

# CHƯƠNG 3. LỚP VÀ ĐỐI TƯỢNG

Visual C++

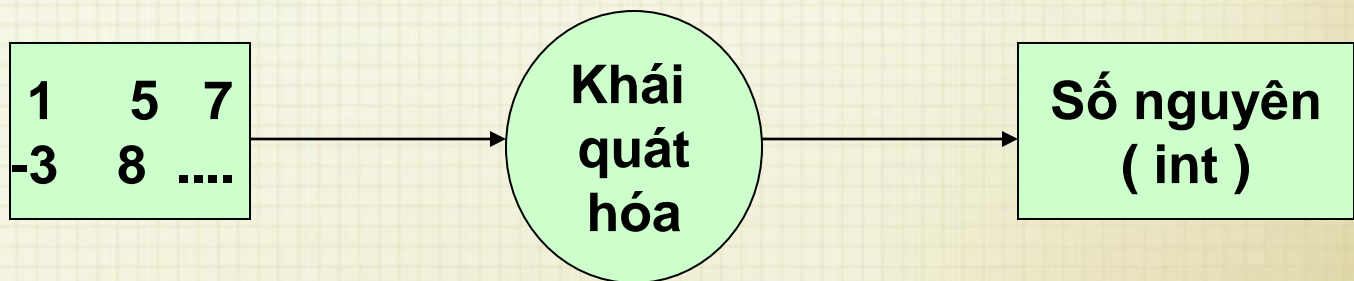


# Giới thiệu

- ❖ Một lớp bao gồm các thành phần **dữ liệu** (**thuộc tính**) và các **phương thức** (hàm thành phần).
- ❖ Lớp trong C++ thực chất là một kiểu dữ liệu do người sử dụng định nghĩa.
- ❖ Trong C++, dùng từ khóa **class** để chỉ điểm bắt đầu của một lớp sẽ được cài đặt.

# Lớp đối tượng - class

- ❖ Lớp là một mô tả trừu tượng của nhóm các đối tượng cùng bản chất, ngược lại mỗi một đối tượng là một **thể hiện** cụ thể cho những mô tả trừu tượng đó.
- ❖ Lớp là cái ta thiết kế và lập trình
- ❖ Đối tượng là cái ta tạo (từ một lớp) tại thời gian chạy.





# Nội dung

1. Giới thiệu
2. Khai báo lớp
3. Các thành phần của lớp
4. Cơ chế tạo lập các lớp
5. Định nghĩa hàm thành phần
6. Tạo lập đối tượng
7. Phạm vi truy xuất
8. Hàm thiết lập – Constructor
9. Hàm hủy bỏ – Destructor
10. Hàm bạn, lớp bạn
11. Các phương thức Truy vấn, Cập nhật
12. Thành viên tĩnh – static member
13. Các nguyên tắc xây dựng lớp

# Khai báo lớp

```
class <tên_lớp>
```

```
{
```

```
//Thành phần dữ liệu
```

```
//Thành phần xử lý
```

```
};
```

# Khai báo lớp

```
class <tên_lớp> {
```

```
private:
```

<khai báo thành phần riêng trong từng đối tượng>

```
protected:
```

<khai báo thành phần riêng trong từng đối tượng,  
có thể truy cập từ lớp dẫn xuất >

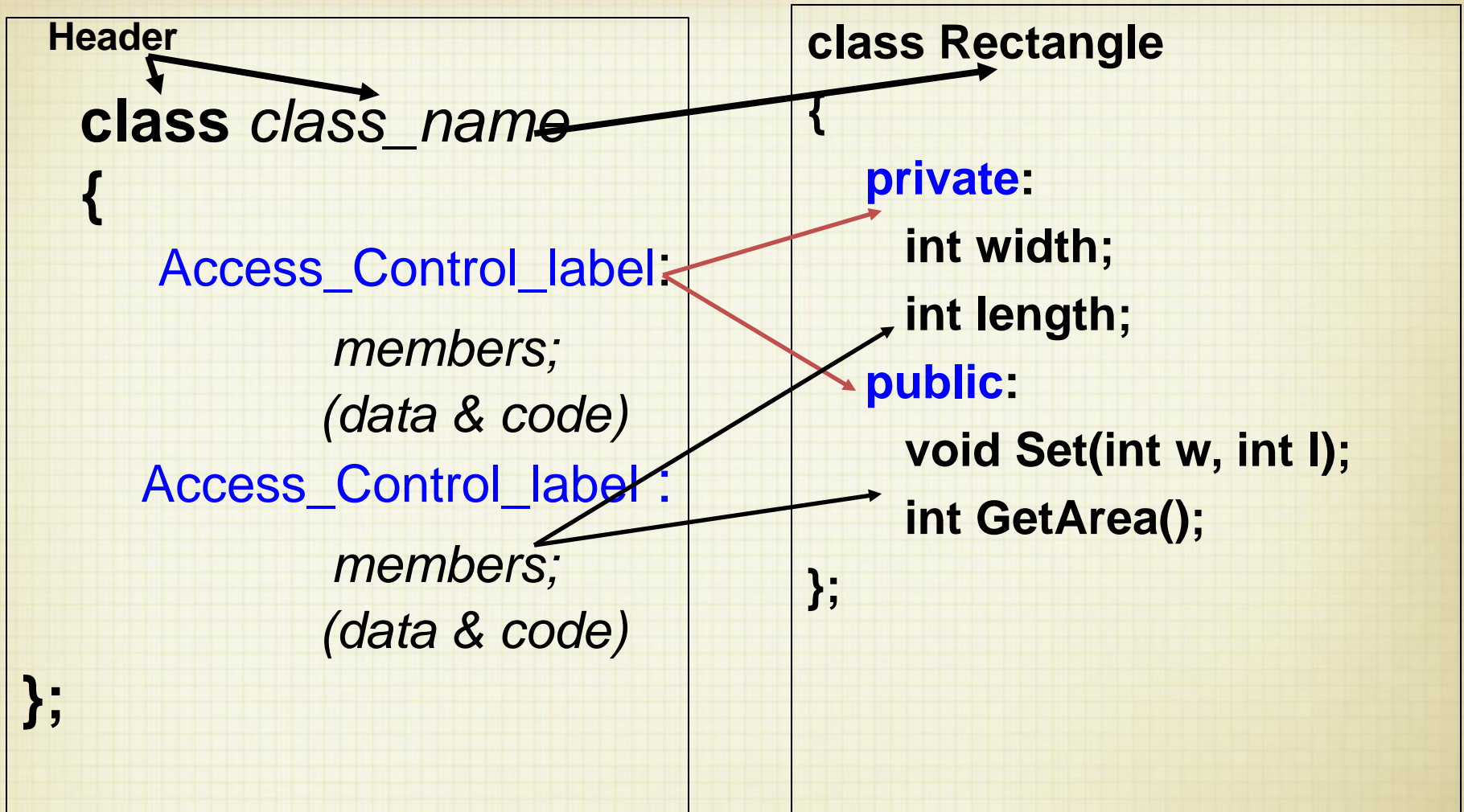
```
public:
```

<khai báo thành phần công cộng>

```
};
```



# Khai báo lớp



# Các thành phần của lớp

- ❖ **Thuộc tính:** Các thuộc tính được khai báo giống như khai báo biến trong C
- ❖ **Phương thức:** Các phương thức được khai báo giống như khai báo hàm trong C. Có hai cách định nghĩa thi hành của một phương thức
  - ❖ Định nghĩa thi hành trong lớp
  - ❖ Định nghĩa thi hành ngoài lớp



# Cơ chế tạo lập các lớp

- ❖ Xác định các thuộc tính (dữ liệu)
  - Những gì mà ta biết về đối tượng – giống như một struct
- ❖ Xác định các phương thức (hành vi)
  - Những gì mà đối tượng có thể làm
- ❖ Xác định các quyền truy xuất
  - Sẽ trình bày sau

