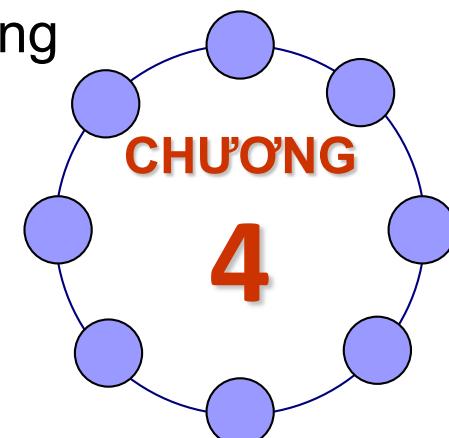


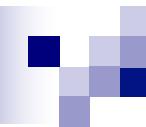
CHƯƠNG 4:

ĐA NĂNG HÓA TOÁN TỬ

(OPERATOR OVERLOADING)

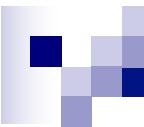
Khoa Công Nghệ Thông Tin và Truyền Thông
Đại học Bách khoa – Đại học Đà Nẵng





Nội dung

- Đa năng hoá hàm.
- Đa năng hoá toán tử.
- Giới hạn của đa năng hoá toán tử
- Chuyển đổi kiểu.
- Đa năng hoá toán tử xuất (<<)- nhập (>>)
- Đa năng hoá toán tử [], toán tử ()
- Khởi tạo ngầm định - Gán ngầm định.
- Đa năng hoá toán tử ++ và --
- Đa năng hoá new và delete



Đa năng hóa hàm

- Định nghĩa các hàm cùng tên
- Đối số phải khác nhau:
 - Số lượng
 - Thứ tự
 - Kiểu dữ liệu
- Có thể dùng đối số mặc định.

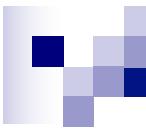
```
long GetTime (void); // số giây tính từ nửa đêm  
void GetTime (int &hours=0,  
              int &minutes=0,  
              int &seconds=0);
```

```
void main() {  
    int h, m, s;  
    long t = GetTime(); // Gọi hàm ???  
    GetTime(h, m, s); // Gọi hàm ???  
}
```

Từng cặp hàm nào sau đây là đa năng hóa hàm.

1. void HV(int a, int b);
2. void HV(int *a, int *b);
3. void HV(int &a, int &b);

Cặp hàm (1,2)
Cặp hàm (2,3)



Đa năng hoá toán tử

- Định nghĩa các phép toán trên đối tượng.
- Các phép toán có thể tái định nghĩa:

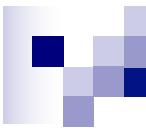
Đơn hạng	+	-	*	!	~	&	++	--	()	->	->*
	new	delete									
Nhi hạng	+	-	*	/	%	&		^	<<	>>	
	=	+=	-=	/=	%=	&=	=	^=	<<=	>>=	
	==	!=	<	>	<=	>=	&&		[]	()	,

- Các phép toán không thể tái định nghĩa:

• .* :: ?: sizeof

Giới hạn của đa năng hóa toán tử

- toán tử gọi hàm () - là một toán tử nhiều ngôi.
- Thứ tự ưu tiên của một toán tử không thể được thay đổi bởi đa năng hóa.
- Tính kết hợp của một toán tử không thể được thay đổi bởi đa năng hóa. Các tham số mặc định không thể sử dụng với một toán tử đa năng hóa.
- Không thể thay đổi số các toán hạng mà một toán tử yêu cầu.
- Không thể thay đổi ý nghĩa của một toán tử làm việc trên các kiểu có sẵn.
- Không thể dùng đối số mặc định.



Đa năng hoá toán tử

- Khai báo và định nghĩa toán tử thực chất không khác với việc khai báo và định nghĩa nghĩa một loại hàm bất kỳ nào khác
- sử dụng tên hàm là "**operator @**" cho toán tử "@"
 - để overload phép "+", ta dùng tên hàm "operator +"
- Số lượng tham số tại khai báo phụ thuộc hai yếu tố:
 - Toán tử là toán tử đơn hay đôi
 - Toán tử được khai báo là hàm toàn cục hay phương thức của lớp

aa@bb

@aa

aa@

aa.operator@(bb)

aa.operator@()

aa.operator@(int)

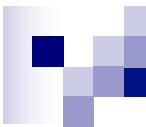
hoặc operator@(aa,bb)

hoặc operator@(aa)

hoặc operator@(aa,int)

Là phương thức của lớp

Là hàm toàn cục



Đa năng hoá toán tử

- Ví dụ: Sử dụng toán tử "+" để cộng hai đối tượng lớp **Complex** và trả về kết quả là một **Complex**
 - Ta có thể khai báo hàm toàn cục sau

```
const Complex operator+(const Complex& num1,  
                           const Complex& num2);
```

