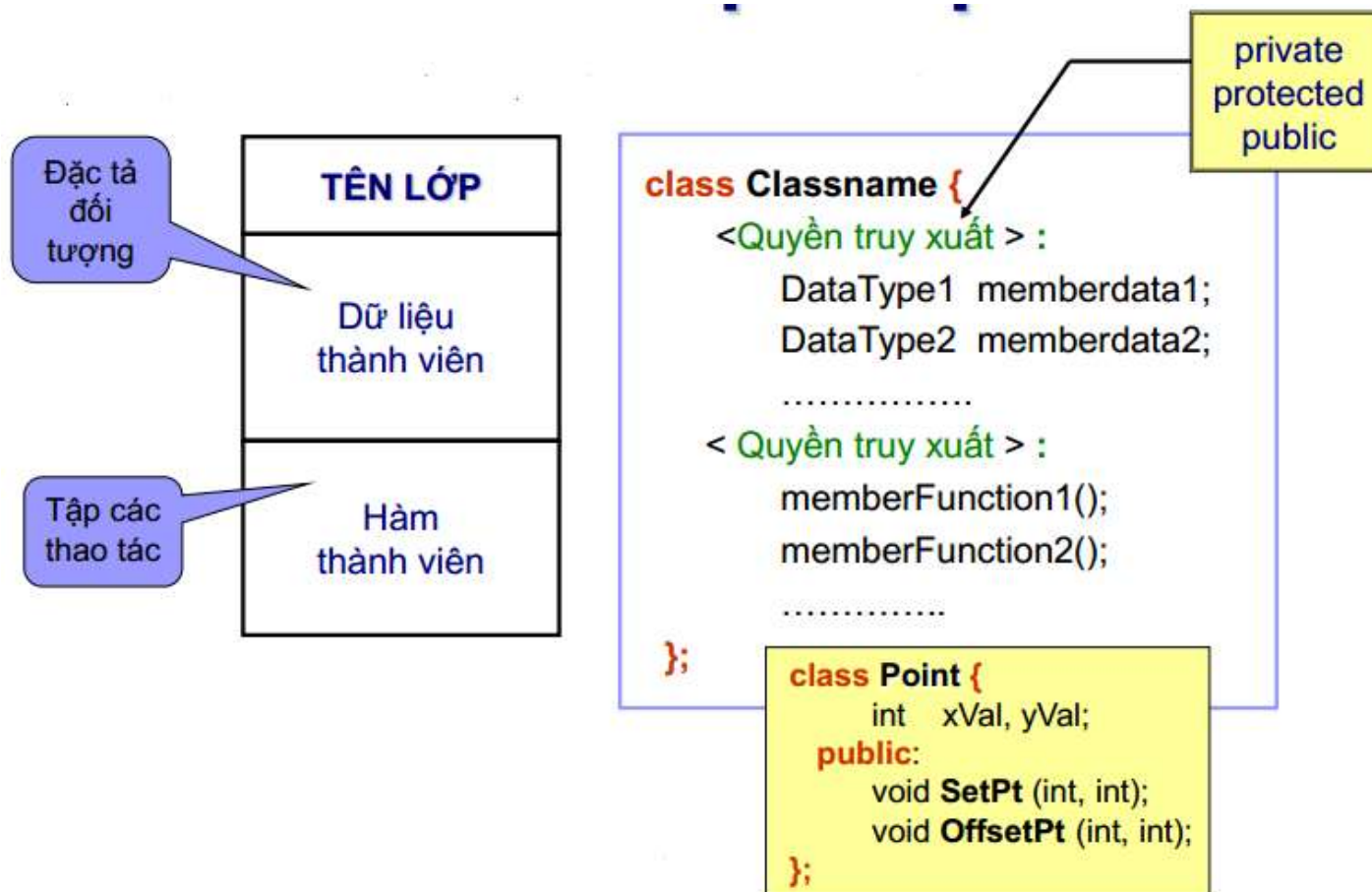




# LỚP & ĐỐI TƯỢNG (C++)



- **Lớp:** kiểu dữ liệu trừu tượng



- 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
}
```

- 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:

**Toán tử định phạm vi**

