CHƯƠNG 5. OVERLOAD TOÁN TỬ VÀ HÀM

ThS. Trần Anh Dũng

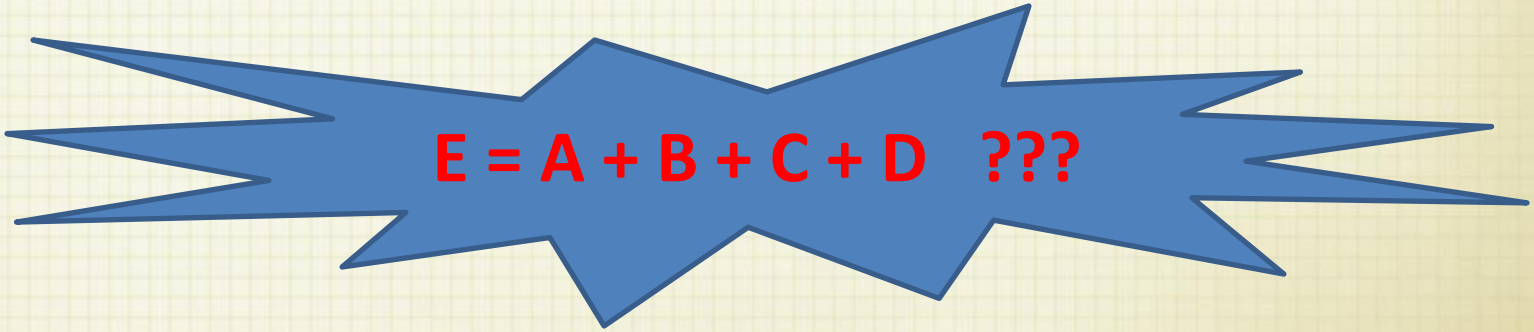
Nội dung

- Giới thiệu
- Các toán tử của C++
- Các toán tử overload được
- Cú pháp Operator Overloading
- Chuyển kiểu
- Sự nhập nhằng
- Phép toán << và >>
- Phép toán lấy phần tử mảng: [ ]
- Phép toán gọi hàm: ()
- Phép toán tăng và giảm: ++ và --

Giới thiệu

- Xét ví dụ sau: Giả sử có lớp PhanSo cung cấp các thao tác Set, Cong, Tru, Nhan, Chia

```
PhanSo A, B, C, D, E;
C.Set(A.Cong(B));
E.Set(D.Cong(C));
```



Giới thiệu

- Các toán tử cho phép ta sử dụng cú pháp toán học đối với các kiểu dữ liệu của C++ thay vì gọi hàm (bản chất vẫn là gọi hàm).
  - Ví dụ thay a.set(b.cong(c)); bằng a = b + c;
  - Gần với kiểu trình bày mà con người quen dùng (mang tính tự nhiên)
  - Đơn giản hóa mã chương trình

```
PhanSo A, B;
cin>>A;  //A.Nhap();
cin>>B;  //B.Nhap();
```

Giới thiệu

- Một lớp ngoài dữ liệu và các phương thức còn có các phép toán giúp người lập trình dễ dàng thể hiện các câu lệnh trong chương trình.
- Tuy nhiên, sự cài đặt phép toán chỉ cho phép tạo ra phép toán mới trên cơ sở ký hiệu phép toán đã có, không được quyền cài đặt các phép toán mới →sự cài đặt thêm phép toán là sự nạp chồng phép toán (operator overloading)
- Đối với các kiểu dữ liệu người dùng: C++ cho phép định nghĩa các toán tử trên các kiểu dữ liệu người dùng →overload

