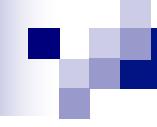


## **CHƯƠNG 3**

# **LỚP VÀ ĐỐI TƯỢNG (CLASS & OBJECT)**

TS. LÊ THỊ MỸ HẠNH  
Bộ môn Công nghệ phần mềm  
Khoa Công Nghệ Thông Tin  
Đại Học Bách khoa – Đại học Đà Nẵng



# Nội dung

- Lớp – Quyền truy xuất
- Khai báo, định nghĩa 1 lớp đơn giản
- Hàm thành viên nội tuyến (inline)
- Hàm xây dựng (constructor)
- Hàm hủy (destructor)
- Hàm bạn (friend) – Lớp bạn
- Đối số mặc định
- Đối số thành viên ẩn (con trỏ this)

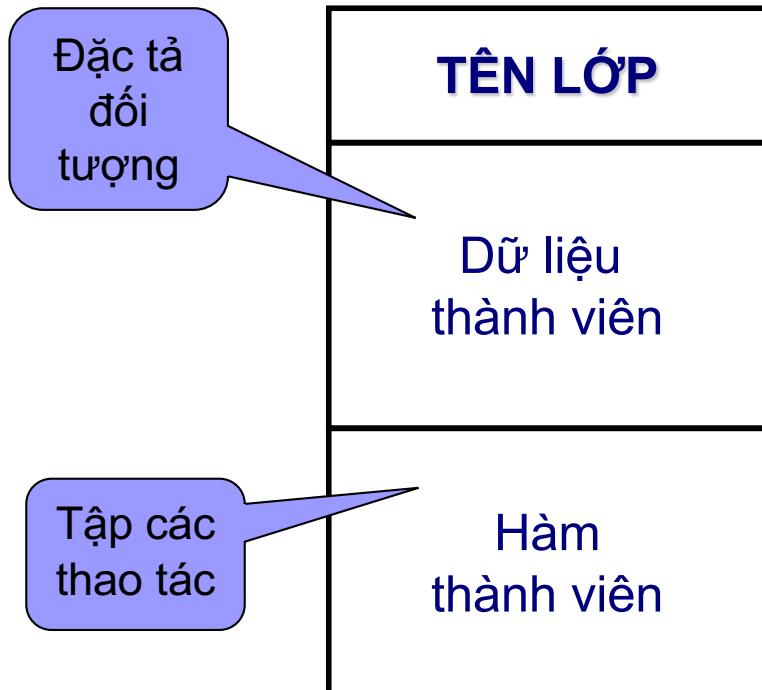


# Nội dung (tt)

- Toán tử phạm vi
- Danh sách khởi tạo thành viên
- Thành viên hằng - Thành viên tĩnh
- Thành viên tham chiếu
- Thành viên là đối tượng của 1 lớp
- Mảng các đối tượng
- Phạm vi lớp
- Cấu trúc (structure) và hợp (union)
- Các trường bit

# Khái niệm lớp

- **Lớp:** kiểu dữ liệu trừu tượng.



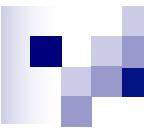
```
class Classname {  
    <Quyền truy xuất> :  
        DataType1 memberdata1;  
        DataType2 memberdata2;  
        .....  
    < Quyền truy xuất > :  
        memberFunction1();  
        memberFunction2();  
        .....  
};
```

```
class Point {  
    int xVal, yVal;  
    public:  
        void SetPt (int, int);  
        void OffsetPt (int, int);  
};
```

private  
protected  
public

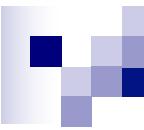
# Đối tượng

- Đối tượng(Object): là một thể hiện thuộc lớp, một thực thể có thực
  - Khai báo:  
**<Classname> <Objectname>;**
  - Để truy xuất đến một thành phần của đối tượng, truy xuất giống như kiểu struct
    - Thành viên dữ liệu:  
**Objectname.datamember**
    - Hàm thành viên:  
**Objectname. Memberfunction(parameter)**
  - Ví dụ:  
Point pt;  
pt.SetPt(10,20);  
pt.OffsetPt(2,2);



# Đóng gói trong C++

- Khái niệm đóng gói có sẵn trong C++ class: ta có thể hạn chế quyền truy nhập đến các thành viên của đối tượng
- Sử dụng một bộ từ khoá để mô tả quyền truy nhập:
  - **private**
    - nếu một thành viên của một lớp được khai báo là **private**, nó chỉ được truy nhập đến từ bên trong lớp đó
    - Mặc định: *mọi* thành viên của class là **private**, do đó nhấn mạnh khái niệm đóng gói của lập trình hướng đối tượng.
  - **Public**
    - Các thành viên được khai báo là **public** có thể được truy nhập từ bên ngoài đối tượng
    - là mặc định đối với các thành viên của **struct**
  - **protected**
  - **friend**



# Đóng gói trong C++

## ■ Khi nào sử dụng quyền nào?

- Theo phong cách lập trình hướng đối tượng tốt, ta sẽ giữ mọi thành viên dữ liệu ở dạng **private** (che dấu dữ liệu).
- Các phương thức thường khai báo là **public** để có thể liên lạc được với đối tượng từ bên ngoài(giao diện của đối tượng).
- Các phương thức tiện ích chỉ được dùng bởi các phương thức khác trong cùng lớp nên được khai báo **private**.

# Khai báo các phương thức

- Giao diện của phương thức luôn đặt trong định nghĩa lớp, cũng như các khai báo thành viên dữ liệu.
- Phần cài đặt (định nghĩa phương thức) có thể đặt trong định nghĩa lớp hoặc đặt ở ngoài.
- Hai lựa chọn:

```
class Point {  
    int xVal, yVal;  
public:  
    void SetPt (int, int);  
    void OffsetPt (int, int);  
};  
void Point:: SetPt (int x, int y) {  
    xVal = x;    yVal = y;  
}
```

```
class Point {  
    int xVal, yVal;  
public:  
    void SetPt (int x, int y) {  
        xVal = x;  
        yVal = y;  
    }  
    void OffsetPt (int, int);  
};
```

# Khai báo các phương thức

## ■ Hàm **inline**:

- Cải thiện tốc độ thực thi
- Tốn bộ nhớ (dành cho mã lệnh) khi thực thi.

**Cách 1:**

Định  
nghĩa  
bên  
trong  
lớp

```
class Point {  
    int xVal, yVal;  
public:  
    void SetPt (int x, int y) {  
        xVal = x;  
        yVal = y;  
    }  
    void OffsetPt (int x, int y) {  
        xVal += x;  
        yVal += y;  
    }  
};
```

**Cách 2:**

thêm  
Tù  
khóa  
**inline**

**class Point {**

int xVal, yVal;

**public:**

void **SetPt** (int, int);

void **OffsetPt** (int, int);

**}**

**inline void Point::SetPt** (int x, int y) {

xVal = x;

yVal = y;

**}**

.....