```
// car.cpp
...
void Car::drive(int speed, int distance)
{
    //method definition
}
```

**Tên lớp**

**Tên phương thức**

- Khai báo 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:

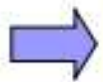
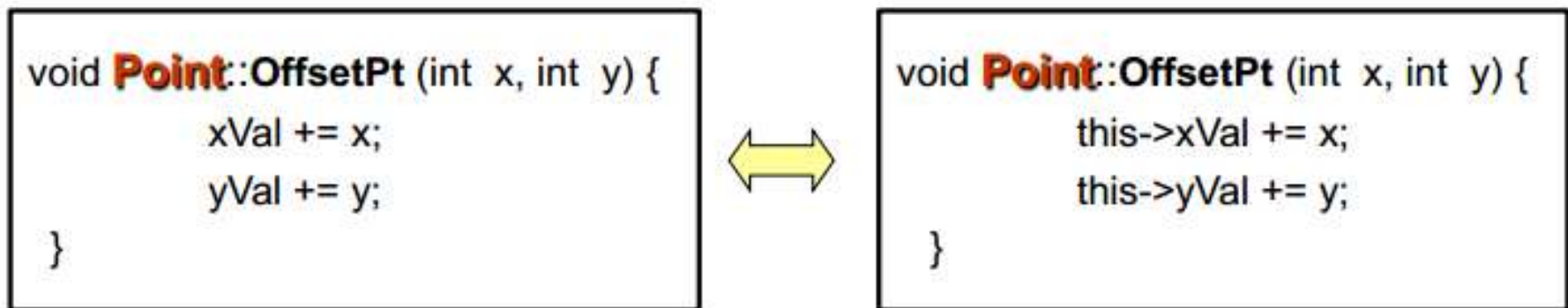
```
#include <iostream>
using namespace std;
class Point
{
    int xVal, yVal;
public:
    Point();
    ~Point();
    void Show()
    {
        cout << xVal << yVal;
    }
};
```

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

//Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;
void Point::Show()
{
    cout << xVal << yVal;
}
```



- Con trỏ **\*this**:
  - Là 1 thành viên ẩn, có thuộc tính là private;
  - Trỏ tới chính bản thân đối tượng.



- 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**

- 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 trỏ 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.



- Toán tử `::` dùng để xác định chính xác hàm (thuộc tính) được truy xuất thuộc lớp nào.

- Câu lệnh:

`pt.OffsetPt(2,2);`  $\leftrightarrow$  `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;
}
```

- 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!

## PhanSo

- Tử số??
- Mẫu số??

## HocSinh

- Họ tên??
- Điểm văn??
- Điểm toán??

**Hàm dựng ra đời!!**



# CONSTRUCTOR

- Dùng để **định nghĩa** và **khởi tạo** đối tượng cùng 1 lúc;
- Có tên 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 hoá) 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.



# DEFAULT CONSTRUCTOR

- Hàm dựng mặc định (default constructor):
  - Đối với constructor mặc định, 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
  - 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.



# 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;
}
```



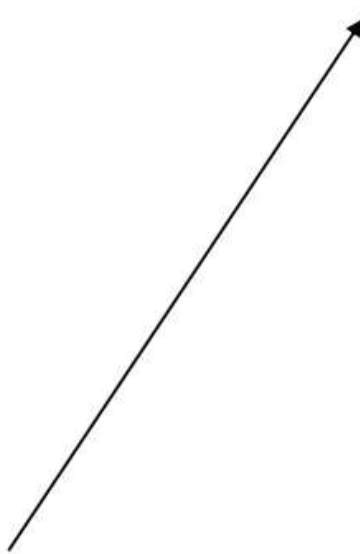
# COPY CONSTRUCTOR

- 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 → có thể định nghĩa lại copy constructor.