 - "x+y" sẽ được hiểu là "operator+(x,y)"
 - dùng từ khoá const để đảm bảo các toán hạng gốc không bị thay đổi
 - Hoặc khai báo toán tử dưới dạng thành viên của **Complex**:

```
const Complex operator+(const Complex& num);
```

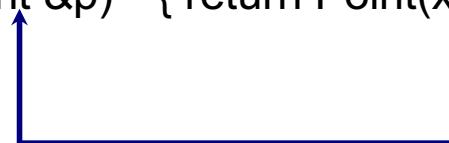
 - đối tượng chủ của phương thức được hiểu là toán hạng thứ nhất của toán tử.
 - "x+y" sẽ được hiểu là "x.operator+(y)"

```
Complex x(5);  
Complex y(10);  
...  
z = x + y;
```

Đa năng hóa toán tử bằng hàm thành viên

- Khi đa năng hóa (), [] -> hoặc =, hàm toán tử phải được khai báo là hàm thành viên của lớp
- Toán tử một ngôi hàm không có tham số, toán tử 2 ngôi hàm sẽ có 1 tham số

```
class Point {  
    public:  
        Point (int x, int y)    { Point::x = x; Point::y = y; }  
        Point operator + (Point &p) { return Point(x + p.x,y + p.y); }  
        Point operator - (Point &p) { return Point(x - p.x, y - p.y); }  
    private:  
        int x, y;  
};
```



Có 1 tham số
(Nếu là toán tử hai ngôi)

```
void main() {  
    Point p1(10,20), p2(10,20);  
    Point p3 = p1 + p2;           Point p4 = p1 - p2;  
    Point p5 = p3.operator + (p4); Point p6 = p3.operator - (p4);  
}
```

Đa năng hóa toán tử bằng hàm thành viên

- $a @ b$: $a.\text{operator } @ (b)$
- $x @ b$: với x là thuộc kiểu float, int, ...
không thuộc kiểu lớp đang định nghĩa
 - $\text{operator } @ (x, b)$

Đa năng hóa toán tử bằng hàm toàn cục

- Nếu toán hạng cực trái của toán tử là đối tượng thuộc lớp khác hoặc thuộc kiểu dữ liệu có sẵn
 - thường khai báo **friend**

```
class Point{  
public:  
    Point (int x, int y) { Point::x = x; Point::y = y; }  
    friend Point operator + (Point &p, Point &q);  
    friend Point operator - (Point &p, Point &q);  
private:  
    int x, y;  
};  
Point operator + (Point &p, Point &q)  
    {return Point(p.x + q.x,p.y + q.y); }  
Point operator - (Point &p, Point &q)  
    {return Point(p.x - q.x,p.y - q.y); }
```

Có 2 tham số
(Nếu là toán tử hai ngôi)

```
void main() {  
    Point p1(10,20), p2(10,20);  
    Point p3 = p1 + p2;           Point p4 = p1 - p2;  
    Point p5 =operator + (p3, p4); Point p6 = operator - (p3, p4);  
}
```

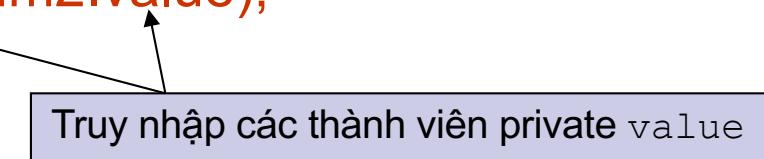
Đa năng hóa toán tử bằng hàm toàn cục

- Ví dụ về phép cộng cho **Complex**, ta có thể khai báo hàm định nghĩa phép cộng tại mức toàn cục:

```
const Complex operator+(const Complex& num1, const Complex& num2);
```

- Khi đó, ta có thể định nghĩa toán tử đó như sau:

```
const Complex operator+(const Complex& num1,const Complex& num2) {  
    Complex result(num1.value + num2.value);  
    return result;  
}
```


Truy nhập các thành viên private value

- **Giải pháp: dùng hàm friend**

- **friend** cho phép một lớp cấp quyền truy nhập tới các phần nội bộ của lớp đó cho một số cấu trúc được chọn

Đa năng hóa toán tử bằng hàm toàn cục

- Để khai báo một hàm là friend của một lớp, ta phải khai báo hàm đó bên trong khai báo lớp và đặt từ khoá friend lên đầu khai báo.

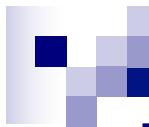
```
class Complex {  
public:  
    Complex(int value = 0);  
    ~Complex();  
    ...  
    friend const Complex operator + (const Complex& num1,  
                                      const Complex& num2);  
    ...  
};
```

- Lưu ý: tuy khai báo của hàm **friend** được đặt trong khai báo lớp và hàm đó có quyền truy nhập ngang với các phương thức của lớp, hàm đó *không phải* phương thức của lớp

Đa năng hóa toán tử bằng hàm toàn cục

- Không cần thêm sửa đổi gì cho định nghĩa của hàm đã được khai báo là **friend**.
 - Định nghĩa trước của phép cộng vẫn giữ nguyên

```
const Complex operator+(const Complex& num1,  
                      const Complex& num2) {  
    Complex result(num1.value + num2.value);  
    return result;  
}
```



Khi nào dùng toán tử toàn cục?

