

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (C++)



Giảng viên:

Đặng Hoài Phương

Bộ môn:

Công nghệ phần mềm

Khoa:

Công nghệ Thông tin

Trường Đại học Bách Khoa

Đại học Đà Nẵng





GIỚI THIỆU

- Đặng Hoài Phương.
- E-mail: danghoaiphuongdn@gmail.com
- Tel: 0935578555
- Khoa Công nghệ thông tin, trường Đại học
Bách khoa, Đại học Đà Nẵng.



NỘI DUNG MÔN HỌC

Chương 1: Extern C++.

Chương 2: OOP.

Chương 3: Operator Overloading.

Chương 4: Inheritance.

Chương 5: Polymorphism.

Chương 6: Template.

Chương 7: Exception.



TÀI LIỆU THAM KHẢO

1. Đặng Hoài Phương, *Slide bài giảng Lập trình hướng đối tượng*, Khoa CNTT, trường Đại học Bách Khoa, Đại học Đà Nẵng, 2018.
2. Lê Thị Mỹ Hạnh, *Giáo trình Lập trình hướng đối tượng*, Khoa CNTT, trường Đại học Bách khoa, Đại học Đà Nẵng.
3. Phạm Văn Ất, *C++ và Lập trình hướng đối tượng*, NXB Giao thông vận tải.
4. Bruce Eckel, *Thinking in C++*. Second Edition. MindView Inc., 2000.



ĐÁNH GIÁ & YÊU CẦU

- Chuyên cần và bài tập: 20%
 - Làm 03 bài tập về nhà;
 - Tham gia 04 bài test kiểm tra.
- Thi giữa kỳ: 20%
 - Trắc nghiệm trên máy tính.
- Thi cuối kỳ: 60%
 - Trắc nghiệm trên máy tính.
- Yêu cầu:
 - Mang theo laptop và dây cắm nối mỗi buổi học;
 - Nộp đủ bài tập và tham gia đủ các bài test kiểm tra → **SEE YOU NEXT TIME** nếu thiếu 01 bài tập hoặc bài test kiểm tra.



CHƯƠNG 1

EXTERN C++





PROGRAMMING LANGUAGE

- Programming Language: là ngôn ngữ dùng để diễn tả thuật toán sao cho máy tính hiểu và thực hiện được.
- 2 loại:
 - Ngôn ngữ lập trình bậc thấp (low-level programming language):
 - Mã máy;
 - Hợp ngữ (Assembly).
 - Ngôn ngữ bậc cao (high-level programming language):
 - FORTRAN (1956);
 - PASCAL (1970), C (1972), C++ (1980), Java (1995), C# (2001), Python (1990), Swift (2014), Kotlin (2017), ...
 - **FRAMEWORK & SOFTWARE LIBRARIES**



NON-STRUCTURED PROGRAMMING

- Là phương pháp xuất hiện đầu tiên.
- Giải quyết các bài toán nhỏ, tương đối đơn giản (lĩnh vực tính toán).

```
10      k=1
20      gosub 100
30      if y > 120 goto 60
40      k = k + 1
50      goto 20
60      print k, y
70      stop
100     y = 3*k*k + 7*k - 3
110     return
```

Lệnh nhảy đến vị trí bất kỳ trong chương trình

- Đặc trưng:
 - Đơn giản;
 - Đơn luồng;
 - Sử dụng biến toàn cục, lạm dụng lệnh GOTO;
 - Chỉ gồm một chương trình chính.



PROCEDURE-ORIENTED PROGRAMMING (POP)

- Tổ chức chương trình thành các **chương trình con**
 - PASCAL: thủ tục & hàm, C: hàm.
 - Chương trình hướng cấu trúc = cấu trúc dữ liệu + tập hợp hàm.
 - Trừu tượng hóa chức năng (Functional Abstraction):
 - Không quan tâm đến cấu trúc hàm;
 - Chỉ cần biết kết quả thực hiện của hàm.
- Nền tảng của lập trình hướng cấu trúc.



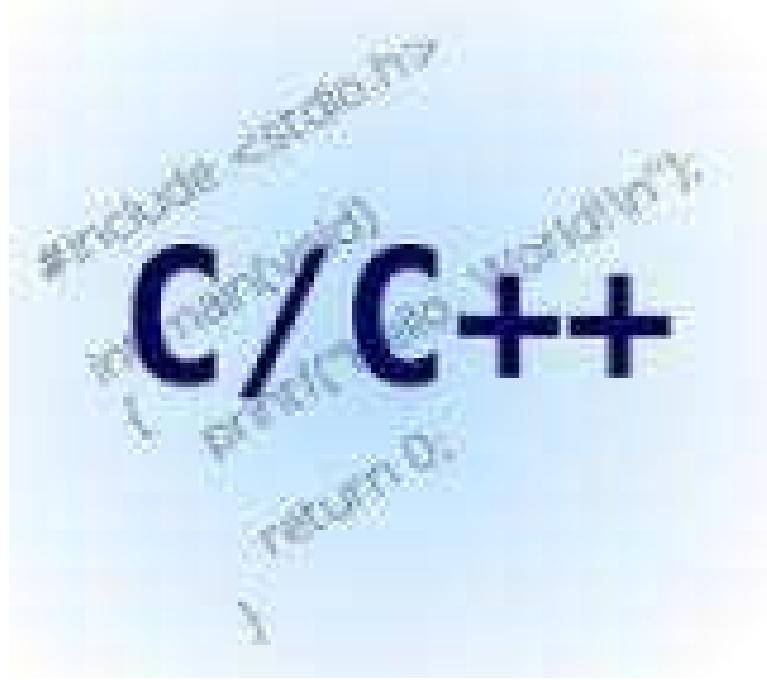
PROCEDURE-ORIENTED PROGRAMMING (POP)

- Ví dụ:

```
int func(int j)
{
    return (3*j*j + 7*j-3);
}
int main()
{
    int k = 1
    while (func(k) < 120)
        k++;
    printf("%d\t%d\n", k, func(k));
    return(0);
}
```



NGÔN NGỮ C VÀ C++



- Ngôn ngữ C ra đời năm 1972.
- Phát triển thành C++ vào năm 1983.



KEYWORDS

- Một số từ khóa (keyword) mới đã được đưa vào C++ ngoài các từ khóa có trong C.
- Các chương trình sử dụng các tên trùng với các từ khóa cần phải thay đổi trước khi chương trình được dịch lại bằng C++.

asm catch class delete friend inline

new operator private protected public

template this throw try virtual



- C:
 - Chú thích bằng /* ... */ → dùng cho khối chú thích lớn gồm nhiều dòng.
- C++: giống như C
 - Đưa thêm chú thích bắt đầu bằng // → dùng cho các chú thích trên một dòng.
- Ví dụ:

```
/* Đây là
   chú thích trong C */
// Đây là chú thích trong C++
```



QUY TẮC ĐẶT TÊN

- Quy tắc Pascal: viết hoa chữ cái đầu tiên của mỗi từ
 - Method, Interface, Enum, Events, Exception, Namespace, Property
- Quy tắc Camel: viết thường từ đầu tiên & chữ cái đầu tiên của từ kế tiếp
 - Bổ sung truy cập, Parameter



DATA TYPE

- 2 loại:
 - Kiểu dữ liệu gốc (Primitive Type);
 - Kiểu dữ liệu gốc modifier: sử dụng 1 trong các modifier sau:
 - signed
 - unsigned
 - short
 - long

Kiểu dữ liệu	Keyword	Byte
Boolean	bool	1
Ký tự	float	1
Số nguyên	int	4
Số thực	float	4
Số thực dạng double	double	8
Kiểu không có giá trị	void	



IDENTIFIER

- Biến: vị trí bộ nhớ được dành riêng để lưu giá trị.
- 3 loại:
 - Biến giá trị;
 - Biến tham chiếu;
 - Biến con trỏ.
- Khai báo & Khởi tạo:
 - `int x;`
 - `x = 5;`
 - `int x = 5;`



IDENTIFIER

- Phân loại theo phạm vi:
 - Biến cục bộ;
 - Biến toàn cục
 - ❖ Toán tử định phạm vi ::

```
#include <iostream>
using namespace std;
int x = 5;
int main()
{
    int x = 1;
    cout << x << ::x;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int x = 5;
int main()
{
    int y = 1;
    x = y;
    cout << x << y;
    return 0;
}
```



TYPEDEF

- **typedef:** tạo một tên mới cho một kiểu dữ liệu đang tồn tại:

typedef type name;

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    typedef int SoNguyen;
    SoNguyen x = 5;
    cout << x;
    return 0;
}
```



CASTING

- C: **(new_type) expression;**
- C++: vẫn sử dụng như trên.
 - Phép chuyển kiểu mới: **new_type (expression);**
→ Phép chuyển kiểu này có dạng như một hàm số
chuyển kiểu đang được gọi.
 - ❖ **type_cast <new_type> (expression);**
 - type_cast: const_cast, static_cast, dynamic_cast,
reinterpret_cast.



CONSTANT/LITERAL

- Có 2 cách định nghĩa:
 - Sử dụng bộ tiền xử lý **#define**;
#define identifier value
 - Sử dụng từ khóa **const**;
const type identifier = value;
type const identifier = value;

```
#include <iostream>
using namespace std;
#define PI 3.14
int main()
{
    const int x = 5;
    int const y = 6;
    cout << PI << x << y;
    return 0;
}
```



ENUM

- **Kiểu liệt kê:** Là kiểu dữ liệu mà tất cả các phần tử của nó (có ý nghĩa tương đương nhau) có giá trị là hằng số:
 - Được định nghĩa thông qua từ khóa **enum**;
 - Khai báo enum không yêu cầu cấp phát bộ nhớ, chỉ cấp phát bộ nhớ lúc tạo biến kiểu enum;
 - Mặc định các phần tử của enum có kiểu dữ liệu **int**;
 - Enum sẽ được ép kiểu ngầm định sang kiểu int, nhưng kiểu int phải ép kiểu tương ứng mình sang enum;
 - Biến kiểu enum chỉ có thể được gán giá trị là một trong số các hằng đã khai báo bên trong kiểu enum đó, không thể sử dụng hằng của kiểu enum khác;
 - Phạm vi sử dụng của một khai báo enum cũng tương tự như phạm vi sử dụng khi khai báo biến.

```
#include <iostream>
using namespace std;
int main()
{
    enum Color
    {
        RED = 1,
        BLUE = 2,
        GREEN = 3,
        YELLOW = 4
    };
    Color c = GREEN;
    cout << c;
    return 0;
}
```



OPERATOR

- Toán tử số học: +, -, *, /, %, ++, --
- Toán tử quan hệ: ==, !=, >, <, >=, <=
- Toán tử logic: &&, ||, !
- Toán tử so sánh bit: &, |, ^, ~, <<, >>
- Toán tử gán: =, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=
- Toán tử hỗn hợp: **sizeof**, ?x:y, , , &, *
- ❖ *Toán tử có thứ tự ưu tiên.*



IOSTREAM.H

- Thư viện:
 - Khai báo tệp tiêu đề:
 - include <iostream.h>
 - Sử dụng câu lệnh nhập, xuất dữ liệu trong C++ cần sử dụng thêm **using namespace std**:

```
#include <iostream>
using namespace std;
int main()
{
    //code
    return 0;
}
```



OUTPUT

- Toán tử xuất:
 - C: ???
 - C++:
 - Cú pháp:
cout << biểu_thức_1 << ... << biểu_thức_n;
 - Trong đó **cout** được định nghĩa trước như một đối tượng biểu diễn cho thiết bị xuất chuẩn của C++ là màn hình;
 - **cout** được sử dụng kết hợp với toán tử chèn **<<** để hiển thị giá trị các biểu thức 1, 2, ..., n ra màn hình;
 - Sử dụng “\n” hoặc **endl** để xuống dòng mới.



OUTPUT

- Toán tử xuất:
 - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5;
    cout << x << endl;
    cout << "x = " << x << endl;
    return 0;
}
```



INPUT

- Toán tử nhập:

- C: ???

- C++:

- Cú pháp:

- $\text{cin} \gg \text{biến_1} \gg \dots \gg \text{biến_n};$

- Toán tử **cin** được định nghĩa trước như một đối tượng biểu diễn cho thiết bị vào chuẩn của C++ là bàn phím;

- **cin** được sử dụng kết hợp với toán tử trích **>>** để nhập dữ liệu từ bàn phím cho các biến 1, 2, ..., n.