# Đặt khai báo lớp ở đâu?

- Để đảm bảo tính đóng gói, ta thường đặt khai báo của lớp trong file header
  - tên file thường trùng với tên lớp. Ví dụ khai báo lớp Car đặt trong file “car.h”
- Phần cài đặt (định nghĩa) đặt trong một file nguồn tương ứng
  - “car.cpp” hoặc “car.cc”
- Quy ước đặt khai báo/định nghĩa của lớp trong file trùng tên lớp được chấp nhận rộng rãi trong C++
  - là quy tắc bắt buộc đối với các lớp của Java

# File header car.h

```
// car.h
#ifndef CAR_H
#define CAR_H
class Car {
public:
    //...
    void drive(int speed, int distance);
    //...
    void stop();
    //...
    void turnLeft();
private:
    int vin;           //...
    string make; //...
    string model;      //...
    string color;       //...
};
#endif
```

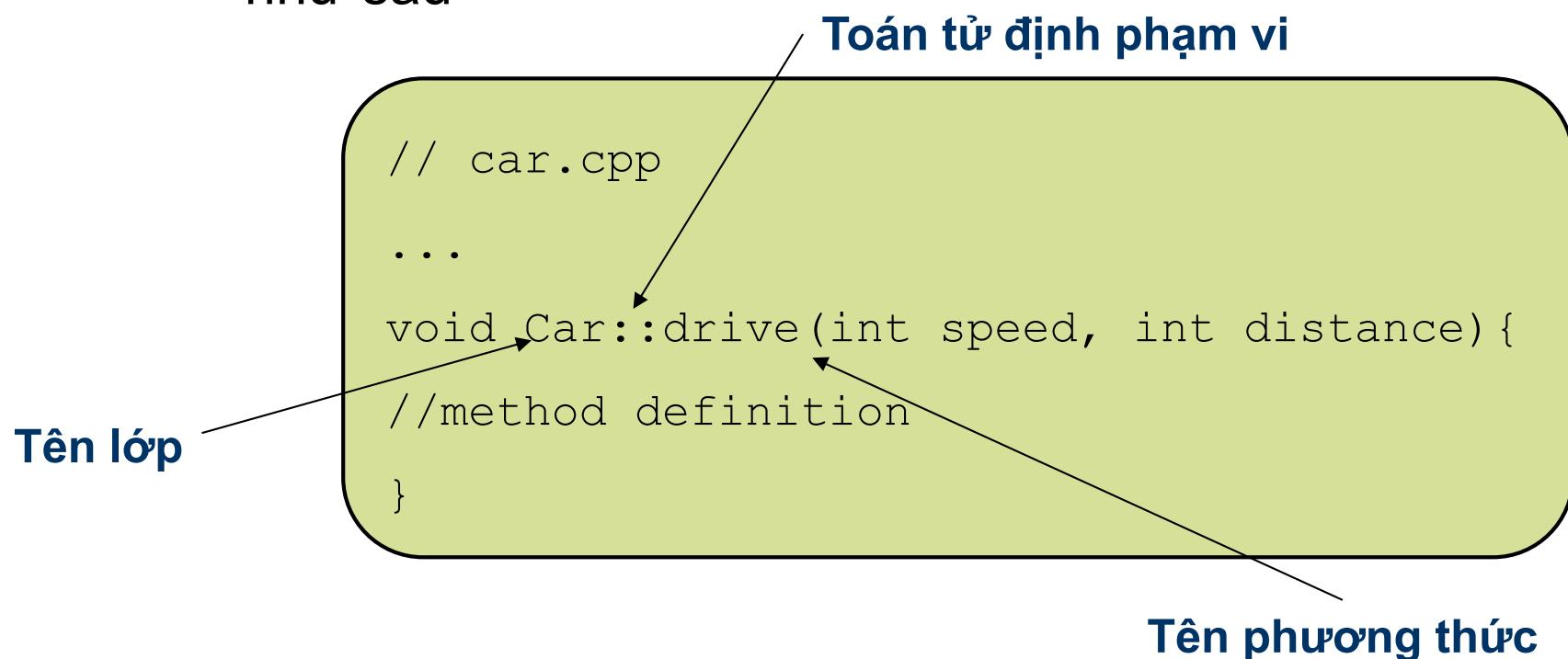
# Định nghĩa các phương thức

- Định nghĩa của các phương thức cần đặt trong 1 file nguồn trùng tên với tên lớp
- File bắt đầu với các lệnh **#include** và có thể có các khai báo **using** cho các namespace
- Bên cạnh việc include các thư viện C++ cần thiết, ta còn phải include header file chứa khai báo lớp

```
// car.cpp
#include <iostream>
#include <string>
#include "car.h"
using namespace std;
...
```

# Định nghĩa các phương thức

- Khi định nghĩa một phương thức, ta cần sử dụng toán tử phạm vi để trình biên dịch hiểu đó là phương thức của một lớp cụ thể chứ không phải một hàm thông thường khác
  - Ví dụ, định nghĩa phương thức drive của lớp Car được viết như sau

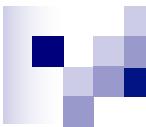


# Định nghĩa các phương thức

- Vậy cấu trúc của file nguồn lớp Car có thể như sau:

```
// car.cpp
#include <iostream.h>
#include <string.h>
#include "car.h"

void Car::drive(int speed, int distance) { ... }
void Car::stop() { ... }
void Car::turnLeft() { ... }
```



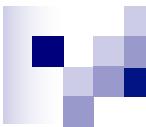
# Khai báo lớp – Ví dụ

- Khai báo lớp: file.h
  - file trùng tên lớp
- Point.h

```
class Point
{
    int xVal, yVal;
public:
    Point();
    ~Point();
    void Show();
};
```

- Cài đặt phương thức: file.cpp
  - Point.cpp

```
#include "Point.h"
Point::Point()
{
    //code
}
Point::~Point()
{
    //code
}
void Point::Show()
{
    //code
}
```



# Đối số thành viên ẩn

## ■ Con trỏ **\*this**:

- Là 1 thành viên ẩn, có thuộc tính là private.
- Trỏ tới chính bản thân đối tượng.

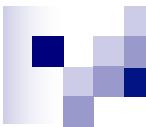
```
void Point::OffsetPt (int xValL, int yVal) {  
    this->xVal += xVal;  
    this->yVal += yVal;  
}
```



```
void Point::OffsetPt (int x, int y) {  
    this->xVal += x;  
    this->yVal += y;  
}
```



- Có những trường hợp sử dụng **\*this** là dư thừa (Ví dụ trên)
- Tuy nhiên, có những trường hợp phải sử dụng con trỏ **\*this**



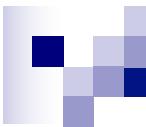
# Con trỏ this

- Tuy không bắt buộc sử dụng tường minh con trỏ this, ta có thể dùng nó để giải quyết vấn đề tên trùng và phạm vi

```
void Foo::bar()
{
    int x;
    x = 5;           // local x
    this->x = 6;    // this instance's x
}
```

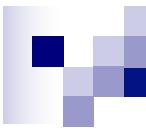
hoặc

```
void Foo::bar(int x)
{
    this->x = x;
}
```



