

# LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (C++)

Giảng viên:

Đặng Hoài Phương

Bộ môn:

Công nghệ phần mềm

Khoa:

Công nghệ Thông tin

Trường Đại học Bách Khoa

Đại học Đà Nẵng





# CHƯƠNG 3

## ĐA NĂNG HÓA

- Đa năng hóa hàm  $\leftrightarrow$  Nạp chồng phương thức (Overload);
- Là cách định nghĩa các hàm cùng tên nhưng khác nhau:
  - Kiểu trả về:
    - ❖ Các hàm cùng tên, cùng danh sách tham số chỉ khác nhau kiểu trả về của hàm thì không được phép.
  - Danh sách tham số:
    - Số lượng;
    - Thứ tự;
    - Kiểu dữ liệu.
  - ❖ Có thể sử dụng đối số mặc định.



# ĐA NĂNG HÓA HÀM

- Ví dụ:
  - `void HV(int a, int b);`
  - `void HV(int *a, int *b);`
  - `void HV(int &a, int &b);`Từng cặp hàm nào là đa năng hóa hàm.

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

Đơn hạng	+	-	*	!	~	&	++	--	()	->	->*
	new	delete									
Nhị 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.



- Khai báo và định nghĩa toán tử thực chất không khác với việc khai báo và định nghĩa một loại hàm bất kỳ nào khác
- Sử dụng tên hàm là “**operator@**” cho toán tử “**@**”;
  - Để overload toán tử “**+**”, 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 là hàm toàn cục hay là 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

- 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)”.



- Đa năng hóa toán tử **bằng hàm thành viên**:
  - Khi đa năng hóa **()**, **[]**, **->** hoặc **=**, hàm đa năng hóa toán tử phải được khai báo như một thành viên 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**:
  - $a @ b$ :  $a.operator @ (b)$
  - $x @ b$ : với  $x$  là thuộc kiểu float, int, ... không thuộc kiểu lớp đang định nghĩa
    - $Operator@ (x, b)$

- 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);  
};
```

- Toán tử là hàm toàn cục
  - Quay lại với 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.

- Toán tử là 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;

- Toán tử là 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;
}
```





# ĐA NĂNG HÓA TOÁN TỬ

- 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 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:  
 $\text{Complex} + \text{float}$  và  $\text{float} + \text{Complex}$
    - `cout << obj;`
    - Không thể sửa định nghĩa kiểu `int` hay kiểu của `cout`;
    - Lựa chọn duy nhất: overload toán tử bằng hàm toàn cục.



# ĐA NĂNG HÓA TOÁN TỬ SỐ HỌC (+, -, \*, /)

- Khai báo lớp Point:

```
#include <iostream>
using namespace std;
class Point
{
    int xval, yval;
public:
    Point(int = 1, int = 1);
    ~Point();
    void Show();
    friend Point operator+(Point, Point);
    Point operator+(Point);
};
```



# ĐA NĂNG HÓA TOÁN TỬ SỐ HỌC (+, -, \*, /)

- Constructor, Destructor & Show():

```
Point::Point(int xval, int yval)
{
    this->xval = xval;
    this->yval = yval;
}
Point::~~Point()
{ }
void Point::Show()
{
    cout << "Point(" << this->xval << ", "
         << this->yval << ")" << endl;
}
```



# ĐA NĂNG HÓA TOÁN TỬ SỐ HỌC (+, -, \*, /)

- Đa năng hóa toán tử +:

```
Point Point::operator+(Point p)
{
    return Point(this->xval + p.xval, this->yval + p.yval);
}
Point operator+(Point p1, Point p2)
{
    return Point(p1.xval + p2.xval, p1.yval + p2.yval);
}
int main()
{
    Point p1;
    Point p2(1, 2);
    Point p3 = p1 + p2;
    p3.Show();
    system("pause");
    return 0;
}
```



# ĐA NĂNG HÓA TOÁN TỬ SỐ HỌC (+, -, \*, /)

- Lưu ý:
  - Truyền tham chiếu hay tham trị, có const hay không?

```
friend Point operator+(Point, Point);  
friend Point operator+(const Point, const Point);  
friend Point operator+(Point&, Point&);  
friend Point operator+(const Point&, const Point&);
```