- Hàm dựng sao chép (copy constructor):

- Ví dụ:

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



Kiểu tham số là tham chiếu  
đến đối tượng kiểu Foo

tham số là đối tượng  
được sao chép

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



# 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;  
}
```

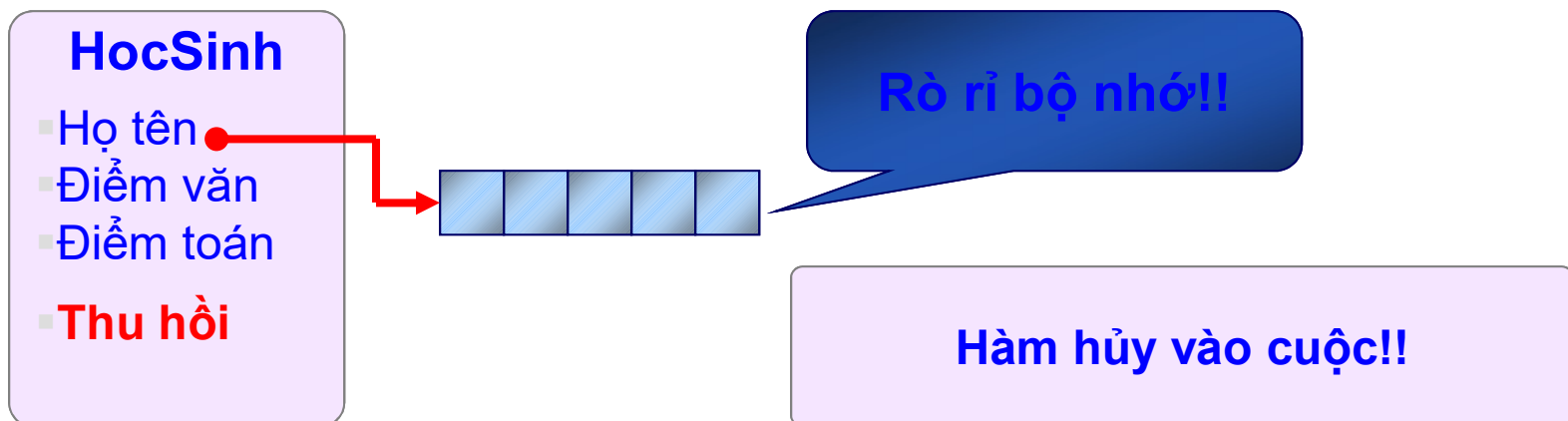


- Dr. Guru khuyên:
  - Một lớp nên có tối thiểu 3 hàm dựng:
    - Hàm dựng mặc định.
    - Hàm dựng có đầy đủ tham số.
    - Hàm dựng sao chép.





- 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.



- Dọn dẹp 1 đối tượng *trước khi* nó được thu hồi;
- Destructor không có giá trị trả về, và không thể định nghĩa lại (nó không bao giờ có tham số):
  - Mỗi lớp chỉ có 1 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 ta không cung cấp destructor, C++ sẽ tự sinh một destructor rỗng (không làm gì cả).

- Tính chất hàm hủy (destructor):
  - Tự động gọi khi đối tượng bị hủy.
  - Mỗi lớp có duy nhất một hàm hủy.
  - Trong C++, hàm hủy có tên  $\sim$ <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;
}
```

- Ví dụ:

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

```
Set TestFunc1(Set s1) {  
    Set *s = new Set(50);  
    return *s;  
}  
  
void main() {  
    Set s1(40), s2(50);  
    s2 = TestFunc1(s1);  
}
```

Tổng cộng  
có **bao  
nhiều lần**  
hàm hủy  
được gọi?



Tập Các  
Số Nguyên

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

Tập Các  
Số Thực

```
class RealSet {
public:
    //...
private:
    float elems[maxCard];
    int card;
};
```

