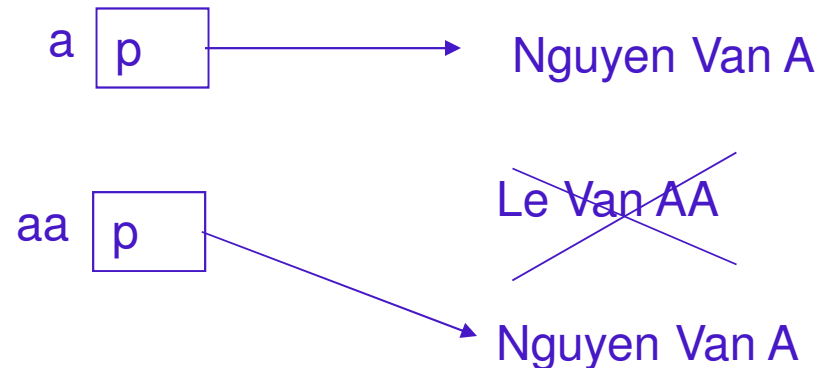


# Gán và khởi động

## ■ Trước khi gán



## ■ Sau khi gán



## ■ Khi thực hiện đoạn chương trình trên ta được xuất liệu sau:

```
aa = La van AA  
aa = Nguyen Van A  
delete 0x0d5a  
delete 0x0d48  
delete 0x0d36
```

## ■ Phần tài nguyên (cũ) của aa được giải phóng, và được tạo tài nguyên mới.

# Gán và khởi động

- Phép gán cũng có thể được thực hiện với các đối tượng chứa sê tài nguyên bằng cơ chế sao chép nông.

```
class StringRep
{
friend class String;
    char *p;
    int n;
    StringRep(const char *s) {p = strdup(s); n = 1;}
    ~StringRep() { cout << "delete " << (void *)p <<
        "\n"; delete [] p;}
};
```

# Gán và khởi động

```
class String
{
    StringRep *rep;
    void CleanUp() { if (--rep->n <= 0) delete rep; }
    void Copy(const String &s){rep = s.rep; rep->n++;}
public:
    String(const char *s = "") {rep = new
    StringRep(s);}
    String(const String &s) {Copy(s);}
    ~String() {CleanUp();}
    String & operator = (const String &s);
    void Output() const {cout << rep->p;}
};
```

# Gán và khởi động

```
String & String::operator = (const String &s)
{
    if (this != &s)    {
        CleanUp();
        Copy(s);
    }
    return *this;
}

void main()
{
    clrscr();
    String a("Nguyen Van A");
    String b = a; // Khoi dong
    String aa = "La van AA";
    cout << "aa = "; aa.Output(); cout << "\n";
    aa = a;
```



# Gán và khởi động

- Xuất liệu khi thực hiện hàm main trên như sau:

```
aa = La van AA
```

```
delete 0x0d58
```

```
aa = Nguyen Van A
```

```
delete 0x0d3e
```

- Lưu ý: Ta chỉ cần định nghĩa phép gán khi đối tượng có nhu cầu cấp phát tài nguyên. Khi đó lớp phải có đủ bộ bốn: phương thức thiết lập, phương thức thiết lập sao chép, phương thức huỷ bỏ và phép gán.
- Trong một lớp có định nghĩa hai phép toán: gán và gán kết hợp với phép toán khác, nên định nghĩa phép toán += trước và phép toán + sau hay ngược lại. Tại sao?

## 3.5 Một số phép toán thông dụng

- Phép toán << và >>
- Phép toán []
- Phép toán ()
- Phép toán ++ và --

## 3.5.1 Phép toán << và >>

- << và >> là hai phép toán thao tác trên từng bit khi các toán hạng là số nguyên.
- C++ định nghĩa lại hai phép toán để dùng với các đối tượng thuộc lớp ostream và istream để thực hiện các thao tác xuất, nhập.
- Khi định nghĩa hai phép toán trên, cần thể hiện ý nghĩa sau:

`a >> b; // chuyển dữ liệu của a vào b`

`a << b; // chuyển nội dung của b vào a`

`cout << a << "\n"; // chuyển a và "\n" vào cout`

`cin >> a >> b; // chuyển dl từ cin vào a và b`

- Lớp ostream (dòng dữ liệu xuất) định nghĩa phép toán << áp dụng cho các kiểu dữ liệu cơ bản (nguyên, thực, char \*,...).
- Lớp istream (dòng dữ liệu nhập) định nghĩa phép toán >> áp dụng cho các kiểu dữ liệu cơ bản (nguyên, thực, char

## Phép toán << và >>

- cout, cerr là các biến thuộc lớp ostream đại diện cho thiết bị xuất chuẩn (mặc nhiên là màn hình) và thiết bị báo lỗi chuẩn (luôn luôn là màn hình).
- cin là một đối tượng thuộc lớp istream đại diện cho thiết bị nhập chuẩn, mặc nhiên là bàn phím.
- Với khai báo của lớp ostream như trên ta có thể thực hiện phép toán << với toán hạng thứ nhất là một dòng dữ liệu xuất (cout, cerr, tập tin...), toán hạng thứ hai thuộc các kiểu cơ bản (nguyên, thực, char \*, con trỏ...).
- Tương tự, ta có thể áp dụng phép toán >> với toán hạng thứ nhất thuộc lớp istream (ví dụ cin), toán hạng thứ hai là tham chiếu đến kiểu cơ bản hoặc con trỏ (nguyên, thực, char \*).



# Lớp ostream

```
class ostream : virtual public ios
{
public:
    // Formatted insertion operations
    ostream & operator<< ( signed char);
    ostream & operator<< (unsigned char);
    ostream & operator<< (int);
    ostream & operator<< (unsigned int);
    ostream & operator<< (long);
    ostream & operator<< (unsigned long);
    ostream & operator<< (float);
    ostream & operator<< (double);
    ostream & operator<< (const signed char *);
    ostream & operator<< (const unsigned char *);
    ostream & operator<< (void *);
    // ...
```



# Lớp istream

```
class istream : virtual public ios
{
public:
    istream & getline(char *, int, char = '\n');
    istream & operator>> (signed char *);
    istream & operator>> (unsigned char *);
    istream & operator>> (unsigned char &);
    istream & operator>> (signed char &);
    istream & operator>> (short &);
    istream & operator>> (int &);
    istream & operator>> (long &);
    istream & operator>> (unsigned short &);
    istream & operator>> (unsigned int &);
    istream & operator>> (unsigned long &);
    istream & operator>> (float &);
    istream & operator>> (double &);
```



## Phép toán << và >>

- Để định nghĩa phép toán << theo nghĩa xuất ra dòng dữ liệu xuất cho kiểu dữ liệu đang định nghĩa, ta định nghĩa phép toán như hàm *toàn cục* với tham số thứ nhất là *tham chiếu đến đối tượng thuộc lớp ostream*, kết quả trả về là *tham chiếu đến chính ostream đó*. Toán hạng thứ hai thuộc lớp đang định nghĩa.
- Để định nghĩa phép toán >> theo nghĩa nhập từ dòng dữ liệu nhập cho kiểu dữ liệu đang định nghĩa, ta định nghĩa phép toán >> như hàm *toàn cục* với tham số thứ nhất là *tham chiếu đến một đối tượng thuộc lớp istream*, kết quả trả về là *tham chiếu đến chính istream đó*. Toán hạng thứ hai là *tham chiếu đến đối tượng thuộc lớp đang định nghĩa*.
- Thông thường ta khai báo hai phép toán trên là hàm bạn của lớp để có thể truy xuất dữ liệu trực tiếp.