# ĐA NĂNG HÓA 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 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 HÓA 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 << ") ";
    return out;
};
```



# ĐA NĂNG HÓA 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 HÓA TOÁN TỬ NHẬP >>

- Khai báo toán tử đọc 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<<"Nhap phan thuc:"; in >> num.R;
    cout<<"Nhap phan ao:"; in >> num.I;
    return in;
};
```



# ĐA NĂNG HÓA TOÁN TỬ >>, <<

- Khai báo lớp Point:

```
#include <iostream>
using namespace std;
class Point
{
    int xval, yval;
public:
    Point(int = 1, int = 1);
    ~Point();
    friend ostream& operator<<(ostream&, const Point&);
    friend istream& operator>>(istream&, Point&);
};
```



- Constructor & Destructor

```
Point::Point(int xval, int yval)
{
    this->xval = xval;
    this->yval = yval;
}
Point::~~Point()
{ }
```



# ĐA NĂNG HÓA TOÁN TỬ >>, <<

- Đa năng hóa toán tử >>, <<:

```
ostream& operator<<(ostream& o, const Point& p)
{
    o << "Point(" << p.xval << ", "
        << p.yval << ")" << endl;
    return o;
}

istream& operator>>(istream& i, Point& p)
{
    cout << "Input xval = ";
    i >> p.xval;
    cout << "Input yval = ";
    i >> p.yval;
    return i;
}
```





# ĐA NĂNG HÓA TOÁN TỬ >>, <<

- Đa năng hóa toán tử >>, <<:

```
int main()
{
    Point p;
    cin >> p;
    cout << p;
    system("pause");
    return 0;
}
```



# ĐA NĂNG HÓA TOÁN TỬ >>, <<

- Lưu ý:
  - Kiểu trả về của hàm đa năng hóa: **ostream&**, **istream&**; nhưng nếu **void** thì sao?
  - Thứ tự đối số trong hàm đa năng hóa;
  - Truyền dữ liệu cho đối số: tham chiếu, tham trị.

```
friend ostream& operator<<(ostream&, const Point&);  
friend ostream& operator<<(ostream&, Point);  
friend ostream& operator<<(ostream&, Point&);  
friend void operator>>(Point&, istream&);  
friend void operator<<(Point, ostream&);  
friend void operator>>(istream&, Point&);  
friend void operator<<(ostream&, Point);
```

- 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.

❖ Lưu ý:

```
int& operator[](const int&);  
int& operator[](const int);  
int& operator[](int);  
int& operator[](int &);
```



- Khai báo lớp Vector

```
#include <iostream>
#include <iomanip>
using namespace std;
class Vector
{
    int size;
    int *data;
public:
    Vector(int = 2, int = 1);
    ~Vector();
    friend ostream& operator<<(ostream&, const Vector&);
    int& operator[](const int);
};
```

- Constructor (cấp phát động) & Destructor

```
Vector::Vector(int size, int n)
{
    this->size = size;
    this->data = new int[this->size];
    for (int i = 0; i < this->size; i++)
        *(this->data + i) = n;
        /* ((*this).data + i) = n;
}

Vector::~~Vector()
{
    delete[] this->data;
}
```

- Đa năng hóa toán tử [] và <<:

```
ostream& operator<<(ostream& o, const Vector& v)
{
    for (int i = 0; i < v.size; i++)
        o << setw(3) << *(v.data + i);
    return o;
}

int& Vector::operator[](const int i)
{
    static int temp = 0;
    return (i >= 0 && i < this->size) ?
        *(this->data + i) : temp;
}
```



- Đa năng hóa toán tử []:

```
int main()
{
    Vector v;
    v[0] = 5;
    cout << v;
    system("pause");
    return 0;
}
```



# ĐA NĂNG HÓA TOÁN TỬ 0

- Định nghĩa là hàm thành viên.

```
#include <iostream>
#include <iomanip>
using namespace std;
class Matrix
{
    int row, col;
    int **data;
public:
    Matrix(int = 2, int = 3, int = 1);
    ~Matrix();
    friend ostream& operator<<(ostream&, const Matrix&);
    int& operator()(const int, const int);
};
```