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

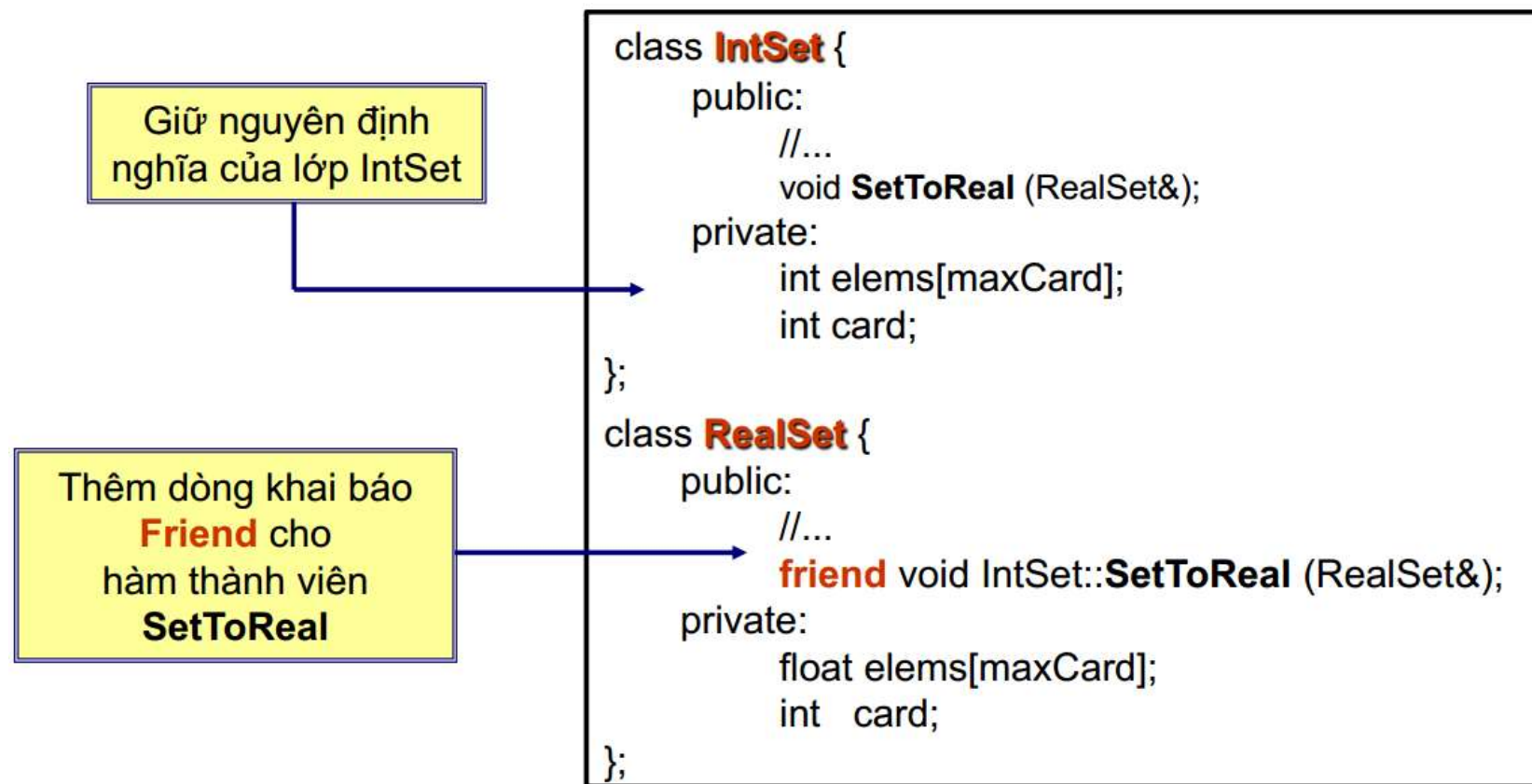
```
void IntSet::SetToReal (RealSet &set) {
    set.card = card;
    for (register i = 0; i < card; ++i)
        set.elems[i] = (float) elems[i];
}
```



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



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



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

<pre>class <b>IntSet</b> {     public:         //...     <b>friend</b> void <b>SetToReal</b> (IntSet &amp;, RealSet&amp;);     private:         int elems[maxCard];         int card; }; class <b>RealSet</b> {     public:         //...     <b>friend</b> void <b>SetToReal</b> (IntSet &amp;, RealSet&amp;);     private:         float elems[maxCard];         int card; };</pre>	<pre>void <b>SetToReal</b> (IntSet&amp; iSet,                 RealSet&amp; rSet ) {     rSet.card = iSet.card;     for (int i = 0; i &lt; iSet.card; ++i)         rSet.elems[i] =             (float) iSet.elems[i]; }</pre>
---	--

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

- 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;  
};
```

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

```
class RealSet
{
    public:
    friend void IntSet::SetToReal (RealSet&) ;
    private:
    //...
};
```
  - Khi xử lý, 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**

- Bài tập: Khai báo hàm nhân ma trận với vecto sử dụng hàm bạn & 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;}
};
```

- Bài tập:
  - Không sử dụng 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;
}
```

- Bài tập:
  - Sử dụng 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);
};
```



- Bài tập:
  - Sử dụng hàm bạn

```
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;
}
```



- Có 2 cách khởi tạo:
  - 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.