# Định nghĩa hàm thành phần

❖ Định nghĩa các hàm thành phần ở bên ngoài khai báo lớp:

<tên kiểu giá trị trả về> <tên lớp>::<tên hàm> (<danh sách tham số>)

```
{  
    <nội dung >  
}
```

Ví dụ:

```
void Point::Xuat() {  
    //.....  
}
```

# Định nghĩa hàm thành phần

```
class Rectangle{  
    private:  
        int width, length;  
    public:  
        void set (int w, int l);  
        int area() { return width*length; }  
};
```

inline

```
r1.set(5,8);  
rp->set(8,10);
```

```
void Rectangle :: set (int w, int l)  
{  
    width = w;  
    length = l;  
}
```

class name

member function name

scope operator



# Tạo lập đối tượng

❖ Khai báo và tạo đối tượng:

<tên lớp> <tên đối tượng>;

❖ Gọi hàm thành phần của lớp

<tên đối tượng>.<tên hàm thành phần> (<danh sách các tham số nếu có>);

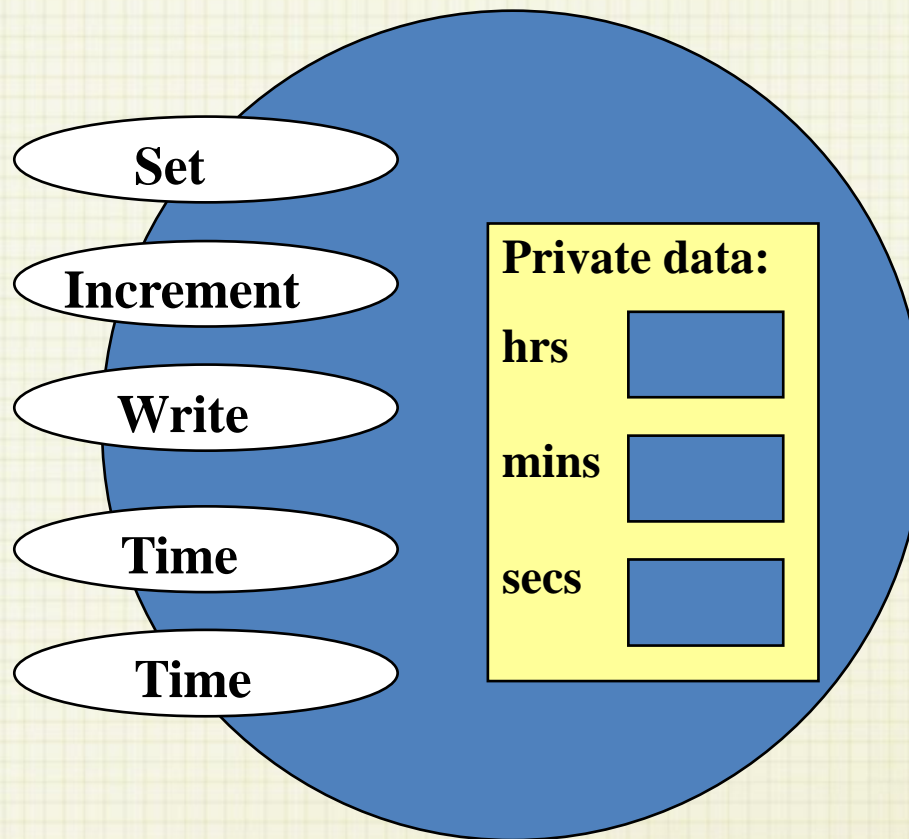
<tên con trỏ đối tượng>→<tên hàm thành phần> (<danh sách các tham số nếu có>);

# Ví dụ: Class Time

```
class Time {  
    public:  
        void Set (int hours , int minutes , int seconds);  
        void Increment ( );  
        void Write ( ) const;  
        Time (int initHrs, int initMins, int initSecs ); //constructor  
        Time ( ); //default constructor  
    private:  
        int      hrs;  
        int      mins;  
        int      secs;  
};
```

# Sơ đồ mô tả lớp Time

## Time class





# Khai báo đối tượng

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

**r1 is statically allocated**

```
main()
{
    Rectangle r1;
    ➔ r1.set(5, 8);
}
```

**r1**

<b>width = 5</b> <b>length = 8</b>
---------------------------------------

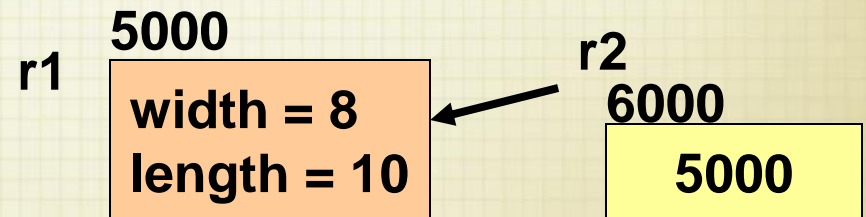
# Khai báo đối tượng

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r2 is a Pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);    //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);  //arrow notation
}
```



# Khai báo đối tượng

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r3 is dynamically allocated

```
main()
{
    Rectangle *r3;
    r3 = new Rectangle();

    r3->set(80,100); //arrow notation

    delete r3;
    ➡ r3 = NULL;
}
```

r3      6000  
[ NULL ]



# Ví dụ

## ❖ Xây dựng lớp **Điểm (Point)** trong hình học 2D

- Thuộc tính
  - Tung độ (float)
  - Hoành độ (float)
- Thao tác (phương thức)
  - Nhập, Xuất
  - Di chuyển theo vector (m,n)
  - Tính khoảng cách với một điểm

Viết chương trình nhập vào 2 điểm, xuất ra 2 điểm vừa nhập, khoảng cách giữa 2 điểm đó và tọa độ mới của 2 điểm khi tịnh tiến theo vector có giá trị do người dùng nhập vào.

# Ví dụ

```
/*Point.h*/  
#include <iostream.h>  
using namespace std;  
class Point {  
    /*khai báo các thành phần dữ liệu riêng*/  
    private:  
        int x,y;  
    /*khai báo các hàm thành phần công cộng*/  
    public:  
        void KhoiTao(int ox, int oy);  
        void DiChuyen(int dx, int dy);  
        void Xuat();  
};
```

# Ví dụ

```
/*Point.cpp*/
void Point::KhoiTao(int ox, int oy) {
    cout<<"Ham thanh phan khoi tao gia tri\n";
    x = ox; y = oy;
    /*x,y là các thành phần của đối tượng gọi hàm thành phần*/
}
void Point::DiChuyen(int dx, int dy) {
    cout<<"Ham thanh phan di chuyen\n";
    x += dx; y += dy;
}
void Point::Xuat() {
    cout<<"Ham thanh phan xuat\n";
    cout<<"Toa do: "<<x<<","<<y<<"\n";
}
```



# Ví dụ

```
void main() {  
    Point p;  
    p.KhoiTao(2,4); /*gọi hàm thành phần từ đối tượng*/  
    p.Xuat();  
    p.DiChuyen(1,2);  
    p.Xuat();  
}
```

Hàm thành phần khoi tao

Hàm thành phần xuất

Toa do: 2,4

Hàm thành phần di chuyển

Hàm thành phần xuất

Toa do: 3,6

# Phạm vi truy xuất

- ❖ Trong định nghĩa của lớp ta có thể xác định **khả năng truy xuất thành phần** của một lớp nào đó từ bên ngoài phạm vi lớp.
- ❖ **private**, **protected** và **public** là các **từ khoá** xác định phạm vi truy xuất
- ❖ Mọi thành phần được liệt kê trong phần **public** đều có thể truy xuất trong **bất kỳ** hàm nào.
- ❖ Những thành phần được liệt kê trong phần **private** chỉ được truy xuất **bên trong phạm vi lớp (và hàm bạn, lớp bạn)**.

# Phạm vi truy xuất

- ❖ Trong lớp có thể có nhiều nhãn `private` và `public`
- ❖ Mỗi nhãn này có phạm vi ảnh hưởng cho đến khi gặp một nhãn kế tiếp hoặc hết khai báo lớp.
- ❖ Nhãn `private` đầu tiên có thể bỏ qua vì C++ ngầm hiểu rằng các thành phần trước nhãn `public` đầu tiên là `private`.



# Phạm vi truy xuất – Ví dụ