- Đối với toán tử được khai báo là phương thức của lớp, đối tượng chủ (xác định bởi con trỏ **this**) luôn được hiểu là toán hạng đầu tiên (trái nhất) của phép toán.
 - Nếu muốn dùng cách này, ta phải được quyền bổ sung phương thức vào định nghĩa của lớp/kiểu của toán hạng trái
- Không phải lúc nào cũng có thể overload toán tử bằng phương thức
 - phép cộng giữa Complex và float cần cả hai cách **Complex + float** và **float+ Complex**
 - **cout << obj;**
 - không thể sửa định nghĩa kiểu **int** hay kiểu của **cout**
 - lựa chọn duy nhất: đa năng hóa toán tử bằng hàm toàn cục

Đa năng hoá toán tử xuất <<

- prototype như thế nào? xét ví dụ:
 - `cout << num;` // num là đối tượng thuộc lớp Complex
- Toán hạng trái `cout` thuộc lớp **ostream**, không thể sửa định nghĩa lớp này nên ta overload bằng hàm toàn cục
 - Tham số thứ nhất : tham chiếu tới **ostream**
 - Tham số thứ hai : kiểu **Complex**,
 - **const** (do không có lý do gì để sửa đổi đối tượng được in ra)
 - giá trị trả về: tham chiếu tới **ostream**
(để thực hiện được `cout << num1 << num2;`)
- Kết luận:

ostream& operator<<(ostream& out, const Complex& num)

Đa năng hoá toán tử xuất <<

- Khai báo toán tử được overload là **friend** của lớp **Complex**

```
class Complex {  
public:  
    Complex(float R = 0, float I = 0);  
    ~Complex();  
    ...  
    friend ostream& operator<<( ostream& out, const Complex& num);  
    ...  
};
```

- Định nghĩa toán tử

```
ostream& operator<<(ostream& out, const Complex& num) {  
    out << "("<<num.R<<","<<num.I<<")";  
    // Use version of insertion operator defined for float  
    return out; // Return a reference to the modified stream  
};
```

Đa năng hoá toán tử nhập >>

- prototype như thế nào? xét ví dụ:
 - `cin >> num;` // num là đối tượng thuộc lớp Complex
- Toán hạng trái `cin` thuộc lớp `istream`, không thể sửa định nghĩa lớp này nên ta overload bằng hàm toàn cục
 - Tham số thứ nhất : tham chiếu tới `istream`
 - Tham số thứ hai : kiểu **Complex**,
 - giá trị trả về: tham chiếu tới `istream`
(để thực hiện được `cin >> num1 >> num2;`)
- Kết luận:

`istream& operator>>(istream& in, Complex& num)`

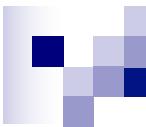
Đa năng hoá toán tử nhập>>

- Khai báo toán tử được overload là **friend** của lớp **Complex**

```
class Complex {  
public:  
    Complex(float R = 0, float I = 0);  
    ~Complex();  
    ...  
    friend istream& operator>>( istream& in, Complex& num);  
    ...  
};
```

- Định nghĩa toán tử

```
istream& operator>>(istream& in, Complex& num) {  
    cout<<"Nhập phần thực:"; in >> num.R;  
    cout<<"Nhập phần ảo:"; in >> num.I;  
    return in; // Return a reference to the modified stream  
};
```



Đa năng hoá toán tử []

- Thông thường để xuất ra giá trị của 1 phần tử tại vị trí cho trước trong đối tượng.
- Định nghĩa là hàm thành viên.
- **Để hàm toán tử [] có thể đặt ở bên trái của phép gán thì hàm phải trả về là một tham chiếu**

```
class Vector {  
    private: int Size; int *Data;  
    public:  
        Vector(int S = 2, int V = 0);  
        ~Vector();  
        void Print() const;  
        int & operator [] (int i);  
};  
int& Vector::operator [] (int i) {  
    static int tam = 0;  
    return (i > 0) && (i < Size)? Data[i] : tam;  
}
```

Đa năng hóa toán tử []

■ Ví dụ:

```
class Vector {  
private:  
    int Size;int *Data;  
public:  
    Vector(int S = 2,int V = 0);  
    ~Vector();  
    void Print() const;  
    int & operator [ ] (int i);  
};  
int& Vector::operator [ ] (int i) {  
    static int tam = 0;  
    return ( i > 0) && (i < Size)? Data[i] : tam;  
}
```