# Con trả this

- Con trả this được các phương thức tự động sử dụng, nên việc ta có sử dụng nó một cách tường minh hay bỏ qua không ảnh hưởng đến tốc độ chạy chương trình
- Nhiều lập trình viên sử dụng this một cách tường minh mỗi khi truy nhập các thành viên dữ liệu
  - để đảm bảo không có rắc rối về phạm vi
  - ngoài ra, còn để tự nhắc rằng mình đang truy nhập thành viên
- Lựa chọn có dùng hay không là tuỳ ở mỗi người



# Toán tử phạm vi

- **Toán tử ::** dùng để xác định chính xác hàm (thuộc tính) được truy xuất thuộc lớp nào.
  - Câu lệnh: pt.OffsetPt(2,2);  
                  <=> pt.**Point**::OffsetPt(2,2);
  - Cần thiết trong một số trường hợp:
    - Cách gọi hàm trong thừa kế.
    - Tên thành viên bị che bởi biến cục bộ.
- Ví dụ: Point(int xVal, int yVal) {  
                  **Point**::xVal = xVal;  
                  **Point**::yVal = yVal;  
}

# Hàm xây dựng (Constructor)

- Khi đối tượng vừa được tạo:
  - Giá trị các thuộc tính bằng bao nhiêu?
  - Đối tượng cần có thông tin ban đầu.
  - Giải pháp:
    - Xây dựng phương thức cung cấp thông tin.
- → Người dùng quên gọi?!
  - “Làm khai sinh” cho đối tượng!

```
class Point {  
    int xVal, yVal;  
public:  
    void OffsetPt (int x, int y) {  
        xVal += x; yVal += y;  
    }  
    void Display(){  
        cout<<x<<","<<y<<endl;  
    }  
};
```

# Hàm xây dựng (Constructor)

## ■ Khi đối tượng vừa được tạo:

- Giá trị các thuộc tính bằng bao nhiêu?
- Đối tượng cần có thông tin ban đầu.
- Giải pháp:
  - Xây dựng phương thức cung cấp thông tin.  
→ Người dùng quên gọi?!  
□ “Làm khai sinh” cho đối tượng!

```
class Point {  
    int xVal, yVal;  
public:  
    void OffsetPt (int x, int y) {  
        xVal += x; yVal += y;  
    }  
    void Display(){  
        cout<<xVal<<","<<yVal<<endl;  
    }  
};
```

**PhanSo**

- Tỷ số??
- Mẫu số??

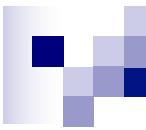
**HocSinh**

- Họ tên??
- Điểm văn??
- Điểm toán??

**Hàm dựng ra đời!!**

# Hàm xây dựng (Constructor)

- Dùng để **định nghĩa** và **khởi tạo** đối tượng cùng 1 lúc.
- Hàm khai báo trùng với tên lớp, không có kiểu trả về.
- Không gọi trực tiếp, sẽ được tự động gọi khi khởi tạo đối tượng.
- **Gán giá trị, cấp vùng nhớ** cho các *dữ liệu thành viên*.
- Constructor có thể được khai báo chồng (đa năng hóa) như các hàm C++ thông thường khác
  - cung cấp các kiểu khởi tạo khác nhau tùy theo các **đối số** được cho khi tạo thể hiện



# Hàm xây dựng

## ■ Hàm dựng mặc định (default constructor)

- nếu ta không cung cấp một phương thức constructor nào, C++ sẽ tự sinh constructor mặc định là một phương thức rỗng (không làm gì)
  - mục đích để luôn có một constructor nào đó để gọi khi không có tham số nào
- Hàm dựng không có đối số, hoặc tất cả đối số đều nhận giá trị mặc định, gọi là hàm dựng mặc định
  - Khai báo đối tượng không cần truyền đối số thực
  - Lớp chỉ có 1 hàm dựng mặc định
- Nếu 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.

# DEFAULT CONSTRUCTOR

```
#include <iostream>
using namespace std;
class Point
{
    int xVal;
    int yVal;
public:
    void Show();
};
void Point::Show()
{
    cout << this->xVal << this->yVal;
}
int main()
{
    Point p;
    p.Show();
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Point
{
    int xVal;
    int yVal;
public:
    Point();
    ~Point();
    void Show();
};
Point::Point()
{
    this->xVal = 1;
    this->yVal = 1;
}
Point::~Point() { }
void Point::Show()
{
    cout << this->xVal << this->yVal;
}
int main()
{
    Point p;
    p.Show();
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Point
{
    int xVal;
    int yVal;
public:
    Point();
    Point(int, int);
    ~Point();
    void Show();
};
Point::Point()
{
    this->xVal = 1;
    this->yVal = 1;
}
Point::Point(int x, int y)
{
    this->xVal = x;
    this->yVal = y;
}
Point::~Point() { }
void Point::Show()
{
    cout << this->xVal << this->yVal;
}
int main()
{
    Point p(1, 2);
    p.Show();
    return 0;
}
```

# DEFAULT CONSTRUCTOR

- Hàm dựng mặc định với đối số mặc định

```
//Point.h
class Point
{
    int xVal, yVal;
public:
    Point(int = 1, int = 1);
    ~Point();
    void Show();
};
```

```
//Point.h
#include <iostream>
#include "Point.h"
using namespace std;
Point::Point(int x, int y)
{
    this->xVal = x;
    this->yVal = y;
}
Point::~Point() { }
void Point::Show()
{
    cout << this->xVal
        << this->yVal;
}
```

```
//main.cpp
#include <iostream>
#include "Point.h"
using namespace std;
int main()
{
    Point p1;
    Point p2(2, 3);
    p1.Show();
    p2.Show();
    return 0;
}
```

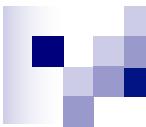
# Hàm dựng sao chép (Copy constructor)

- Copy constructor là constructor đặc biệt được gọi khi ta tạo đối tượng mới là bản sao của một đối tượng đã có sẵn

MyClass x(5);

MyClass y = x; **hoặc** MyClass y(x);

- C++ cung cấp sẵn một copy constructor, nó chỉ đơn giản copy từng thành viên dữ liệu từ đối tượng cũ sang đối tượng mới.
- Tuy nhiên, trong nhiều trường hợp, ta cần thực hiện các công việc Khởi tạo khác trong copy constructor. → Ta có thể định nghĩa lại copy constructor
  - Ví dụ: lấy giá trị cho một ID duy nhất từ đâu đó, hoặc thực hiện sao chép “sâu” (chẳng hạn khi một trong các thành viên là con trỏ giữ hộ nhớ cần phát động)



# Hàm dựng sao chép (Copy constructor)

- Khai báo cho copy constructor của lớp Foo:

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



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

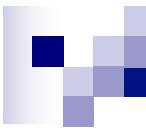
# Hàm dựng sao chép (Copy constructor)

```
//Point.h
class Point
{
    int xVal, yVal;
public:
    Point(int = 1, int = 1);
    Point(const Point &);

    ~Point();
    void Show();
};

//Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;
Point::Point(int x, int y)
{
    this->xVal = x;
    this->yVal = y;
}
Point::Point(const Point &p)
{
    this->xVal = p.xVal;
    this->yVal = p.yVal;
}
Point::~Point() { }
void Point::Show()
{
    cout << this->xVal
        << this->yVal;
}

//main.cpp
#include <iostream>
#include "Point.h"
using namespace std;
int main()
{
    Point p1(2, 3);
    Point p2(p1);
    p1.Show();
    p2.Show();
    return 0;
}
```



# Hàm xây dựng

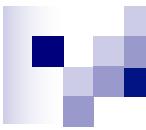
## ■ Dr. Guru khuyên:

- Một lớp nên có tối thiểu 3 hàm dựng:
  - Hàm dựng mặc định.
  - Hàm dựng có đầy đủ tham số.
  - Hàm dựng sao chép.



