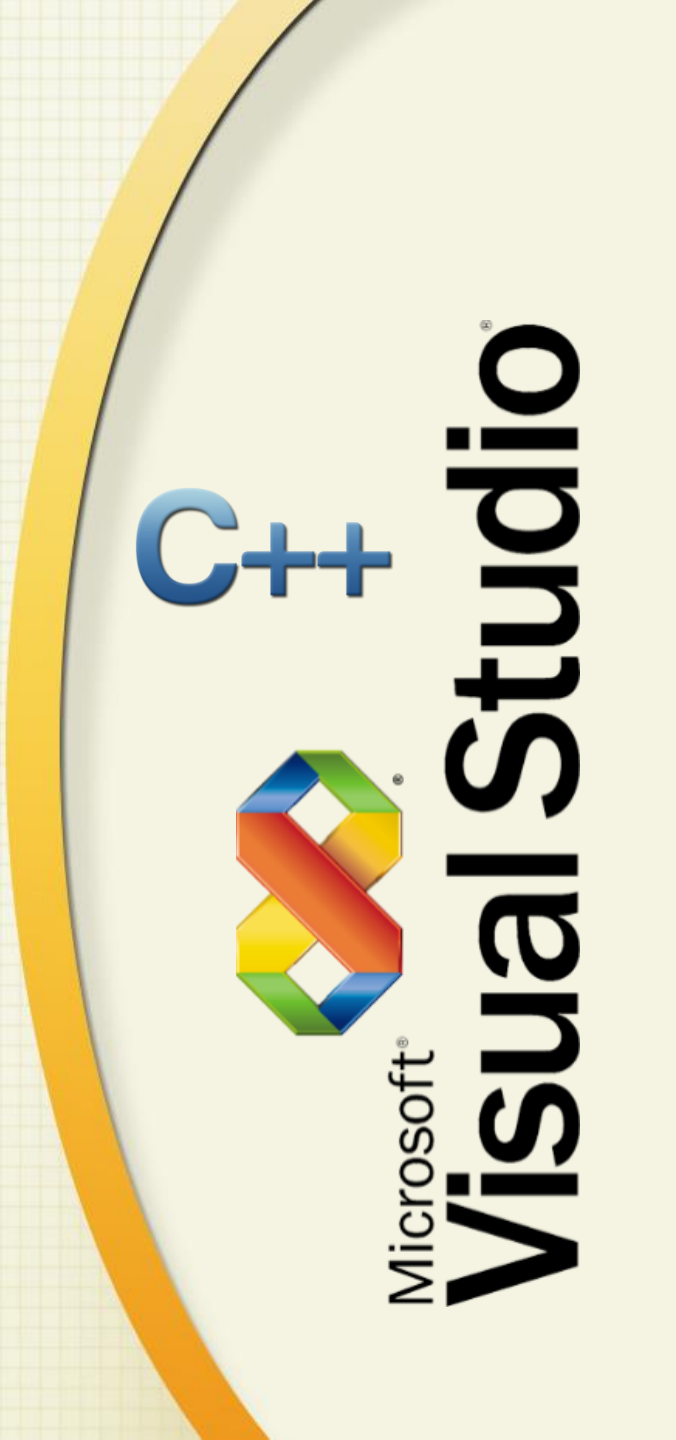
OVERLOAD TOÁN TỬ



Khoa Công nghệ phần mềm

Nội dung

1. Toán tử
2. Overload toán tử
3. Chuyển kiểu
4. Overload một số toán tử thông dụng

1. Toán tử

Operator: toán tử

Operation: phép toán

Operand: toán hạng

Expression: biểu thức

- **Toán tử** là các ký hiệu được dùng để thực hiện một **phép toán** trong ngôn ngữ lập trình.
- Toán tử thao tác trên hằng hoặc biến, hằng hoặc biến này được gọi là **toán hạng**.
- Một **biểu thức** là tổ hợp các toán tử và toán hạng, một biểu thức sẽ được tính toán để cho ra một giá trị.

Các loại toán tử

1) **Toán tử số học:** + - * / %(chỉ ad cho số nguyên) ++ --
2) **Toán tử quan hệ:** < <= > >= == !=
3) **Toán tử logic:** && || !(not)
4) **Toán tử gán:** = += -= *= /= %=
5) **Toán tử thao tác trên bit:** & | ^(XOR) ~(đảo bit) >> <<
   và &= |= ^= >>= <<= (ví dụ A &= 2 ⇔ A = A & 2)
6) **Toán tử làm việc với con trỏ:** & *
7) **Toán tử khác:** () //gọi hàm

   [ ] //truy xuất phần tử mảng

2. Overload toán tử

- Khái niệm
- Phân loại các toán tử của C++
- Cú pháp Overload toán tử
- Hạn chế của Overload toán tử
- Một số lưu ý khi Overload toán tử
- Hàm thành phần và hàm toàn cục

2.1 Khái niệm

Giả sử có lớp PhanSo cung cấp các thao tác Nhap, Gan, Cong, Tru, Nhan, Chia

E = A + B + D ???