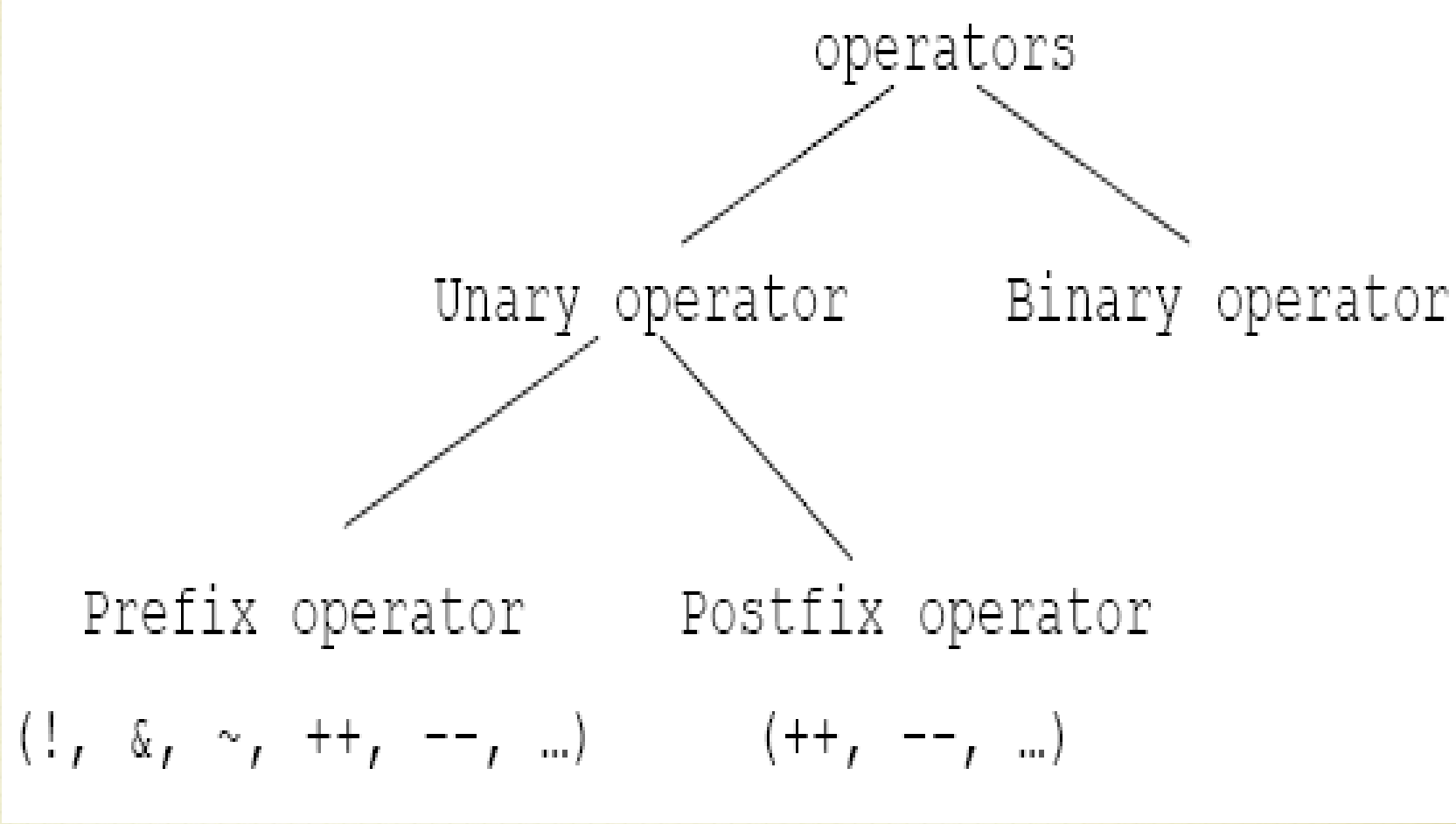
Operator overload

- Một toán tử có thể dùng cho nhiều kiểu dữ liệu.
- Như vậy, ta có thể tạo các kiểu dữ liệu đóng gói hoàn chỉnh (fully encapsulated) để kết hợp với ngôn ngữ như các kiểu dữ liệu cài sẵn.
- Ví dụ:

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

Các toán tử của C++

- Các loại toán tử:



Các toán tử của C++

- Một số toán tử đơn có thể được dùng làm cả toán tử trước và toán tử sau. Ví dụ phép tăng (++), phép giảm (--)
- Một số toán tử có thể được dùng làm cả toán tử đơn và toán tử đôi: *
- Toán tử chỉ mục ("[…]") là toán tử đôi
- Các từ khoá "new" và "delete" cũng được coi là toán tử và có thể được định nghĩa lại

Các toán tử overload được

- Các toán tử có thể overload:

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

Cú pháp Operator Overloading

- Sử dụng tên hàm là "operator@" cho toán tử "@".
  - Ví dụ: operator+
- Số lượng tham số tại khai báo hàm phụ thuộc hai yếu tố:
  - Toán tử là toán tử đơn hay đôi
  - Toán tử được khai báo là phương thức toàn cục hay phương thức của lớp