```
class TamGiac{  
    private:  
        float a,b,c;/*độ dài ba cạnh*/  
    public:  
        void Nhap();/*nhập vào độ dài ba cạnh*/  
        void Xuat();/*in ra các thông tin liên quan đến tam giác*/  
    private:  
        int LayLoai();//cho biết kiểu của tam giác: 1-d,2-v,3-c,4-v,5-t  
        float TinhDienTich();/*tính diện tích của tam giác*/  
};
```

# Phạm vi truy xuất – Ví dụ

```
class TamGiac{  
    private:  
        float a,b,c; /*độ dài ba cạnh*/  
        int LayLoai(); //cho biết kiểu của tam giác: 1-d,2-vc,3-  
            c,4-v,5-t  
        float TinhDienTich(); /*tính diện tích của tam giác*/  
    public:  
        void Nhap(); /*nhập vào độ dài ba cạnh*/  
        void Xuat(); /*in các thông tin liên quan đến tam giác*/  
};
```

# Tham số hàm thành phần

```
void Point::KhoiTao (int xx, int yy){  
    x = xx;  
    y = yy; //x, y la thanh phan cua lop Point  
}
```

❖ **Hàm thành phần** có quyền truy nhập đến các thành phần **private** của đối tượng gọi nó



# Tham số hàm thành phần

- ❖ Hàm thành phần có quyền truy cập đến tất cả các thành phần `private` của các đối tượng, tham chiếu đối tượng hay con trỏ đối tượng có cùng kiểu lớp khi được dùng là tham số hình thức của nó.

# Tham số hàm thành phần

```
int KiemTraTrung(Point pt){  
    return (x==pt.x && y==pt.y);  
}  
  
int KiemTraTrung(Point *pt){  
    return (x==pt→x && y==pt→y);  
}  
  
int KiemTraTrung(Point &pt) {  
    return (x==pt.x && y==pt.y);  
}
```



# Con trỏ this

- ❖ Từ khoá **this** trong định nghĩa của các hàm thành phần lớp dùng để xác định địa chỉ của đối tượng dùng làm **tham số ngầm định** cho hàm thành phần.
- ❖ Con trỏ **this** tham chiếu đến đối tượng đang gọi hàm thành phần.
- ❖ Ví dụ:

```
int KiemTraTrung(Point pt){  
    return (this → x == pt.x && this → y == pt.y);  
}
```



# Phép gán đối tượng

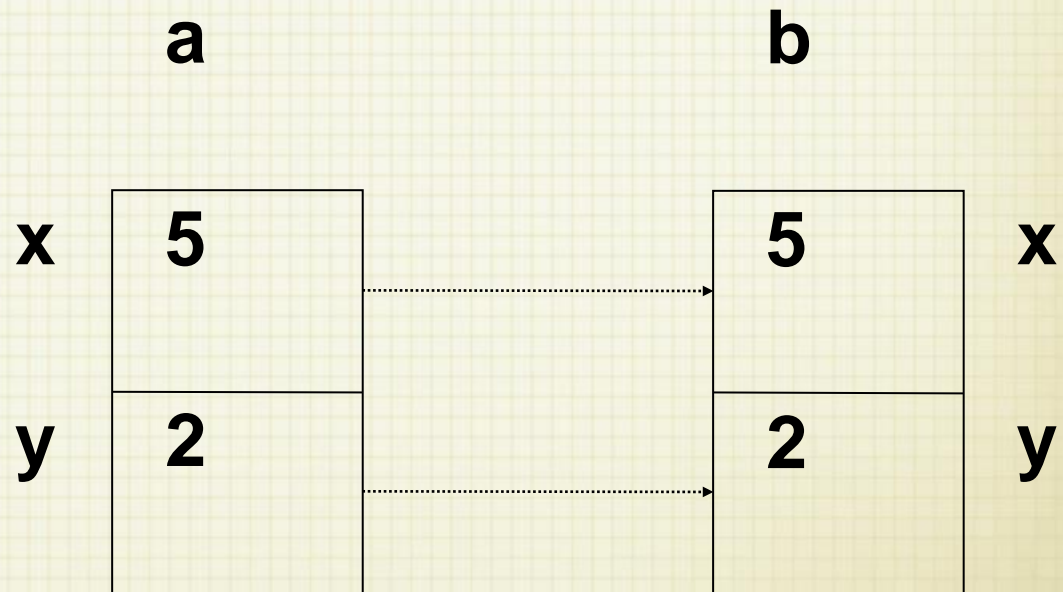
❖ Là việc sao chép giá trị các thành phần dữ liệu từ đối tượng **a** sang đối tượng **b** tương ứng từng đối một

❖ Ví dụ:

Point a, b;

a.KhoiTao(5,2);

b = a;



# Hàm thiết lập – Constructor

- ❖ Trong hầu hết các thuật giải, để giải quyết một vấn đề → thường phải thực hiện các công việc:
  - Khởi tạo giá trị cho biến, cấp phát vùng bộ nhớ của biến con trỏ, mở tập tin để truy cập,...
  - Hoặc khi kết thúc, chúng ta phải thực hiện quá trình ngược lại như: Thu hồi vùng bộ nhớ đã cấp phát, đóng tập tin,...
- ❖ Các ngôn ngữ OOP có các phương thức để thực hiện công việc này một cách “*tự động*” gọi là *phương thức thiết lập* và *phương thức hủy bỏ*.

# Hàm thiết lập – Constructor

- ❖ Constructor là một loại phương thức đặc biệt dùng để khởi tạo thể hiện của lớp.
- ❖ Bất kỳ một đối tượng nào được khai báo đều phải sử dụng một hàm thiết lập để khởi tạo các giá trị thành phần của đối tượng.
- ❖ Hàm thiết lập được khai báo giống như một phương thức với tên phương thức trùng với tên lớp và không có giá trị trả về (kể cả void).
- ❖ Constructor phải có phạm vi là public



# Hàm thiết lập – Constructor

- ❖ Constructor có thể được khai báo chồng như các hàm C++ thông thường
- ❖ Constructor có thể được khai báo với các tham số có giá trị ngầm định (tham số mặc nhiên)



# Ví dụ

```
class Point{  
    /*Khai báo các thành phần dữ liệu*/  
    int x, y;  
    public:  
        Point() { x = 0; y = 0; } /*Hàm thiết lập mặc định*/  
        Point(int ox, int oy) { x = ox; y = oy; } /*Gán giá trị*/  
        void DiChuyen(int dx, int dy);  
        void Xuat();  
};  
Point a(5,2);    //ok?  
Point b;         //ok?  
Point c(3);      //ok?
```

# Ví dụ