# Ví dụ phép toán << và >>: Lớp PS

```
// phanso.h
class PhanSo
{
    long tu, mau;
    void UocLuoc();
public:
    PhanSo(long t = 0, long m = 1) {Set(t,m);}
    void Set(long t, long m); long LayTu() const {return
    tu;}    long LayMau() const {return mau;}
    friend PhanSo operator + (PhanSo a, PhanSo b);
    friend PhanSo operator - (PhanSo a, PhanSo b);
    friend PhanSo operator * (PhanSo a, PhanSo b);
    friend PhanSo operator / (PhanSo a, PhanSo b);
    PhanSo operator -() const {return PhanSo(-tu,mau);}
    friend istream& operator >> (istream &is, PhanSo &p);
    friend ostream& operator << (ostream &os, PhanSo &p);
};
```

# Phép toán << và >>

```
// phanso.cpp
#include <iostream.h>
#include "phanso.h"
istream & operator >> (istream &is, PhanSo &p)
{
    is >> p.tu >> p.mau;
    while (!p.mau)
    {
        cout << "Nhap lai mau so: ";
        is >> p.mau;
    }
    p.UocLuoc();
    return is;
}
ostream & operator << (ostream &os, PhanSo p)
{
    os << p.tu;
    if (p.tu != 0 && p.mau != 1)
        os << "/" << p.mau;
    return os;
}
```

# Phép toán << và >>

```
// tps.cpp
#include <iostream.h>
#include "phanso.h"
void main()
{
    PhanSo a, b;
    cout << "Nhap phan so a: "; cin >> a;
    cout << "Nhap phan so b: "; cin >> b;
    cout << a << " + " << b << " = " << a + b << "\n";
    cout << a << " - " << b << " = " << a - b << "\n";
    cout << a << " * " << b << " = " << a * b << "\n";
    cout << a << " / " << b << " = " << a / b << "\n";
}
```

# V dụ phép toán << và >>: Lớp String

```
class String
{
    char *p;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s2) {p = strdup(s2.p);}
    ~String() {delete [] p;}
    String& operator = (const String& p2);
    friend String operator +(const String &s1,
                             const String &s2);
    friend istream& operator >> (istream &i,
                                   String& s);
    friend ostream& operator << (ostream &o, const
                                   String& s);
};
```

# V dụ phép toán << và >>: Lớp String

```
const MAX = 512;
istream& operator >> (istream &is, String& s)
{
    char st[MAX];
    is.getline(st, sizeof(st));
    s = st;
    return is;
}
ostream& operator << (ostream &os, const String& s)
{
    return os << s.p;
}
```



# Phép toán << và >>

- Phép toán << và >> cũng có thể được định nghĩa với toán hạng thứ nhất thuộc lớp đang xét, không thuộc lớp ostream hoặc istream. Trong trường hợp đó, ta dùng hàm thành phần. Kiểu trả về là chính đối tượng ở vế trái để có thể thực hiện phép toán liên tiếp.
- Các ví dụ về sử dụng phép toán trên theo cách này là các lớp Stack, Tập hợp, Danh sách, Mảng, Tập tin...

**Mang a;**

**a << 5 << 15; // bỏ 5 và 15 vào mảng**

- Ví dụ sau minh họa cách sử dụng phép toán trên với lớp Stack

# Phép toán << và >>

```
class Stack
{
    Item *st, *top;
    int size;
    void Init(int sz) {st = top = new Item[size=sz];}
    void CleanUp() {if (st) delete [] st;}
public:
    Stack(int sz = 20) {Init(sz);}
    ~Stack() {CleanUp();}
    static Stack *Create(int sz);
    bool Full() const {return (top - st >= size);}
    bool Empty() const {return (top <= st);}
    bool Push(Item x);
    bool Pop(Item *px);
    Stack &operator << (Item x) {Push(x); return *this;}
    Stack &operator >> (Item &x) {Pop(&x); return *this;}
```

# Phép toán << và >>

```
void main()  
{  
  
    Stack s(10);  
    Item a,b,c,d,e;  
    a = b = c = d = e = 10;  
    s << 1 << 3 << 5 << 7;  
    s >> a >> b >> c >> d >> e;  
    cout << setw(4) << a << setw(4) << b <<  
    setw(4) << c << setw(4) << d << setw(4) << e  
    << "\n";  
}
```

- Xuất liệu khi thực hiện đoạn chương trình trên: ?