INPUT

- Xuất dữ liệu:
 - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cout << "Input x = ";
    cin >> x;
    cout << "x = " << x << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char x, int y;
    cout << "Input x = ";
    cin >> x;
    y = x;
    cout << x << y << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char x, int y;
    cout << "Input y = ";
    cin >> y;
    x = y;
    cout << x << y << endl;
    return 0;
}
```



ĐỊNH DẠNG IN

- Quy định số thực được hiển thị ra màn hình với p chữ số sau dấu chấm thập phân, sử dụng đồng thời các hàm sau:

```
cout << setiosflags(ios::showpoint) << setprecision(p);
```

- setiosflags(ios::showpoint): bật cờ hiệu showpoint(p).
- Định dạng trên sẽ có hiệu lực đối với tất cả các toán tử xuất tiếp theo cho đến khi gặp một câu lệnh định dạng mới.



ĐỊNH DẠNG IN

- Ví dụ:

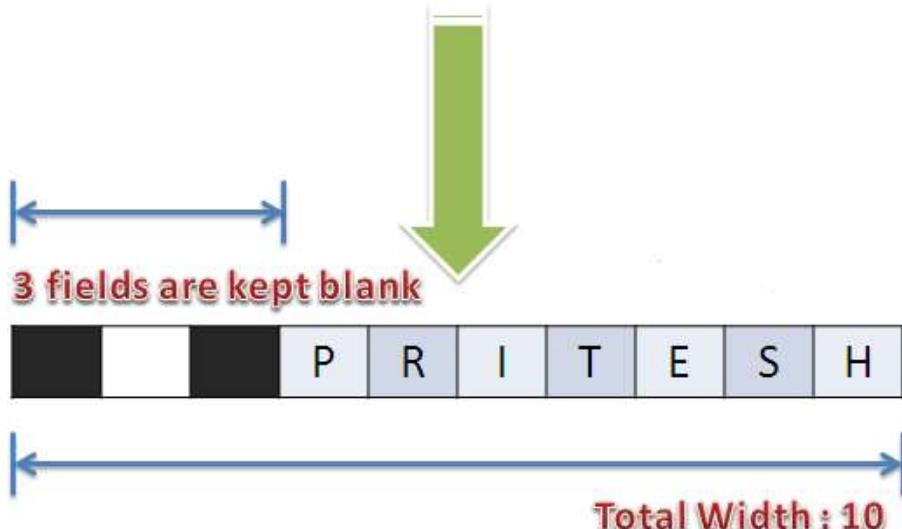
```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x = 1234.5;
    //x = 1.234; x = 12.34; x = 123.4; x = 1234.5
    cout << setiosflags(ios::showpoint) << setprecision(3);
    cin >> x;
    return 0;
}
```



ĐỊNH DẠNG IN

- Để quy định độ rộng tối thiểu để hiển thị là k vị trí cho giá trị (nguyên, thực, chuỗi) ta dùng hàm: `setw(k);` (`#include <iomanip>`).
- Hàm này cần đặt trong toán tử xuất và nó chỉ có hiệu lực cho một giá trị được in gần nhất. Các giá trị in ra tiếp theo sẽ có độ rộng tối thiểu mặc định là 0.

```
cout << setw(10) << "Pritesh";
```



```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << setw(5) << "KHOA" << "CNTT";
    return 0;
}
```



BIẾN THAM CHIẾU

- Biến tham chiếu là một tên khác (bí danh) cho biến đã định nghĩa:

Kiểu_dữ_liệu & Biến_tham_chiếu = biến_hằng;

- Biến tham chiếu có đặc điểm là nó được sử dụng làm bí danh cho một biến (kiểu giá trị) nào đó và sử dụng vùng nhớ của biến này;
- Biến tham chiếu không được cung cấp vùng nhớ (dùng chung địa chỉ vùng nhớ với biến mà nó tham chiếu đến);
- Trong khai báo biến tham chiếu phải chỉ rõ tham chiếu đến biến nào;
- Biến tham chiếu có thể tham chiếu đến một phần tử mảng, nhưng không cho phép khai báo mảng tham chiếu.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5;
    int &y = x;
    y = 1;
    cout << x;
    x++;
    cout << y;
    return 0;
}
```



HÀNG THAM CHIẾU

- Cú pháp:
const Kiểu_dữ_liệu &Hàng = Biến/Hàng;

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    const int a = 1;
    int b = 2;
    const int &x = a; //x++
    const int &y = b; //y++
    const int &z = 3; //z++
    cout << x << y << z;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int x = 1;
    const int &y = x;
    x++; //y++;
    cout << x << y;
    return 0;
}
```



HÀM

- Khai báo nguyên mẫu hàm & Định nghĩa hàm;
- Ví dụ:

```
#include <iostream>
using namespace std;
void Sum(int x, int y);
int main()
{
    int m = 1, n = 2;
    Sum(m, n);
    return 0;
}
void Sum(int x, int y)
{
    cout << x + y;
}
```

```
#include <iostream>
using namespace std;
void Sum(int x, int y)
{
    cout << x + y;
}
int main()
{
    int m = 1, n = 2;
    Sum(m, n);
    return 0;
}
```



HÀM

- Truyền tham trị & tham chiếu (biến tham chiếu hoặc biến con trỏ):

```
#include <iostream>
using namespace std;
void Swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(m, n);
    cout << m << n;
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(m, n);
    cout << m << n;
    return 0;
}
```



HÀM VỚI ĐỐI SỐ HẰNG

- Đối số bình thường:

```
#include <iostream>
using namespace std;
void Show(int m)
{
    cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //1
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Show(int &m)
{
    cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //compile error
    return 0;
}
```



HÀM VỚI ĐỐI SỐ HẰNG

- Đối số hằng: sử dụng khi không muốn thay đổi giá trị đối số truyền vào:

```
#include <iostream>
using namespace std;
void Show(const int m)
{
    cout << m; //cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x); //1
    Show(y); //1
    return 0;
}
```



HÀM VỚI ĐỐI SỐ HẰNG

- Đối số hằng tham chiếu:

```
#include <iostream>
using namespace std;
void Show(int &m)
{
    m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);      //2
    Show(y);      //compile error
    cout << x << y;
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Show(const int &m)
{
    cout << m++;      //compile error
}
void Display(const int &m)
{
    cout << m + 1;
}
int main()
{
    int x = 1;
    Show(x);
    Display(x);
    return 0;
}
```



HÀM VỚI ĐỐI SỐ MẶC ĐỊNH

- Đối số mặc định:
 - Định nghĩa các giá trị tham số mặc định cho các hàm;
 - Các đối số mặc định cần là các đối số cuối cùng tính từ trái qua phải;
 - Nếu chương trình sử dụng khai báo nguyên mẫu hàm thì các đối số mặc định cần được khởi gán trong nguyên mẫu hàm, không được khởi gán lại cho các đối số mặc định trong dòng đầu của định nghĩa hàm;
 - Khi xây dựng hàm, nếu không khai báo nguyên mẫu hàm thì các đối số mặc định được khởi gán trong dòng đầu của định nghĩa hàm;
 - Đối với các hàm có đối số mặc định thì lời gọi hàm cần viết theo quy định; các tham số vẫn mặt trong lời gọi hàm tương ứng với các đối số mặc định cuối cùng (tính từ trái sang phải).



HÀM VỚI ĐỐI SỐ MẶC ĐỊNH

- Đối số mặc định:

```
#include <iostream>
using namespace std;
void Show(int x, int y = 1, int z = 2);
int main()
{
    Show();          //compile error
    Show(1);        //112
    Show(1, 2);     //122
    Show(1, 2, 3);  //123
    Show(1, , 1);   //compile error
    return 0;
}
void Show(int x, int y, int z)
{
    cout << x << y << z;
}
```

```
#include <iostream>
using namespace std;
void Show(int x, int y = 1, int z = 2)
{
    cout << x << y << z;
}
int main()
{
    Show();          //compile error
    Show(1);        //112
    Show(1, 2);     //122
    Show(1, 2, 3);  //123
    Show(1, , 1);   //compile error
    return 0;
}
```



HÀM TRẢ VỀ THAM CHIẾU

- Hàm trả về là một tham chiếu:

Kiểu & Tên_hàm(...)

{

//thân hàm

return <biến phạm vi toàn cục>;

}

- Biểu thức được trả lại trong câu lệnh **return** phải là biến toàn cục, khi đó mới có thể sử dụng được giá trị của hàm;
- Nếu trả về tham chiếu đến một biến cục bộ thì biến cục bộ này sẽ bị mất đi khi kết thúc thực hiện hàm. Do vậy tham chiếu của hàm sẽ không còn ý nghĩa nữa. Vì vậy, nếu hàm trả về là tham chiếu đến biến cục bộ thì biến cục bộ này **NÊN** khai báo **static**;
- Khi giá trị trả về của hàm là tham chiếu, ta có thể gấp các câu lệnh gán hơi khác thường, trong đó vé trái là một lời gọi hàm chứ không phải là tên của một biến.



HÀM TRẢ VỀ THAM CHIẾU

- Ví dụ:

```
#include <iostream>
using namespace std;
int x = 4;
int &Func()
{
    return x;
}
int main()
{
    cout << x;          //4
    cout << Func(); //4
    Func() = 8;
    cout << x;          //8
    system("pause");
    return 0;
}
```

```
#include <iostream>
using namespace std;
int x = 4;
int &Func()
{
    static int x = 5;
    return x;
}
int main()
{
    cout << x;          //4
    cout << Func(); //5
    Func() = 8;
    cout << x;          //4
    system("pause");
    return 0;
}
```



HÀM TRẢ VỀ THAM CHIẾU

- Hàm trả về hằng tham chiếu:

```
#include <iostream>
using namespace std;
int &Func()
{
    static int x = 4;
    return x;
}
int main()
{
    cout << Func();
    cout << Func()++;
    return 0;
}
```

```
#include <iostream>
using namespace std;
const int &Func()
{
    static int x = 4;
    return x;
}
int main()
{
    cout << Func();
    cout << Func()++; //compile error
    return 0;
}
```



HÀM HẰNG

- Hàm hằng:

- const int func(int value);
- int func(int value) const;

```
#include <iostream>
using namespace std;
const int Func()
{
    int x = 4;
    x++;
    return x;
}
int main()
{
    int x = Func();      //x = 4
    int y = Func()++;   //compile error
    return 0;
}
```



HÀM INLINE

- Hàm nội tuyến (inline):

- Hàm: làm chậm tốc độ thực hiện chương trình vì phải thực hiện một số thao tác có tính thủ tục khi gọi hàm:
 - Cấp phát vùng nhớ cho đối số và biến cục bộ;
 - Truyền dữ liệu của các tham số cho các đối;
 - Giải phóng vùng nhớ trước khi thoát khỏi hàm.
 - C++ cho khả năng khắc phục được nhược điểm trên bằng cách dùng hàm nội tuyến → dùng từ khóa inline trước khai báo nguyên mẫu hàm.

- ❖ Chú ý:

- Hàm nội tuyến cần có từ khóa inline phải xuất hiện trước các lời gọi hàm;
 - Chỉ nên khai báo là hàm inline khi hàm có nội dung đơn giản;
 - Hàm đệ quy không thể là hàm inline.



HÀM INLINE

- Ví dụ:

```
#include <iostream>
using namespace std;
inline int f(int a, int b);
int main()
{
    int s;
    s = f(5,6);
    cout << s;
    return 0;
}
int f(int a, int b)
{
    return a*b;
```

```
#include <iostream>
using namespace std;
int f(int a, int b);
int main()
{
    int s;
    s = f(5,6);
    cout << s;
    return 0;
}
int f(int a, int b)
{
    return a*b;
```



CON TRỎ

- Khái niệm:
 - Con trỏ là biến dùng để chứa địa chỉ của biến khác;
 - Cùng kiểu dữ liệu với kiểu dữ liệu của biến mà nó trỏ tới.
- Cú pháp: <kiểu dữ liệu> *<tên con trỏ>;
- ❖ Lưu ý:
 - Có thể viết dấu * ngay sau kiểu dữ liệu;
 - Ví dụ: **int *a;** và **int* a;** là tương đương.
- Thao tác với con trỏ:
 - * là toán tử thâm nhập (dereferencing operator): *p là giá trị nội dung vùng nhớ con trỏ đang trỏ đến;
 - & là toán tử địa chỉ (address of operator): &x là địa chỉ của biến x; nếu **int *p = &x** thì **p ↔ x**.



CON TRỎ

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10, y = 20;
    int *p1, *p2;
    p1 = &x; p2 = &y;
    cout << "x = " << x;           //x = 10
    cout << "y = " << y;           //y = 20
    cout << "*p1 = " << *p1;     //*p1 = 10
    cout << "*p2 = " << *p2;     //*p2 = 20
    *p1 = 50; *p2 = 90;
    cout << "x = " << x;           //x = 50
    cout << "y = " << y;           //y = 90
    cout << "*p1 = " << *p1;     //*p1 = 50
    cout << "*p2 = " << *p2;     //*p2 = 90
    *p1 = *p2;
    cout << "x = " << x;           //x = 90
    cout << "y = " << y;           //y = 90
    cout << "*p1 = " << *p1;     //*p1 = 90
    cout << "*p2 = " << *p2;     //*p2 = 90
    return 0;
}
```



CON TRỎ HẰNG & HẰNG CON TRỎ

- Lưu ý:
 - **const int x = 1;** và **int const x = 1;** là như nhau;
 - Nếu **int x = 1;** thì
 - **const int *p = &x;** (**CON TRỎ HẰNG**)
Không cho phép thay đổi giá trị vùng nhớ mà con trỏ đang trỏ đến thông qua con trỏ (*p).
 - **int* const p = &x;** (**HẰNG CON TRỎ**)
Không cho phép thay đổi vùng nhớ con trỏ đang trỏ tới, nhưng có thể thay đổi giá trị vùng nhớ đó thông qua con trỏ.