# ĐA NĂNG HÓA TOÁN TỬ 0

- Định nghĩa là hàm thành viên.

```
#include <iostream>
#include <iomanip>
using namespace std;
class Matrix
{
    int row, col;
    int **data;
public:
    Matrix(int = 2, int = 3, int = 1);
    ~Matrix();
    friend ostream& operator<<(ostream&, const Matrix&);
    int& operator()(const int, const int);
};
```

- Hàm dựng (cấp phát động)

```
Matrix::Matrix(int row, int col, int n)
{
    this->row = row;
    this->col = col;
    this->data = new int*[this->row];
    for (int i = 0; i < this->row; i++)
    {
        *(this->data + i) = new int[this->col];
        /* ((*this).data + i) = new int[this->col];
        for (int j = 0; j < this->col; j++)
            *((this->data + i) + j) = n;
        //this->data[i] = n;
    }
}
```

- Hàm dựng (cấp phát động)

```
Matrix::Matrix(int row, int col, int n)
{
    this->row = row;
    this->col = col;
    this->data = new int*[this->row];
    int *temp = new int[this->row * this->col];
    for (int i = 0; i < this->row; i++)
    {
        *(this->data + i) = temp;
        temp += this->col;
        for (int j = 0; j < this->col; j++)
            *(*this->data + i) + j) = n;
    }
}
```

- Đa năng hóa toán tử () và <<:

```
int& Matrix::operator()(const int i, const int j)
{
    static int temp = 0;
    return (i >= 0 && i < this->row && j >= 0 && j < this->col) ?
        (*(this->data + i) + j) : temp;
}

ostream& operator<<(ostream& o, const Matrix& m)
{
    for (int i = 0; i < m.row; i++)
    {
        for (int j = 0; j < m.col; j++)
            o << setw(3) << (*(m.data + i) + j);
        cout << endl;
    }
    return o;
}
```





# ĐA NĂNG HÓA TOÁN TỬ ()

- Đa năng hóa toán tử () và <<:
  - Lưu ý, toán tử () sẽ được sử dụng ở đâu với 2 tham số **const int**.

```
int main()
{
    Matrix m;
    m(1, 1) = 5;
    cout << m;
    system("pause");
    return 0;
}
```

- 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 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 ⇔ Point(5)



# CHUYỂN KIỂU

- Ví dụ: (nếu dùng hàm dựng với đối số mặc định thì chuyện gì sẽ xảy ra)

```
#include <iostream>
using namespace std;
class Point
{
    int xval, yval;
public:
    Point(int, int);    //Point(int = 1, int = 1)
    Point(int);
    ~Point();
    friend ostream& operator<<(ostream&, const Point&);
    friend istream& operator>>(istream&, Point&);
    friend Point operator+(Point, Point);
};
```

- Ví dụ: Constructor & Destructor

```
Point::Point(int xval, int yval)
{
    this->xval = xval;
    this->yval = yval;
}
Point::Point(int xval)
{
    this->xval = xval;
    this->yval = xval;
}
Point::~~Point()
{ }
```

- Ví dụ: Đa năng hóa toán tử >>, <<:

```
ostream& operator<<(ostream& o, const Point& p)
{
    o << "Point(" << p.xval << ", "
        << p.yval << ")" << endl;
    return o;
}

istream& operator>>(istream& i, Point& p)
{
    cout << "Input xval = ";
    i >> p.xval;
    cout << "Input yval = ";
    i >> p.yval;
    return i;
}
```



- Ví dụ: Đa năng hóa toán tử + cho 2 đối tượng Point

```
Point operator+(Point p1, Point p2)
{
    return Point(p1.xval + p2.xval, p1.yval + p2.yval);
}

int main()
{
    Point p1(1, 2);
    Point p2 = p1 + 2;
    cout << p2;
    system("pause");
    return 0;
}
```

- 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.
- Toán tử chuyển đổi kiểu như thế phải là hàm thành viên không tĩnh và không là hàm **friend**.
- Prototype của hàm thành viên này có cú pháp:
  - **operator <data type> ();**