## 3.5.2 Phép toán lấy ph. tử mảng: []

- Ta có thể định nghĩa phép toán [] để truy xuất phần tử của một đối tượng có ý nghĩa mảng.

```
class String
{
friend ostream& operator << (ostream &o, const
    String& s);
    char *p;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s) {p = strdup(s.p);}
    ~String() {delete [] p;}
    String & operator = (const String &s);
    char & operator[](int i) {return p[i];}
};
```

- Kết quả trả về là tham chiếu để phần tử trả về có thể đứng ở bên trái của phép toán gán (lvalue).



# Phép toán lấy phần tử mảng: []

- Sau khi định nghĩa như trên, có thể sử dụng đối tượng trả về ở cả hai vế của phép toán gán.

```
void main() {  
    clrscr();  
    String a("Nguyen van A");  
    cout << a[7] << "\n"; // a.operator[] (7)  
    a[7] = 'V';  
    cout << a[7] << "\n"; // a.operator[] (7)  
    cout << a << "\n";  
}
```

- Ta có thể cải tiến để phép toán trên có thể được sử dụng an toàn khi chỉ số không hợp lệ:

```
char a[] = "Dai hoc Tu nhien";  
a[300] = 'H'; // Nguy hiem  
String aa("Dai hoc Tu nhien");  
aa[300] = 'H'; // Nguy hiem, nhưng co the sua
```



# Phép toán lấy phần tử mảng: []

- Sử dụng phép toán trên như giá trị trái (lvalue) với chỉ số không hợp lệ thường gây ra những lỗi khó tìm và sửa. Ta có thể khắc phục bằng cách kiểm tra.

```
class String
{
    char *p;
    static char c;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s) {p = strdup(s.p);}
    ~String() {delete [] p;}
    String & operator = (const String &s);
    char & operator[](int i) {return (i >= 0 && i
        < strlen(p)) ? p[i] : c;}
};

char String::c = 'A';
```



# Phép toán lấy phần tử mảng: []

- Sau khi định nghĩa như trên, ta có thể “yên trí” gán vào các phần tử có chỉ số không hợp lệ.

```
void main()  
{  
    clrscr();  
    String a("Nguyen van A");  
    cout << a[7] << "\n";  
    a[7] = 'V';  
    cout << a[7] << "\n";  
    cout << a[200] << "\n"; // Xuat String::c  
    a[200] = 'X'; // Gan String::c = 'X';  
    cout << a[300]; // Xuat String::c  
}
```

- Phép toán [] định nghĩa như trên có thể hoạt động tốt cho cả hai trường hợp: ở bên trái và bên phải phép toán gán.

# Phép toán [] cho đối tượng hằng

- Tuy nhiên sử dụng phép toán [] như trên là không hợp lệ đối với đối tượng hằng.

```
void main()
{
    clrscr();
    String a("Nguyen van A");
    const String aa("Dai Hoc Tu Nhien");
    cout << a[7] << "\n";
    a[7] = 'V';
    cout << a[7] << "\n";
    cout << aa[4] << "\n"; // Bao Loi: sai khai niem
    aa[4] = 'L';           // Bao Loi: tot
    cout << aa[4] << "\n"; // Bao Loi: sai khai niem
    cout << aa << "\n";
}
```



# Phép toán [] cho đối tượng hằng

- Lỗi trên được khắc phục bằng cách định nghĩa một phiên bản áp dụng được cho đối tượng hằng.

```
class String
{
    char *p;
    static char c;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s) {p = strdup(s.p);}
    ~String() {delete [] p;}
    String & operator = (const String &s);
    char & operator[](int i) {return (i >= 0 && i <
        strlen(p)) ? p[i] : c;}
    char operator[](int i) const {return p[i];}
};

char String::c = 'A';
```