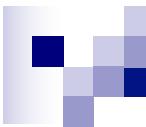
```
...  
int main() {  
    Vector V(5,1);  
    V.Print();  
    for(int I=0;I<5;++I)  
        V[I]*=(I+1);  
    V.Print();  
    V[0]=10;//V.operator[](0)  
    V.Print();  
    return 0;  
}
```

Đa năng hoá toán tử ()

■ Định nghĩa là hàm thành viên.

```
class Matrix {  
public:  
    Matrix (const short rows, const short cols);  
    ~Matrix (void) {delete elems;}  
    double& operator () (const short row,  
                          const short col);  
    friend ostream& operator << (ostream&, Matrix&);  
    friend Matrix operator + (Matrix&, Matrix&);  
    friend Matrix operator - (Matrix&, Matrix&);  
    friend Matrix operator * (Matrix&, Matrix&);  
private:  
    const short    rows;      // số hàng  
    const short    cols;      // số cột  
    double        *elems;    // các phần tử  
};
```

```
double& Matrix::operator ()  
    (const short row, const short col)  
{  
    static double dummy = 0.0;  
    return (row >= 0 && row < rows  
            && col >= 0 && col < cols)  
        ? elems[row *cols  
                + col]  
        : dummy;  
}  
void main() {  
    Matrix m(3,2);  
    m(1,1) = 10; m(1,2) = 20;  
    m(2,1) = 30; m(2,2) = 40;  
    m(3,1) = 50; m(3,2) = 60;  
    cout<<m<<endl;  
}
```



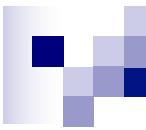
Chuyển kiểu

- Muốn thực hiện các phép cộng:

```
void main() {  
    Point p1(10,20), p2(30,40), p3, p4, p5;  
    p3 = p1 + p2;  
    p4 = p1 + 5; p5 = 5 + p1;  
};
```

→ Có thể định nghĩa thêm 2 toán tử:

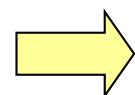
```
class Point {  
    //...  
    friend Point operator + (Point, Point);  
    friend Point operator + (int, Point);  
    friend Point operator + (Point, int);  
};
```



Chuyển kiểu (tt)

- Chuyển đổi kiểu: ngôn ngữ định nghĩa sẵn.

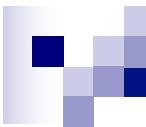
```
void main() {  
    Point p1(10,20), p2(30,40), p3, p4, p5;  
    p3 = p1 + p2;  
    p4 = p1 + 5; // tương đương p1 + Point(5)  
    p5 = 5 + p1; // tương đương Point(5) + p1  
}
```



Định nghĩa phép chuyển đổi kiểu

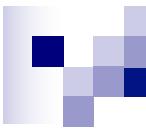
```
class Point {  
    //...  
    Point (int x) { Point::x = Point::y = x; }  
    friend Point operator + (Point, Point);  
};
```

Chuyển kiểu
 $5 \Leftrightarrow \text{Point}(5)$



Chuyển kiểu (tt)

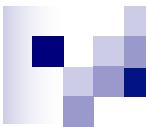
- Một toán tử chuyển đổi kiểu có thể được sử dụng để chuyển đổi một đối tượng của một lớp thành đối tượng của một lớp khác hoặc thành một đối tượng của một kiểu có sẵn.
 - Định nghĩa là hàm thành viên không tĩnh và không là hàm **friend**.
 - Prototype của hàm thành viên này có cú pháp:
 - **operator <data type> ()**;



Chuyển kiểu (tt)