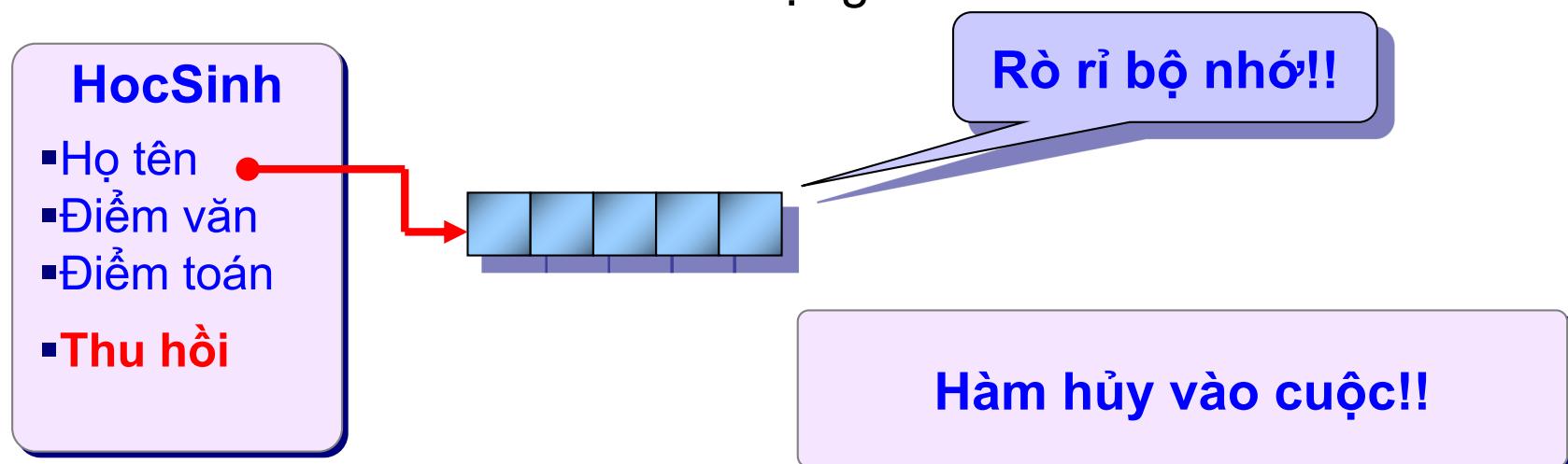
```
class PhanSo{  
    private:  
        int      m_tuSo;  
        int      m_mauSo;  
    public:  
        PhanSo();  
        PhanSo(int tuSo, int mauSo);  
        PhanSo(const PhanSo &p);  
};
```

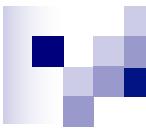
```
class Point{  
    private:  
        int      x, y;  
    public:  
        Point(int xx=0, int yy=0);  
        Point(const Point &p);  
};
```



# Hàm hủy (Destructor)

- Vấn đề rò rỉ bộ nhớ (memory leak):
  - Khi hoạt động, đối tượng có cấp phát bộ nhớ.
  - **Khi hủy đi, bộ nhớ không được thu hồi!!**
  - Giải pháp:
    - Xây dựng phương thức thu hồi.  
→ Người dùng quên gọi!
    - Làm “khai tử” cho đối tượng.





# Hàm hủy (Destructor)

- Dọn dẹp 1 đối tượng **trước khi** nó được thu hồi.
- Hàm huỷ không có giá trị trả về, và không thể định nghĩa lại (nó không bao giờ có tham số)
  - mỗi lớp chỉ có duy nhất 1 hàm huỷ (destructor)
- Không gọi trực tiếp, sẽ được tự động gọi khi hủy bỏ đối tượng.
- **Thu hồi vùng nhớ** cho các *dữ liệu thành viên* là *con trỏ*.
- Nếu không cung cấp destructor, C++ sẽ tự sinh một destructor rỗng(không làm gì cả)

# Hàm hủy (Destructor)

- Tự động gọi khi đối tượng bị hủy.
- Mỗi lớp có duy nhất một hàm hủy -> không đa năng hóa hàm hủy
- Trong C++, hàm hủy có tên **<Tên lớp>**

```
class HocSinh
{
    private:
        char    *m_hoTen;
        float   m_diemVan;
        float   m_diemToan;
    public:
        ~HocSinh() { delete m_hoTen; }
};

int main()
{
    HocSinh     h;
    HocSinh     *p = new HocSinh;
    delete p;
    return 0;
}
```

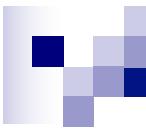
# Hàm hủy (Destructor)

## ■ Ví dụ:

```
class Set {  
private:  
    int *elems;  
    int maxCard;  
    int card;  
public:  
    Set(const int size) { ..... }  
    ~Set() { delete[] elems; }  
    ....  
};
```

```
Set TestFunct1(Set s1){  
    Set *s = new Set(50);  
    return *s;  
}  
  
void main()  
{  
    Set s1(40), s2(50);  
    s2 = TestFunct1(s1);  
}
```

Tổng cộng  
có bao  
nhiêu lần  
hàm hủy  
được gọi ?



# Hàm dựng – Hàm huỷ

## ■ Hàm dựng:

- Khởi tạo thông tin ban đầu cho đối tượng.
- Tự động gọi khi định nghĩa đối tượng.
- Mỗi lớp có thể có nhiều hàm dựng => có thể đa năng hóa hàm dựng
  - Hàm dung mặc định
  - Hàm dung có tham số
  - Hàm dung sao chép

## ■ Hàm hủy:

- Dọn dẹp bộ nhớ cho đối tượng.
- Tự động gọi khi đối tượng bị hủy.
- Mỗi lớp có duy nhất một hàm hủy => không thể đa năng hóa hàm dựng



# Friend – Đặt vấn đề

Tập Các  
Số Nguyên

```
class IntSet {  
public:  
    //...  
    void SetToReal (RealSet&);  
private:  
    int elems[maxCard];  
    int card;  
};  
  
class RealSet {  
public:  
    //...  
private:  
    float elems[maxCard];  
    int card;  
};
```

Hàm SetToReal  
dùng để chuyển  
tập số nguyên  
thành tập số thực

```
void IntSet::SetToReal (RealSet &set) {  
    set.card = card;  
    for (register i = 0; i < card; ++i)  
        set.elems[i] = (float) elems[i];  
}
```



Làm thế nào  
để thực hiện  
được việc truy  
xuat  
đến thành viên  
**Private** ?