CON TRỎ

- Phép gán giữa các con trỏ:
 - Các con trỏ cùng kiểu có thể gán cho nhau thông qua phép gán và lấy địa chỉ con trỏ
$$<\text{tên con trỏ 1}> = <\text{tên con trỏ 2}>;$$
 - Lưu ý:
 - Bắt buộc phải dùng phép **lấy địa chỉ của biến** do con trỏ trỏ tới mà không được dùng phép **lấy giá trị của biến**;
 - Hai con trỏ phải cùng kiểu dữ liệu (nếu khác kiểu phải sử dụng các phương thức ép kiểu).



CON TRỎ

- Phép gán giữa các con trỏ:

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 1;
    int *p1, *p2;
    p1 = &x;
    cout << *p1;      // *p1 = 1
    *p1 += 2;
    cout << x;       // x = 3
    p2 = p1;
    *p2 += 3;
    cout << x;       // x = 6
    return 0;
}
```



CON TRỎ

- Sử dụng `typedef`:

```
#include <iostream>
using namespace std;
typedef int P;
typedef int* Q;
int main()
{
    int x = 1, y = 2;
    P *p1, *p2;
    p1 = &x; p2 = &y;
    cout << *p1 << *p2; //12
    Q p3, p4;
    p3 = &x; p4 = &y;
    cout << *p3 << *p4; //12
    P* p5, p6;
    p5 = &x;
    p6 = &y;      //compile error
    return 0;
}
```



CONST_CAST

- **const_cast**: thêm hoặc loại bỏ const khỏi một biến (là cách duy nhất để thay đổi tính hằng của biến).
- Ví dụ:

```
#include <iostream>
using namespace std;
void Show(char *str)
{
    cout << str;
}
int main()
{
    const char *str = "DUT";
    Show(str); //compile error
    Show(const_cast<char*>(str));
    return 0;
}
```



MỘT SỐ LUU Ý

- Đối số con trỏ;
- Hàm trả về là tham chiếu.

```
#include <iostream>
using namespace std;
void Swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(&m, &n);
    cout << m << n;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int &Func()
{
    int x = 5;
    return x;
}
int main()
{
    int *p = &Func();
    //cout << *p << *p;
    cout << *p;
    cout << *p;
    return 0;
}
```



MẢNG

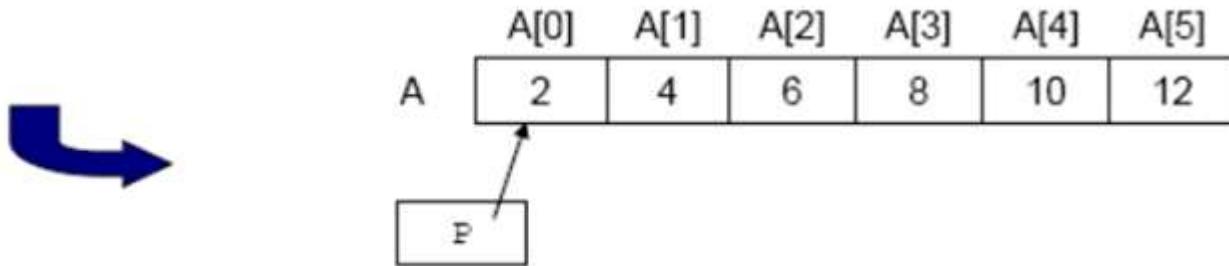
- Khai báo & khởi tạo mảng:
 - int A[5];
 - int A[5] = {1, 2, 3, 4, 5};
 - int A[] = {1, 2, 3, 4};
 - int A[2][3];
 - int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
 - int A[2][3] = {1, 2, 3, 4, 5, 6};
 - int A[][3] = {{1, 2, 3}, {4, 5, 6}};
 - ❖ Không được bỏ trống cột với mảng 2 chiều.
- Truy cập phần tử trong mảng:
 - A[i] – giá trị phần tử thứ i; &A[i] – địa chỉ phần tử thứ i;
 - A[i][j] – giá trị phần tử hàng i, cột j;
 - &A[i] – địa chỉ hàng i;
 - &A[i][j] – địa chỉ phần tử hàng i, cột j.



CON TRỎ & MẢNG

- Quan hệ giữa con trỏ và mảng 1 chiều:
 - Tên mảng được coi như một con trỏ tới phần tử đầu tiên của mảng:

```
int A[6] = {2,4,6,8,10,12};  
int *P;  
P = A; // P points to A
```



- Do tên mảng và con trỏ là tương đương, ta có thể dùng P như tên mảng. Ví dụ:

$P[3] = 7$; tương đương với $A[3] = 7$;



CON TRỎ & MẢNG

- Phép toán trên con trỏ và mảng 1 chiều:
 - Khi con trỏ trỏ đến mảng, thì các phép toán tăng hay giảm trên con trỏ sẽ tương ứng với phép dịch chuyên trên mảng;
 - ❖ Lưu ý: nếu `int *p = A;` thì `p++` và `*p++` là khác nhau
 - `p++` thao tác trên con trỏ: đưa con trỏ pa đến địa chỉ phần tử tiếp theo;
 - `*p++` là phép toán trên giá trị, tăng giá trị phần tử hiện tại của mảng.



CONTROL & MẠNG

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int A[ ] = {1, 2, 3, 4, 5};
    int *p = A;
    p += 2;
    cout << *p; /*p = A[2] = 3
    p--;
    cout << *p; /*p = A[1] = 2
    *p++;
    cout << *p; /*p = A[1] = 3
    return 0;
}
```



CIN.GET()

- **Lưu ý:**

- **cin.get(a, n);** - nhập một chuỗi không quá n ký tự và lưu vào mảng một chiều a (kiểu char);
 - Toán tử nhập **cin>>** sẽ để lại ký tự chuyển dòng ‘\n’ trong bộ đệm → làm trôi phương thức **cin.get**;
 - Để khắc phục tình trạng trên cần dùng phương thức **cin.ignore(1)** để bỏ qua một ký tự chuyển dòng.



CON TRỎ & MẢNG

- Con trỏ & mảng 2 chiều

- Địa chỉ ma trận A:

$$A = A[0] = *(A+0) = \&A[0][0];$$

- Địa chỉ của hàng thứ nhất:

$$A[1] = *(A+1) = \&A[1][0];$$

- Địa chỉ của hàng thứ i:

$$A[i] = *(A+i) = \&A[i][0];$$

- Địa chỉ phần tử:

$$\&A[i][j] = (*(A+i)) + j;$$

- Giá trị phần tử:

$$A[i][j] = *((*(A+i)) + j);$$

→ int A[3][3]; tương đương int (*A)[3];



CON TRỎ & MẢNG

- Con trỏ tới con trỏ:
 - **int A[3][3];** tương đương **int (*A)[3];**
 - **int A[3];** tương đương **int *A;**
→ Mảng 2 chiều có thể thay thế bằng một mảng các con trỏ tương đương một con trỏ trỏ đến con trỏ.
 - Ví dụ:
 - int A[3][3];**
 - int (*A)[3];**
 - int **A;**



CON TRỎ & MẢNG

- Con trỏ & mảng 2 chiều
 - Bài tập: Chương trình sau gồm các hàm:
 - Nhập một ma trận thực cấp mxn
 - In một ma trận thực dưới dạng bảng
 - Tìm phần tử lớn nhất và phần tử nhỏ nhất của dãy số thực;

Viết chương trình sẽ nhập một ma trận, in ma trận vừa nhập và in các phần tử lớn nhất và nhỏ nhất trên mỗi hàng của ma trận.



FUNCTION POINTER

- Mỗi hàm đều có 1 địa chỉ xác định trên bộ nhớ → có thể sử dụng con trỏ để trỏ đến địa chỉ của hàm.

```
#include <iostream>
using namespace std;
int Show(int x)
{
    return x;
}
int main()
{
    int n = 5;
    cout << Show(n);      //5
    cout << Show;        //Address of function Show
    return 0;
}
```

- ❖ Với compiler của Visual Studio (Window) thì kết quả đúng, với compiler của Java thì kết quả bằng 1.



FUNCTION POINTER

- Khai báo:
<kiểu dữ liệu trả về> (*<tên hàm>) [<danh sách tham số>]
- Lưu ý:
 - Dấu “()” bao bọc tên hàm để thông báo đó là con trỏ hàm;
 - Nếu không có dấu “()” thì trình biên dịch sẽ hiểu ta đang khai báo một hàm có giá trị trả về là một con trỏ.
 - Ví dụ:
int (*Cal) (int a, int b) //Khai báo con trỏ hàm
int *Cal (int a, int b) //Khai báo hàm trả về kiểu con trỏ
 - Đối số mặc định không áp dụng cho con trỏ hàm, vì đối số mặc định được cấp phát khi biên dịch, còn con trỏ hàm được sử dụng lúc thực thi.



FUNCTION POINTER

- Ví dụ:

```
#include <iostream>
using namespace std;
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 5, n = 10;
    void(*pSwap) (int &, int &) = Swap;
    (*pSwap)(m, n); //m = 10, n = 5
    return 0;
}
```



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
 - Sử dụng khi cần gọi 1 hàm như là tham số của 1 hàm khác;
 - Ví dụ: int (*Cal) (int a, int b);
 - Thị có thể gọi các hàm có hai tham số kiểu int và trả về cũng kiểu int:
 int add(int a, int b);
 int sub(int a, int b);
 - Nhưng không được gọi các hàm khác kiểu tham số hoặc kiểu trả về:
 int add(float a, int b);
 int add(int a);
 char* sub(char* a, char* b);



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
 - Ví dụ: sắp xếp dữ liệu trong mảng số nguyên theo thứ tự tăng dần

```
#include <iostream>
#include <iomanip>
using namespace std;
void Show(int *p, int length)
{
    for (int i = 0; i < length; i++) }
    cout << setw(3) << *(p + i);
}
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

void SelectionSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (*(p + i) > *(p + j))
                Swap(*(p + i), *(p + j));
}

int main()
{
    int A[] = {1, 4, 2, 3, 6, 5, 8, 9, 7};
    int l = sizeof(A) / sizeof(int);
    SelectionSort(A, l);
    Show(A, l);
    return 0;
}
```

- Nhưng nếu muốn sắp xếp theo thứ tự giảm dần ???



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
 - Thêm 2 hàm so sánh:

```
bool ascending(int left, int right)
{
    return left > right;
}
bool descending(int left, int right)
{
    return left < right;
}
```

- ascending: sắp xếp tăng dần
- Descending: sắp xếp giảm dần



FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:

```
void SelectionSort(int *p, int length, bool CompFunc(int, int))
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (CompFunc(*(p + i), *(p + j)))
                Swap(*(p + i), *(p + j));
}
int main()
{
    int A[] = {1, 4, 2, 3, 6, 5, 8, 9, 7};
    int l = sizeof(A) / sizeof(int);
    SelectionSort(A, l, ascending);
    SelectionSort(A, l, descending);
    return 0;
}
```

- ❖ Lưu ý: Con trỏ hàm là đối số mặc định

```
void SelectionSort(int *p, int length,  
    bool CompFunc(int, int) = ascending)
```



STATIC & DYNAMIC ALLOWCATION

- **Cấp phát tĩnh (static allocation):** biến được cấp phát vùng nhớ khi biên dịch;
- **Cấp phát động (dynamic allocation):** biến được cấp phát vùng nhớ lúc thực thi:
 - Sử dụng từ khóa **new**;
 - Dùng biến con trỏ (**p**) để lưu trữ địa chỉ vùng nhớ:
 - Cấp phát bộ nhớ 1 biến: **p = new type;**
 - Cấp phát bộ nhớ 1 mảng: **p = new type[n];**



STATIC & DYNAMIC ALLOWCATION

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    //Cấp phát tĩnh
    int x = 1; int *p1 = &x;
    //Cấp phát động
    int *p2 = new int; *p2 = 2;
    cout << x;           //1
    cout << *p1;         //1
    cout << *p2;         //2
    p1 = p2;
    p2 = new int; *p2 = 3;
    cout << *p1;         //2
    cout << *p2;         //3
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    //Cấp phát tĩnh
    int A[] = {1, 2, 3, 4}; int *p1 = A;
    //Cấp phát động
    int m = 5;
    int *p2 = new int[m];
    for (int i = 0; i < m; i++)
        *(p2 + i) = i++;
    return 0;
}
```



STATIC & DYNAMIC ALLOWCATION

- Kiểm tra vùng nhớ cấp phát có thành công hay không?
 - *Thành công*: con trỏ chứa địa chỉ đầu vùng nhớ được cấp phát; *Không thành công*: p = NULL.
 - Kiểm tra vùng nhớ cấp phát có thành công hay không bằng con trỏ hàm: new_handler (new.h)
 - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int *p = new int;
    if (p == NULL)
    {
        cout << "Error";
        exit(0);
    }
    return 0;
}
```