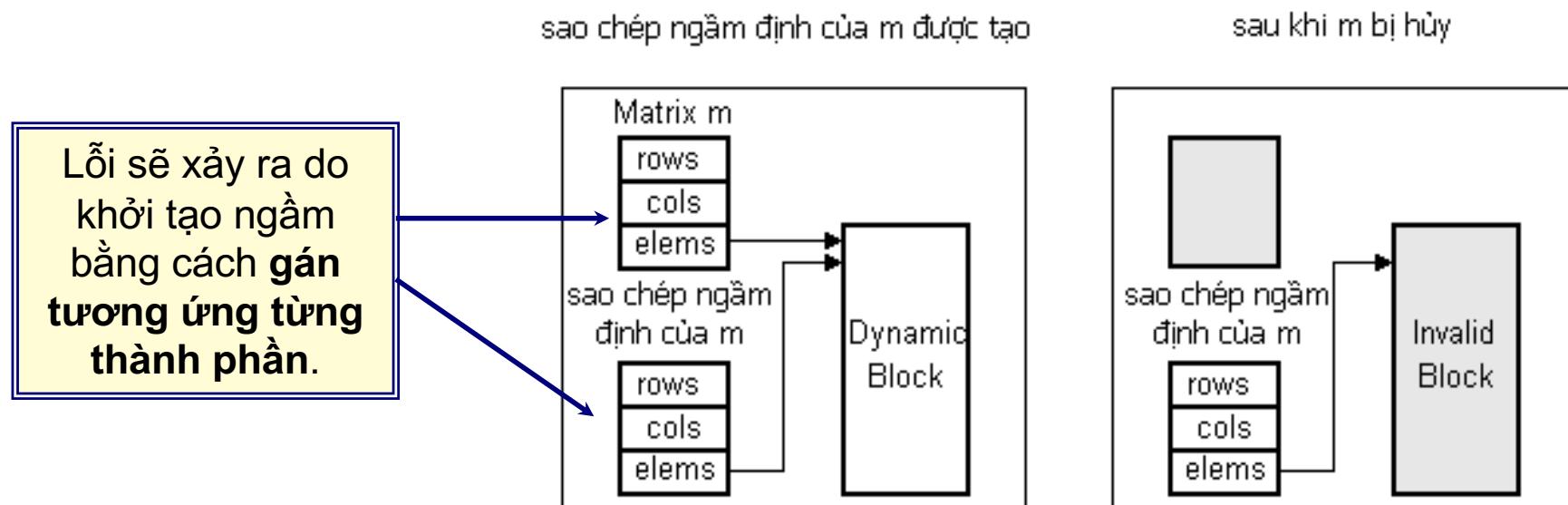
Ví dụ:

```
class Number  {
    private:
        float Data;
    public:
        Number(float F = 0.0) { Data = F; }
        operator float() { return Data; }
        operator int() { return (int)Data; }
    };
int main() {
    Number N1(9.7), N2(2.6);
    float X = (float)N1; //Gọi operator float()
    cout << X << endl;
    int Y = (int)N2; //Gọi operator int()
    cout << Y << endl;
    Number Z = 3.5;
    return 0;
}
```

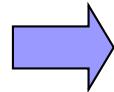


Khởi tạo ngầm định

- Được định nghĩa sẵn trong ngôn ngữ:
VD: Point p1(10,20); Point p2 = p1;
- Sẽ gây ra lỗi (kết quả SAI) khi bên trong đối tượng có **thành phần dữ liệu là con trỏ**.
VD: Matrix m(5,6); Matrix n = m;



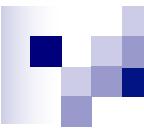
Khởi tạo ngầm định (tt)



Khi lớp có **thành phần dữ liệu con trỏ**,
phải định nghĩa hàm **xây dựng sao chép**

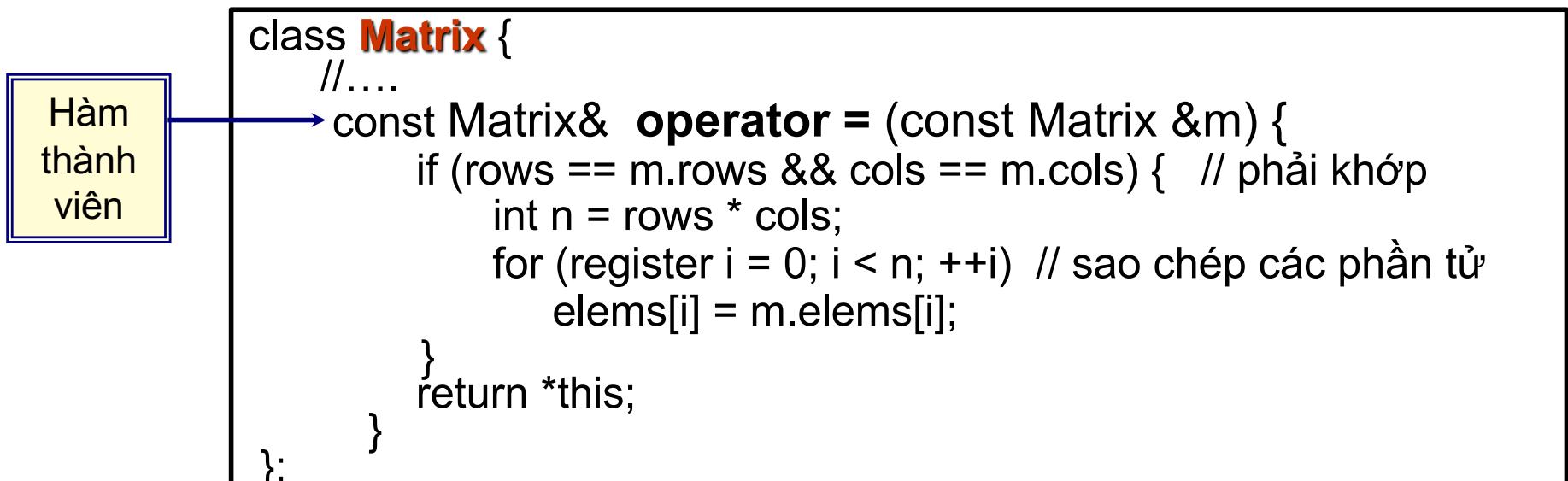
```
class Point {  
    int x, y;  
public:  
    Point (int =0; int =0 );  
    // Không cần thiết DN  
    Point (const Point& p) {  
        x= p.x;  
        y = p.y;  
    }  
    // .....  
};  
// .....
```

```
class Matrix {  
    //....  
    Matrix(const Matrix&);  
};  
Matrix::Matrix (const Matrix &m)  
    : rows(m.rows), cols(m.cols)  
{  
    int n = rows * cols;  
    elems = new double[n];      // cùng kích thước  
    for (register i = 0; i < n; ++i) // sao chép phần tử  
        elems[i] = m.elems[i];  
}
```