# Friend

- Cách 1: Khai báo hàm thành viên của lớp IntSet là **bạn (friend)** của lớp RealSet.

```
class IntSet {  
public:  
    //...  
    void SetToReal (RealSet&);  
private:  
    int elems[maxCard];  
    int card;  
};  
class RealSet {  
public:  
    //...  
    friend void IntSet::SetToReal (RealSet&);  
private:  
    float elems[maxCard];  
    int card;  
};
```

# Friend

## ■ Cách 2:

- Chuyển hàm SetToReal ra ngoài (**độc lập**).
- Khai báo hàm đó là **bạn** của cả 2 lớp.

```
class IntSet {  
public:  
    //...  
    friend void SetToReal (IntSet &, RealSet&);  
private:  
    int elems[maxCard];  
    int card;  
};  
class RealSet {  
public:  
    //...  
    friend void SetToReal (IntSet &, RealSet&);  
private:  
    float elems[maxCard];  
    int card;  
};
```

```
void SetToReal (IntSet& iSet,  
                RealSet& rSet )  
{  
    rSet.card = iSet.card;  
    for (int i = 0; i < iSet.card; ++i)  
        rSet.elems[i] =  
            (float) iSet.elems[i];  
}
```

Hàm độc lập  
là bạn(friend)  
của cả 2 lớp.

# Friend

## ■ Hàm bạn:

- Có quyền truy xuất đến tất cả các dữ liệu và hàm thành viên (protected + private) của 1 lớp.
- Lý do:
  - Cách định nghĩa hàm chính xác.
  - Hàm cài đặt không hiệu quả.

## ■ Lớp bạn:

- Tất cả các hàm trong lớp bạn: là hàm bạn.

```
class A;  
class B { // .....  
    friend class A;  
};
```

```
class IntSet { ..... }  
class RealSet { // .....  
    friend class IntSet;  
};
```

# friend – khai báo forward

- Lưu ý khi khai báo 01 phương thức là friend:
  - Khai báo phương thức **SetToReal** (**RealSet&**) là friend của **RealSet**

```
class RealSet{  
    public:  
        friend void IntSet::SetToReal (RealSet&);  
    private:  
        //...  
};
```

- Khi xử lý phần này, trình biên dịch cần phải biết là đã có lớp **IntSet**
- Tuy nhiên các phương thức của **IntSet** lại dùng đến **RealSet** nên phải có lớp **RealSet** trước khi định nghĩa **IntSet**
- Cho nên ta không thể tạo **IntSet** khi chưa tạo **RealSet** và không thể tạo **RealSet** khi chưa tạo **IntSet**

# friend – khai báo forward

## ■ Giải pháp:

- sử dụng **khai báo forward** (forward declaration) cho lớp cấp quan hệ **friend** (trong ví dụ là **RealSet**)
- Ta khai báo các lớp trong ví dụ như sau:

```
Class RealSet; // Forward declaration
class IntSet
{
    public: void SetToReal (RealSet&);
    private:
    //...
};

class RealSet
{
    public: friend void IntSet::SetToReal (RealSet&);
    private:
    //...
};
```

# friend – khai báo forward

- Tuy nhiên, không thể làm ngược lại (khai báo forward cho lớp IntSet)

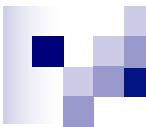
```
class IntSet; // Forward declaration
class RealSet {
    public:
        friend void IntSet::SetToReal (RealSet&);
    private:
        ...
};

class IntSet {
    public:
        void SetToReal (RealSet&);
    private:
        ...
};
```

Trình biên dịch chưa biết **SetToReal**

# friend – khai báo forward

- Lý do: trình biên dịch phải nhìn thấy khai báo phương thức trong *lớp nhận* trước khi tạo mối quan hệ **friend** tại lớp *cho* (granting class)
  - Trong ví dụ, trình biên dịch phải biết khai báo IntSet::**SetToReal** (RealSet&) tại khai báo của **IntSet** trước khi có thể tạo mối quan hệ **friend** của IntSet::**SetToReal** (RealSet&) với **RealSet**
  - Khai báo forward cho một lớp chỉ cho trình biên dịch biết về sự có mặt của lớp mà không cho biết về các thành viên của lớp đó
- Vậy: cần khai báo forward cho *lớp cấp quyền friend*
  - trong ví dụ trên là **RealSet**

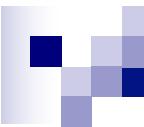


# friend – Ví dụ

- Khai báo hàm nhân ma trận với vecto, sử dụng hàm bạn và không sử dụng hàm bạn

```
const int N = 4;
class Vector
{
    double a[N];
public:
    double Get(int i) const {return a[i];}
    void Set(int i, double x) {a[i] = x;}
};

class Matrix
{
    double a[N][N];
public:
    double Get(int i, int j) const {return a[i][j];}
    void Set(int i, int j, double x) {a[i][j] = x;}
};
```



# friend – Ví dụ

- Khai báo hàm nhân ma trận với vecto không dùng hàm bạn

```
Vector Multiply(const Matrix &m, const Vector &v)
{
    Vector r;
    for (int i = 0; i < N; i++)
    {
        r.Set(i, 0);
        for (int j = 0; j < N; j++)
            r.Set(i, r.Get(i) + m.Get(i, j) * v.Get(j));
    }
    return r;
}
```

# friend – Ví dụ

- Khai báo hàm nhân ma trận với vecto có dùng hàm bạn

```
const int N = 4;  
class Matrix; // khai báo forward  
class Vector {  
    double a[N];  
public:  
    double Get(int i) const {return a[i];}  
    void Set(int i, double x) {a[i] = x;}  
    friend Vector Multiply(const Matrix &m, const Vector &v);  
};  
class Matrix {  
    double a[N][N];  
public:  
    double Get(int i, int j) const {return a[i][j];}  
    void Set(int i, int j, double x) {a[i][j] = x;}  
    friend Vector Multiply(const Matrix &m, const Vector &v);  
};
```

# friend – Ví dụ

```
Vector Multiply(const Matrix &m, const Vector &v)
{
    Vector r;
    for (int i = 0; i < N; i++)
    {
        r.a[i] = 0;
        for (int j = 0; j < N; j++)
            r.a[i] += m.a[i][j]*v.a[j];
    }
    return r;
}
```

## **KHỞI TẠO THÀNH VIÊN DỮ LIỆU**

- Có 2 cách khởi tạo cho thành viên dữ liệu ở hàm dựng:
  - Sử dụng phép gán trong thân hàm dựng;
  - Sử dụng 1 danh sách khởi tạo thành viên (member initialization list) trong định nghĩa hàm dựng
    - thành viên được khởi tạo trước khi thân hàm dựng được thực hiện.