# Phép toán [] cho đối tượng hằng

- Khi đó việc sử dụng phép toán [] để đọc phần tử thì hợp lệ nhưng cố tính gán sẽ gây ra lỗi sai lúc biên dịch.

```
void main()
{
    clrscr();
    String a("Nguyen van A");
    const String aa("Dai Hoc Tu Nhlen");
    cout << a[7] << "\n";
    a[7] = 'V';
    cout << a[7] << "\n";
    cout << aa[4] << "\n";
        // String::operator[](int) const : Ok
    aa[4] = 'L'; // Bao Loi: Khong the la lvalue
    cout << aa[4] << "\n";
        // String::operator[](int) const : Ok
    cout << aa << "\n";    //
```



# Phép toán [] với tham số kiểu bất kỳ

- Phép toán [] có thể được định nghĩa với tham số thuộc kiểu bất kỳ không nhất thiết là kiểu đếm được.
- Dùng phép toán [] với các kiểu chỉ số khác với số nguyên rất thuận tiện khi ta muốn thực hiện ánh xạ một giá trị có kiểu bất kỳ sang một kiểu khác. Ví dụ mảng ánh xạ từ tiếng Anh sang tiếng Việt, mảng đếm sự xuất hiện của một từ trong một văn bản.

```
ifstream f("data.txt");  
MangDem md(f);  
cout << md["Hello"] << "\n";  
//...  
ifstream f("tudien.txt");  
TuongUng av(f);  
cout << av["Hello"] << "\n";
```

- Ví dụ sau minh họa phép toán [] dùng để tương ứng tên hàm với bản thân hàm đó.



# Phép toán [] với tham số kiểu bất kỳ

```
typedef double (*PF) (double);  
#define dim(a) (sizeof(a)/sizeof(a[0]))  
struct BoDoi  
{  
    char *TenHam;  
    PF Ham;  
};  
class TuongUng  
{  
    int n;  
    const BoDoi * const bang;  
public:  
    TuongUng(int nn, const BoDoi * const b):n(nn),  
    bang(b) {}  
    PF operator [] (char *th);  
};
```



# Phép toán [] với tham số kiểu bất kỳ

```
BoDoi bb[] =  
{  
    "sin", sin,  
    "cos", cos,  
    "sqrt", sqrt,  
    "...", KhongHopLe,  
};
```

```
PF TuongUng::operator [] (char *th)  
{  
    for (int i = 0; i < n; i++)  
        if (strcmp(bang[i].TenHam, th) == 0)  
            return bang[i].Ham;  
    return KhongHopLe;  
}
```

# Phép toán [] với tham số kiểu bất kỳ

```
double KhongHopLe(double)
{
    cerr << "Ham khong hop le\n";
    return 0;
}
void main()
{
    TuongUng af(dim(bb),bb);
    cout << af["sqrt"](9) << "\n";
    cout << af["abcd"](9) << "\n";
    char th[20];
    double x;
    cout << "Nhap ten ham va doi so:";
    cin >> th >> x;
    cout << th << "(" << x << ") = " << af[th](x)
        << "\n";
```



# Phép toán [] với tham số kiểu bất kỳ

- Ta có thể trả về giá trị NULL cho tên hàm không hợp lệ

```
PF TuongUng::operator [] (char *th){
    for (int i = 0; i < n; i++)
        if (strcmp(bang[i].TenHam, th) == 0)
            return bang[i].Ham;
    return NULL;}

void main(){
    TuongUng af(dim(bb),bb);
    char th[20]; double x;
    cout << "Nhap ham va doi so:"; cin >> th >> x;
    if (af[th])
        cout << th << "(" << x << ") = " << af[th](x)
            << "\n";
    else
        cerr << "Ham " << th << " khong hop le\n";
}
```



# Phép toán [] và chuyển kiểu

- Sử dụng chuyển kiểu có thể tạo các đối tượng mang ý nghĩa danh sách với phép toán [] hoạt động khác nhau khi dùng như một giá trị trái và giá trị phải. Điều đó có thể được thực hiện bằng cách tạo một loại đối tượng trung gian có thể chuyển kiểu qua lại với ký tự.
- Theo cách trên ta có thể định nghĩa phép toán [] cho phép quan điểm một tập tin như một mảng (vô hạn) các ký tự. Xem chương trình nguồn [cfile.cpp](#). Có thể thực hiện chương trình [cfile.exe](#)