```
class Point{  
    /*Khai báo các thành phần dữ liệu*/  
    int x, y;  
    public:  
        Point() { x = 0; y = 0; } /*Hàm thiết lập mặc định*/  
        Point(int ox, int oy = 1){ x = ox; y = oy;} /*Hàm thiết lập*/  
        void DiChuyen(int dx, int dy);  
        void Xuat();  
};  
Point a(5,2);    //ok?  
Point b;         //ok?  
Point c(3);      //ok?
```

# Constructor mặc định

- ❖ Constructor mặc định (default constructor) là constructor được gọi khi thể hiện được khai báo mà không có đối số nào được cung cấp
  - `MyClass x;`
  - `MyClass* p = new MyClass();`
- ❖ Ngược lại, nếu tham số được cung cấp tại khai báo thể hiện, trình biên dịch sẽ gọi constructor khác (overload)
  - `MyClass x(5);`
  - `MyClass* p = new MyClass(5);`

# Constructor mặc định

- ❖ Đối với constructor mặc định, nếu ta không cung cấp bất kỳ constructor nào, C++ sẽ tự sinh constructor mặc định là một phương thức rỗng.
- ❖ Tuy nhiên, nếu ta không định nghĩa constructor mặc định nhưng lại có các constructor khác, trình biên dịch sẽ báo lỗi không tìm thấy constructor mặc định nếu ta không cung cấp tham số khi tạo thể hiện.



# Ví dụ

```
class Point{  
    /*Khai báo các thành phần dữ liệu*/  
    int x, y;  
    public:  
        Point(int ox, int oy = 1){ x = ox; y = oy;}  
        void DiChuyen(int dx, int dy);  
        void Xuat();  
};  
  
Point a(5,2);    //ok?  
Point b;         //ok?  
Point c(3);      //ok?
```

# Copy constructor

- ❖ Chúng ta có thể **tạo đối tượng mới giống đối tượng cũ** một số đặc điểm, không phải hoàn toàn như phép gán bình thường, hình thức “giống nhau” được định nghĩa theo quan niệm của người lập trình. Để làm được vấn đề này, trong các ngôn ngữ OOP cho phép ta xây dựng **phương thức thiết lập sao chép**.
- ❖ Đây là phương thức thiết lập có tham số là tham chiếu đến đối tượng thuộc chính lớp này.
- ❖ Trong phương thức thiết lập sao chép có thể ta chỉ sử dụng một số thành phần nào đó của đối tượng ta tham chiếu → “gần giống nhau”

# Copy constructor

```
Foo(const Foo& existingFoo);
```

tham số là đối tượng  
được sao chép

Kiểu tham số là tham chiếu  
đến đối tượng kiểu Foo

từ khoá const được dùng để đảm bảo đối  
tượng được sao chép sẽ không bị sửa đổi

# Hàm hủy bỏ – Destructor

- ❖ Destructor, được gọi ngay trước khi một đối tượng bị thu hồi.
- ❖ Destructor thường được dùng để thực hiện việc dọn dẹp cần thiết trước khi một đối tượng bị hủy.
- ❖ Một lớp chỉ có duy nhất một Destructor
- ❖ Phương thức Destructor trùng tên với tên lớp nhưng có dấu ~ đặt trước
- ❖ Được tự động gọi thực hiện khi đối tượng hết phạm vi sử dụng.
- ❖ Destructor phải có thuộc tính public



# Ví dụ

```
class Vector{  
    int n;           //số chiều  
    float *v;        //vùng nhớ tọa độ  
public:  
    Vector();         //Hàm thiết lập không tham số  
    Vector(int size); //Hàm thiết lập một tham số  
    Vector(int size, float *a);  
    ~Vector();        //Hàm hủy bỏ  
    void Xuat();  
};
```

# Hàm bạn, lớp bạn

- ❖ Giả sử có lớp Vector, lớp Matrix
  - ❖ Cần viết hàm nhân Vector với một Matrix
  - ❖ Hàm nhân:
    - Không thể thuộc lớp Vector
    - Không thể thuộc lớp Matrix
    - Không thể tự do
- Giải pháp: Xây dựng hàm truy cập dữ liệu?

# Hàm bạn (Friend function)

- ❖ Hàm bạn không thuộc lớp. Tuy nhiên, có quyền truy cập các thành viên `private` của lớp.
- ❖ Khi định nghĩa một lớp, có thể khai báo một hay nhiều hàm “bạn”
- ❖ Ưu điểm:
  - Kiểm soát các truy nhập ở cấp độ lớp – không thể áp đặt hàm bạn cho lớp nếu điều đó không được dự trù trước trong khai báo của lớp.

# Hàm bạn (Friend function)

- ❖ Dùng từ khóa **friend** để khai báo, định nghĩa hàm bạn
- ❖ Đây là cách cho phép chia sẻ dữ liệu giữa các đối tượng với một hàm tùy ý trong chương trình (**hàm friend**) hoặc chia sẻ các thành phần của đối tượng có thuộc tính **private** hay **protected** với các đối tượng khác (**lớp friend**).



# Hàm bạn (Friend function)

❖ Các tính chất của quan hệ **friend**:

- Phải được cho, không được nhận
  - Lớp B là bạn của lớp A, lớp A phải khai báo rõ ràng B là bạn của nó
- Không đối xứng, Không bắc cầu
- Quan hệ friend có vẻ như vi phạm khái niệm đóng gói (encapsulation) của OOP nhưng có khi lại cần đến nó để cài đặt các mối quan hệ giữa các lớp và khả năng **đa năng hóa toán tử** trên lớp (sẽ đề cập ở chương sau)

# Ví dụ

```
class CounterClass{
    int Counter;
public:
    char CounterChar;
    void Init( char );
    void AddOne( ){
        Counter++;
    }
    friend int Total (int);
};
```

# Ví dụ

```
CounterClass MyCounter[26];    //Có 26 đối tượng
int Total(int NumberObjects)
{
    for (int i=0, sum=0; i<NumberObjects; i++)
        sum += MyCounter[i].Counter
        //Tính tổng số ký tự trong số các Objects ký tự
    return sum;
}
```

# Hàm bạn (Friend function)

## ❖ Lưu ý:

- Vị trí của khai báo “bạn bè” trong lớp hoàn toàn tùy ý
- Trong hàm bạn, không còn **tham số ngầm định this** như trong hàm thành phần
- Hàm bạn của một lớp có thể có một hay nhiều tham số, hoặc có thể có giá trị trả về thuộc kiểu lớp đó



# Lớp bạn (Friend class)

- ❖ Một lớp **có thể** truy cập đến các thành phần có thuộc tính **private** của một lớp khác.
- ❖ Để thực hiện được điều này, chúng ta có thể lấy toàn bộ một lớp làm bạn (**lớp friend**) cho lớp khác.

# Ví dụ

```
class Tom{
public:
    friend class Jerry;           //Có lớp bạn là Jerry
private:
    int SecretTom;               //Bí mật của Tom
};

class Jerry{
public:
    void Change(Tom T){
        T.SecretTom++;          //Bạn nên có thể truy cập
    }
};
```

# Thao tác với dữ liệu private

- ❖ Khi muốn truy xuất dữ liệu private từ các đối tượng thì phải làm thế nào?
- ❖ Khi muốn cập nhật dữ liệu private từ các đối tượng thì phải làm thế nào?

# Phương thức Truy vấn

- ❖ Có nhiều loại câu hỏi truy vấn có thể:
  - Truy vấn đơn giản (“giá trị của x là bao nhiêu?”)
  - Truy vấn điều kiện (“thành viên x có  $> 10$  không?”)
  - Truy vấn dẫn xuất (“tổng giá trị của các thành viên x và y là bao nhiêu?”)
- ❖ Đặc điểm quan trọng của phương thức truy vấn là nó không nên thay đổi trạng thái hiện tại của đối tượng



# Phương thức Truy vấn

- ❖ Đối với các truy vấn đơn giản, quy ước đặt tên phương thức như sau: **Tiền tố “Get”**, tiếp theo là **tên của thành viên** cần truy vấn
  - `int GetX();`
  - `int GetSize();`
- ❖ Các loại truy vấn khác nên có tên có tính mô tả
- ❖ Truy vấn điều kiện nên có **tiền tố “is”**

# Phương thức Cập nhật

- ❖ Thường để thay đổi trạng thái của đối tượng bằng cách sửa đổi một hoặc nhiều thành viên dữ liệu của đối tượng đó.
- ❖ Dạng đơn giản nhất là gán một giá trị nào đó cho một thành viên dữ liệu.
- ❖ Đối với dạng cập nhật đơn giản, quy ước đặt tên như sau: Dùng tiền tố “Set” kèm theo tên thành viên cần sửa
  - `int SetX(int);`

# Truy vấn và Cập nhật

- ❖ Nếu phương thức **Get/Set** chỉ có nhiệm vụ cho ta đọc/ghi giá trị cho các thành viên dữ liệu → Quy định các thành viên **private** để được ích lợi gì?
  - Ngoài việc **bảo vệ các nguyên tắc đóng gói**, ta cần **kiểm tra xem giá trị mới** cho thành viên dữ liệu có hợp lệ hay không.
  - Sử dụng phương thức truy vấn cho phép ta thực hiện việc **kiểm tra trước khi thực sự thay đổi giá trị** của thành viên.
  - Chỉ cho phép các dữ liệu có thể truy vấn hay thay đổi mới được truy cập đến.



# Ví dụ