- Khởi tạo thành viên dữ liệu sử dụng phép gán trong thân hàm dựng

```
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;
}
```


- Tương đương với việc gán giá trị dữ liệu thành viên:

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



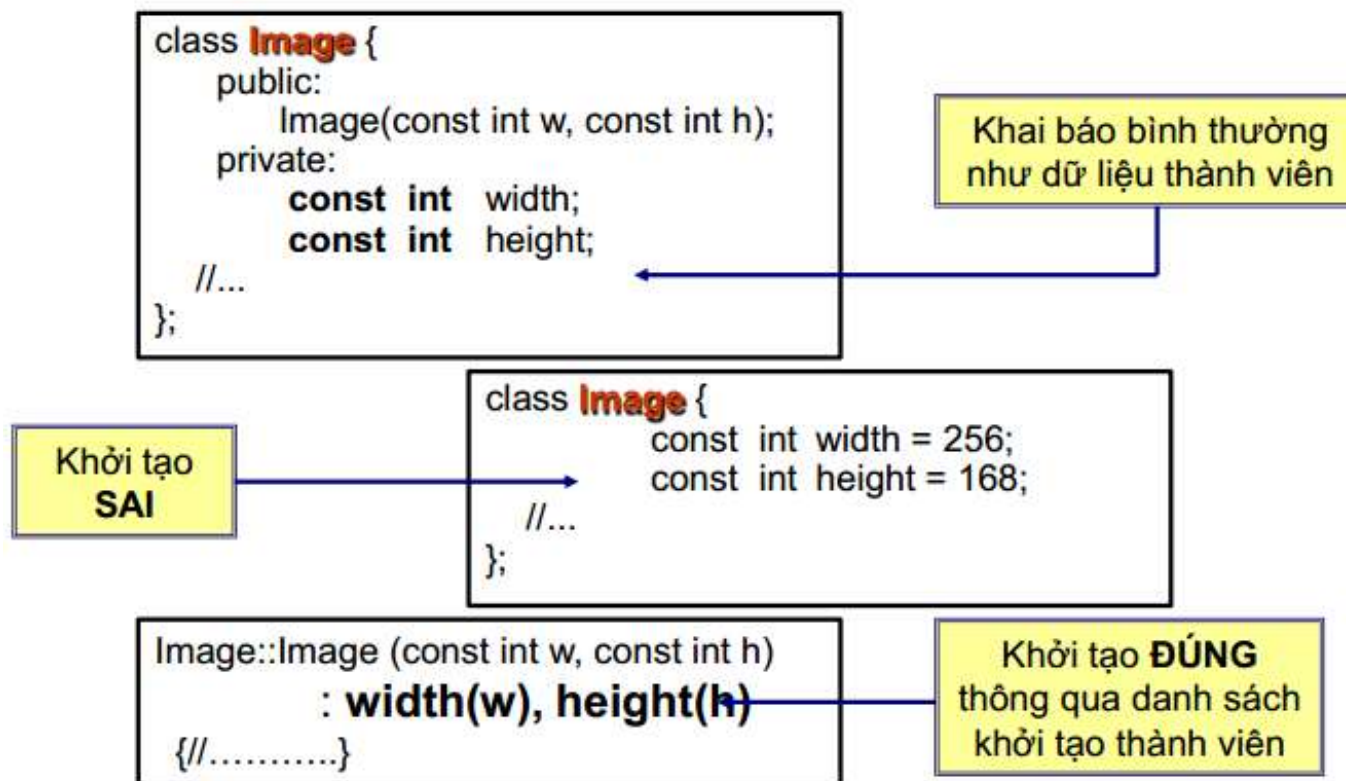
```
Point::Point (int x, int y)  
    : xVal(x), yVal(y)  
{ }
```

```
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)  
{ //..... }
```

- 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.



- 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 báo với trình biên dịch, vừa để tự gọi nhớ.