STATIC & DYNAMIC ALLOWCATION

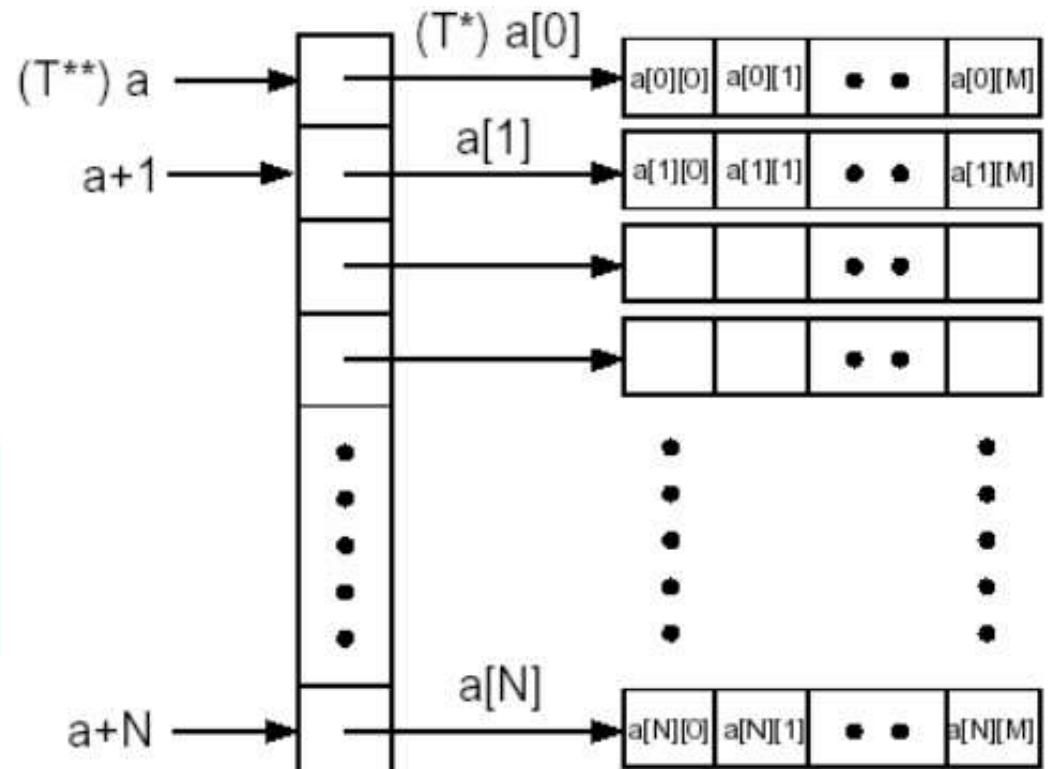
- Giải phóng bộ nhớ trong C++:

```
#include <iostream>
using namespace std;
int main()
{
    int *p1 = new int;
    *p1 = 1;
    delete p1;
    int n = 3;
    int *p2 = new int[n];
    for (int i = 0; i < n; i++)
        *(p2 + i) = i++;
    delete[] p2;
    return 0;
}
```



STATIC & DYNAMIC ALLOWCATION

- Cấp phát động mảng đa chiều:
 - Cấp phát động mảng hai chiều $(N+1)(M+1)$ gồm các đối tượng IQ:



```
IQ **a = new (IQ*) [N+1];
for (int i=0; i<N+1; i++)
    a[i] = new IQ[M+1];
```



STATIC & DYNAMIC ALLOWCATION

- Huỷ mảng động bất hợp lệ:

```
#include <iostream>
int main ()
{
    int* A = new int[6];
    // dynamically allocate array
    A[0] = 0; A[1] = 1; A[2] = 2;
    A[3] = 3; A[4] = 4; A[5] = 5;
    int *p = A + 2;
    cout << "A[1] = " << A[1] << endl;
    delete [] p; // illegal!!!
    // results depend on particular compiler
    cout << "A[1] = " << A[1] << endl;
}
```

P không trả tới
đầu mảng A

Huỷ không
hợp lệ

Kết quả phụ
thuộc trình
biên dịch

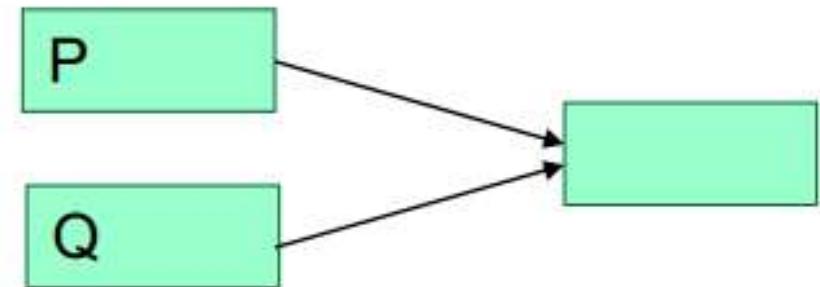


DANGLING POINTER

- Con trỏ lạc: khi delete P, ta cần chú ý không xoá vùng bộ nhớ mà một con trỏ Q khác đang trỏ tới:

```
int *P;  
int *Q;  
P = new int;  
Q = P;
```

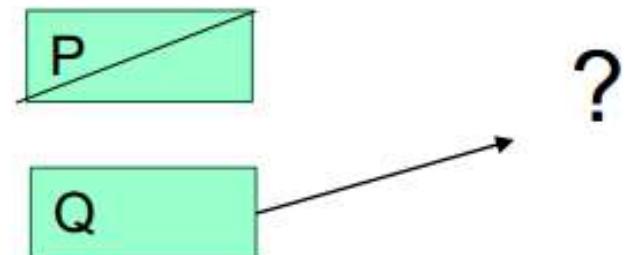
tạo



- Sau đó:

```
delete P;  
P = NULL;
```

Làm Q bị lạc





RÒ RỈ BỘ NHỚ

- Rò rỉ bộ nhớ:

- Một vấn đề liên quan: *mất* mọi con trỏ đến một vùng bộ nhớ được cấp phát. Khi đó, vùng bộ nhớ đó bị mất dấu, không thể trả lại cho heap được:

```
int *P;  
int *Q;  
P = new int;  
Q = new int;
```

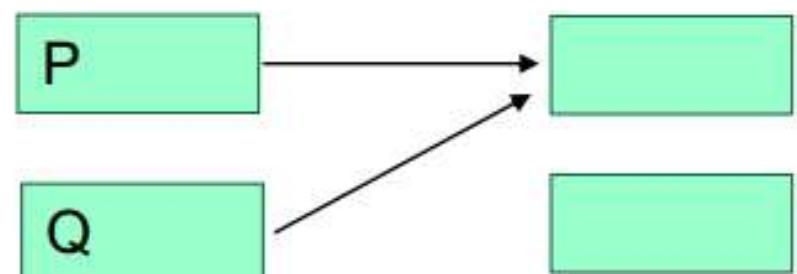
tạo



- Sau đó:

```
Q = P;
```

Làm mất vùng
nhớ đã từng
được Q trả tới





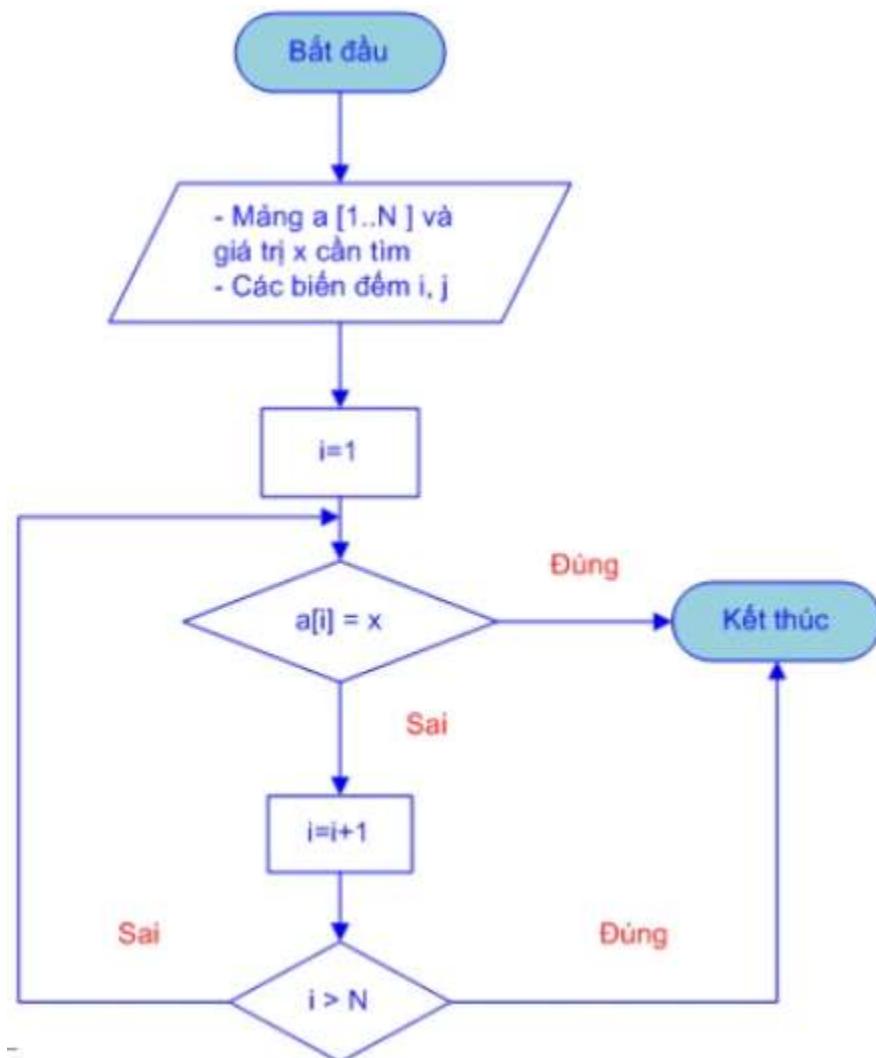
LINEAR SEARCH

- Thường sử dụng với các mảng chưa được sắp xếp.
- Ý tưởng: tiến hành so sánh số x với các phần tử trong mảng cho đến khi gặp được phần tử có giá trị cần tìm.
- Input:
 - Mảng $A[n]$ và giá trị cần tìm x
- Output:
 - True: nếu tìm thấy;
 - False: nếu không tìm thấy.



LINEAR SEARCH

- Lưu đồ giải thuật:





LINEAR SEARCH

- Ví dụ:

```
bool LinearSearch(int *p, int length, int x)
{
    for (int i = 0; i < length; i++)
        if (*(p + i) == length)
            return true;
    return false;
}
```



BINARY SEARCH

- Thường dùng cho mảng đã sắp xếp thứ tự;
- Ý tưởng:
 - Tìm kiếm kiểu tra “**tù điển**”;
 - Tìm cách giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong mảng đã được sắp xếp;
 - Tại mỗi bước, so sánh x với phần tử nằm ở vị trí giữa của mảng tìm kiếm hiện hành:
 - Nếu x nhỏ hơn thì sẽ tìm kiếm ở nửa mảng trước;
 - Ngược lại, tìm kiếm ở nửa mảng sau.



BINARY SEARCH

Giả sử chúng ta cần tìm vị trí của giá trị 31 trong một mảng bao gồm các giá trị như hình dưới đây bởi sử dụng Binary Search:

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Đầu tiên, chúng ta chia mảng thành hai nửa theo phép toán sau:

$$\text{chi-mục-giữa} = \text{ban-đầu} + (\text{cuối} + \text{ban-đầu}) / 2$$

Với ví dụ trên là $0 + (9 - 0) / 2 = 4$ (giá trị là 4.5). Do đó 4 là chỉ mục giữa của mảng.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Bây giờ chúng ta so sánh giá trị phần tử giữa với phần tử cần tìm. Giá trị phần tử giữa là 27 và phần tử cần tìm là 31, do đó là không kết nối. Bởi vì giá trị cần tìm là lớn hơn nên phần tử cần tìm sẽ nằm ở mảng con bên phải phần tử giữa.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Chúng ta thay đổi giá trị ban-đầu thành chỉ-mục-giữa + 1 và lại tiếp tục tìm kiếm giá trị chỉ-mục-giữa.

$$\text{ban-đầu} = \text{chi-mục-giữa} + 1$$

$$\text{chi-mục-giữa} = \text{ban-đầu} + (\text{cuối} + \text{ban-đầu}) / 2$$



BINARY SEARCH

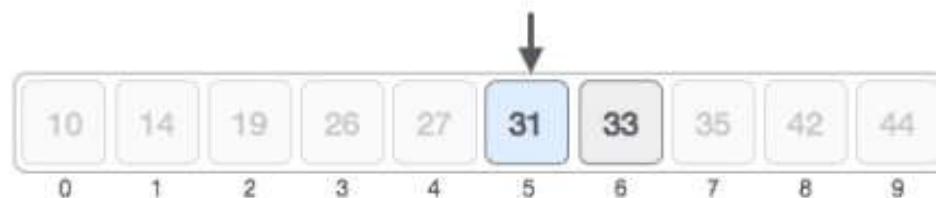
Bây giờ chỉ mục giữa của chúng ta là 7. Chúng ta so sánh giá trị tại chỉ mục này với giá trị cần tìm.



Giá trị tại chỉ mục 7 là không kết nối, và ngoài ra giá trị cần tìm là nhỏ hơn giá trị tại chỉ mục 7 do đó chúng ta cần tìm trong mảng con bên trái của chỉ mục giữa này.



Tiếp tục tìm chỉ-mục-giữa lần nữa. Lần này nó có giá trị là 5.