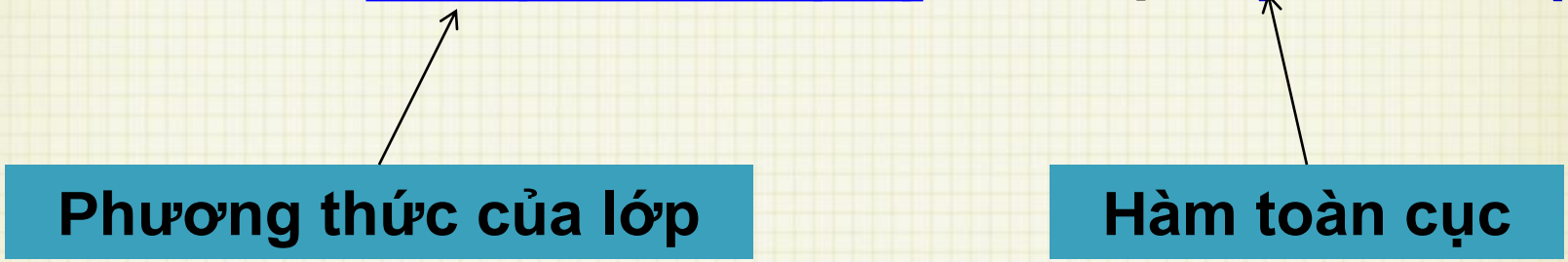
2/3 + 5 – 6/5 = ?

Cú pháp Operator Overloading

aa@bb ➔ aa.operator@(bb) hoặc operator@(aa,bb)

@aa ➔ aa.operator@() hoặc operator@(aa)

aa@ ➔ aa.operator@(int) hoặc operator@(aa,int)



Ví dụ - Lớp PhanSo

```
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 - 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;
    }
```

Ví dụ - Lớp PhanSo

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

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

Hạn chế của overload toán tử

- Không thể tạo toán tử mới hoặc kết hợp các toán tử có sẵn theo kiểu mà trước đó chưa được định nghĩa.
- Không thể thay đổi thứ tự ưu tiên của các toán tử
- Không thể tạo cú pháp mới cho toán tử
- Không thể định nghĩa lại một định nghĩa có sẵn của một toán tử

Một số ràng buộc của phép toán

- Hầu hết các phép toán không ràng buộc ý nghĩa, chỉ một số trường hợp cá biệt như operator =, operator [], operator (), operator -> đòi hỏi phải được định nghĩa là hàm thành phần của lớp để toán hạng thứ nhất có thể là một đối tượng trái (lvalue).
- Ta phải chủ động định nghĩa phép toán +=, -=, *=,… dù đã định nghĩa phép gán và các phép toán +,-,*,…

Lưu ý khi định nghĩa lại toán tử

- Tôn trọng ý nghĩa của toán tử gốc, cung cấp chức năng mà người dùng mong đợi/chấp nhận
- Cố gắng tái sử dụng mã nguồn một cách tối đa
- Trong ví dụ trên, ta định nghĩa hàm thành phần có tên đặc biệt bắt đầu bằng từ khóa 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

Hàm thành phần và 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 –();
```

Hàm thành phần và hàm toàn cục

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

Hàm thành phần và hàm toàn cục

- Dùng hàm thành phần hay hàm toàn cục?
- Các phép toán =, [ ], (), →, định nghĩa hàm toàn cục được không?
- Nếu toán hạng thứ nhất không thuộc lớp đang xét?

Ví dụ minh họa

```
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)
a + 5; // a.operator + (5)
3 + a; // 3.operator + (a): ???
```

Ví dụ minh họa

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

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.

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 + (int a, PhanSo b);
    PhanSo operator - (PhanSo b) const;
    PhanSo operator - (long b) const;
    friend PhanSo operator - (int a, PhanSo b);
    PhanSo operator * (PhanSo b) const;
    PhanSo operator * (long b) const;
    friend PhanSo operator * (int a, PhanSo b);
```

Chuyển kiểu

```
    PhanSo operator / (PhanSo b) const;
    PhanSo operator / (long b) const;
    friend PhanSo operator / (int a, PhanSo b);
    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);
    //Tương tự cho các phép toán còn lại
};
```

Chuyển kiểu

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

```
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;
}
```

Chuyển kiểu

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

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

Chuyển kiểu bằng constructor

- 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 là số 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,…).