- Ví dụ:

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

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

- Ví dụ:

```
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);}
};
```

- Ví dụ:

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

- 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.

```
class Window {  
    // danh sách liên kết tất cả Window  
    static Window *first;  
    // con trỏ tới window kế tiếp  
    Window *next;  
    //...  
};  
  
Window *Window::first = &myWindow;  
// .....
```

Khai báo

Khởi tạo  
dữ liệu  
thành viên  
tĩnh

- Ví dụ: đếm số đối tượng MyClass
  - Khai báo lớp MyClass:

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



- Ví dụ: đếm số đối tượng MyClass
  - Cài đặt các phương thức lớp MyClass:

```
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.

- Định nghĩa & 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 chứa 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;
```

- Ví dụ:

```
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.

- 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**

Tóm lại, ta nên khai báo:

- **static:**
  - Đối với các thành viên dữ liệu ta muốn dùng chung cho mọi thể hiện (đối tượng) 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.



# 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::PainProc();  
}
```

Khai báo  
Định nghĩa  
hàm thành  
viên tĩnh

Truy xuất  
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.

- Ví dụ:

```
typedef int bool;
const bool false = 0, true = 1;
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=1, int m=1, int y=2010)
{    if (ValidDate(d,m,y)) {day=d;month=m;year=y;}    }
```



- Ví dụ: 

```
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;
    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;
}
```

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

Khai báo bình thường  
như dữ liệu thành viên

Khởi tạo  
**SAI**

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

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

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



- 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; //Error
```

- 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), (xB,yB), C(xC,yC) {}
    void Ve() const;
    //...
};
TamGiac t(100,100,200,400,300,300);
```

- Sử dụng hàm xây dựng không đối số (hàm xây dựng mặc nhiên - default constructor).
  - Ví dụ: `Point pentagon[5];`
- Sử dụng bộ khởi tạo mảng:
  - Ví dụ:  
`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) };`

- Sử dụng dạng con trỏ:

- Cấp vùng nhớ:

`Point *pentagon = new Point[5];`

- Thu hồi vùng nhớ:

`delete[] pentagon;`

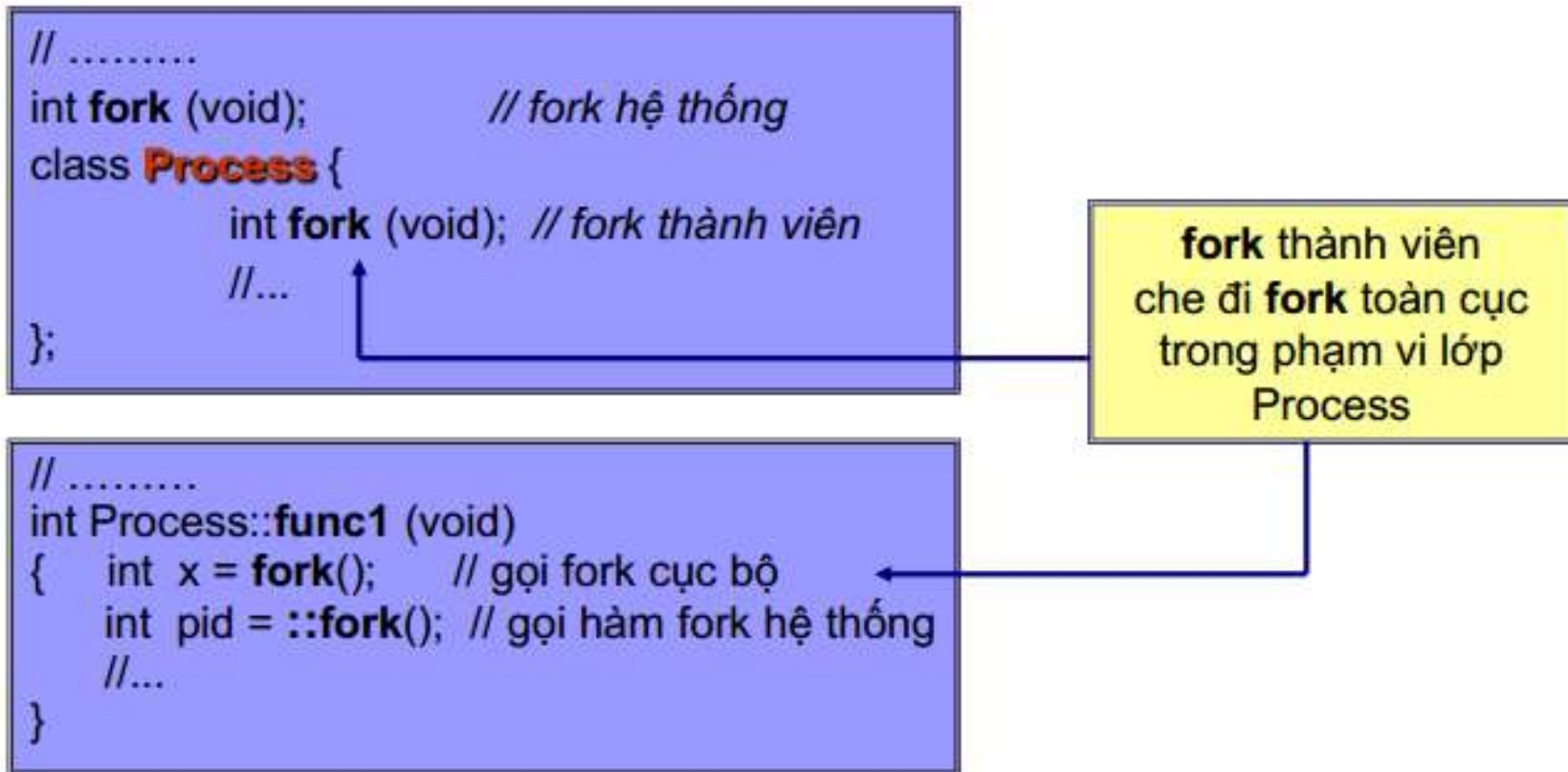
`delete pentagon; // Thu hồi vùng nhớ đầu`