### 3.5.3 Phép toán gọi hàm: ()

- Phép toán [] chỉ có thể có một tham số, vì vậy dùng phép toán trên không thuận tiện khi ta muốn lấy phần tử của một ma trận hai chiều. Phép toán gọi hàm cho phép có thể có số tham số bất kỳ, vì vậy thuận tiện khi ta muốn truy xuất phần tử của các đối tượng thuộc loại mảng hai hay nhiều chiều hơn.
- Lớp ma trận sau đây định nghĩa phép toán () với hai tham số, nhờ vậy ta có thể truy xuất phần tử của ma trận thông qua số dòng và số cột.

# Phép toán gọi hàm

```
class Matrix
{
    int m,n;
    double *px;
    int DataSize(){return m*n*sizeof(double);}
    void CopyData(double *pData)
        {memcpy(px,pData,DataSize());}
    void Create(int mm, int nn, double *pData = NULL);
public:
    Matrix(int mm, int nn, double *_px)
        {Create(mm,nn,_px);}
    Matrix(const Matrix &m2){Create(m2.m,m2.n,m2.px);}
    ~Matrix() { delete [] px;}
    Matrix operator = (const Matrix &m2);
    double &operator()(int i, int j){return px[i*n+j];}
    double operator()(int i, int j) const
        {return px[i*n+j];}
```



# Phép toán gọi hàm ()

- Sau khi định nghĩa phép toán gọi hàm như trên, ta có thể sử dụng ma trận như một mảng hai chiều.

```
void main()
{
    double a[] = {1, 2, 3, 4, 5, 6};
    Matrix m(2, 3, a);
    cout << "m = " << m << "\n";
    cout << m(1, 2) << "\n"; // m.operator()(1, 2)
    m(1, 2) = 50;
    cout << m(1, 2) << "\n";
}
```

- Phép toán gọi hàm cũng được dùng như một dạng vắn tắt thay cho một hàm thành phần thông thường. Như ví dụ sau, phép toán gọi hàm hoạt động tương đương với hàm thành phần SubStr để lấy một chuỗi con.

# Phép toán gọi hàm ()

```
class String
{
friend ostream& operator<<(ostream &o,const String& s);
    char *p;
    static char c;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s) {p = strdup(s.p);}
    ~String() {delete [] p;}
    String & operator = (const String &s);
    void Output() const {cout << p;}
    char & operator[](int i) {return (i >= 0 && i <
        strlen(p)) ? p[i] : c;}
    char operator[](int i) const {return p[i];}
    String SubStr(int start, int count) const;
    String operator()(int start, int count) const;
};
```

# Phép toán gọi hàm ()

```
String String::SubStr(int start, int count) const
{
    char buf[512];
    strncpy(buf, p+start, count); buf[count] = '\\0';
    return buf;
}

String String::operator()(int start, int count) const
{
    char buf[512];
    strncpy(buf, p+start, count); buf[count] = '\\0';
    return buf;
}

void main()
{
    String a("Nguyen van A");
    cout << a.SubStr(7,3) << "\\n";
    cout << a(7,3) << "\\n";
```



# Phép toán gọi hàm ()

- Ứng dụng quan trọng nhất của phép toán gọi hàm là cơ chế cho phép duyệt qua một danh sách (mảng, danh sách liên kết...). Xem minh họa trong tập tin chương trình nguồn [plist.cpp](#). Có thể thực hiện chương trình [plist.exe](#)

### 3.5.4 Phép toán tăng và giảm: ++ và --

- ++ là phép toán một ngôi có vai trò tăng giá trị một đối tượng lên giá trị kế tiếp. Tương tự -- là phép toán một ngôi có vai trò giảm giá trị một đối tượng xuống giá trị trước đó.
- ++ và -- chỉ áp dụng cho các kiểu dữ liệu đếm được, nghĩa là mỗi giá trị của đối tượng đều có giá trị kế tiếp hoặc giá trị trước đó.
- ++ và -- có thể được dùng theo hai cách, tiếp đầu ngữ hoặc tiếp vĩ ngữ.
- Khi dùng như tiếp đầu ngữ, ++a có hai vai trò:
  - Tăng a lên giá trị kế tiếp.
  - Trả về tham chiếu đến chính a.
- Khi dùng như tiếp vĩ ngữ, a++ có hai vai trò:
  - Tăng a lên giá trị kế tiếp.
  - Trả về giá trị bằng với a trước khi tăng.