- 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;
}
```

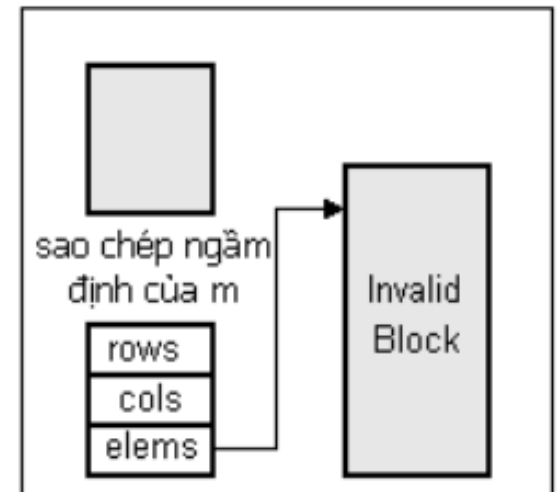
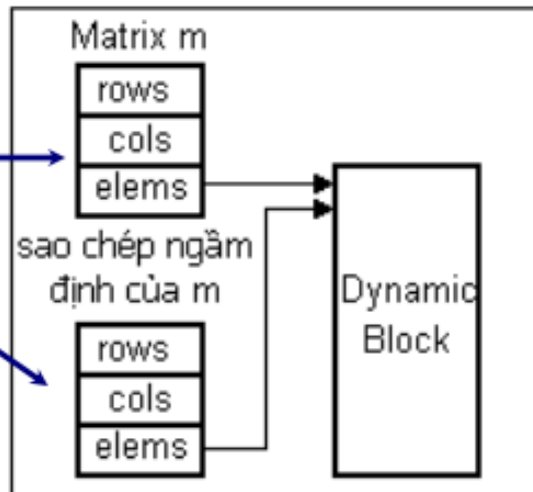
# KHỞI TẠO NGẦM ĐỊNH

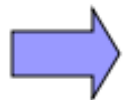
- Được định nghĩa sẵn trong ngôn ngữ:
  - Ví dụ: Point p1(10,20); Point p2 = p1;
- Sẽ gây ra lỗi (kết quả SAI) khi bên trong đối tượng có **thành phần dữ liệu là con trỏ**:
  - Ví dụ: Matrix m(5,6); Matrix n = m;

sao chép ngầm định của m được tạo

sau khi m bị hủy

Lỗi sẽ xảy ra do khởi tạo ngầm bằng cách **gán tương ứng từng thành phần**.





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];  
        for (register i = 0; i < n; ++i) // cùng kích thước  
            elems[i] = m.elems[i]; // sao chép phần tử  
    }
```

- Được định nghĩa sẵn trong ngôn ngữ:
  - Gán tương ứng từng thành phần;
  - Đúng khi đối tượng không có dữ liệu con trỏ:
    - Ví dụ: `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.





# 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 → 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 nảy 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:
  - 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.



# PHÉP GÁN =

- Ví dụ:

```
#include <iostream>
using namespace std;
class Vector
{
    int size;
    int *data;
public:
    Vector(int = 2, int = 1);
    ~Vector();
    friend ostream& operator<<(ostream&, const Vector&);
    const Vector& operator=(const Vector&);
    int& operator[](const int);
};
```



# PHÉP GÁN =

- Ví dụ:

```
const Vector& Vector::operator=(const Vector& v)
{
    if (this->size == v.size)
    {
        for (int i = 0; i < this->size; i++)
            *(this->data + i) = *(v.data + i);
    }
    return *this;
}
```



# ĐA NĂNG HÓA TOÁN TỬ ++ & --

- Toán tử ++ (hoặc toán tử --) có 2 loại:
  - Tiền tố: ++**n**;
  - Hậu tố: **n**++ (Hàm toán tử ở dạng hậu tố có thêm đối số giả kiểu int).



# ĐA NĂNG HÓA 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 chứa 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 ++ & --

- Phép tăng trước tăng giá trị trước khi trả kết quả, trong khi phép tăng sau trả lại giá trị trước khi tăng;
- Định nghĩa từng phiên bản của phép tăng như sau:

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

**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 & 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
- 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.

- 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 & 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;
    - 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 & KIỂU TRẢ VỀ

- Cách 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**

- C++ cung cấp một cách hiệu quả hơn:

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



# THAM SỐ & 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;
- Vậy, 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 rằng quá trình này không chỉ áp dụng được đối với các toán tử. Ta nên sử dụng mỗi khi tạo một đối tượng chỉ để trả về.





# ĐA NĂNG HÓA TOÁN TỬ 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 TOÁN TỬ 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.



# Thank You !