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

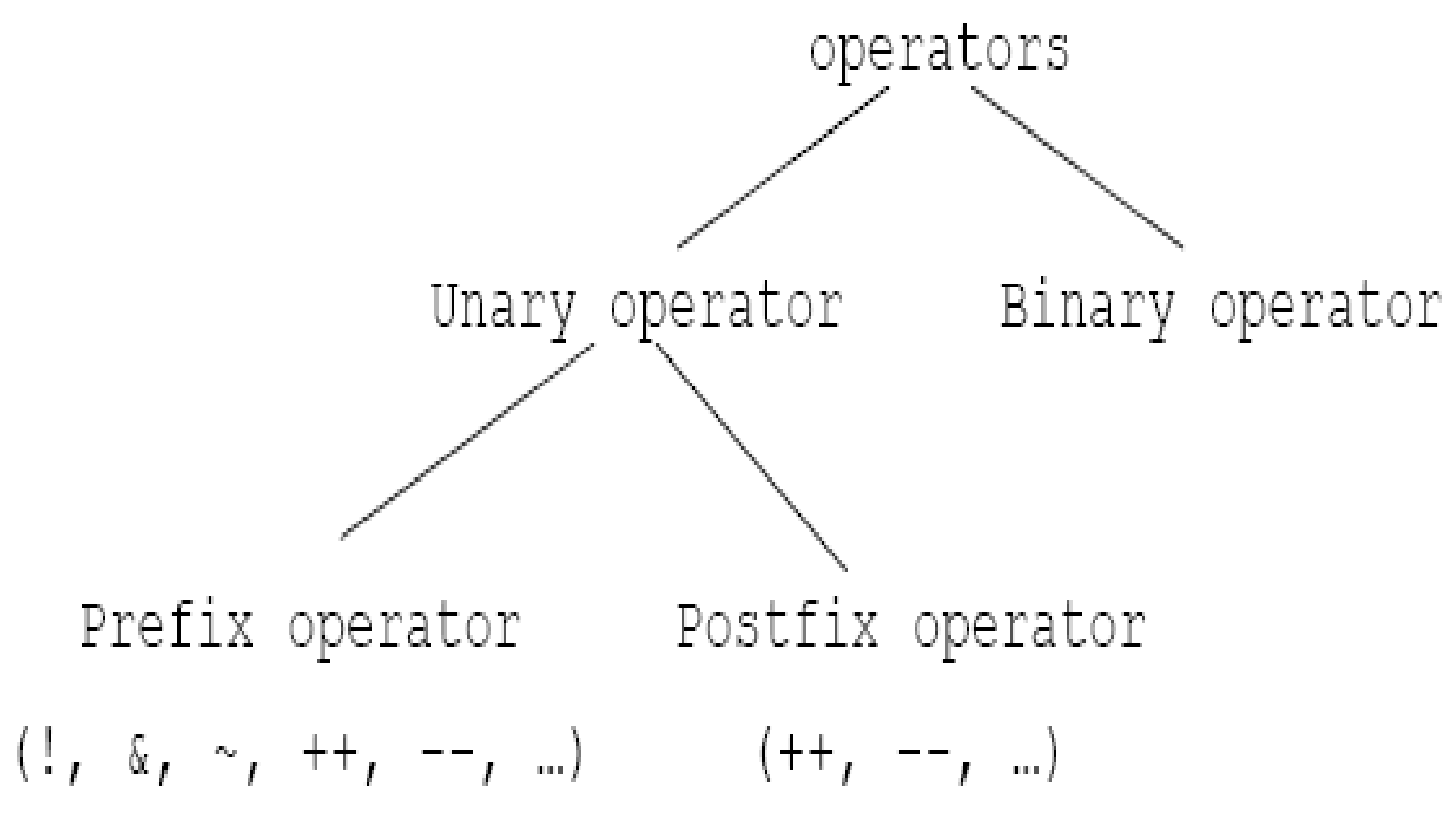
E = A + B + D ???

```
=>   PhanSo A, B, C, D, E;
     cin >> A;
     cin >> B;
     cin >> D;
     C = A + B;
     E = D + C;
```

2.1 Khái niệm (tt)

- Việc cài đặt các toán tử cho các kiểu dữ liệu tự định nghĩa cho phép ta viết các biểu thức một cách tự nhiên như trên các kiểu dữ liệu cơ sở, thay vì phải gọi hàm (bản chất vẫn là gọi hàm).
- Đây là một mở rộng đáng kể của **C++** so với **C**
- **C/C++** đã cài đặt sẵn các toán tử cho các kiểu dữ liệu cơ sở **(int, float…).**
- Đối với các kiểu dữ liệu tự định nghĩa, C++ cho phép cài đặt các toán tử là các ký hiệu toán học đã có **(overload toán tử).**

2.2 Phân loại các toán tử của C++



2.2 Phân loại các toán tử của C++ (tt)

- Các toán tử được chia thành hai loại, dựa trên số lượng toán hạng tham gia vào toán tử:
  - Toán tử đơn (toán tử một ngôi) nhận một toán hạng.
  - Toán tử đôi (toán tử hai ngôi) nhận hai toán hạng.
- **Các toán tử đơn** cũng được chia thành hai loại:
  - Toán tử trước đặt trước toán hạng.
  - Toán tử sau đặt sau toán hạng.

2.2 Phân loại các toán tử của C++ (tt)

- Toán tử vừa là toán tử trước, vừa là toán tử sau: **++** và **--** **(**tăng giảm một đơn vị).
- Toán tử vừa là toán tử đơn, vừa là toán tử đôi: ***** (toán tử con trỏ và phép nhân); - (đổi dấu và phép trừ).
- Toán tử truy xuất phần tử mảng **[ ]** là toán tử đôi: arg1**[**arg2**]**

2.3 Cú pháp Overload toán tử

- Khai báo và định nghĩa hàm toán tử giống như một hàm bình thường, chỉ khác là có thêm từ khóa operator trước tên hàm (tên hàm là tên toán tử cần overload).