# Danh sách khởi tạo thành viên

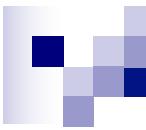
- Tương đương việc gán giá trị dữ liệu thành viên.

```
class Point {  
    int xVal, yVal;  
public:  
    Point (int x, int y) {  
        xVal = x;  
        yVal = y;  
    }  
    // .....  
};
```

```
Point::Point (int x, int y)  
    : xVal(x), yVal(y)  
{ }
```

```
class Image {  
public:  
    Image(const int w, const int h);  
private:  
    int width;  
    int height;  
    //...  
};  
Image::Image(const int w, const int h) {  
    width = w;  
    height = h;  
    //.....  
}
```

```
Image::Image (const int w, const int h)  
    : width(w), height(h)  
{ //..... }
```



# Thành viên hằng

## ■ Thành viên dữ liệu hằng:

- Khi một thành viên dữ liệu được khai báo là `const`, thành viên đó sẽ giữ nguyên giá trị trong suốt thời gian sống của đối tượng chủ.

```
class Image {  
public:  
    Image(const int w, const int h);  
private:  
    const int width;  
    const int height;  
    //...  
};
```

Khai báo bình thường  
như dữ liệu thành viên

Khởi tạo  
SAI

```
class Image {  
    const int width = 256;  
    const int height = 168;  
    //...  
};
```

```
Image::Image (const int w, const int h)  
    : width(w), height(h)  
{  
    //.....  
}
```

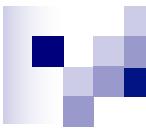
Khởi tạo ĐÚNG  
thông qua danh sách  
khởi tạo thành viên

# Thành viên hằng

- Hằng đối tượng: không được thay đổi giá trị.
- Hàm thành viên hằng:
  - Được phép gọi trên hằng đối tượng (đảm bảo không thay đổi giá trị của đối tượng chủ)
  - Không được thay đổi giá trị dữ liệu thành viên.
- nên khai báo mọi phương thức truy vấn là hằng, vừa để bão với trình biên dịch, vừa để tư gợi nhớ.

```
class Set {  
public:  
    Set(void){ card = 0; }  
    Bool Member(const int) const;  
    void AddElem(const int);  
    //...  
};  
Bool Set::Member (const int elem) const  
{    //...  
}
```

```
void main() {  
    const Set s;  
    s.AddElem(10); // SAI  
    s.Member(10); // OK  
}
```

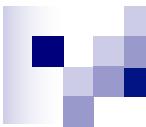


# Thành viên hằng

```
inline char *strdup(const char *s) {  
    return strcpy(new char[strlen(s) + 1], s);  
}  
  
class string{  
    char *p;  
public:  
    string(char *s = "") {p = strdup(s);}  
    ~string() {delete [] p;}  
    string(const string &s2) {p = strdup(s2.p);}  
    void Output() const {cout << p;}  
    void ToLower() {strlwr(p);}  
};
```

# Thành viên hằng

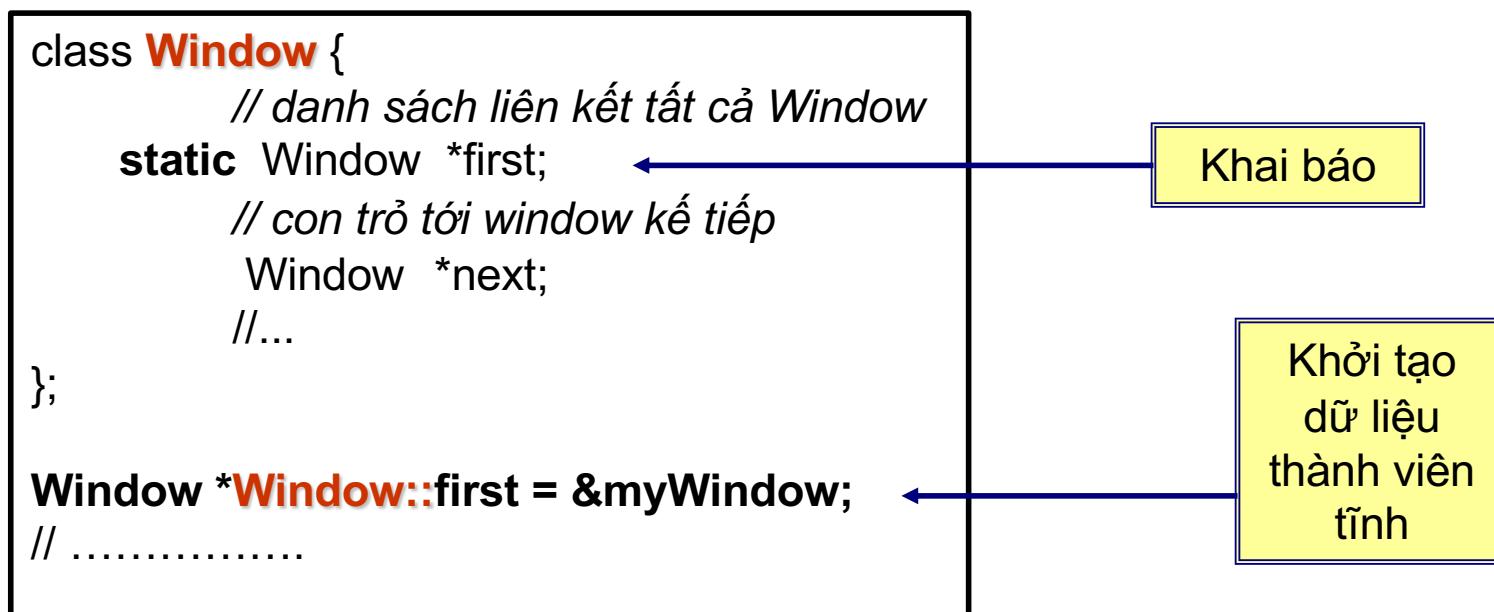
```
void main()
{
    const string Truong("DH BC TDT");
    string s("ABCdef");
    s.Output();
    s.ToLower();
    s.Output();
    Truong.Output();
    Truong.ToLower(); // Bao loi
}
```

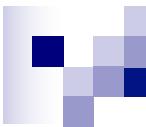


# Thành viên tĩnh

## ■ Thành viên dữ liệu tĩnh:

- Dùng chung 1 bản sao chép (1 vùng nhớ) chia sẻ cho tất cả đối tượng của lớp đó.
- Sử dụng: <TênLớp>::<TênDữLiệuThànhViên>
- Thường dùng để đếm số lượng đối tượng.

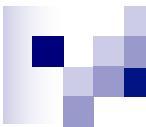




# Thành viên tĩnh - Ví dụ

- Đếm số đối tượng MyClass
  - khai báo lớp MyClass

```
class MyClass
{
    public:
        MyClass(); // Constructor
        ~MyClass(); // Destructor
        void printCount(); //Output current value of count
    private:
        //static member to store
        //number of instances of MyClass
        static int count;
};
```



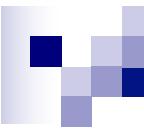
# Thành viên tĩnh - Ví dụ

## ■ Đếm số đối tượng MyClass

- Cài đặt các phương thức

```
int MyClass::count=0;  
MyClass::MyClass() {  
    this->count++; //Increment the static count  
}  
MyClass::~MyClass() {  
    this->count--; //Decrement the static count  
}  
void MyClass::printCount() {  
    cout << "There are currently " << this->count  
        << " instance(s) of MyClass.\n";  
}
```

*Khởi tạo biến đếm bằng 0 vì ban đầu không có đối tượng nào*



# Thành viên tĩnh

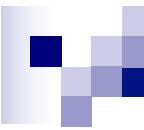
Định nghĩa và khởi tạo

- thành viên tĩnh được lưu trữ độc lập với các thể hiện của lớp, do đó, các thành viên tĩnh phải được định nghĩa

```
int MyClass::count;
```