```
int Student::SetGPA (double NewGPA){  
    if ((NewGPA >= 0.0) && (NewGPA <= 10.0)){  
        this->GPA= NewGPA;  
        return 0; // Return 0 to indicate success  
    }  
    else  
    {  
        return -1; // Return -1 to indicate failure  
    }  
}
```



# Thành viên tĩnh – static member

- ❖ Trong C, **static** xuất hiện trước dữ liệu được khai báo trong một hàm nào đó thì giá trị của dữ liệu đó vẫn được lưu lại như một biến toàn cục.
- ❖ Trong C++, nếu **static** xuất hiện trước một dữ liệu hoặc một phương thức **của lớp** thì giá trị của nó vẫn được lưu lại và **có ý nghĩa cho đối tượng khác của cùng lớp này**.
- ❖ Các thành viên **static** có thể là **public**, **private** hoặc **protected**.

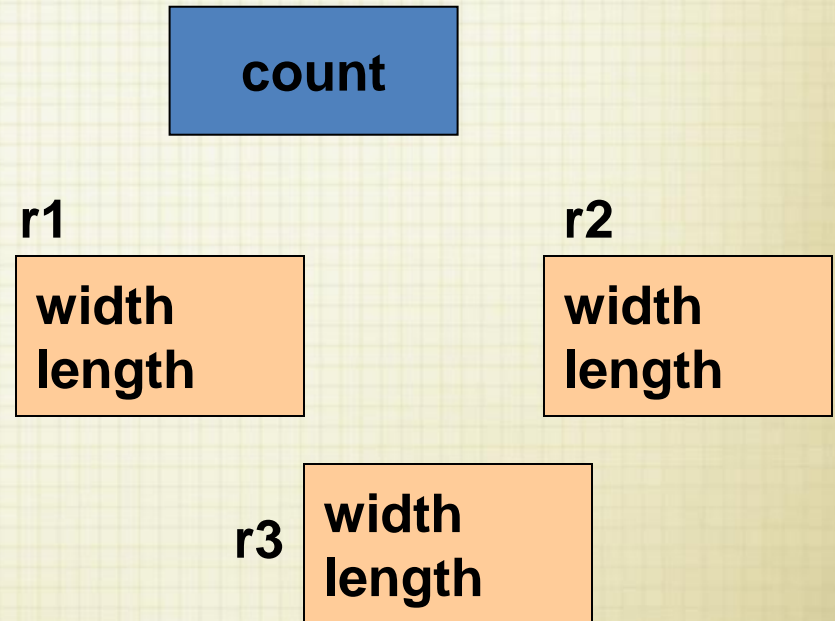
# Thành viên tĩnh – static member

- ❖ Đối với class, **static** dùng để **khai báo thành viên dữ liệu dùng chung cho mọi thể hiện của lớp**:
  - Một bản duy nhất tồn tại trong suốt quá trình chạy của chương trình.
  - Dùng chung cho tất cả các thể hiện của lớp.
  - Bất kể lớp đó có bao nhiêu thể hiện.

# Ví dụ

```
class Rectangle
{
    private:
        int width;
        int length;
        static int count;
    public:
        Rectangle(){count++;}
        void Set(int w, int l);
        int GetArea();
}
```

```
Rectangle r1;
Rectangle r2;
Rectangle r3;
```



# Ví dụ

❖ Đếm số đối tượng MyClass:

```
class MyClass{  
    public:  
        MyClass();  
        ~MyClass();  
        void PrintCount();  
    private:  
        static int count;  
};
```



# Ví dụ

```
int MyClass::count = 0;
MyClass::MyClass(){
    this → count++;
}
MyClass::~MyClass(){
    this → count--;
}
void MyClass::PrintCount(){
    cout << "There are currently " << this → count << "
instance(s) of MyClass.\n";
}
```

# Ví dụ

```
void main()
{
    MyClass* x = new MyClass();
    x → PrintCount();
    MyClass* y = new MyClass();
    x → PrintCount();
    y → PrintCount();
    delete x;
    y → PrintCount();
}
```

# Thành viên tĩnh – static member

## ❖ Phương thức static?

- Đối với các phương thức static, ngoài ý nghĩa tương tự với dữ liệu, còn có sự khác biệt cơ bản đó là việc cho phép truy cập đến các phương thức static khi **chưa khai báo đối tượng** (thông qua tên lớp)

# Thành viên tĩnh – static member

- ❖ Các thành viên lớp tĩnh **public** có thể được truy cập thông qua bất kỳ đối tượng nào của lớp đó, hoặc chúng có thể được truy cập thông qua tên lớp sử dụng toán tử định phạm vi.
- ❖ Các thành viên lớp tĩnh **private** và **protected** phải được truy cập thông qua các hàm thành viên **public** của lớp hoặc thông qua các **friend** của lớp.
- ❖ Các thành viên lớp tĩnh tồn tại ngay cả khi đối tượng của lớp đó không tồn tại.



# Ví dụ về đối tượng toàn cục

- ❖ Xét đoạn chương trình sau:

```
#include <iostream.h>
void main(){
    cout << "Hello, world.\n";
}
```

- ❖ Hãy sửa lại đoạn chương trình trên để có kết xuất:

Entering a C++ program saying...

Hello, world.

And then exiting...

- ❖ Yêu cầu không thay đổi hàm main() dưới bất kỳ hình thức nào.

# Ví dụ về đối tượng toàn cục

```
#include <iostream.h>

class Dummy{
public:
    Dummy(){cout << "Entering a C++ program saying...\n";}
    ~Dummy(){cout << "And then exiting...";}
};

Dummy A;

void main(){
    cout << "Hello, world.\n";
}
```

# Đối tượng là thành phần của lớp

```
class Diem{
    double x, y;
public:
    Diem (double xx, double yy) { x = xx; y = yy; }
    // ...
};

class TamGiac{
    Diem A, B, C;
public:
    void Ve( );
    // ...
};

TamGiac t;    //Error ?
Diem d;       //Error ?
```

# Đối tượng là thành phần của lớp

- ❖ Đối tượng có thể là thành phần của đối tượng khác, khi một đối tượng thuộc lớp “lớn” được tạo ra, các thành phần của nó cũng được tạo ra.
- ❖ Phương thức thiết lập (nếu có) sẽ được tự động gọi cho các đối tượng thành phần.
- ❖ Khi đối tượng kết hợp bị hủy → đối tượng thành phần của nó cũng bị hủy, nghĩa là phương thức hủy bỏ sẽ được gọi cho các đối tượng thành phần, sau khi phương thức hủy bỏ của đối tượng kết hợp được gọi.



# Đối tượng là thành phần của lớp

- ❖ Nếu đối tượng thành phần phải cung cấp tham số khi thiết lập thì đối tượng kết hợp (đối tượng lớn) **phải có phương thức thiết lập** để cung cấp tham số thiết lập cho các đối tượng thành phần.
- ❖ Cú pháp để khởi động đối tượng thành phần là **dùng dấu hai chấm (:)** theo sau bởi tên thành phần và tham số khởi động.

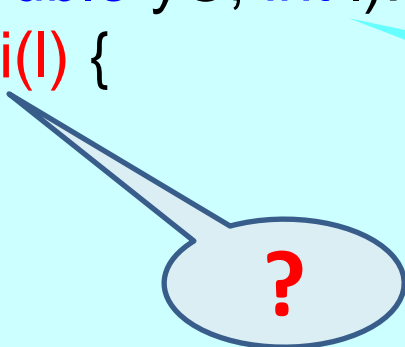
# Ví dụ