  Kiểu_trả_về operator Tên_toán_tử (danh sách đối số)

  VD: PhanSo operator + (PhanSo b) const;

- Số lượng đối số của hàm toán tử phụ thuộc vào:
  - Toán tử đơn hay toán tử đôi;
  - Hàm toán tử là ***hàm thành phần của lớp*** hay ***hàm toàn cục.***

Ví dụ Overload toán tử – 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 Overload toán tử – Lớp PhanSo (tt)

```
class PhanSo{
    long tu, mau;
    void UocLuoc();
public:
    void Set(long t, long m);
    PhanSo(long t, long m) {
            Set(t,m);
    }
    long LayTu() const {
            return tu;
    }
    long LayMau() const {
            return mau;
    }
```

Ví dụ Overload toán tử – Lớp PhanSo (tt)

```
    PhanSo Cong(PhanSo b) const;
    PhanSo operator + (PhanSo b) const;
    PhanSo operator - () const //đổi dấu phân số
    {
        return PhanSo(-tu, mau);
    }
    bool operator == (PhanSo b) const;
    bool operator != (PhanSo b) const;
    void Xuat() const;
}; //end of class
```

Ví dụ Overload toán tử – Lớp PhanSo (tt)

```
void PhanSo::UocLuoc(){
    long usc = USCLN(tu, mau);
    tu /= usc;
    mau /= usc;
    if (mau < 0) mau = -mau, tu = -tu; //dấu của phân số là dấu của tử
    if (tu == 0) mau = 1;
}
void PhanSo::Set(long t, long m) {
    if (m) {
            tu = t;
            mau = m;
            UocLuoc();
    }
}
```

Ví dụ Overload toán tử – Lớp PhanSo (tt)

```
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;
}
```

2.4 Hạn chế của Overload toán tử

- Không thể overload một toán tử mà chưa từng được định nghĩa trước đó, hay nói cách khác là không thể tạo toán tử mới.
- Không thể tạo cú pháp mới cho toán tử.
- Không thể thay đổi định nghĩa có sẵn của một toán tử.
- Không thể thay đổi thứ tự ưu tiên của các toán tử.

2.5 Một số lưu ý

- Các hàm toán tử sau phải là hàm thành phần của lớp, khi đó đối tượng gọi hàm sẽ là toán hạng thứ nhất của toán tử.

  operator =

  operator [ ]

  operator ()

  operator ->

- Nếu có sử dụng các toán tử gán (+=, -=, *=, …) thì chúng phải được định nghĩa (là các hàm toán tử), cho dù đã định nghĩa toán tử gán (=) và các toán tử số học (+,-,*, …) cho lớp.
- Cài đặt hàm toán tử đúng ý nghĩa của toán tử được overload.

2.6 Hàm thành phần và hàm toàn cục

Hàm toán tử có thể là ***hàm thành phần của lớp*** hay ***hàm toàn cục.***

- **Nếu là hàm thành phần thì:**
  - Số đối số của hàm toán tử = số toán hạng của toán tử **- 1**
  - Toán hạng thứ nhất = đối tượng gọi hàm

VD: a - b; ➔ a.operator -(b);

-a; ➔ a.operator -();

2.6 Hàm thành phần và hàm toàn cục (tt)

- **Nếu là hàm toàn cục thì:**
  - Số đối số của hàm toán tử = số toán hạng của toán tử

VD: a - b; ➔ operator -(a,b);

-a; ➔ operator -(a);

2.6 Hàm thành phần và hàm toàn cục (tt)

Hàm toán tử nên là hàm thành phần hay hàm toàn cục?

- Các hàm toán tử **=, [ ], (), →**
  phải là hàm thành phần của lớp.
- Toán tử có toán hạng thứ nhất thuộc lớp đang xét thì hàm toán tử có thể là hàm thành phần/hàm toàn cục.
- Nếu toán hạng thứ nhất không thuộc lớp đang xét thì hàm toán tử phải là hàm toàn cục.

2.6 Hàm thành phần và hàm toàn cục – Ví dụ

```
class PhanSo {
    long tu, mau;
public:
    void Set(long t, long m);
    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); c(0,1);
c = a + b; //Khi đó sẽ gọi hàm a.operator + (b)
c = a + 5; //Gọi hàm a.operator + (5)
c = 3 + a;  ??? SAI
```

2.6 Hàm thành phần và hàm toàn cục – Ví dụ (tt)

```
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); //k.báo hàm bạn của lớp
}; //end of class
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; //Gọi hàm a.operator + (b)
c = a + 5; //Gọi hàm a.operator + (5)
c = 3 + a; //Gọi hàm toàn cục operator + (3,a) ĐÚNG
```

3. Chuyển kiểu

- Nhu cầu chuyển kiểu
- Chuyển kiểu bằng Constructor
- Chuyển kiểu bằng phép toán chuyển kiểu
- Sự nhập nhằng

Ví dụ

```
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); //Hàm bạn của lớp
    PhanSo operator - (PhanSo b) const;
    PhanSo operator - (long b) const;
    friend PhanSo operator - (int a, PhanSo b); //Hàm bạn của lớp
    PhanSo operator * (PhanSo b) const;
    PhanSo operator * (long b) const;
    friend PhanSo operator * (int a, PhanSo b); //Hàm bạn của lớp
```

Ví dụ (tt)

```
    PhanSo operator / (PhanSo b) const;
    PhanSo operator / (long b) const;
    friend PhanSo operator / (int a, PhanSo b); //Hàm bạn của lớp
    bool operator == (PhanSo b) const;
    bool operator == (long b) const;
    friend bool operator == (long a, PhanSo b); //Hàm bạn của lớp
    bool operator != (PhanSo b) const;
    bool operator != (long b) const;
    friend bool operator != (int a, PhanSo b); //Hàm bạn của lớp
    bool operator < (PhanSo b) const;
    bool operator < (long b) const;
    friend bool operator < (int a, PhanSo b); //Hàm bạn của lớp
    //Tương tự cho các phép toán còn lại
};
```

3.1 Nhu cầu chuyển kiểu

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

```
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;
}
```

3.1 Nhu cầu chuyển kiểu (tt)

- Tuy nhiên, việc 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 dựng sẵn:

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

3.1 Nhu cầu chuyển kiểu (tt)

Khi tính toán giá trị của một biểu thức, nếu kiểu dữ liệu của các toán hạng không giống nhau, trình biên dịch sẽ tìm cách chuyển kiểu.