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

Không cần biết kích  
thước mảng.



- Thành viên trong 1 lớp:
  - Che các thực thể trùng tên trong phạm vi.



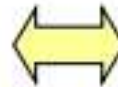
- Lớp toàn cục: đại đa số lớp trong C++;
- Lớp lồng nhau: lớp chứa đựng lớp;
- Lớp cục bộ: trong 1 hàm hoặc 1 khối.

```
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
```

- 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

- Struct:

- Giống như C:

```
struct Tên_kiểu_ct  
{  
    // Khai báo các thành phần của cấu trúc  
};
```

- Khai báo biến (struct):

- C: **struct Tên\_kiểu\_ct** danh sách biến, mảng cấu trúc;
    - C++: **Tên\_kiểu\_ct** danh sách biến, mảng cấu trúc;

- Struct:

- Ví dụ: Định nghĩa kiểu cấu trúc TS (thí sinh) gồm các thành phần : ht (họ tên), sobd (số báo danh), dt (điểm toán), dl (điểm lý), dh (điểm hoá) và td (tổng điểm), sau đó khai báo biến cấu trúc h và mảng cấu trúc ts.

```
struct TS
{
    char ht [25];
    long sobd;
    float dt, dl, dh, td;
};
TS h, ts[1000];
```



- 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)

- Union:
  - Giống như C:

```
union Tên_kiểu_hợp  
{  
    // Khai báo các thành phần của hợp  
};
```

- Khai báo biến (struct):
  - C: **union Tên\_kiểu\_hợp** danh sách biến, mảng kiểu **hợp**;
  - C++: **Tên\_kiểu\_ct** danh sách biến, mảng kiểu **hợp**;

- Union không tên:
  - C++ cho phép khai báo các union không tên:

```
union
```

```
{
```

```
// Khai báo các thành phần
```

```
} ;
```

→ Khi đó các thành phần (khai báo trong union) sẽ dùng chung một vùng nhớ → tiết kiệm bộ nhớ và cho phép dễ dàng tách các byte của một vùng nhớ.

- Union không tên:

- Ví dụ: nếu các biến nguyên i , biến ký tự ch và biến thực x không đồng thời sử dụng thì có thể khai báo chúng trong một union không tên như sau:

```
union
{
    int i ;
    char ch ;
    float x ;
};

union
{
    unsigned long u;
    unsigned char b[4];
};
```

- `u = 0xDDCCBBAA; // Số hệ 16 → b[4] ???`

Thank You !