# Phép toán tăng và giảm: ++ và --

Khi chỉ định nghĩa một phiên bản của phép toán ++ (hay --) phiên bản này sẽ được dùng cho cả hai trường hợp: tiếp đầu ngữ và tiếp vĩ ngữ.

```
class ThoiDiem
{
    long tsgiaiy;
    static bool HopLe(int g, int p, int gy);
public:
    ThoiDiem(int g = 0, int p = 0, int gy = 0);
    void Set(int g, int p, int gy);
    int LayGio() const {return tsgiaiy / 3600;}
    int LayPhut() const {return (tsgiaiy%3600)/60;}
    int LayGiay() const {return tsgiaiy % 60;}
    void Tang();
    void Giam();
    ThoiDiem &operator ++();
```





# Phép toán tăng và giảm: ++ và --

```
void ThoiDiem::Tang()
{
    tsgiaiy = ++tsgiaiy%SOGIAY_NGAY;
}
void ThoiDiem::Giam()
{
    if (--tsgiaiy < 0) tsgiaiy = SOGIAY_NGAY-1;
}
ThoiDiem &ThoiDiem::operator ++()
{
    Tang();
    return *this;
}
```

# Phép toán tăng và giảm: ++ và --

```
void main()
{
    clrscr();
    ThoiDiem t(23,59,59),t1,t2;
    cout << "t = " << t << "\n";
    t1 = ++t; // t.operator ++();
    // t = 0:00:00, t1 = 0:00:00
    cout << "t = " << t << "\tt1 = " << t1 << "\n";
    t1 = t++; // t.operator ++();
    // t = 0:00:01, t1 = 0:00:00
    cout << "t = " << t << "\tt1 = " << t1 << "\n";
}
```

# Phép toán tăng và giảm: ++ và --

Để có thể có phép toán ++ và -- hoạt động khác nhau cho hai cách dùng (++a và a++) ta cần định nghĩa hai phiên bản ứng với hai cách dùng kể trên. Phiên bản tiếp đầu ngữ có thêm một tham số giả để phân biệt.

```
class ThoiDiem {  
    long tsgiaiy;  
public:  
    ThoiDiem(int g = 0, int p = 0, int gy = 0);  
    void Set(int g, int p, int gy);  
    int LayGio() const {return tsgiaiy / 3600;}  
    int LayPhut() const {return (tsgiaiy%3600)/60;}  
    int LayGiay() const {return tsgiaiy % 60;}  
    void Tang();  
    void Giam();  
    ThoiDiem &operator ++();  
    ThoiDiem operator ++(int);
```



# Phép toán tăng và giảm: ++ và --

```
void ThoiDiem::Tang() {
    tsgiaiy = ++tsgiaiy%SOGIAY_NGAY;
}

void ThoiDiem::Giam() {
    if (--tsgiaiy < 0) tsgiaiy = SOGIAY_NGAY-1;
}

ThoiDiem &ThoiDiem::operator ++()
{
    Tang();
    return *this;
}

ThoiDiem ThoiDiem::operator ++(int)
{
    ThoiDiem t = *this;
    Tang();
    return t;
}
```



# Phép toán tăng và giảm: ++ và --

```
void main()
{
    clrscr();
    ThoiDiem t(23,59,59),t1,t2;
    cout << "t = " << t << "\n";
    t1 = ++t; // t.operator ++();
    // t = 0:00:00, t1 = 0:00:00
    cout << "t = " << t << "\tt1 = " << t1 << "\n";
    t1 = t++; // t.operator ++(int);
    // t = 0:00:01, t1 = 0:00:00
    cout << "t = " << t << "\tt1 = " << t1 << "\n";
}
```