So sánh giá trị tại chỉ mục 5 với giá trị cần tìm và thấy rằng nó kết nối.

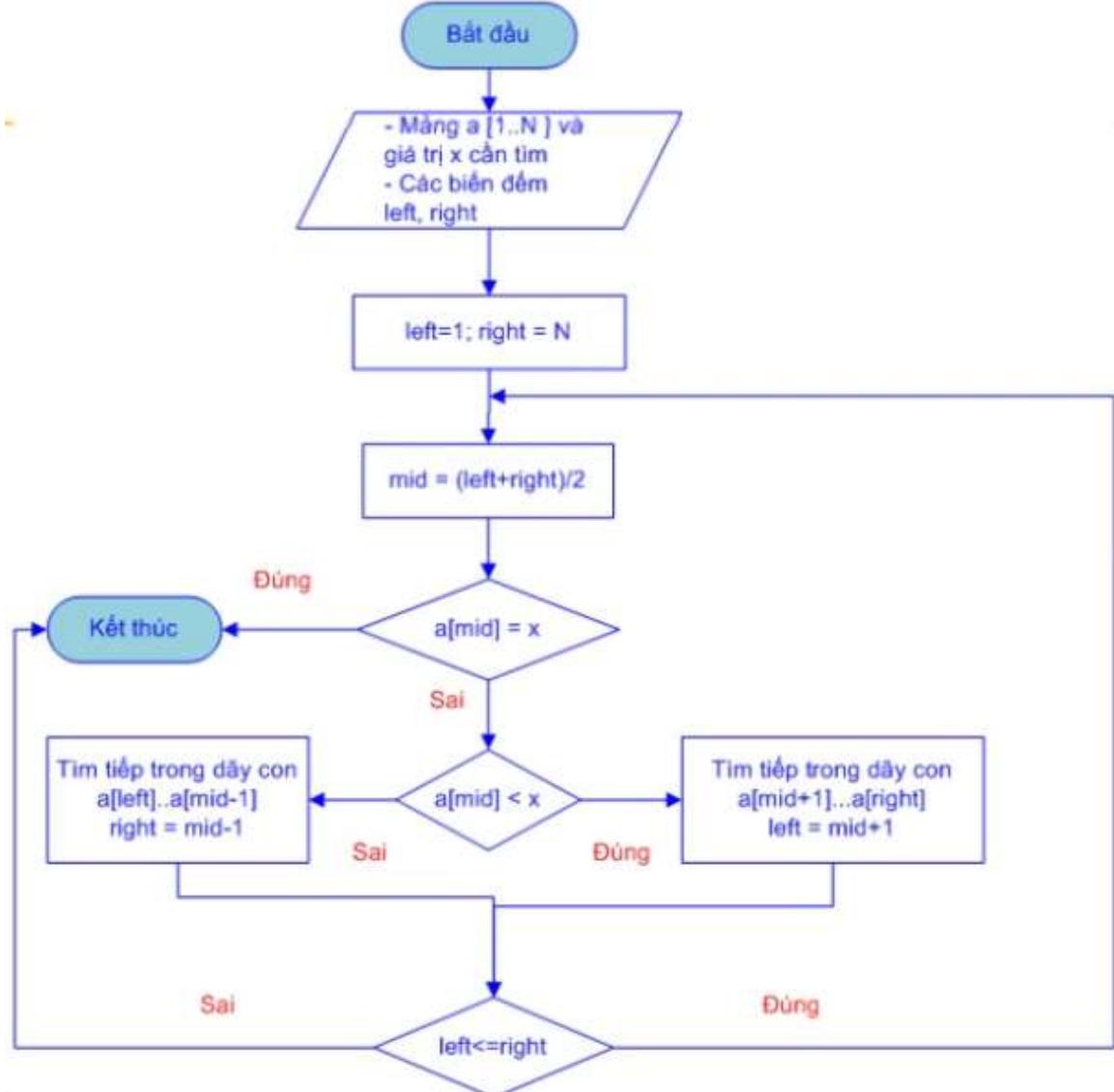


Do đó chúng ta kết luận rằng giá trị cần tìm 31 được lưu giữ tại vị trí chỉ mục 5.



BINARY SEARCH

- Lưu đồ thuật toán:





BINARY SEARCH

- Ví dụ:

```
bool BinarySearch(int *p, int length, int x)
{
    int left = 0, right = length - 1;
    while (left <= right)
    {
        int k = (left + right) / 2;
        if (*(p + k) == x)
            return true;
        if (*(p + k) > x)
            right = k - 1;
        else
            left = k + 1;
    }
    return false;
}
```



INTERPOLATION SEARCH

- Biến thể của Binary Search, sử dụng với mảng đã được sắp xếp;
- Ý tưởng:
 - Step 1:
 - Binary Search: $pos = left + (right - left)/2;$
 - Cải tiến: $pos = left + (X - T[left]) * (right - left)/(T[right] - T[left])$
 - Step 2:
 - Kiểm $A[pos]$ nếu bằng X thì pos là vị trí cần tìm;
 - Nếu nhỏ hơn X thì ta tăng $left$ lên một đơn vị và tiếp tục thực hiện lại bước 1;
 - Nếu lớn hơn X thì ta giảm $right$ một đơn vị và tiếp tục thực hiện lại bước 1.



INTERPOLATION SEARCH

Tìm kiếm nội suy tìm kiếm một phần tử cụ thể bằng việc tính toán vị trí dò (Probe Position). Ban đầu thì vị trí dò là vị trí của phần tử nằm ở giữa nhất của tập dữ liệu.



Nếu tìm thấy kết nối thì chỉ mục của phần tử được trả về. Để chia danh sách thành hai phần, chúng ta sử dụng phương thức sau:

$$mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])$$

Trong đó:

A = danh sách

Lo = chỉ mục thấp nhất của danh sách

Hi = chỉ mục cao nhất của danh sách

A[n] = giá trị được lưu giữ tại chỉ mục n trong danh sách

Nếu phần tử cần tìm có giá trị lớn hơn phần tử ở giữa thì phần tử cần tìm sẽ ở mảng con bên phải phần tử ở giữa và chúng ta lại tiếp tục tính vị trí dò; nếu không phần tử cần tìm sẽ ở mảng con bên trái phần tử ở giữa. Tiến trình này tiếp tục diễn ra trên các mảng con cho tới khi kích cỡ của mảng con giảm về 0.



INTERPOLATION SEARCH

- Ví dụ:

```
bool InterpolationSearch(int *p, int length, int x)
{
    int left = 0;
    int right = length - 1;
    while (left <= right && x >= *(p + left) && x <= *(p + right))
    {
        int val1 = (x - *(p + left)) / (*(p + right) - *(p + left));
        int val2 = right - left;
        int pos = left + val1 * val2;
        if (*(p + pos) == x)
            return true;
        if (*(p + pos) < x)
            left = pos + 1;
        else
            right = pos - 1;
    }
    return false;
}
```



BUBBLE SORT

- Sắp xếp nổi bọt;
- Ý tưởng: xuất phát từ đầu mảng, so sánh 2 phần tử cạnh nhau để đưa phần tử nhỏ hơn lên trước; sau đó lại xét cặp tiếp theo cho đến khi tiến về đầu mảng. Nhờ vậy, ở lần xử lý thứ i sẽ tìm được phần tử ở vị trí đầu mảng là i.
 - Step 1: $i = 0$
 - Step 2: $j = n - 1$
 - Trong khi $j > i$ thực hiện: nếu $a[j] < a[j - 1]$: hoán vị $a[j]$ & $a[j - 1]$, $j++$;
 - Step 3: $i++$
 - Nếu $i > n + 1$ thì dừng, ngược lại, lặp lại step 2.



BUBBLE SORT

Giả sử chúng ta có một mảng không có thứ tự gồm các phần tử như dưới đây. Bây giờ chúng ta sử dụng giải thuật sắp xếp nổi bọt để sắp xếp mảng này.



Giải thuật sắp xếp nổi bọt bắt đầu với hai phần tử đầu tiên, so sánh chúng để kiểm tra xem phần tử nào lớn hơn.



Trong trường hợp này, 33 lớn hơn 14, do đó hai phần tử này đã theo thứ tự. Tiếp đó chúng ta so sánh 33 và 27.





BUBBLE SORT

Chúng ta thấy rằng 33 lớn hơn 27, do đó hai giá trị này cần được tráo đổi thứ tự.



Mảng mới thu được sẽ như sau:



Tiếp đó chúng ta so sánh 33 và 35. Hai giá trị này đã theo thứ tự.



Sau đó chúng ta so sánh hai giá trị kế tiếp là 35 và 10.



Vì 10 nhỏ hơn 35 nên hai giá trị này chưa theo thứ tự.





BUBBLE SORT

Tráo đổi thứ tự hai giá trị. Chúng ta đã tiến tới cuối mảng. Vậy là sau một vòng lặp, mảng sẽ trông như sau:



Để đơn giản, tiếp theo mình sẽ hiển thị hình ảnh của mảng sau từng vòng lặp. Sau lần lặp thứ hai, mảng sẽ trông giống như:



Sau mỗi vòng lặp, ít nhất một giá trị sẽ di chuyển tới vị trí cuối. Sau vòng lặp thứ 3, mảng sẽ trông giống như:



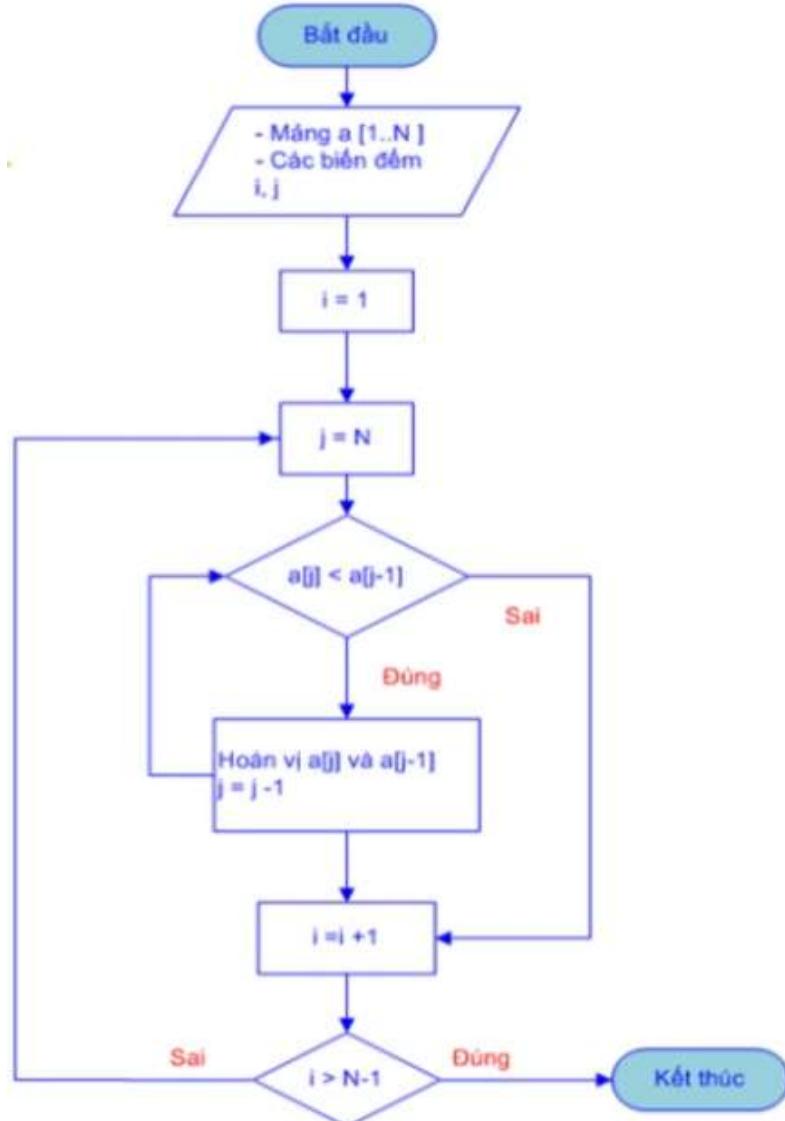
Và khi không cần tráo đổi thứ tự phần tử nào nữa, giải thuật sắp xếp nổi bọt thấy rằng mảng đã được sắp xếp xong.





BUBBLE SORT

- Lưu đồ giải thuật:





BUBBLE SORT

- Ví dụ:

	3	10	4	6	2	6	15	3	9	7
<i>Bước 1</i>	2	3	10	4	6	3	6	15	7	9
<i>Bước 2</i>		3	3	10	4	6	6	7	15	9
<i>Bước 3</i>			3	4	10	6	6	7	9	15
<i>Bước 4</i>				4	6	10	6	7	9	15
<i>Bước 5</i>					6	6	10	7	9	15
<i>Bước 6</i>						6	7	10	9	15
<i>Bước 7</i>							7	9	10	15
<i>Bước 8</i>								9	10	15
<i>Bước 9</i>									10	15
Kết quả	2	3	3	4	6	6	7	9	10	15



BUBBLE SORT

- Ví dụ:

```
void BubbleSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = length - 1; j > i; j--)
            if (*(p + j) < *(p + j - 1))
                Swap(*(p + j - 1), *(p + j));
}
```



SELECTION SORT

- Sắp xếp chọn;
- Ý tưởng:
 - Chọn phần tử nhỏ nhất trong n phần tử ban đầu của mảng, đưa phần tử này về vị trí đầu tiên của mảng; sau đó loại nó ra khỏi danh sách sắp xếp;
 - Mảng hiện hành còn $n - 1$ phần tử của mảng ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho mảng hiện hành đến khi mảng hiện hành còn 1 phần tử.