- Trong một biểu thức số học, nếu có sự tham gia của một toán hạng kiểu số thực ➔ các toán hạng khác sẽ được chuyển kiểu sang số thực.
- Các trường hợp khác chuyển kiểu được thực hiện theo nguyên tắc nâng cấp.

  VD: int ➔ long, float ➔ double,…

3.2 Chuyển kiểu bằng Constructor

Trình biên dịch sẽ tự động "chuyển kiểu" sang kiểu dữ liệu tự định nghĩa khi ta cung cấp phương thức thiết lập phù hợp.

Ví dụ: xây dựng phương thức thiết lập với đối số là số nguyên để tạo một phân số, khi đó được xem như đã chuyển kiểu từ số nguyên sang phân số:

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

Hoặc `PhanSo a(3);`

3.2 Chuyển kiểu bằng Constructor (tt)

```
class PhanSo{
    long tu, mau;
public:
    void Set( long t, long m);
    PhanSo (long t, long m) { Set(t,m); }
    PhanSo (long t) { Set(t,1); } //Hàm tạo có 1 đối là số nguyên
    const PhanSo& operator=(const PhanSo &a); //Toán tử gán
    PhanSo operator + (PhanSo b) const;
    //PhanSo operator + (long b) const; //Bỏ bớt hàm cộng 1 ps với 1 số nguyên
    friend PhanSo operator + (int a, PhanSo b);
    PhanSo operator - (PhanSo b) const;
    friend PhanSo operator - (int a, PhanSo b);
};
```

3.2 Chuyển kiểu bằng Constructor (tt)

```
/*Cài đặt toán tử gán*/
const Phanso& Phanso::operator=(const Phanso &a) {
	x = a.x;
	y = a.y;
	return a;
}
int main() {
	PhanSo a(2,3), b(4,1), c(0);
	PhanSo d = 5; // PhanSo d = PhanSo(5); hoặc 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): gọi hàm toàn cục
	              // nếu không có hàm toàn cục tương ứng sẽ báo lỗi
	system("pause");
	return 0;
}
```

3.2 Chuyển kiểu bằng Constructor (tt)

Ta nhận thấy nếu cơ chế chuyển kiểu được thực hiện trên cả hai toán hạng thì ta chỉ cần cài đặt 1 hàm toán tử cho một phép toán ở trên.

=> Chuyển hàm toán tử là hàm thành phần của lớp sang hàm toán tử là hàm toàn cục.

3.2 Chuyển kiểu bằng Constructor (tt)

```
class PhanSo{
    long tu, mau;
public:
    void Set (long t, long m);
    PhanSo (long t, long m) { Set(t,m); }
    PhanSo (long t) { Set(t,1); }
    const PhanSo& operator=(const PhanSo &a); //Toán tử gán
    friend PhanSo operator + (PhanSo a, PhanSo b); //Chỉ cần 1 hàm toàn cục
};
int main() {
      PhanSo a(2,3), b(4,1), c(0);
      c = a + b;      // c = operator + (a,b)
      c = a + 5;      // c = operator + (a,PhanSo(5))
      c = 3 + a;      // c = operator + (PhanSo(3),a)
      system("pause");
      return 0;
}
```

3.2 Chuyển kiểu bằng Constructor (tt)

Ta dùng cách chuyển kiểu bằng phương thức thiết lập khi thỏa hai điều kiện sau:

- Chuyển từ kiểu đã có sang kiểu đang định nghĩa.
- Có quan hệ "là một" giữa kiểu đã có và kiểu đang định nghĩa (ví dụ một số nguyên là một phân số).

Các ví dụ chuyển kiểu bằng phương thức thiết lập:

+ Số thực ➔ số phức

+ char * ➔ String

+ số thực ➔ điểm trong mặt phẳng

3.3 Chuyển kiểu bằng phép toán chuyển kiểu

Chuyển kiểu bằng Constructor có nhược điểm:

- Không thể chuyển từ kiểu dữ liệu tự định nghĩa sang kiểu dữ liệu chuẩn, vì không thể sửa đổi kiểu dữ liệu chuẩn.
- Phương thức thiết lập với một tham số sẽ dẫn đến cơ chế chuyển kiểu tự động có thể không mong muốn.

=> Chuyển kiểu bằng cách định nghĩa phép toán chuyển kiểu.

3.3 Chuyển kiểu bằng phép toán chuyển kiểu (tt)

- Phép toán chuyển kiểu là hàm thành phần có dạng:

  Tên_lớp::operator Kiểu_dữ_liệu_chuẩn()

  => Sẽ có cơ chế chuyển kiểu tự động từ kiểu dữ liệu tự định nghĩa (Tên_lớp) sang kiểu dữ liệu chuẩn.
- Ta dùng phép toán chuyển kiểu khi định nghĩa kiểu dữ liệu mới và muốn sử dụng các phép toán đã có của kiểu dữ liệu chuẩn.
- Phép toán chuyển kiểu cũng được dùng để biểu diễn quan hệ "là một" giữa kiểu đang định nghĩa và kiểu đã có (ví dụ một phân số là một số thực).