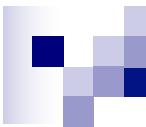


Gán ngầm định

- Được định nghĩa sẵn trong ngôn ngữ:
 - Gán tương ứng từng thành phần.
 - **Đúng** khi đối tượng không có dữ liệu con trỏ.
VD: Point p1(10,20); Point p2; p2 = p1;
- Khi thành phần dữ liệu có con trỏ, bắt buộc phải định nghĩa phép gán = cho lớp.



```
class Matrix {  
    //....  
    const Matrix& operator = (const Matrix &m) {  
        if (rows == m.rows && cols == m.cols) { // phải khớp  
            int n = rows * cols;  
            for (register i = 0; i < n; ++i) // sao chép các phần tử  
                elems[i] = m.elems[i];  
        }  
        return *this;  
    }  
};
```



Phép gán =

- Một trong những toán tử hay được overload nhất
 - Cho phép gán cho đối tượng này một giá trị dựa trên một đối tượng khác
 - Copy constructor cũng thực hiện việc tương tự, cho nên, định nghĩa toán tử gán gần như giống hệt định nghĩa của copy constructor
- Ta có thể khai báo phép gán cho lớp **MyNumber** như sau:
const MyNumber& operator=(const MyNumber& num);
 - Phép gán nên luôn luôn trả về một tham chiếu tới đối tượng đích (đối tượng được gán trị cho)
 - Tham chiếu được trả về phải là **const** để tránh trường hợp **a** bị thay đổi bằng lệnh "**(a = b) = c;**" (lệnh đó không tương thích với định nghĩa gốc của phép gán)

Phép gán "="

```
const MyNumber& MyNumber::operator=(const MyNumber& num)
{
    if (this != &num) {
        this->value = num.value;
    }
    return *this;
}
```

- Định nghĩa trên có thể dùng cho phép gán
 - Lệnh **if** dùng để ngăn chặn các vấn đề có thể xảy sinh khi một đối tượng được gán cho chính nó (thí dụ khi sử dụng bộ nhớ động để lưu trữ các thành viên)
 - Ngay cả khi gán một đối tượng cho chính nó là an toàn, lệnh **if** trên đảm bảo không thực hiện các công việc thừa khi gán

Phép gán "="

- Khi nói về copy constructor, ta đã biết rằng C++ luôn cung cấp một copy constructor mặc định, nhưng nó chỉ thực hiện sao chép đơn giản (sao chép nông)
- Đối với phép gán cũng vậy → chỉ cần định nghĩa lại phép gán nếu:
 - Ta cần thực hiện phép gán giữa các đối tượng
 - Phép gán nông (memberwise assignment) không đủ dùng vì
 - Cần sao chép sâu - chẳng hạn sử dụng bộ nhớ động
 - Khi sao chép đòi hỏi cả tính toán - chẳng hạn gán một số hiệu có giá trị duy nhất hoặc tăng số đếm

Đa năng hoá toán tử ++ & --

- Toán tử ++ (hoặc toán tử --) có 2 loại:

- Tiên tố: ++n
 - Hậu tố: n++

(Hàm toán tử ở dạng hậu tố có thêm đối số giả kiểu int)

Đa năng hoá toán tử ++ & --

■ Phép tăng ++

- giá trị trả về

- tăng trước **++num**

- trả về tham chiếu (**MyNumber &**)
 - giá trị trái - lvalue (có thể được gán trị)

- tăng sau **num++**

- trả về giá trị (giá trị cũ trước khi tăng)
 - trả về đối tượng tạm thời chưa giá trị cũ.
 - giá trị phải - rvalue (không thể làm đích của phép gán)

- prototype

- tăng trước: **MyNumber& MyNumber::operator++()**

- tăng sau: **const MyNumber MyNumber::operator++(int)**

Đa năng hóa toán tử ++ & --

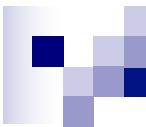
- Nhớ lại rằng phép tăng trước tăng giá trị trước khi trả kết quả, trong khi phép tăng sau trả lại giá trị trước khi tăng
- Ta định nghĩa từng phiên bản của phép tăng như sau:

```
MyNumber& MyNumber::operator++() { // Prefix
    this->value++;
    return *this;
}

const MyNumber MyNumber::operator++(int) { // Postfix
    MyNumber before(this->value); // Create temporary MyNumber
                                    // with current value
    this->value++;
    return before;
}
```