- ta thường định nghĩa các thành viên tĩnh trong file chưa định nghĩa các phương thức
- nếu muốn khởi tạo giá trị cho thành viên tĩnh ta cho giá trị khởi tạo tại định nghĩa

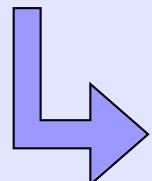
```
int MyClass::count = 0;
```



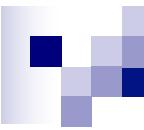
# Thành viên tĩnh

## ■ chương trình demo sử dụng MyClass

```
int main()
{
    MyClass* x = new MyClass;
    x->PrintCount();
    MyClass* y = new MyClass;
    x->PrintCount();
    y->PrintCount();
    delete x;
    y->PrintCount();
}
```



```
There are currently 1 instance(s) of MyClass.  
There are currently 2 instance(s) of MyClass.  
There are currently 2 instance(s) of MyClass.  
There are currently 1 instance(s) of MyClass.
```



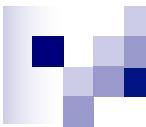
# Thành viên hằng tĩnh

- Kết hợp hai từ khoá const và static, ta có hiệu quả kết hợp
  - một thành viên dữ liệu được định nghĩa là static const là một hằng được chia sẻ giữa tất cả các đối tượng của một lớp.
- Không như các thành viên khác, các thành viên static const phải được khởi tạo khi khai báo

```
class MyClass {  
public:  
    MyClass();  
    ~MyClass();  
private:  
    static const int thirteen=13;  
};
```

```
int main() {  
    MyClass x;  
    MyClass y;  
    MyClass z;  
}
```

x, y, z dùng chung một thành viên  
**thirteen** có giá trị không đổi là 13



# Thành viên hằng tĩnh

## ■ Tóm lại, ta nên khai báo:

### **static**

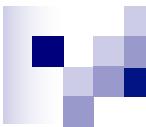
- đối với các thành viên dữ liệu ta muốn dùng chung cho mọi thể hiện của một lớp

### **const**

- đối với các thành viên dữ liệu cần giữ nguyên giá trị trong suốt thời gian sống của một thể hiện

### **static const**

- đối với các thành viên dữ liệu cần giữ nguyên cùng một giá trị tại tất cả các đối tượng của một lớp



# Thành viên tĩnh

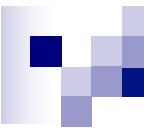
## ■ Hàm thành viên tĩnh:

- Tương đương với hàm toàn cục.
- Phương thức tĩnh không được truyền con trỏ this làm tham số ẩn.
- Không thể sửa đổi các thành viên dữ liệu từ trong phương thức tĩnh.
- Gọi thông qua: <TênLớp>::<TênHàm>

```
class Window {  
    // .....  
    static void PaintProc () { ..... }  
    // .....  
};  
void main() {  
    // .....  
    Window::PaintProc();  
}
```

Khai báo  
Định nghĩa  
hàm thành  
viên tĩnh

Truy xuất  
hàm thành  
viên tĩnh



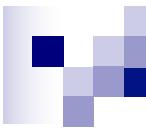
# Thành viên tĩnh

## ■ Hàm thành viên tĩnh: ví dụ

```
class MyClass {  
public:  
    MyClass(); // Constructor  
    ~MyClass(); // Destructor  
    static void printCount(); //Output current value of count  
private:  
    static int count; // count  
};
```

```
int main()  
{  
    MyClass::printCount();  
    MyClass* x = new MyClass;  
    x->printCount();  
    MyClass* y = new MyClass;  
    x->printCount();  
    y->printCount();  
    delete x;  
    MyClass::printCount();  
}
```

There are currently 0 instance(s) of MyClass.  
There are currently 1 instance(s) of MyClass.  
There are currently 2 instance(s) of MyClass.  
There are currently 2 instance(s) of MyClass.  
There are currently 1 instance(s) of MyClass.



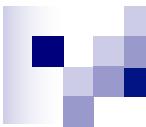
# Thành viên tĩnh

## ■ Ví dụ:

```
class CDate{
    static int dayTab[13];
    int day, month, year;
public:
    CDate(int d=1, int m=1, int y=2010);
    static bool LeapYear(int y) {
        return (y%400 == 0 || (y%4==0 && y%100 != 0));
    }
    static int DayOfMonth(int m, int y);
    static bool ValidDate(int d, int m, int y);
    void Input();
};

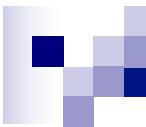
int CDate::dayTab[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};

CDate::CDate(int d, int m, int y){
    if (ValidDate(d,m,y)){
        day=d; month=m; year=y;
    }
}
```



# Thành viên tĩnh

```
int CDate::DayOfMonth(int m, int y){  
    dayTab[2]= LeapYear(y)?29:28;  
    return dayTab[m];  
}  
  
bool betw(int x, int a, int b){  
    return x >= a && x <= b;  
}  
  
bool CDate::ValidDate(int d, int m, int y){  
    return betw(m,1,12) && betw(d,1,DayOfMonth(m,y));  
}  
  
void CDate::Input(){  
    int d,m,y;  
    cout<<"Nhập ngày tháng năm:"; cin >> d >> m >> y;  
    while (!ValidDate(d,m,y))  
    {  
        cout << "Please enter a valid date: ";  
        cin >> d >> m >> y;  
    }  
    day = d; month = m; year = y;  
}
```



# Thành viên tham chiếu

## ■ Tham chiếu dữ liệu thành viên:

```
class Image {  
    int width;  
    int height;  
    int &widthRef;  
    //...  
};
```

Khai báo bình thường  
như dữ liệu thành viên

```
class Image {  
    int width;  
    int height;  
    int &widthRef = width;  
    //...  
};
```

Khởi tạo  
**SAI**

```
Image::Image (const int w, const int h)  
    : widthRef(width)  
{ //..... }
```

Khởi tạo **ĐÚNG**  
thông qua danh sách  
khởi tạo thành viên

# Thành viên là đối tượng của 1 lớp

## ■ Dữ liệu thành viên có thể có kiểu:

- Dữ liệu (lớp) chuẩn của ngôn ngữ.
- Lớp do người dùng định nghĩa (có thể là chính lớp đó).

```
class Point { ..... };
class Rectangle {
    public:
        Rectangle (int left, int top, int right, int bottom);
        //...
    private:
        Point topLeft;
        Point botRight;
};
Rectangle::Rectangle (int left, int top, int right, int bottom)
    :topLeft(left, top), botRight(right, bottom)
{}
```

Khởi tạo cho các  
dữ liệu thành viên  
qua danh sách khởi  
tạo thành viên

# Thành viên là đối tượng của 1 lớp

## ■ Ví dụ:

```
class Diem {  
    double x,y;  
public:  
    Diem(double xx, double yy) {x = xx; y = yy;}  
    // ...  
};  
class TamGiac {  
    Diem A,B,C;  
public:  
    void Ve() const;  
    // ...  
};  
TamGiac t;
```



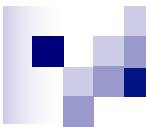
**Báo lỗi**

```
class TamGiac {  
    Diem A,B,C;  
public:  
implicit default constructor for 'TamGiac' must explicitly initialize the member 'A' which does not have a default constructor  
implicit default constructor for 'TamGiac' must explicitly initialize the member 'B' which does not have a default constructor  
implicit default constructor for 'TamGiac' must explicitly initialize the member 'C' which does not have a default constructor
```

# Thành viên là đối tượng của 1 lớp

## ■ Ví dụ:

```
class Diem {  
    double x,y;  
public:  
    Diem(double xx, double yy) {x = xx; y = yy;}  
    // ...  
};  
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){ }  
    void Ve() const;  
    // ...  
};  
TamGiac t(100,100,200,400,300,300);
```



# Mảng các đối tượng

- Sử dụng **hàm xây dựng không đối số** (hàm xây dựng mặc nhiên - default constructor).