3.3 Chuyển kiểu bằng phép toán chuyển kiểu - Ví dụ 1

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

3.3 Chuyển kiểu bằng phép toán chuyển kiểu - Ví dụ 2

```
class NumStr { //chuỗi số
        char *s;
    public:
        NumStr(char *p) { s = _strdup(p); }
        /*Định nghĩa phép toán chuyển kiểu*/
        operator double() { return atof(s); } //double atof(const char *str)
        friend ostream& operator << (ostream &os, NumStr &ns);
};
ostream& operator << (ostream &os, NumStr &ns){
    return os << ns.s;
}
```

3.3 Chuyển kiểu bằng phép toán chuyển kiểu - Ví dụ 2 (tt)

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

3.4 Sự nhập nhằng

- Sự nhập nhằng xảy ra khi trình biên dịch tìm được **ít nhất hai cách chuyển kiểu** để thực hiện một tính toán nào đó.
- Hiện tượng này 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.**

3.4 Sự nhập nhằng – Ví dụ 1

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

3.4 Sự nhập nhằng – Ví dụ 2

```
class PhanSo {
    long tu, mau;
    void UocLuoc();
    int SoSanh(PhanSo b);
public:
    void Set(long t, long m);
    PhanSo(long t = 0, long m = 1) {Set(t,m);}
    PhanSo (long t) { Set(t,1); } //ck bằng Constructor
    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;} //phép toán ck
};
```

3.4 Sự nhập nhằng – Ví dụ 2 (tt)

- Lớp PhanSo có hai cơ chế chuyển kiểu tự động:

  + số nguyên ➔ phân số

  (bằng phương thức thiết lập có 1 đối số)

  + phân số ➔ số thực

  (bằng phép toán chuyển kiểu)

- Sự nhập nhằng xảy ra khi ta thực hiện phép cộng:

  phân số + số nguyên

  phân số + số thực

3.4 Sự nhập nhằng – Ví dụ 2 (tt)

```
void main() {
    PhanSo a(2,3), b(3,4), c;
    cout << sqrt(a) << "\n"; //ck a sang số thực bằng phép toán ck
    c = a + b;
    c = a + 2; c = 2 + a;
    /*Nhập nhằng: chuyển a sang số thực hay là chuyển 2 sang phân
    số??? -> báo lỗi: "more than one operator "+" matches these
    operands" => Bỏ phép toán chuyển kiểu, khi đó trình biên dịch sẽ
    chuyển 2 sang phân số */
    double r = 2.5 + a; r = a + 2.5;
    /*Báo lỗi tương tự vì có sự nhập nhằng: chuyển 2.5 sang phân số hay
    là chuyển a sang số thực??? => Bỏ hàm toán tử +, khi đó trình biên
    dịch sẽ chuyển a sang số thực */
}
```

3.4 Sự nhập nhằng – Ví dụ 2 (tt)

Để tránh sự nhập nhằng ta phải chuyển kiểu một cách tường minh, không áp dụng cơ chế chuyển kiểu tự động, mặc dù điều này làm mất đi sự tiện lợi của cơ chế chuyển kiểu tự động.

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

4. Overload một số toán tử thông dụng

- Toán tử gán
- Toán tử << và >>
- Toán tử lấy phần tử mảng [ ]
- Toán tử gọi hàm ()
- Toán tử ++ và --

4.1 Overload Toán tử gán

Khi lớp có các thuộc tính kiểu con trỏ hay tham chiếu thì không sử dụng hàm tạo sao chép mặc định và toán tử gán mặc định được bởi vì khi đó 2 con trỏ (thuộc tính của 2 đối tượng) cùng trỏ đến một vùng nhớ nên sự thay đổi của đối tượng này sẽ ảnh hưởng đến đối tượng kia.

=> Xây dựng **Hàm tạo sao chép** và **Hàm toán tử gán** cho lớp.

Khái niệm Hàm tạo sao chép

- **Hàm tạo sao chép có một đối kiểu tham chiếu để khởi gán cho đối tượng mới.**

```
Tên_lớp(const Tên_lớp &u){
        this->Tên_thuộc_tính = u.Tên_thuộc_tính;
        this->Tên_con_trỏ = new Kiểu_dữ_liệu[n];
        memcpy(this->Tên_con_trỏ,u.Tên_con_trỏ,n);
}
Tên_lớp u; //Gọi tới hàm tạo mặc định
Tên_lớp v(u); //Gọi tới hàm tạo sao chép
```

- Mục đích của hàm tạo sao chép là tạo ra đối tượng **v** có nội dung ban đầu giống như đối tượng **u** nhưng độc lập với **u**.

Khái niệm Hàm toán tử gán

Trong hàm toán tử gán, đối ẩn (con trỏ this) biểu thị đối tượng đích và 1 đối tường minh biểu thị đối tượng nguồn.

```
Kiểu_trả_về operator=(const Tên_lớp &u){
    this->Tên_thuộc_tính = u.Tên_thuộc_tính;
    this->Tên_con_trỏ = new Kiểu_dữ_liệu[n];
    memcpy(this->Tên_con_trỏ,u.Tên_con_trỏ,n);}
```

Ví dụ: HT v; //gọi tới hàm tạo mặc định của lớp HT

v = u; //gọi tới hàm toán tử gán để gán u cho v

So sánh Hàm toán tử gán và Hàm tạo sao chép

- Hàm toán tử gán không tạo ra đối tượng mới, chỉ thực hiện phép gán giữa hai đối tượng đã tồn tại.
- Hàm tạo sao chép được dùng để tạo một đối tượng mới và gán nội dung của một đối tượng đã tồn tại cho đối tượng mới vừa tạo.
- **Câu lệnh new và câu lệnh khai báo -> gọi hàm tạo:**

  HT *h = new HT(50,6); //gọi hàm tạo có đối

  HT k = *h; //kg gọi hàm toán tử gán mà gọi hàm tạo sao chép
- **Câu lệnh gán -> gọi hàm toán tử gán:**