SELECTION SORT



Từ vị trí đầu tiên trong danh sách đã được sắp xếp, toàn bộ danh sách được duyệt một cách liên tục. Vị trí đầu tiên có giá trị 14, chúng ta tìm toàn bộ danh sách và thấy rằng 10 là giá trị nhỏ nhất.



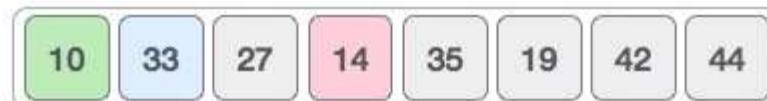
Do đó, chúng ta thay thế 14 với 10. Sau một vòng lặp, giá trị 10 thay thế cho giá trị 14 tại vị trí đầu tiên trong danh sách đã được sắp xếp. Chúng ta tráo đổi hai giá trị này.



Tại vị trí thứ hai, giá trị 33, chúng ta tiếp tục quét phần còn lại của danh sách theo thứ tự từng phần tử.



Chúng ta thấy rằng 14 là giá trị nhỏ nhất thứ hai trong danh sách và nó nên xuất hiện ở vị trí thứ hai. Chúng ta tráo đổi hai giá trị này.



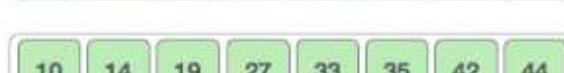
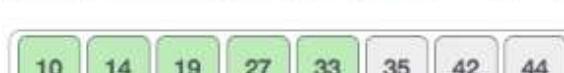
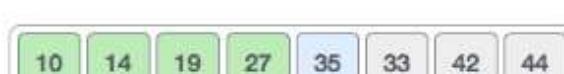


SELECTION SORT

Sau hai vòng lặp, hai giá trị nhỏ nhất đã được đặt tại phần đầu của danh sách đã được sắp xếp.



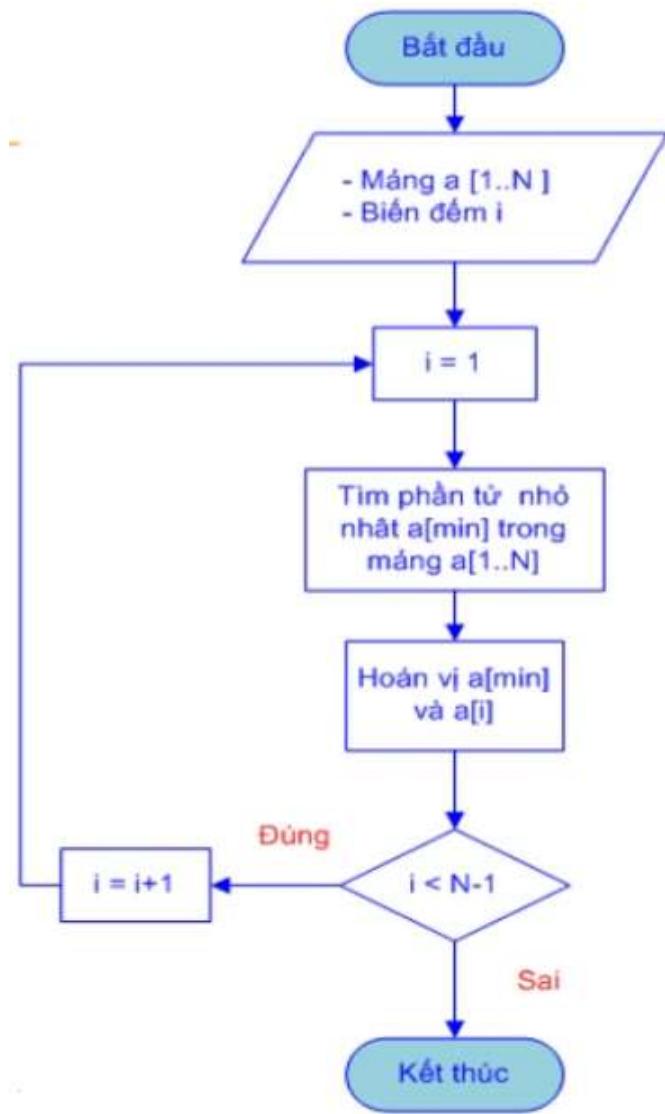
Tiến trình tương tự sẽ được áp dụng cho phần còn lại của danh sách. Các hình dưới minh họa cho các tiến trình này.





SELECTION SORT

- Lưu đồ giải thuật:





SELECTION SORT

- Ví dụ:

```
void SelectionSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (*(p + i) > *(p + j))
                Swap(*(p + i), *(p + j));
}
```



INSERTION SORT

- Sắp xếp chèn;
- Ý tưởng:

Ví dụ chúng ta có một mảng gồm các phần tử không có thứ tự:



Giải thuật sắp xếp chèn so sánh hai phần tử đầu tiên:



Giải thuật tìm ra rằng cả 14 và 33 đều đã trong thứ tự tăng dần. Bây giờ, 14 là trong danh sách con đã qua sắp xếp.



Giải thuật sắp xếp chèn tiếp tục di chuyển tới phần tử kế tiếp và so sánh 33 và 27.





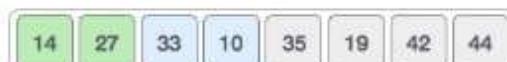
INSERTION SORT



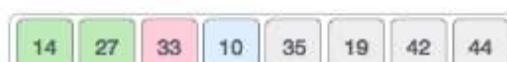
Giải thuật sắp xếp chèn tráo đổi vị trí của 33 và 27. Đồng thời cũng kiểm tra tất cả phần tử trong danh sách con đã sắp xếp. Tại đây, chúng ta thấy rằng trong danh sách con này chỉ có một phần tử 14 và 27 là lớn hơn 14. Do vậy danh sách con vẫn giữ nguyên sau khi đã tráo đổi.



Bây giờ trong danh sách con chúng ta có hai giá trị 14 và 27. Tiếp tục so sánh 33 với 10.



Hai giá trị này không theo thứ tự.



Vì thế chúng ta tráo đổi chúng.



Việc tráo đổi dẫn đến 27 và 10 không theo thứ tự.



Vì thế chúng ta cũng tráo đổi chúng.



Chúng ta lại thấy rằng 14 và 10 không theo thứ tự.



Và chúng ta tiếp tục tráo đổi hai số này. Cuối cùng, sau vòng lặp thứ 3 chúng ta có 4 phần tử.

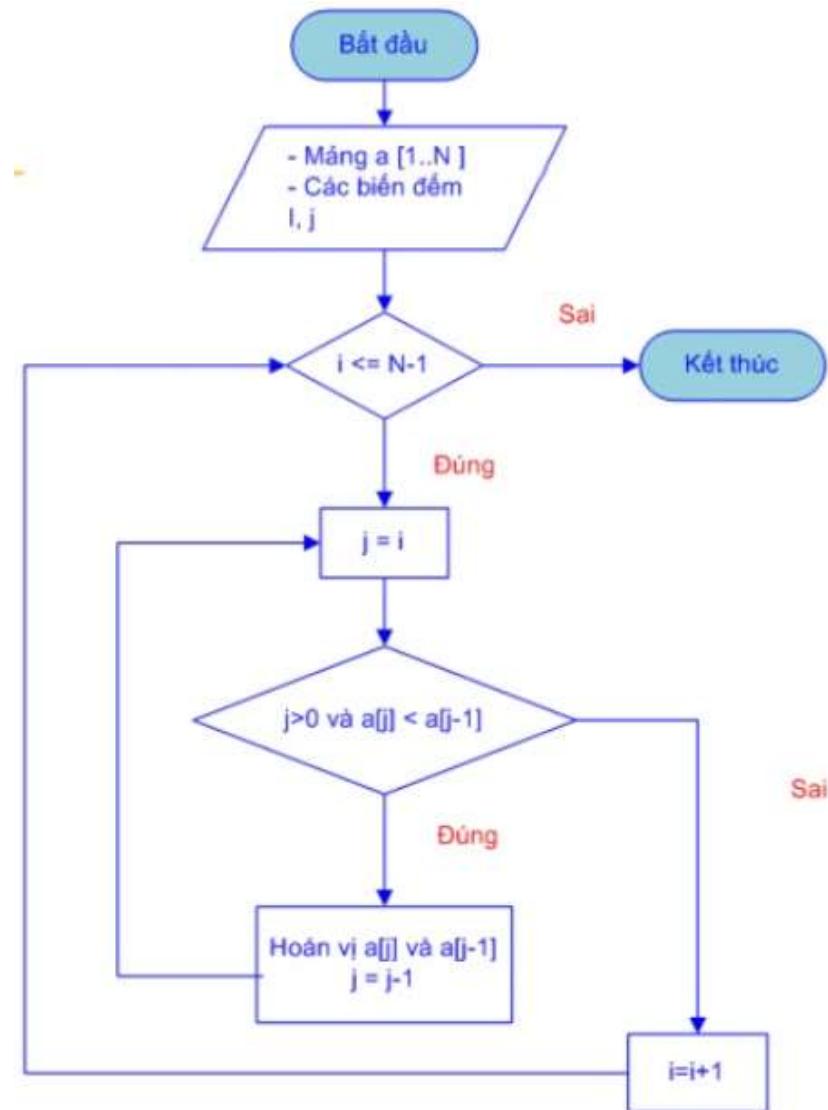


Activat
Go to Se



INSERTION SORT

- Lưu đồ giải thuật:





INSERTION SORT

- Ví dụ:

	3	7	22	3	1	5	8	4	3	9
Bước 0	3									
Bước 1	3	7								
Bước 2	3	7	22							
Bước 3	3	3	7	22						
Bước 4	1	3	3	7	22					
Bước 5	1	3	3	5	7	22				
Bước 6	1	3	3	5	7	8	22			
Bước 7	1	3	3	4	5	7	8	22		
Bước 8	1	3	3	3	4	5	7	8	22	
Bước 9	1	3	3	3	4	5	7	8	9	22



INSERTION SORT

- Ví dụ:

```
void InsertionSort(int *p, int length)
{
    for (int i = 0; i < length; i++)
        for (int j = i; j > 0; j--)
            if (*p + j) < *(p + j - 1))
                Swap(*(p + j), *(p + j - 1));
}
```



MEGRE SORT

- Sắp xếp trộn;
- Ý tưởng:
 - Chia mảng lớn thành những mảng con nhỏ hơn;
 - Tiếp tục chia cho đến khi mảng con nhỏ nhất là 1 phần tử;
 - Tiến hành so sánh 2 mảng con có cùng mảng cơ sở (vừa so sánh, vừa sắp xếp & ghép) cho đến khi gộp thành 1 mảng duy nhất



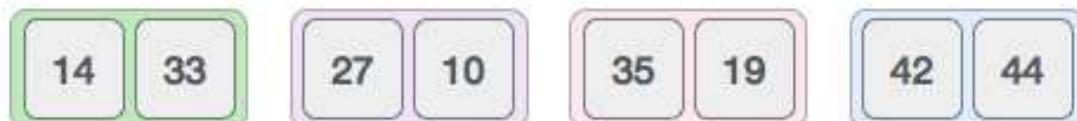
MEGRE SORT



Đầu tiên, giải thuật sắp xếp trộn chia toàn bộ mảng thành hai nửa. Tiến trình chia này tiếp tục diễn ra cho đến khi không còn chia được nữa và chúng ta thu được các giá trị tương ứng biểu diễn các phần tử trong mảng. Trong hình dưới, đầu tiên chúng ta chia mảng kích cỡ 8 thành hai mảng kích cỡ 4.



Tiến trình chia này không làm thay đổi thứ tự các phần tử trong mảng ban đầu. Bây giờ chúng ta tiếp tục chia các mảng này thành 2 nửa.



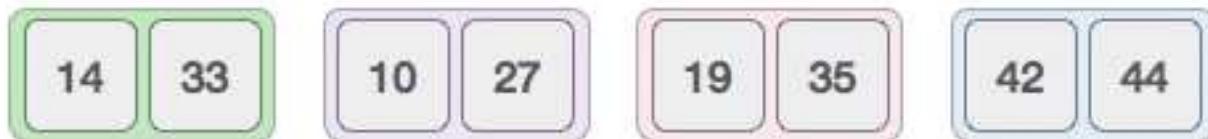
Tiến hành chia tiếp cho tới khi không còn chia được nữa.