VD: Point pentagon[5];

- Sử dụng bộ khởi tạo mảng:

VD: Point triangle[3] =  
    { Point(4,8), Point(10,20), Point(35,15) };

Ngắn gọn:

Set s[4] = { 10, 20, 30, 40 };

tương đương với:

Set s[4] = { Set(10), Set(20), Set(30), Set(40) };

# Mảng các đối tượng

## ■ Sử dụng dạng con trỏ:

- Cấp vùng nhớ:

VD: Point \*pentagon = new Point[5];

- Thu hồi vùng nhớ:

delete[] pentagon;

delete pentagon; // Thu hồi vùng nhớ đầu

```
class Polygon {
    public:
        //...
    private:
        Point *vertices; // các đỉnh
        int nVertices; // số các đỉnh
};
```

Không cần biết kích  
thước mảng.

# Phạm vi lớp

## ■ Thành viên trong 1 lớp:

- Che các thực thể trùng tên trong phạm vi.

```
// .....
int fork (void);           // fork hệ thống
class Process {
    int fork (void); // fork thành viên
    //...
};
```

fork thành viên  
che đi fork toàn cục  
trong phạm vi lớp  
Process

```
// .....
int Process::func1 (void)
{
    int x = fork(); // gọi fork cục bộ
    int pid = ::fork(); // gọi hàm fork hệ thống
    //...
}
```

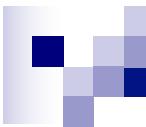
# Phạm vi lớp

- Lớp toàn cục: đại đa số lớp trong C++.
- Lớp lồng nhau: lớp chứa đựng lớp.
- Lớp cục bộ: trong 1 hàm hoặc 1 khối.

```
class Rectangle { // Lớp lồng nhau
public:
    Rectangle (int, int, int, int);
    //..
private:
    class Point {
        public:
            Point(int a, int b) { ... }
        private:
            int x, y;
    };
    Point topLeft, botRight;
};

Rectangle::Point pt(1,1); // sd ở ngoài
```

```
void Render (Image &i)
{
    class ColorTable {
public:
    ColorTable () { /* ... */ }
    AddEntry (int r, int g, int b)
        { /* ... */ }
    //...
    };
    ColorTable colors;
    //...
}
ColorTable ct; // SAI
```



# Cấu trúc và hợp

## ■ Cấu trúc (structure):

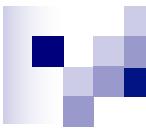
- Bắt nguồn từ ngôn ngữ C.
- Tương đương với **class** với các thuộc tính là public.
- Sử dụng như **class**.

```
struct Point {  
    Point (int, int);  
    void OffsetPt(int, int);  
    int x, y;  
};
```

```
class Point {  
public:  
    Point(int, int);  
    void OffsetPt(int, int);  
    int x, y;  
};
```

```
Point p = { 10, 20 };
```

Có thể khởi tạo dạng này  
nếu không có định nghĩa  
hàm xây dựng



# Cấu trúc và hợp

## ■ Hợp (union):

- Tất cả thành viên ánh xạ đến cùng 1 địa chỉ bên trong đối tượng chính nó (không liên tiếp).
- Kích thước = kích thước của dữ liệu lớn nhất.

```
union Value {  
    long integer;  
    double real;  
    char *string;  
    Pair list;  
    //...  
};
```

```
class Pair {  
    Value *head;  
    Value *tail;  
    //...  
};
```

```
class Object {  
private:  
    enum ObjType {intObj, realObj,  
                  strObj, listObj};  
    ObjType type; // kiểu đối tượng  
    Value val; // giá trị của đối tượng  
    //...  
};
```

Kích thước của Value là  
8 bytes = sizeof(double)

# Bài tập

## ■ Bài tập 3.1:

Xây dựng lớp **phân số** cho phép thực hiện các thao tác:

- 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.
- Nhập, xuất.
- Lấy tử số, mẫu số.
- Gán giá trị cho tử số, mẫu số.
- Nghịch đảo, rút gọn.
- Cộng, trừ, nhân, chia, so sánh với phân số khác.

# Bài tập

## ■ Bài tập 3.2:

Xây dựng lớp **số phức** cho phép thực hiện các thao tác:

- 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
- 
- Nhập, xuất.
  - Lấy phần thực, phần ảo.
  - Gán giá trị cho phần thực, phần ảo.
  - Tính module.
  - Cộng, trừ, nhân, chia, so sánh với số phức khác.

# Bài tập

## ■ Bài tập 3.3:

Xây dựng lớp **đơn thức** cho phép thực hiện các thao tác:

- Khởi tạo mặc định đơn thức = 0.
- Khởi tạo với hệ số và số mũ cho trước.
- Khởi tạo từ một đơn thức khác.
- Nhập, xuất đơn thức.
- Lấy hệ số, số mũ.
- Gán giá trị cho hệ số, số mũ.
- Tính giá trị, đạo hàm, nguyên hàm.
- Cộng, trừ, nhân, chia, so sánh với đơn thức khác cùng bậc.

# Bài tập

## ■ Bài tập 3.4:

Thông tin một học sinh bao gồm:

- Họ tên.
- Điểm văn, toán.

Xây dựng lớp **học sinh** cho phép thực hiện các thao tác:

- Khởi tạo với họ tên và điểm văn, toán cho trước.
- Khởi tạo từ một học sinh khác.
- Hủy đổi tượng học sinh, thu hồi bộ nhớ.
- Nhập, xuất.
- Lấy họ tên, điểm văn, toán.
- Gán giá trị cho họ tên, điểm văn, điểm toán.
- Tính điểm trung bình.
- Xếp loại theo tiêu chí:
  - Giỏi ( $\geq 8.0$ ), Khá ( $\geq 7.0$ ).
  - Trung bình ( $\geq 5.0$ ), Yếu ( $< 5$ ).

# Bài tập

## ■ Bài tập 3.5:

Xây dựng lớp **mảng** cấp phát động cho phép thực hiện các thao tác sau:

- 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.
- Nhập, xuất mảng.
- Lấy kích thước mảng.
- Lấy phần tử tại vị trí nào đó.
- Gán giá trị cho phần tử tại vị trí nào đó.
- Tìm phần tử nào đó trong mảng.
- Sắp xếp tăng, giảm.