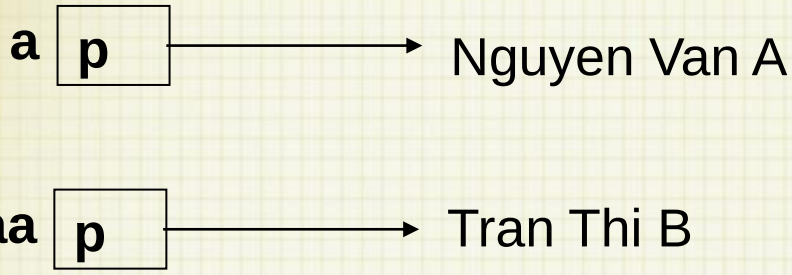
  HT u; u = *h; //gọi hàm toán tử gán

4.1 Overload Toán tử gán – Ví dụ

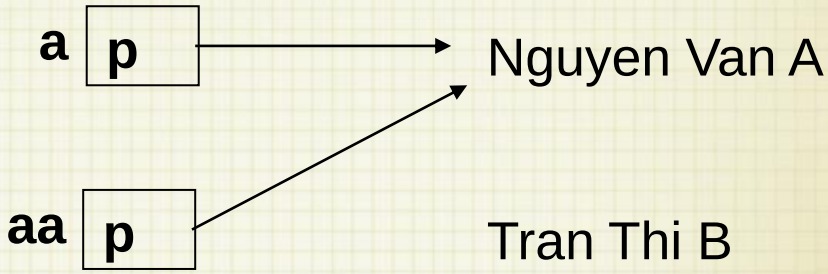
```
class String{
    char *p;
public:
    String(char *s = "") { p = _strdup(s); }
    String(const String &s) { p = _strdup(s.p); } //Hàm tạo sao chép
    ~String() { cout <<"delete"<<(void*)p<<"\n"; delete [] p; }
    void Output() const { cout << p; }
};
void main(){
    String a("Nguyen Van A");
    String b = a; //Gọi hàm tạo sao chép
    String aa = “Tran Thi B";
    cout << "aa = ";   aa.Output();   cout << "\n";
    aa = a; //Gọi toán tử gán mặc định vì không đn hàm toán tử gán
    cout << "aa = ";   aa.Output();   cout << "\n";
}
```

4.1 Overload Toán tử gán – Ví dụ (tt)

Trước khi gán



Sau khi gán



Kết quả thực hiện chương trình:

aa = Tran Thi B

aa = Nguyen Van A

delete 0x0d36

delete 0x0d48

delete 0x0d36

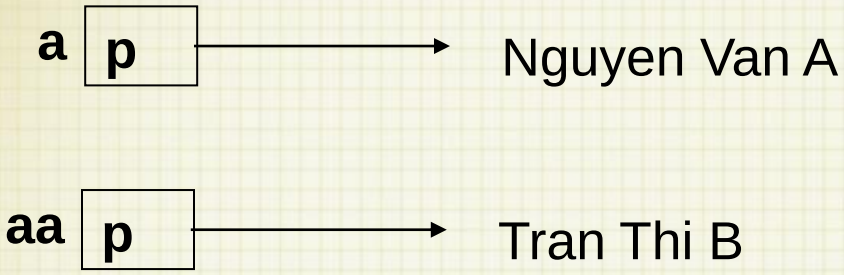
Null pointer assignment

4.1 Overload Toán tử gán – Ví dụ (tt)

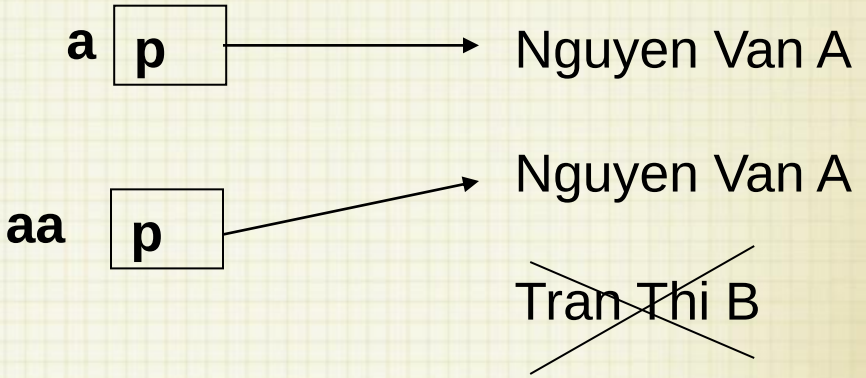
```
class String {
    char *p;
public:
    String(char *s = "") {p = _strdup(s);}
    String(const String &s) {p = _strdup(s.p);} //Hàm tạo sao chép
    ~String() {cout << "delete "<< (void *)p << "\n"; delete [] p;}
    String& operator = (const String &s); //Thêm hàm toán tử gán
    void Output() const {cout << p;}
};
String& String::operator = (const String &s) {
    if (this != &s){
            delete [] p; //Dọn dẹp tài nguyên cũ
            p = _strdup(s.p);} //Sao chép mới
    return *this;
}
```

4.1 Overload Toán tử gán – Ví dụ (tt)

Trước khi gán



Sau khi gán



Kết quả thực hiện chương trình: aa = Tran Thi B

aa = Nguyen Van A

delete 0x0d5a

delete 0x0d48

delete 0x0d36

4.2 Overload toán tử << và >>

- C++ định nghĩa chồng hai toán tử thao tác trên bit là **dịch trái <<** và **dịch phải >>** trong lớp **ostream** và **istream**.
- Lớp **ostream** định nghĩa **toán tử xuất <<** (áp dụng cho các kiểu dữ liệu chuẩn) và một số phương thức xuất khác (flush, put, write, …).
- Lớp **istream** định nghĩa **toán tử nhập >>** (áp dụng cho các kiểu dữ liệu chuẩn) và một số phương thức nhập khác (get, getline, ignore, ...).