Chuyển kiểu bằng constructor

- Như vậy 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); }
    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 constructor

- Như vậy có thể giảm bớt việc khai báo và định nghĩa phép toán + phân số với số nguyên, cơ chế chuyển kiểu tự động cho phép thực hiện thao tác cộng đó.

```
//...
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
c = a + 5;      // c = a.operator + PhanSo(5)
c = 3 + a;      // c = operator + (3,a)
```

Chuyển kiểu bằng constructor

- Như vậy có thể giảm việc định nghĩa 3 phép toán còn 2.
- 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.
- Có cách nào để đơn giản hơn, mỗi phép toán phải định nghĩa 2 hàm thành phần tương ứng?

Chuyển kiểu bằng constructor

- Ta có thể giảm số phép toán cần định nghĩa từ 3 xuống 1 bằng cách dùng hàm toàn cục

```
class PhanSo{
    long tu, mau;
public:
    PhanSo (long t, long m) { Set(t,m); }
    PhanSo (long t) { Set(t,1); }
    void Set (long t, long m);
    friend PhanSo operator + (PhanSo a, PhanSo b);
    friend PhanSo operator - (PhanSo a, PhanSo b);
    //...
};
```

Khi nào chuyển kiểu bằng constructor

- Ta dùng chuyển kiểu bằng phương thức thiết lập khi thỏa hai điều kiện sau:
  - Chuyển từ kiểu đã có (số nguyên) sang kiểu đang định nghĩa (phân số).
  - Có quan hệ là một từ kiểu đã có sang kiểu đang định nghĩa (một số nguyên là một phân số).

Chuyển kiểu bằng phép toán chuyển kiểu

- Chuyển kiểu bằng constructor có một số nhược điểm sau:
  - Muốn chuyển từ kiểu đang định nghĩa sang một kiểu đã có, ta phải sửa đổi kiểu đã có.
  - Không thể chuyển từ kiểu đang định nghĩa sang kiểu cơ bản có sẵn.

Chuyển kiểu bằng phép toán chuyển kiểu

- 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.

Chuyển kiểu bằng phép toán chuyển kiểu

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

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

Chuyển kiểu bằng 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
}
```

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

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

```
1  void main() {
2          int a = 3, b = 7;
3          double r = 3.2, s = 6.3;
4          cout << a+b << "\n";
5          cout << r+s << "\n";
6          cout << a+r << "\n";
7          cout << Sum(a,b) << "\n";
8          cout << Sum(r,s) << "\n";
9          cout << Sum(a,r) << "\n";
10 }
```

Sự nhập nhằng

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

- Xét lớp phân số

Sự nhập nhằng

```
class PhanSo {
    long tu, mau;
    void UocLuoc();
    int SoSanh(PhanSo b);
public:
    PhanSo(long t = 0, long m = 1) {Set(t,m);}
    PhanSo (long t) { Set(t,1); }
    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.

Sự nhập nhằng

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

Sự nhập nhằng

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

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.