```
class TamGiac{
    Diem A, B, C;
public:
    TamGiac(double xA, double yA, double xB, double yB,
double xC, double yC) : A(xA,yA), B(xB,yB),C(xC,yC){
        cout<<"Khoi tao tam giac";
    }
    void Ve();
    // ...
};

TamGiac t(100,100,200,400,300,300);
```

# Ví dụ

```
class TamGiac{  
    Diem A,B,C;  
    int loai;  
public:  
    TamGiac(double xA, double yA, double xB, double yB,  
            double xC, double yC, int l): A(xA,yA), B(xB,yB),  
            C(xC,yC), loai(l) {  
    }  
    void Ve();  
    // ...  
};
```

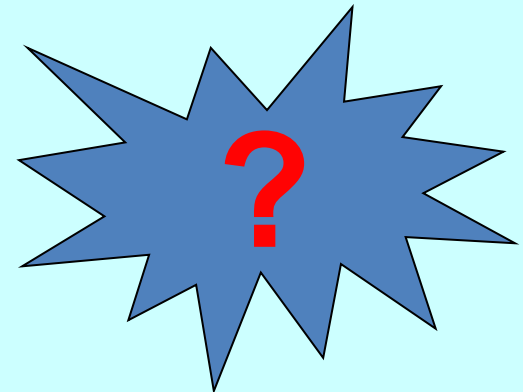


Cú pháp dấu hai chấm  
cũng được dùng cho đối  
tượng thành phần thuộc  
kiểu cơ sở

```
TamGiac t (100, 100, 200, 400, 300, 300, 1);
```

# Ví dụ

```
class Diem{  
    double x,y;  
public:  
    Diem(double xx = 0, double yy = 0) : x(xx), y(yy){  
    }  
    void Set(double xx, double yy){  
        x = xx;  
        y = yy;  
    }  
};
```





# Đối tượng là thành phần của mảng

- ❖ Khi một mảng được tạo ra → các phần tử của nó cũng được tạo ra → phương thức thiết lập sẽ được gọi cho từng phần tử.
- ❖ Vì không thể cung cấp tham số khởi động cho tất cả các phần tử của mảng → khi khai báo mảng, mỗi đối tượng trong mảng phải có **khả năng tự khởi động**, nghĩa là có thể thiết lập không cần tham số.

# Đối tượng là thành phần của mảng

❖ Đối tượng có khả năng tự khởi động trong những trường hợp nào?

1. Lớp không có phương thức thiết lập
2. Lớp có phương thức thiết lập không tham số
3. Lớp có phương thức thiết lập mà mọi tham số đều có giá trị mặc nhiên

# Đối tượng là thành phần của mảng

```
class Diem
{
    double x,y;
    public:
        Diem(double xx, double yy) : x(xx), y(yy) { }
        void Set(double xx, double yy) {
            x = xx, y = yy;
        }
        // ...
};
```

# Đối tượng là thành phần của mảng

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



# Đối tượng là thành phần của mảng

```
class SinhVien{  
    String MaSo;  
    String HoTen;  
    int NamSinh;  
public:  
    SinhVien(char *ht, char *ms, int ns) : HoTen(ht),  
        MaSo(ms), NamSinh(ns){ }  
};
```

```
String arrString[3];  
Diem arrDiem[5];  
SinhVien arrSV[7];
```



# Dùng phương thức thiết lập với tham số có giá trị mặc nhiên

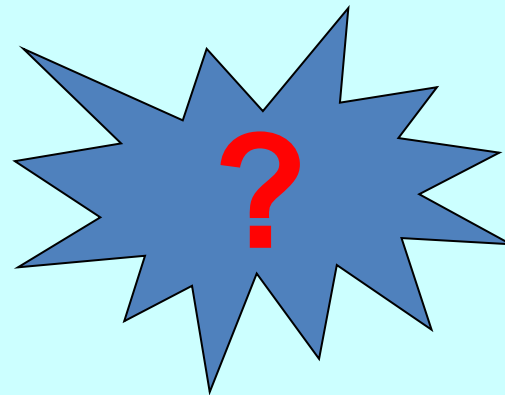
```
class Diem
{
    double x,y;
public:
    Diem(double xx = 0, double yy = 0) : x(xx), y(yy) { }
    void Set(double xx, double yy) {
        x = xx, y = yy;
    }
    // ...
};
```

# Dùng phương thức thiết lập với tham số có giá trị mặc nhiên

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

# Dùng phương thức thiết lập với tham số có giá trị mặc nhiên

```
class SinhVien{  
    String MaSo, HoTen;  
    int NamSinh;  
public:  
    SinhVien(char *ht="Nguyen Van A", char  
        *ms="19920014", int ns = 1982) : HoTen(ht), MaSo(ms),  
        NamSinh(ns) { }  
};  
String as[3];  
Diem ad[5];  
SinhVien asv[7];
```





# Dùng phương thức thiết lập không tham số

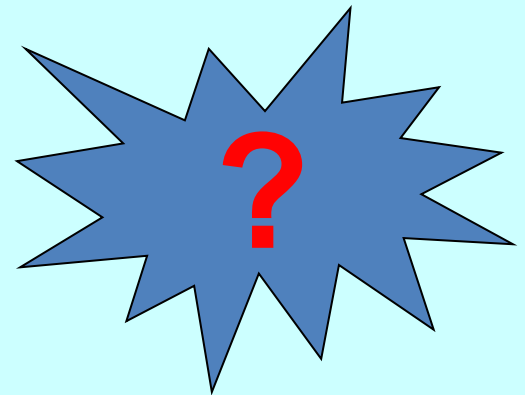
```
class Diem
{
    double x,y;
public:
    Diem(double xx, double yy) : x(xx), y(yy)
    {}
    Diem() : x(0), y(0)
    {}
    // ...
};
```

# Dùng phương thức thiết lập không tham số

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

# Dùng phương thức thiết lập không tham số

```
class SinhVien {  
    String MaSo, HoTen;  
    int NamSinh;  
public:  
    SinhVien(char *ht, char *ms, int ns) : HoTen(ht),  
        MaSo(ms), NamSinh(ns) { }  
    SinhVien() : HoTen("Nguyen Van A"), MaSo("19920014"),  
        NamSinh(1982) { }  
};  
String as[3];  
Diem ad[5];  
SinhVien asv[7];
```



# Đối tượng được cấp phát động

- ❖ Đối tượng được cấp phát động là các đối tượng được tạo ra bằng phép toán **new** và bị hủy đi bằng phép toán **delete**
- ❖ Phép toán **new** cấp đối tượng trong vùng heap và gọi phương thức thiết lập cho đối tượng được cấp.



# Đối tượng được cấp phát động

```
class String {  
    char *p;  
public:  
    String( char *s ) { p = strdup(s); }  
    String( const String &s ) { p = strdup(s.p); }  
    ~String() { delete [] p; }  
    //...  
};  
  
class Diem {  
    double x,y;  
public:  
    Diem(double xx, double yy) : x(xx), y(yy) { }  
    //...  
};
```

# Cấp phát và hủy một đối tượng