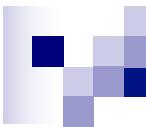
before là một đối tượng địa phương của phương thức và sẽ chấm dứt tồn tại khi lời gọi hàm kết thúc
Khi đó, tham chiếu tới nó trở thành bất hợp lệ

Không thể trả về tham chiếu



Tham số và kiểu trả về

- Cũng như khi overload các hàm khác, khi overload một toán tử, ta cũng có nhiều lựa chọn về việc truyền tham số và kiểu trả về
 - chỉ có hạn chế rằng ít nhất một trong các tham số phải thuộc kiểu người dùng tự định nghĩa
- Ở đây, ta có một số lời khuyên về các lựa chọn



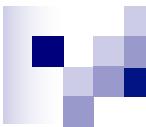
Tham số và kiểu trả về

■ Các toán hạng:

- Nên sử dụng tham chiếu mỗi khi có thể (đặc biệt là khi làm việc với các đối tượng lớn)
- Luôn luôn sử dụng tham số là hằng tham chiếu khi đối số sẽ không bị sửa đổi

```
bool String::operator==(const String &right) const
```

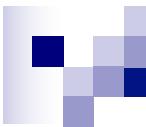
- Đối với các toán tử là phương thức, điều đó có nghĩa ta nên khai báo toán tử là hằng thành viên nếu toán hạng đầu tiên sẽ không bị sửa đổi
- Phần lớn các toán tử (tính toán và so sánh) không sửa đổi các toán hạng của nó, do đó ta sẽ rất hay dùng đến hằng tham chiếu



Tham số và kiểu trả về

■ Giá trị trả về

- không có hạn chế về kiểu trả về đối với toán tử được overload, nhưng nên cố gắng tuân theo tinh thần của các cài đặt có sẵn của toán tử
 - Ví dụ, các phép so sánh (==, !=...) thường trả về giá trị kiểu **bool**, nên các phiên bản overload cũng nên trả về **bool**
- là tham chiếu (tới đối tượng kết quả hoặc một trong các toán hạng) hay một vùng lưu trữ mới
- Hằng hay không phải hằng

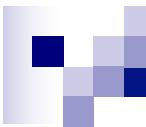


Tham số và kiểu trả về

■ Giá trị trả về ...

- Các toán tử sinh một giá trị mới cần có kết quả trả về là một giá trị (thay vì tham chiếu), và là const (để đảm bảo kết quả đó không thể bị sửa đổi như một l-value)
 - Hầu hết các phép toán số học đều sinh giá trị mới
 - ta đã thấy, các phép tăng sau, giảm sau tuân theo hướng dẫn trên
- Các toán tử trả về một tham chiếu tới đối tượng ban đầu (đã bị sửa đổi), chẳng hạn phép gán và phép tăng trước, nên trả về tham chiếu không phải là hằng
 - để kết quả có thể được tiếp tục sửa đổi tại các thao tác tiếp theo

```
const MyNumber MyNumber::operator+(const MyNumber& right) const  
MyNumber& MyNumber::operator+=(const MyNumber& right)
```



Tham số và kiểu trả về

- Xem lại cách ta đã dùng để trả về kết quả của toán tử:

```
const MyNumber MyNumber::operator+(const MyNumber& num)
{
    MyNumber result(this->value + num.value);
    return result;
}
```

1. Gọi constructor để tạo đối tượng **result**

2. Gọi copy-constructor để tạo bản sao dành cho giá trị trả về khi hàm thoát

3. Gọi destructor để huỷ đối tượng **result**

- Cách trên không sai, nhưng C++ cung cấp một cách hiệu quả hơn

```
const MyNumber MyNumber::operator+(const MyNumber& num)
{
    return MyNumber(this->value + num.value);
}
```

Tham số và kiểu trả về

```
return MyNumber(this->value + num.value);
```

- Cú pháp của ví dụ trước tạo một đối tượng tạm thời (temporary object)
- Khi trình biên dịch gặp đoạn mã này, nó hiểu đối tượng được tạo chỉ nhằm mục đích làm giá trị trả về, nên nó tạo thẳng một đối tượng bên ngoài (để trả về) - bỏ qua việc tạo và huỷ đối tượng bên trong lời gọi hàm
- Chỉ có một lời gọi duy nhất đến constructor của MyNumber (không phải copy-constructor) thay vì dãy lời gọi trước
- Quá trình này được gọi là tối ưu hoá giá trị trả về
- *Ghi nhớ: quá trình này không chỉ áp dụng được đối với các toán tử → sử dụng mỗi khi tạo một đối tượng chỉ để trả về*

Đa năng hóa new & delete

■ Hàm **new** và **delete** mặc định của ngôn ngữ:

- Nếu đối tượng kích thước nhỏ, có thể sẽ gây ra quá nhiều khối nhỏ => chậm.
- Không đáng kể khi đối tượng có kích thước lớn.