4.2 Overload toán tử << và >> (tt)

Dòng **cout:**

- Là một đối tượng kiểu **ostream**
- Là dòng xuất chuẩn gắn với màn hình.
- Các thao tác xuất trên dòng **cout** đồng nghĩa với xuất dữ liệu ra màn hình.

Dòng **cin:**

- Là một đối tượng kiểu **istream**
- Là dòng nhập chuẩn gắn với bàn phím.
- Các thao tác nhập trên dòng **cin** đồng nghĩa với nhập dữ liệu từ bàn phím.

Toán tử << (Lớp ostream)

C++ định nghĩa chồng **toán tử dịch trái <<** để gửi dữ liệu có kiểu dựng sẵn ra dòng xuất chuẩn **cout**

```
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 ...
};
```

Toán tử >> (Lớp istream)

C++ định nghĩa chồng **toán tử dịch phải >>** để nhận dữ liệu có kiểu dựng sẵn từ dòng nhập chuẩn **cin**

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

4.2.1 Overload toán tử <<

Do **cout** là một đối tượng của lớp **ostream** nên có thể sử dụng **toán tử xuất <<** cũng như gọi các phương thức xuất khác của lớp **ostream** (flush, put, write, …).

cout << Tham_số_1 **<<** Tham_số_2 **<<**…**<<** Tham_số_n;

Các tham số có thể là:

- **Giá trị** nguyên, giá trị thực, một ký tự.
- **Con trỏ ký tự** (char *) để xuất chuỗi ký tự.

Ví dụ: **cout <<** a **<<** "\n"; //Nghĩa là bỏ a và "\n" vào **cout**

4.2.1 Overload toán tử << (tt)

Định nghĩa chồng **toán tử <<** để gửi ra dòng xuất chuẩn **cout** dữ liệu có kiểu tự định nghĩa.

```
friend ostream& operator<<(ostream& os, const Kiểu_tự_đn &a)
{
    os << a.Thuộc_tính; //Hàm bạn nên có thể truy xuất dl của lớp
    return os;
}
```

=> Sử dụng: cout << Đối_tượng_có_kiểu_tự_định_nghĩa;

4.2.1 Overload toán tử << (tt)

Nếu không phải là hàm bạn mà là hàm thành phần của lớp thì hàm toán tử trên chỉ có một đối số như sau:

```
ostream& operator<<(ostream& os) {
    os << this.Thuộc_tính; hoặc os << Dữ_liệu_của_lớp;
    return os;
}
```

=> Sử dụng:

```
Đối_tượng_có_kiểu_tự_định_nghĩa.operator<<(cout);
```

4.2.2 Overload toán tử >>

Do **cin** là một đối tượng của lớp **istream** nên có thể sử dụng **toán tử nhập >>** cũng như gọi các phương thức nhập khác của lớp **istream** (get, getline, ignore, …).

cin >> Tham_số_1 **>>** Tham_số_2 **>>**…**>>** Tham_số_n;

Các tham số có thể là:

- **Biến hoặc phần tử mảng** kiểu số nguyên, số thực, ký tự để nhận một số nguyên, một số thực, một ký tự.
- **Con trỏ ký tự** (char *) để nhận một dãy ký tự đến khi gặp một **ký tự trắng (dấu cách, dấu tab, dấu chuyển dòng).**

Ví dụ: **cin >>** a **>>** b; //Nghĩa là bỏ **cin** vào a và b

4.2.2 Overload toán tử >> (tt)

Định nghĩa chồng **toán tử <<** để nhận dữ liệu từ dòng nhập chuẩn **cin** và gán cho biến có kiểu tự định nghĩa.

```
friend istream& operator>>(istream& is, Kiểu_tự_đn &a) {
    is >> a.Thuộc_tính;
    return is;
}
```

=> Sử dụng: cin >> Đối_tượng_có_kiểu_tự_định_nghĩa;

4.2.2 Overload toán tử >> (tt)

Nếu không phải là hàm bạn mà là hàm thành phần của lớp thì hàm toán tử trên chỉ có một đối số như sau:

```
istream& operator>>(istream& is) {
    is >> this.Thuộc_tính; hoặc is >> Dữ_liệu_của_lớp;
    return is;
}
```

=> Sử dụng:

```
Đối_tượng_có_kiểu_tự_định_nghĩa.operator>>(cin);
```

4.2 Overload toán tử << và >> – Ví dụ

```
class PhanSo {
    long tu, mau;
    void UocLuoc();
public:
    void Set ( long t, long m);
    PhanSo ( long t = 0, long m = 1) {Set(t,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); //nhập phân số
    friend ostream& operator << (ostream &os, PhanSo p); //xuất phân số
};
```

4.2 Overload toán tử << và >> – Ví dụ (tt)

```
istream& operator >> (istream &is, PhanSo &p){
    is >> p.tu >> p.mau;
    while (!p.mau){
        cout << “Nhap lai mau so: ”; //nhập lại mẫu số nếu mẫu số = 0
        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;
}
```

4.2 Overload toán tử << và >> – Ví dụ (tt)

```
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";
    system("pause");
}
```

4.3 Overload toán tử lấy phần tử mảng [ ]

Định nghĩa chồng toán tử **[ ]** 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]; } /*Kết quả trả về phải là tham
    chiếu để phần tử trả về có thể đứng bên trái của phép gán (lvalue)*/
    friend ostream& operator << (ostream& os, const String &s);
};
```

4.3 Overload toán tử lấy phần tử mảng [ ] (tt)