```
int *pi = new int;
```

```
int *pj = new int(15);
```

```
Diem *pd = new Diem(20,40);
```

```
String *pa = new String("Nguyen Van A");
```

```
//...
```

```
delete pa;
```

```
delete pd;
```

```
delete pj;
```

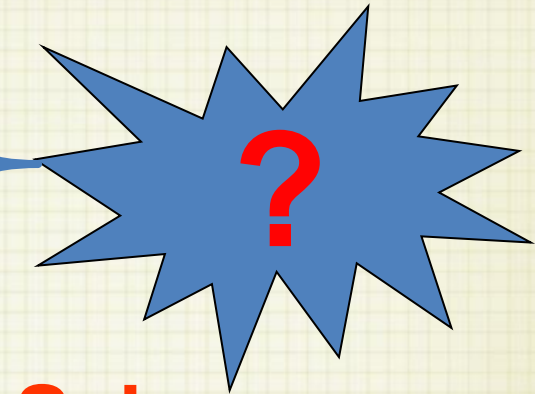
```
delete pi;
```

# Cấp phát và hủy nhiều đối tượng

```
int *pai = new int[10];
```

```
Diem *pad = new Diem[5];
```

```
String *pas = new String[5];
```



**Sai**

- ❖ Trong trường hợp cấp phát nhiều đối tượng, ta không thể cung cấp tham số cho từng phần tử được cấp phát.

# Cấp và hủy nhiều đối tượng

❖ Thông báo lỗi cho đoạn chương trình trên như sau:

- *Cannot find default constructor to initialize array element of type 'Diem'*
- *Cannot find default constructor to initialize array element of type String'*

❖ Khắc phục lỗi?

Lỗi trên được khắc phục bằng cách **cung cấp phương thức thiết lập để đối tượng có khả năng tự khởi động.**



# Cấp và hủy nhiều đối tượng

```
class String{
    char *p;
public:
    String (char *s = "Alibaba") { p = strdup(s); }
    String (const String &s) { p = strdup(s.p); }
    ~String () {delete [] p;}
    //...
};

class Diem {
    double x,y;
public:
    Diem (double xx, double yy) : x(xx),y(yy){};
    Diem () : x(0),y(0){};
};
```

# Cấp và hủy nhiều đối tượng

- ❖ Khi đó mọi phần tử được cấp đều được khởi động với cùng giá trị.

```
int *pai = new int[10];
```

```
Diem *pad = new Diem[5];
```

```
//Ca 5 diem co cung toa do (0,0)
```

```
String *pas = new String[5];
```

```
//Ca 5 chuai cung duoc khoi dong la "Alibaba"
```

# Cấp và hủy nhiều đối tượng

- ❖ Việc hủy nhiều đối tượng được thực hiện bằng cách dùng `delete` và có thêm dấu `[]` ở trước.

```
delete [] pas;
```

```
delete [] pad;
```

```
delete [] pai;
```

# Giao diện và chi tiết cài đặt

- ❖ Lớp có hai phần tách rời
  - ❖ Phần giao diện khai báo trong phần **public** để người sử dụng “thấy” và sử dụng.
  - ❖ Chi tiết cài đặt bao gồm dữ liệu khai báo trong phần **private** của lớp và chi tiết mã hóa các hàm thành phần, vô hình đối với người dùng.
- ❖ Lớp **ThoiDiem** có thể được cài đặt với các thành phần dữ liệu là giờ, phút, giây hoặc tổng số giây tính từ 0 giờ.



# Giao diện và chi tiết cài đặt

- ❖ Ta có thể thay đổi uyển chuyển chi tiết cài đặt, nghĩa là có thể thay đổi tổ chức dữ liệu của lớp, cũng như có thể thay đổi chi tiết thực hiện các hàm thành phần (do sự thay đổi tổ chức dữ liệu hoặc để cải tiến giải thuật). Nhưng nếu bảo đảm không thay đổi phần giao diện thì không ảnh hưởng đến người sử dụng, và do đó không làm đổ vỡ kiến trúc của hệ thống.

# Lớp ThoiDiem – Cách 1

```
class ThoiDiem{
    int gio, phut, giay;
    static bool KiemTraHopLe(int g, int p, int gy);
public:
    ThoiDiem(int g = 0, int p = 0, int gy = 0) {Set(g,p,gy);}
    void Set(int g, int p, int gy);
    int GetGio() const {return gio; }
    int GetPhut() const {return phut; }
    int GetGiay() const {return giay; }
    void Nhap();
    void Xuat() const;
    void Tang();
    void Giam();
};
```

# Lớp ThoiDiem – Cách 2

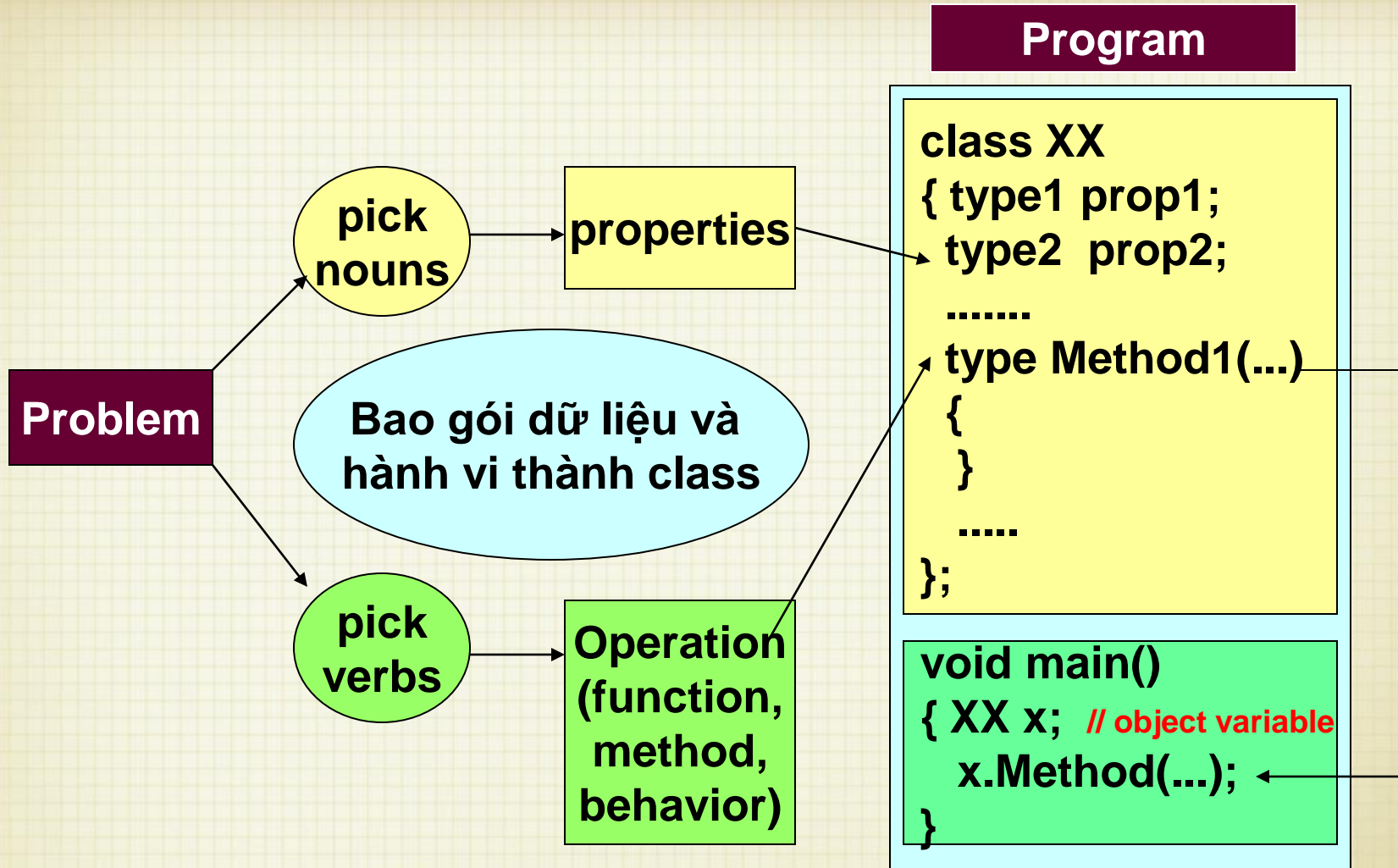
```
class ThoiDiem{
    long tsgiai;
    static bool KiemTraHopLe(int g, int p, int gy);
public:
    ThoiDiem(int g = 0, int p = 0, int gy = 0) {Set(g,p,gy);}
    void Set(int g, int p, int gy);
    int GetGio() const {return tsgiai/3600;}
    int GetPhut() const {return (tsgiai%3600)/60;}
    int GetGiay() const {return tsgiai%60;}
    void Nhap();
    void Xuat() const;
    void Tang();
    void Giam();
};
```

# Các nguyên tắc xây dựng lớp

- ❖ **Hình thành lớp:** Khi ta nghĩ đến “nó” như một khái niệm riêng lẻ → Xây dựng lớp biểu diễn khái niệm đó.
- ❖ **Lớp** là biểu diễn cụ thể của một khái niệm vì vậy tên lớp luôn là danh từ.
- ❖ **Các thuộc tính** của lớp là các thành phần dữ liệu nên chúng **luôn là danh từ**.
- ❖ **Các hàm thành phần** (các hành vi) là các thao tác chỉ rõ hoạt động của lớp nên **các hàm là động từ**.



# Các nguyên tắc xây dựng lớp



# Các nguyên tắc xây dựng lớp

- ❖ Các thuộc tính có thể suy diễn từ những thuộc tính khác thì dùng hàm thành phần để thực hiện tính toán. Ví dụ chu vi, diện tích của một tam giác