=> Toán tử new và delete ít được tái định nghĩa.

■ Có 2 cách đa năng hóa toán tử new và delete

- có thể đa năng hóa một cách toàn cục nghĩa là thay thế hẳn các toán tử **new** và **delete** mặc định.
- Đa năng hóa các toán tử **new** và **delete** với tư cách là hàm thành viên của lớp nếu muốn các toán tử **new** và **delete** áp dụng đối với lớp đó.
 - Khi chúng ta dùng **new** và **delete** đối với lớp nào đó, trình biên dịch sẽ kiểm tra xem **new** và **delete** có được định nghĩa riêng cho lớp đó hay không; nếu không thì dùng **new** và **delete** toàn cục (có thể đã được đa năng hóa).

Đa năng hóa new & delete

- Hàm toán tử của toán tử **new** và **delete** có prototype như sau:

void * operator new(size_t size);

void operator delete(void * ptr);

- Trong đó tham số kiểu **size_t** được trình biên dịch hiểu là kích thước của kiểu dữ liệu được trao cho toán tử **new**.

- **Ví dụ:**

- Đa năng hóa toán tử new và delete toàn cục
 - Đa năng hóa toán tử new và delete cho một lớp

Bài tập

■ Bài tập 4.1:

Bổ sung vào lớp **phân số** những phương thức sau:
(Nhóm tạo hủy)

- Khởi tạo mặc định phân số = 0.
- Khởi tạo với tử và mẫu cho trước.
- Khởi tạo từ giá trị nguyên cho trước.
- Khởi tạo từ một phân số khác.

(Nhóm toán tử)

- Toán tử số học: +, -, *, /, =, +=, -=.
- Toán tử so sánh: >, <, ==, >=, <=, !=.
- Toán tử một ngôi: ++, -- (tăng, giảm 1).
- Toán tử ép kiểu: (float), (int).
- Toán tử nhập, xuất: >>, <<.

Bài tập

■ Bài tập 4.2:

Bổ sung vào lớp **số phức** những phương thức sau:

(Nhóm tạo hủy)

- Khởi tạo mặc định số phức = 0.
- Khởi tạo với phần thực và phần ảo cho trước.
- Khởi tạo từ giá trị thực cho trước.
- Khởi tạo từ một số phức khác.

(Nhóm toán tử)

- Toán tử số học: +, -, *, /, =, +=, -=.
- Toán tử so sánh: >, <, ==, >=, <=, !=.
- Toán tử một ngôi: ++, -- (tăng, giảm 1).
- Toán tử ép kiểu: (float), (int).
- Toán tử nhập, xuất: >>, <<.

Bài tập

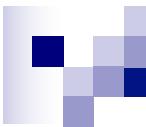
■ Bài tập 4.3:

Bổ sung vào lớp **đơn thức** những phương thức sau:
(Nhóm tạo hủy)

- Khởi tạo mặc định đơn thức = 0.
- Khởi tạo với hệ số và số mũ cho trước.
- Khởi tạo từ một đơn thức khác.

(Nhóm toán tử)

- Toán tử số học: +, -, *, /, =, +=, -=.
- Toán tử so sánh: >, <, ==, >=, <=, !=.
- Toán tử một ngôi:
 - ++, -- (tăng, giảm bậc).
 - ! (đạo hàm), ~ (nguyên hàm).
- Toán tử nhập, xuất: >>, <<.



Bài tập

■ Bài tập 4.4:

Bổ sung vào lớp **học sinh** những phương thức sau:

(Nhóm tạo hủy)

- Khởi tạo với họ tên và điểm văn, toán cho trước.
- Khởi tạo với họ tên cho trước, điểm văn, toán = 0.
- Khởi tạo từ một học sinh khác.
- Hủy đối tượng học sinh, thu hồi bộ nhớ.

(Nhóm toán tử)

- Toán tử so sánh (ĐTB): >, <, ==, >=, <=, !=.
- Toán tử gán: =.
- Toán tử nhập, xuất: >>, <<.

Bài tập

■ Bài tập 4.5:

Bổ sung vào lớp **mảng** những phương thức sau:

(Nhóm tạo hủy)

- Khởi tạo mặc định mảng kích thước = 0.
- Khởi tạo với kích thước cho trước, các phần tử = 0.
- Khởi tạo từ một mảng int [] với kích thước cho trước.
- Khởi tạo từ một đối tượng IntArray khác.
- Hủy đối tượng mảng, thu hồi bộ nhớ.

(Nhóm toán tử)

- Toán tử gán: =.
- Toán tử lấy phần tử: [].
- Toán tử ép kiểu: (int *).
- Toán tử nhập, xuất: >>, <<.