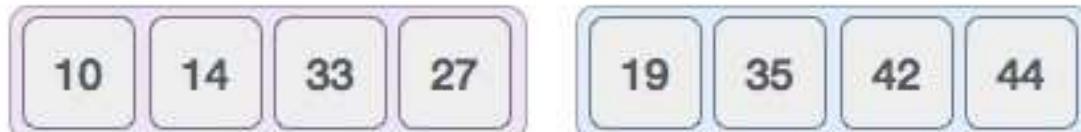


MEGRE SORT

Đầu tiên chúng ta so sánh hai phần tử trong mỗi list và sau đó tổ hợp chúng vào trong một list khác theo cách thức đã được sắp xếp. Ví dụ, 14 và 33 là trong các vị trí đã được sắp xếp. Chúng ta so sánh 27 và 10 và trong list khác chúng ta đặt 10 ở đầu và sau đó là 27. Tương tự, chúng ta thay đổi vị trí của 19 và 35. 42 và 44 được đặt tương ứng.



Vòng lặp tiếp theo là để kết hợp từng cặp list một ở trên. Chúng ta so sánh các giá trị và sau đó hợp nhất chúng lại vào trong một list chứa 4 giá trị, và 4 giá trị này đều đã được sắp thứ tự.



Sau bước kết hợp cuối cùng, danh sách sẽ trông giống như sau:





MEGRE SORT

- Ví dụ:

```
void MergeSort(int *p, int left, int right)
{
    if (right > left)
    {
        int mid;
        mid = (left + right) / 2;
        MergeSort(p, left, mid);
        MergeSort(p, mid + 1, right);
        Merge(p, left, mid, right);
    }
}
```



MEGRE SORT

```
void Merge(int *p, int left, int mid, int right)
{
    int *temp, i = left, j = mid + 1;
    temp = new int[right - left + 1];
    for (int k = 0; k <= right - left; k++)
    {
        if (*(p + i) < *(p + j))
        {
            *(temp + k) = *(p + i);
            i++;
        }
        else
        {
            *(temp + k) = *(p + j);
            j++;
        }
    }
    if (i == mid + 1)
    {
        while (j <= right)
        {
            k++;
            *(temp + k) = *(p + j);
            j++;
        }
        break;
    }
    if (j == right + 1)
    {
        while (i <= mid)
        {
            k++;
            *(temp + k) = *(p + i);
            i++;
        }
        break;
    }
    for (int k = 0; k <= right - left; k++)
        *(p + left + k) = *(temp + k);
    delete temp;
}
```



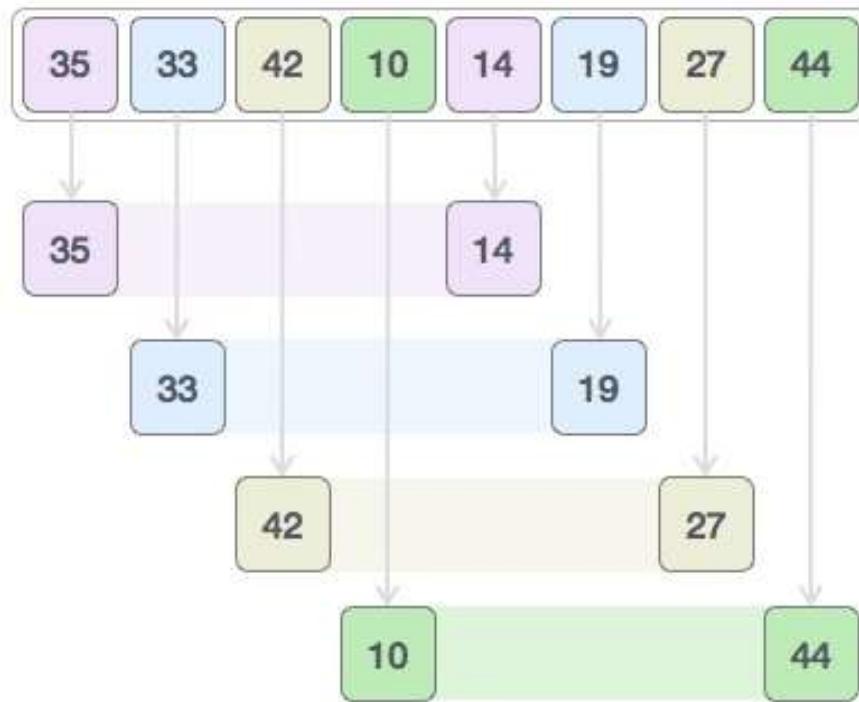
SHELL SORT

- Ý tưởng:
 - Sử dụng Selection Sort trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn;
 - Khoảng cách này là interval: là số vị trí từ phần tử này đến phần tử khác.
 - $h = h * 3 + 1$ (h là interval với giá trị ban đầu là 1).



SHELL SORT

- $h = 4$



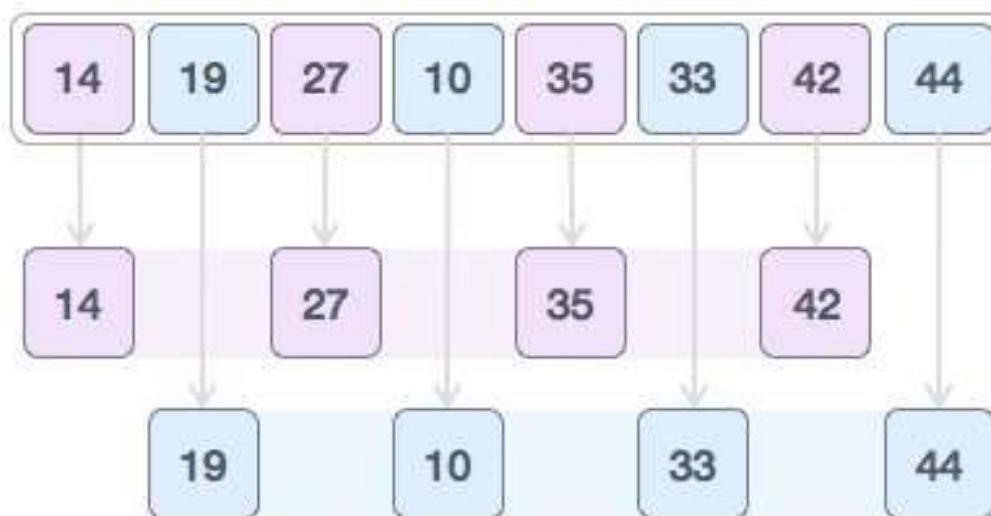
So sánh các giá trị này với nhau trong các danh sách con và tráo đổi chúng (nếu cần) trong mảng ban đầu. Sau bước này, mảng mới sẽ trống như sau:





SHELL SORT

- $h = 2$



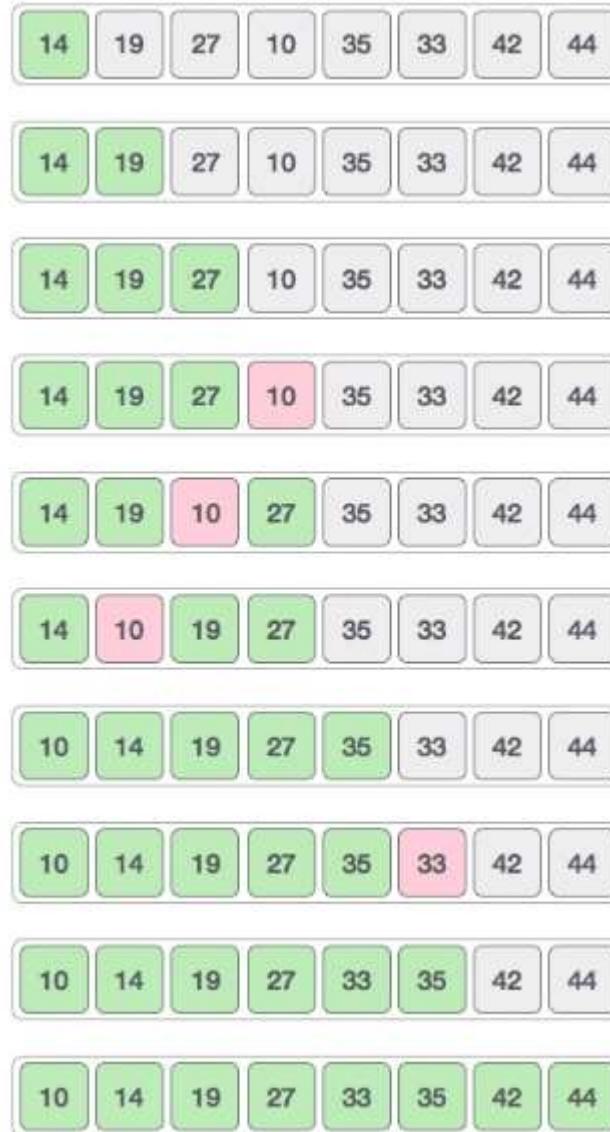
Tiếp tục so sánh và tráo đổi các giá trị (nếu cần) trong mảng ban đầu. Sau bước này, mảng sẽ trông như sau:





SHELL SORT

- $h = 1$





SHELL SORT

- Ví dụ:

```
void ShellSort(int *p, int length)
{
    for (int gap = length / 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < length; i += 1)
        {
            int temp = *(p + i);
            int j;
            for (j = i; j >= gap && *(p + j - gap) > temp; j -= gap)
                *(p + j) = *(p + j - gap);
            *(p + j) = temp;
        }
    }
}
```



QUICK SORT

- Sắp xếp nhanh;
- Ý tưởng:
 - QuickSort chia mảng thành hai danh sách bằng cách so sánh từng phần tử của danh sách với một phần tử được chọn được gọi là phần tử chốt. Những phần tử nhỏ hơn hoặc bằng phần tử chốt được đưa về phía trước và nằm trong danh sách con thứ nhất, các phần tử lớn hơn chốt được đưa về phía sau và thuộc danh sách con thứ hai. Cứ tiếp tục chia như vậy tới khi các danh sách con đều có độ dài bằng 1.
- Cách chọn phần tử chốt:
 - Chọn phần tử đứng đầu hoặc đứng cuối;
 - Chọn phần tử đứng giữa mảng;
 - Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối;
 - Chọn phần tử ngẫu nhiên.



QUICK SORT

1 12 5 26 7 14 3 7 2 unsorted

1 12 5 26 7 14 3 7 2 pivot value = 7
↑ ↑ ↑
i pivot value j

1 12 5 26 7 14 3 7 2 $12 \geq 7 \geq 2$, swap 12 and 2
↑ ↑
i j

1 2 5 26 7 14 3 7 12 $26 \geq 7 \geq 7$, swap 26 and 7
↑ ↑
i j

1 2 5 7 14 3 26 12 $7 \geq 7 \geq 3$, swap 7 and 3
↑ ↑
i j

1 2 5 7 3 14 7 26 12 $i > j$, stop partition
↑ ↑
i j

1 2 5 7 3 14 7 26 12 run quick sort recursively

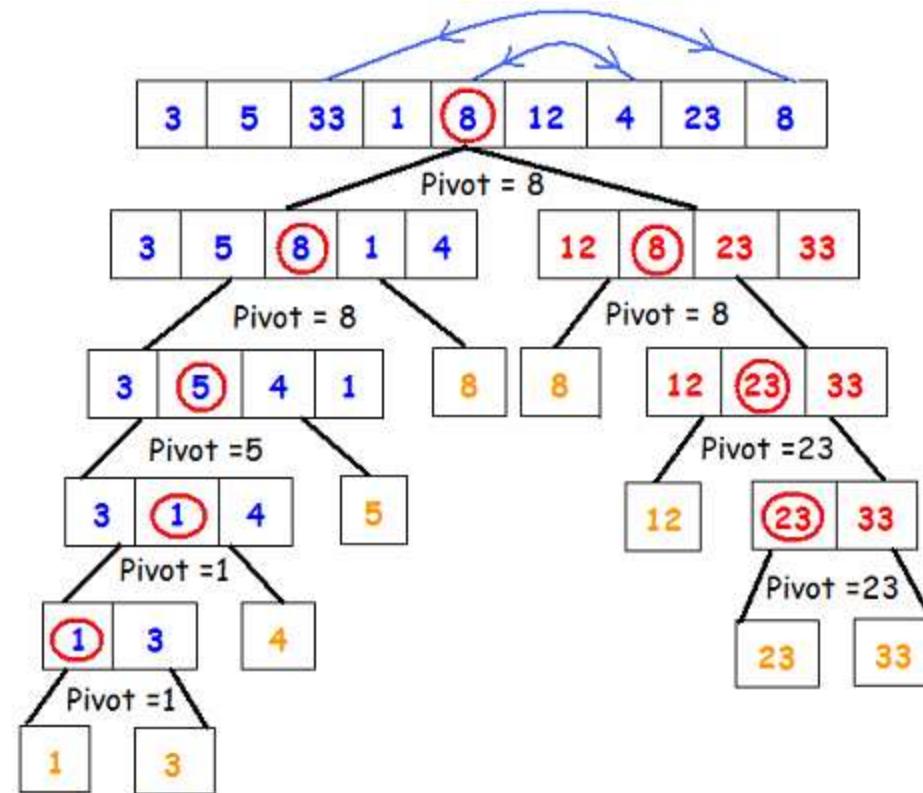
...