```
class TamGiac{  
    Diem A,B,C;  
    double ChuVi;  
    double DienTich;  
public:  
    //...  
};
```

```
class TamGiac{  
    Diem A,B,C;  
public:  
    //...  
    double ChuVi() const;  
    double DienTich() const;  
};
```

# Các nguyên tắc xây dựng lớp

- ❖ Tuy nhiên, nếu các thuộc tính suy diễn đòi hỏi nhiều tài nguyên hoặc thời gian để thực hiện tính toán, ta có thể khai báo là dữ liệu thành phần.

```
class QuocGia{  
    long DanSo;  
    double DienTich;  
    double TuổiTrungBinh;  
public:  
    double TínhTuoiTB() const;  
    //...  
};
```

# Các nguyên tắc xây dựng lớp

❖ Dữ liệu thành phần nên được kết hợp:

```
class TamGiac{
    Diem A,B,C;
public:
    //...
};

class HìnhTron{
    Diem Tam;
    double BanKinh;
public:
    //...
};
```

```
class TamGiac{
    double xA, yA;
    double xB, yB, xC, yC;
public:
    //...
};

class HìnhTron{
    double tx, ty, BanKinh;
public:
    //...
};
```



# Các nguyên tắc xây dựng lớp

- ❖ Trong mọi trường hợp, nên có phương thức thiết lập (Constructor) để khởi động đối tượng
- ❖ Nên có phương thức thiết lập có khả năng tự khởi động không cần tham số
- ❖ Nếu đối tượng có nhu cầu cấp phát tài nguyên thì phải có phương thức thiết lập, copy constructor để khởi động đối tượng bằng đối tượng cùng kiểu và có destructor để dọn dẹp. Ngoài ra còn có phép gán (chương 4).
- ❖ Nếu đối tượng đơn giản không cần tài nguyên riêng  
→ Không cần copy constructor và destructor

# BÀI TẬP

## 1. Xây dựng các lớp sau

- Ma trận các số nguyên
- Đa thức

- Yêu cầu:

- Các thuộc tính, phương thức cần thiết (nhập, xuất, cộng, trừ, nhân, các phương thức truy cập dữ liệu,...)
- Các phương thức khởi tạo (cả khởi tạo sao chép), hủy cần thiết

# BÀI TẬP

2. Xây dựng lớp Vector (N số thực) và các phương khởi tạo, hủy (nếu có) thức nhập, xuất, cộng, trừ, tịnh tiến, tính norm (chuẩn) của vector. Viết chương trình quản lý một danh sách các vector và thực hiện các yêu cầu:

- Nhập N vector và xuất ra thông tin các Vector vừa nhập
- Xuất các vector có norm tăng dần
- Tịnh tiến các vector theo vector là chính nó
- Tính tổng và hiệu các vector với chính nó
- **Ghi chú:**

- $V(x_1, x_2, \dots, x_n), \text{norm}(V) = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$



# BÀI TẬP

**3. Xây dựng lớp **điểm** trong mặt phẳng 2 chiều (tọa độ là số thực) và các phương thức sau:**

- Khởi tạo mặc định (có tọa độ là (0;0)) và khởi tạo có 2 tham số.
- Phương thức khởi tạo sao chép (nếu cần thiết).
- Phương thức hủy bỏ (nếu cần thiết).
- Phương thức nhập cho điểm. Nếu người dùng không muốn nhập liệu thì tọa độ điểm sẽ có giá trị mặc nhiên là (0;0).
- Phương thức xuất cho điểm.
- Phương thức tịnh tiến một điểm theo một vector. VD: A(2;3) tịnh tiến với vector v(2;5) thì A có tọa độ mới là  $A(x_A+x_v; y_A+y_v) = (4;8)$ .
- Phương thức tính khoảng cách với một điểm khác  $d(A,B) = \sqrt{(x_A-x_B)^2+(y_A-y_B)^2}$ .

**Viết chương trình thực hiện yêu cầu tác sau:**

- Nhập vào N điểm, N do người dùng nhập vào.
- Xuất ra N điểm vừa nhập.
- Tìm khoảng cách lớn nhất và nhỏ nhất giữa 2 điểm trong tập điểm vừa nhập.
- Nhập vào một vector cần tịnh tiến V(x,y). Xuất ra tọa độ tất cả các điểm sau khi tịnh tiến theo V.



# BÀI TẬP

4. Xây dựng lớp **tam giác** gồm 3 điểm (tạo trong bài 1) và các phương thức sau

- Khởi tạo mặc định (gọi khởi tạo mặc định của 3 điểm).
- Khởi tạo có 3 tham số là 3 điểm nhằm khởi cho 3 điểm của tam giác.
- Phương thức hủy bỏ (nếu cần thiết).
- Phương thức nhập cho tam giác. Nếu người dùng không muốn nhập liệu thì tọa độ các điểm sẽ có giá trị mặc nhiên là (0;0).
- Phương thức xuất cho tam giác.
- Phương thức tịnh tiến một tam giác theo một vector. Tam giác mới có tọa độ các điểm là các điểm sau khi được tịnh tiến.
- Tính chu vi và diện tích của tam giác. Gợi ý tính diện tích tam giác  $S_{ABC} = \sqrt{p*(p-a)*(p-b)*(p-c)}$ , trong đó: a, b, c là độ dài 3 cạnh và p là nửa chu vi.

Viết chương trình thực hiện yêu cầu tác sau:

- Nhập vào N tam giác, N do người dùng nhập vào.
- Xuất ra N tam giác vừa nhập.
- Xuất ra tam giác có chu vi lớn nhất, tam giác có diện tích lớn nhất.
- Nhập vào một vector cần tịnh tiến V. Xuất ra các tam giác sau khi tịnh tiến theo V.

# Q & A