```
void main() {
    String a("Nguyen van A");
    cout << a[7] << "\n";          // a.operator[ ](7)
    a[7] = 'V'; /*Được phép do kq trả về của hàm toán tử [ ] là tham chiếu.
    Nếu không phép gán trên sẽ bị báo lỗi "expression must be a
    modifiable lvalue" */
     cout << a[7] << "\n";         // a.operator[ ](7)
     cout << a << "\n";
}
```

4.3 Overload toán tử lấy phần tử mảng [ ] (tt)

Cải tiến hàm toán tử trên để sử dụng an toàn khi chỉ số không hợp lệ:

```
void main() {
        char *a = “Dai hoc Tu nhien”;
        a[300] = ‘H’;    // Chỉ số không hợp lệ
        String aa(“Dai hoc Tu nhien”);
        aa[300] = ‘H’;  // Chỉ số không hợp lệ
}
char& String::operator[ ](int i) {
          return (i > 0 && i < _strlen(p)) ? p[i] : p[0];
}
```

4.3 Overload toán tử lấy phần tử mảng [ ] (tt)

Tuy nhiên, hàm toán tử trên không sử dụng được với đối tượng hằng.

```
void main() {
    String a("Nguyen van A");
    const String aa("Dai Hoc Tu Nhien"); //aa là đối tượng hằng
    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";
}
```

4.3 Overload toán tử lấy phần tử mảng [ ] (tt)

Cách khắc phục: định nghĩa một hàm toán tử khác dành 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 (i>=0 && i<strlen(p))?p[i]:c;}
};
char String::c = 'A';
```

4.3 Overload toán tử lấy phần tử mảng [ ] (tt)

```
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"; //Gọi hàm toán tử operator[](int) const
    aa[4] = 'L'; /* Báo lỗi "expression must be a modifiable lvalue"
    và ta cũng không được phép thay đổi giá trị của đt hằng */
    cout << aa[4] << "\n"; //Gọi hàm toán tử operator[](int) const
    cout << aa << "\n";
}
```

4.4 Overload toán tử gọi hàm ()

Toán tử [ ] chỉ có một tham số

➔ Chỉ được dùng để truy xuất phần tử của đối tượng thuộc loại mảng một chiều.

Toán tử gọi hàm ( ) có số tham số bất kỳ

➔ Được dùng để truy xuất phần tử của các đối tượng thuộc loại mảng hai hay nhiều chiều.

Ví dụ: Lớp ma trận sau định nghĩa **hàm toán tử ()** với hai đối số ➔ ta có thể truy xuất phần tử của ma trận thông qua chỉ số dòng và chỉ số cột.

4.4 Overload toán tử gọi hàm () – Ví dụ

```
class MATRIX{
    float **M;
    int row, col;
public:
    MATRIX (int, int);
    ~MATRIX();
    float& operator() (int, int); //Hàm toán tử ()
};
float MATRIX::operator() (int i, int j){
    return M[i][j];
}
```

4.4 Overload toán tử gọi hàm () – Ví dụ (tt)

```
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]; //M[i] = new float[c]
    delete [ ] M; //M = new float* [r]
}
```

4.4 Overload toán tử gọi hàm () – Ví dụ (tt)

```
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;
    }
}
```

4.5 Overload toán tử ++ và --

++ và **--** dùng để tăng giá trị của đối tượng lên ***giá trị kế tiếp*** hoặc giảm giá trị của đối tượng xuống ***giá trị trước đó***.

++ và **--** chỉ áp dụng cho các kiểu dữ liệu đếm được (mỗi giá trị của đối tượng đều có ***giá trị kế tiếp*** và ***giá trị trước đó***).

++ và **--** vừa là toán tử trước vừa là toán tử sau.

4.5 Overload toán tử ++ và – (tt)

- Khi là toán tử trước, **++a** có hai vai trò:
  - Tăng **a** lên ***giá trị kế tiếp***
  - Trả về tham chiếu đến chính **a**
- Khi là toán tử sau, **a++** có hai vai trò:
  - Tăng **a** lên ***giá trị kế tiếp***
  - Trả về giá trị bằng với **a** trước khi tăng

4.5 Overload toán tử ++ và -- – Ví dụ 1

```
class ThoiDiem{
    long tsgiay;
    static bool HopLe(int g, int p, int gy); //Giờ, phút, giây
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;} //1h = 3600s
    int LayPhut() const {return (tsgiay%3600)/60;}
    int LayGiay() const {return tsgiay % 60;}
    void Tang();
    void Giam();
    ThoiDiem& operator ++();
};
```

4.5 Overload toán tử ++ và -- – Ví dụ 1 (tt)

```
void ThoiDiem::Tang(){
    tsgiay = ++tsgiay%SOGIAY_NGAY;
    //SOGIAY_NGAY = 86.400s
}
void ThoiDiem::Giam()
{
    if (--tsgiay < 0) tsgiay = SOGIAY_NGAY-1;
}
ThoiDiem& ThoiDiem::operator ++() {
    Tang();
    return *this;
}
```

4.5 Overload toán tử ++ và -- – Ví dụ 1 (tt)

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

4.5 Overload toán tử ++ và – (tt)

Vì toán tử **++** (hay **--**) hoạt động khác nhau khi là toán tử trước và toán tử sau nên phải định nghĩa hai hàm toán tử khác nhau cho mỗi toán tử này:

```
ThoiDiem& operator ++(); //Khi ++ là toán tử trước
ThoiDiem operator ++(int); /* Khi ++ là toán tử sau.
Có thêm 1 đối số giả để phân biệt với hàm toán tử
trên nhưng không sử dụng đối số này */
```

4.5 Overload toán tử ++ và -- – Ví dụ 2

```
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;
}
```

4.5 Overload toán tử ++ và -- – Ví dụ 2 (tt)

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

Bài tập nộp lần 2

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

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

Trong đó: n là bậc của đa thức

$a_1, a_2, a_3, \ldots, a_j$ 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 yêu cầu 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 hàm toán tử để thực hiện 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