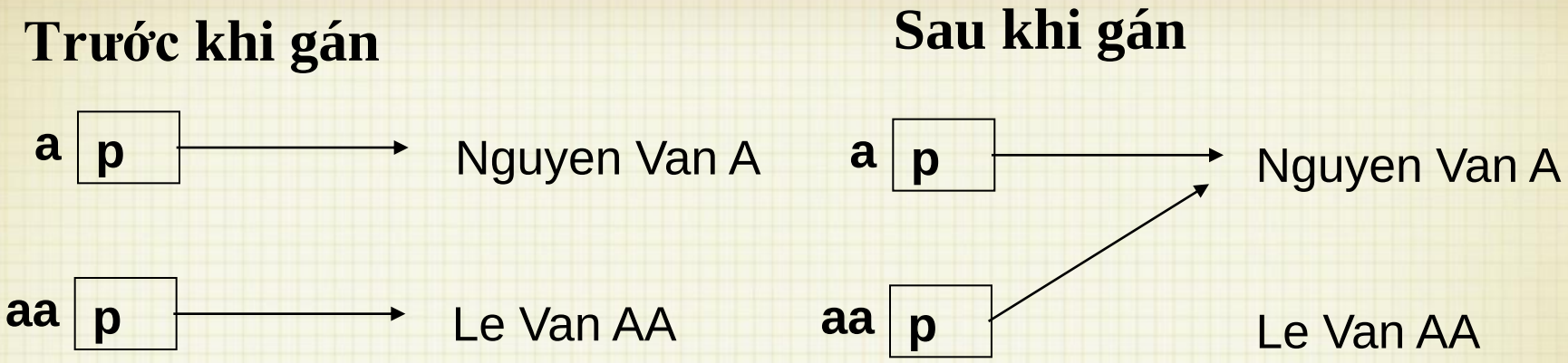
Gán và khởi động

- Khi lớp 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 hủy bỏ.
- Khi thực hiện phép gán trên các đối tượng cùng kiểu, cơ chế gán mặc nhiên là gán từng thành phần →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.

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(){
    String a("Nguyen Van A");
    String b = a;           //Khoi dong
    String aa = "Le van AA";
    cout << "aa = "; aa.Output(); cout << "\n";
    aa = a;                 //Gan
    cout << "aa = "; aa.Output(); cout << "\n";
}
```

Gán và khởi động



❖ Thực hiện chương trình trên ta được kết xuất như sau:

aa = Le van AA

aa = Nguyen Van A

delete 0x0d36

delete 0x0d48

delete 0x0d36

Null pointer assignment

Gán và khởi động

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

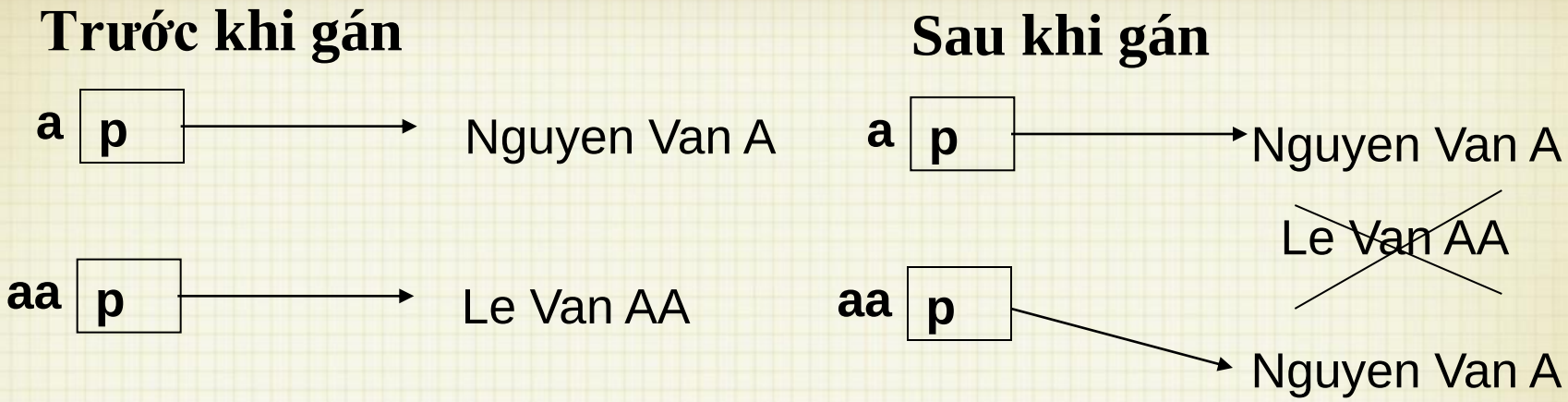
```
class String {
    char *p;
public:
    String(char *s = "") {p = strdup(s);}
    String(const String &s) {p = strdup(s.p);}
    ~String() {cout << "delete "<< (void *)p << "\n"; delete [] p;}
    String & operator = (const String &s);
    void Output() const {cout << p;}
};
```

Gán và khởi động

- Phép gán thực hiện hai thao tác chính là dọn dẹp tài nguyên cũ và sao chép mới.

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

Gán và khởi động



- Thực hiện chương trình trên ta được kết xuất như sau:

aa = La van AA

aa = Nguyen Van A

delete 0x0d5a

delete 0x0d48

delete 0x0d36

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

Phép toán << và >>

- Khi định nghĩa hai phép toán trên, cần thể hiện ý nghĩa sau:

```
a >> b;            //bỏ a vào b
a << b;            //bỏ b vào a
cout << a << "\n";     // bỏ a và "\n" vào cout
cin >> a >> b; // bỏ cin vào a và b
```

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.

Phép toán << và >>

- 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 *);
    // ...
private:
    //data ...
};
```

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  &);
private:
    // data...
};
```

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.

Phép toán << và >>

- 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.

Ví dụ phép toán << và >>

```
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);
};
```

Ví dụ phép toán << và >>

```
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;
}
```

Ví dụ phép toán << và >>

```
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";
}
```

Phép toán lấy phần tử mảng: [ ]

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

```
class String {
    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]; }
    friend ostream& operator << (ostream &o, const String& s);
};
```

Phép toán lấy phần tử mảng: [ ]

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

```
void main() {
    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";
}
```

Phép toán [ ] cho đối tượng hằng

- Phép toán [ ] không hợp lệ với đối tượng hằng

```
void main() {
    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";
    aa[4] = 'L';
    cout << aa[4] << "\n";
    cout << aa << "\n";
}
```

Phép toán [ ] cho đối tượng hằng

- Cách khắc phục?

```
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

```
void main() {
    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";   // 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 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.
- 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{
    float **M;
    int row, col;
public:
    MATRIX (int, int);
    ~MATRIX();
    float& operator() (int, int);
};
float MATRIX::operator() (int i, int j){
    return M[i][j];
}
```

Phép toán gọi hàm: ()

```
MATRIX::MATRIX ( int r, int c){
    M = new float* [r];
    for ( int i=0; i<r; i++)
        M[i] = new float[c];
    row = r;
    col = c;
}
~MATRIX::MATRIX(){
    for ( int i=0; i<col; i++)
        delete [ ] M[i];
    delete [ ] M;
}
```

Phép toán gọi hàm: ()

```
void main(){
    cout<<“Cho ma tran 2x3\n”;
    MATRIX a(2, 3);
    int i, j;
    for (i = 0; i<2; i++)
          for (j = 0; j<3; j++)
                cin>>a(i,j);
     for (i = 0; i<2; i++){
          for (j = 0; j<3; j++)
                cout<<a(i,j)<<“ ”;
          cout<<endl;
    }
}
```

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 -- 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 vị ngữ.

Phép toán tăng và giảm: ++ và --

- 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 vị 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 --

```
class ThoiDiem{
    long tsgiay;
    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 tsgiay / 3600;}
    int LayPhut() const {return (tsgiay%3600)/60;}
    int LayGiay() const {return tsgiay % 60;}
    void Tang();
    void Giam();
    ThoiDiem &operator ++();
};
```

Phép toán tăng và giảm: ++ và --

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

Phép toán tăng và giảm: ++ và --

```
void main()
{
    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.
- Khi đó, phiên bản tiếp vị ngữ có thêm một tham số giả để phân biệt.

```
ThoiDiem &operator ++();
ThoiDiem operator ++(int);
```

Phép toán tăng và giảm: ++ và --

```
void ThoiDiem::Tang() {
    tsgiay = ++tsgiay%SOGIAY_NGAY;
}
void ThoiDiem::Giam() {
    if (--tsgiay < 0) tsgiay = 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()
{
    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";
}
```

Bài tập kiểm tra

Xét **đa thức** theo biến x (đa thức một biến) bậc n có dạng như sau:

$$P(X) = a_1x^n + a_2x^{n-1} + a_3x^{n-2} + \ldots + a_j$$

Trong đó: n là bậc của đa thức. a_1, a_2, a_3,…, a_j là các hệ số tương ứng với từng bậc của đa thức.

a. Xây dựng lớp **DaThuc** biểu diễn khái niệm đa thức với các thao tác như sau:

- Hàm khởi tạo mặc định để tạo một đa thức có bậc bằng 0
- Hàm khởi tạo để tạo một đa thức bậc n.
- Tính giá trị của đa thức khi biết giá trị của x
- Định nghĩa các toán tử tương ứng cho các thao tác sau:
  - Nhập đa thức.
  - Xuất đa thức.
  - Cộng hai đa thức
  - Trừ hai đa thức

b. Viết chương trình cho phép người dùng nhập vào hai đa thức rồi xuất các đa thức ra màn hình. Sau đó tính tổng hai đa thức và xuất kết quả ra màn hình.

Q & A