1 2 3 5 7 7 12 14 26 sorted



QUICK SORT

- Ví dụ:





QUICK SORT

- Ví dụ:

```
void QuickSort(int *p, int left, int right)
{
    srand(time(NULL));
    int key = *(p + (left + rand() % (right - left + 1)));
    int i = left, j = right;
    while (i <= j)
    {
        while (*(p + i) < key) i++;
        while (*(p + j) > key) j--;
        if (i <= j)
        {
            if (i < j)
                Swap(*(p + i), *(p + j));
            i++; j--;
        }
    }
    if (left < j)
        QuickSort(p, left, j);
    if (i < right)
        QuickSort(p, i, right);
}
```



HEAP SORT

- Sắp xếp vun đống;
- Ý tưởng:
 - Phiên bản cải tiến của Selection Sort khi chia các phần tử thành 2 mảng con, 1 mảng các phần tử đã được sắp xếp và mảng còn lại các phần tử chưa được sắp xếp. Trong mảng chưa được sắp xếp, các phần tử lớn nhất sẽ được tách ra và đưa vào mảng đã được sắp xếp. Điều cải tiến ở Heapsort so với Selection sort ở việc sử dụng cấu trúc dữ liệu heap thay vì tìm kiếm tuyến tính (linear-time search) như Selection sort để tìm ra phần tử lớn nhất.



HEAP SORT

- Ví dụ:

```
void Heapify(int *p, int length, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < length && *(p + left) > *(p + largest))
        largest = left;
    if (right < length && *(p + right) > *(p + largest))
        largest = right;
    if (largest != i)
    {
        Swap(*(p + i), *(p + largest));
        Heapify(p, length, largest);
    }
}
```



HEAP SORT

- Ví dụ:

```
void HeapSort(int *p, int length)
{
    for (int i = length / 2 - 1; i >= 0; i--)
        Heapify(p, length, i);
    for (int i = length - 1; i >= 0; i--)
    {
        Swap(*p, *(p + i));
        Heapify(p, i, 0);
    }
}
```



CHƯƠNG 2

OBJECT-ORIENTED PROGRAMMING (OOP)





PROCEDURE-ORIENTED PROGRAMMING (POP)

- Tổ chức chương trình thành các **chương trình con**
 - PASCAL: thủ tục & hàm, C: hàm.
 - Chương trình hướng cấu trúc = cấu trúc dữ liệu + tập hợp hàm.
 - Trừu tượng hóa chức năng (Functional Abstraction):
 - Không quan tâm đến cấu trúc hàm;
 - Chỉ cần biết kết quả thực hiện của hàm.
- Nền tảng của lập trình hướng cấu trúc.



LẬP TRÌNH HƯỚNG CẤU TRÚC

- Tại sao phải thay đổi CTDL:
 - Cấu trúc dữ liệu là mô hình của bài toán cần giải quyết
 - Do thiếu kiến thức về bài toán, về miền ứng dụng, ... không phải lúc nào cũng tạo được CTDL hoàn thiện ngay từ đầu;
 - Tạo ra một cấu trúc dữ liệu hợp lý luôn là vấn đề đau đầu của người lập trình.
 - Bản thân bài toán cũng không bất biến:
 - Cần phải thay đổi cấu trúc dữ liệu để phù hợp với các yêu cầu thay đổi.



LẬP TRÌNH HƯỚNG CẤU TRÚC

- Vấn đề đặt ra khi thay đổi CTDL:
 - Thay đổi cấu trúc:
 - Dẫn đến việc sửa lại mã chương trình (thuật toán) tương ứng và làm chi phí phát triển tăng cao;
 - Không tái sử dụng được các mã xử lý ứng với cấu trúc dữ liệu cũ.
 - Đảm bảo tính đúng đắn của dữ liệu:
 - Một trong những nguyên nhân chính gây ra lỗi phần mềm là gán các dữ liệu không hợp lệ;
 - Cần phải kiểm tra tính đúng đắn của dữ liệu mỗi khi thay đổi giá trị.



LẬP TRÌNH HƯỚNG CẤU TRÚC

- Vấn đề đặt ra khi thay đổi CTDL:

- Ví dụ: struct Date

```
struct Date
{
    int year, month, day;
};

Date d;
d.day = 32;           // invalid day
d.day = 31; d.month = 2; // how to check
d.day = d.day + 1;
```

- Thay đổi CTDL

```
struct Date
{
    short year;
    short mon_n_day;
};
```



TIẾP CẬN HƯỚNG ĐỐI TƯỢNG

- Xuất phát từ hai hạn chế chính của Lập trình hướng cấu trúc:
 - Không quản lý được sự thay đổi dữ liệu khi có nhiều chương trình cùng thay đổi một biến chung;
 - Không tiết kiệm được tài nguyên: giải thuật gắn liền với CTDL, nếu CTDL thay đổi, sẽ phải thay đổi giải thuật.
- Phương pháp tiếp cận mới: phương pháp lập trình hướng đối tượng. Với hai mục đích chính:
 - Đóng gói, che dấu dữ liệu: (che dấu cấu trúc) để hạn chế sự truy nhập tự do vào dữ liệu. Truy cập dữ liệu thông qua giao diện xác định;
 - Cho phép sử dụng lại mã nguồn, hạn chế việc viết mã lại từ đầu.



TIẾP CẬN HƯỚNG ĐỐI TƯỢNG

- **Đóng gói** được thực hiện theo phương pháp trùu tượng hóa đối tượng từ thấp lên cao:
 - Thu thập các thuộc tính của mỗi đối tượng, gắn các thuộc tính vào đối tượng tương ứng;
 - Nhóm các đối tượng có thuộc tính tương tự nhau thành nhóm, loại bỏ các thuộc tính cá biệt, chỉ giữ lại các thuộc tính chung nhất. Đây gọi là quá trình trùu tượng hóa đối tượng thành lớp;
 - Đóng gói các dữ liệu của đối tượng vào lớp tương ứng. Mỗi thuộc tính của đối tượng trở thành thuộc tính của lớp tương ứng;
 - Việc truy nhập dữ liệu được thực hiện thông qua các phương thức được trang bị cho lớp;
 - Khi có thay đổi trong dữ liệu của đối tượng, chỉ thay đổi các phương thức truy nhập thuộc tính của lớp, mà không cần thay đổi mã nguồn của chương trình sử dụng lớp tương ứng.



TIẾP CẬN HƯỚNG ĐỐI TƯỢNG

- **Tái sử dụng mã nguồn** được thực hiện thông qua cơ chế kế thừa trong lập trình hướng đối tượng
 - Các lớp có thể được kế thừa nhau để tận dụng các thuộc tính, các phương thức của nhau;
 - Trong lớp dẫn xuất (lớp được thừa kế) có thể sử dụng lại các phương thức của lớp cơ sở mà không cần cài đặt lại mã nguồn;
 - Khi lớp dẫn xuất định nghĩa lại phương thức cho mình, lớp cơ sở cũng không bị ảnh hưởng và không cần thiết sửa đổi lại mã nguồn.



TIẾP CẬN HƯỚNG ĐỐI TƯỢNG

- **Ưu điểm:**
 - Không có nguy cơ dữ liệu bị thay đổi tự do trong chương trình;
 - Khi thay đổi cấu trúc dữ liệu của một đối tượng, không cần thay đổi mã nguồn của các đối tượng khác, mà chỉ cần thay đổi một số thành phần của đối tượng bị thay đổi;
 - Có thể sử dụng lại mã nguồn, tiết kiệm được tài nguyên;
 - Phù hợp với dự án phần mềm lớn, phức tạp.



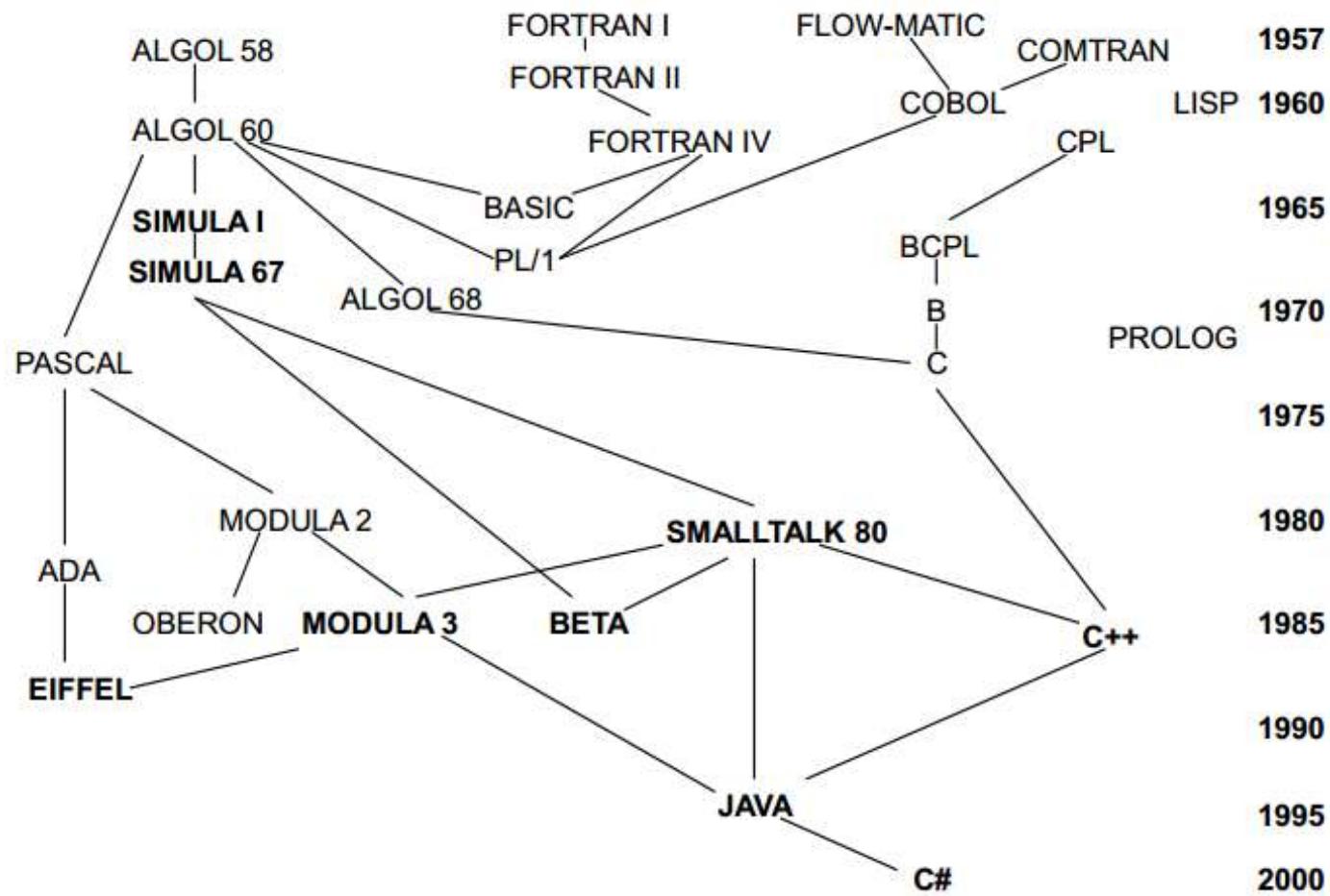
LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- Một số hệ thống “hướng đối tượng” thời kỳ đầu không có các lớp, chỉ có các “đối tượng” và các “thông điệp” (v.d. Hypertalk).
- Hiện giờ, đã có sự thống nhất rằng hướng đối tượng là:
 - Lớp (class);
 - Thừa kế (inheritance) và liên kết động (dynamic binding).
- Một số đặc tính của lập trình hướng đối tượng có thể được thực hiện bằng C hoặc các ngôn ngữ lập trình thủ tục khác;
- Điểm khác biệt sự hỗ trợ và ép buộc ba khái niệm trên được cài hẵn vào trong ngôn ngữ;
- Mức độ hướng đối tượng của các ngôn ngữ không giống nhau:
 - Eiffel (tuyệt đối), Java (rất cao), C++ (nửa nọ nửa kia).



LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- Lịch sử ngôn ngữ lập trình hướng đối tượng:





LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- OOP là phương pháp lập trình:
 - Mô tả chính xác các đối tượng trong thế giới;
 - Lấy đối tượng làm nền tảng xây dựng thuật toán;
 - Thiết kế xoay quanh dữ liệu của hệ thống;
 - Chương trình được chia thành các lớp đối tượng;
 - Dữ liệu được đóng gói, che dấu và bảo vệ;
 - Đối tượng làm việc với nhau qua thông báo;
 - Chương trình được thiết kế theo cách từ dưới lên (bottom-up).



LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- Hệ thống Hướng Đối Tượng:
 - Gồm tập hợp các đối tượng:
 - Sự đóng gói của 2 thành phần:
 - Dữ liệu (thuộc tính của đối tượng);
 - Các thao tác trên dữ liệu.
 - Các đối tượng có thể kế thừa các đặc tính của đối tượng khác;
 - Hoạt động thông qua sự tương tác giữa các đối tượng nhờ cơ chế truyền thông điệp:
 - Thông báo;
 - Gửi & nhận thông báo.



LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- Hướng thủ tục:
 - Lấy hành động làm trung tâm.
 - Hàm là xương sống.
 - Lặt (Rau) - Ướp (Cá) - Luộc (Rau).
 - Kho (Cá) - Nấu (Cơm).
- Hướng đối tượng:
 - Lấy dữ liệu làm trung tâm.
 - Đối tượng là xương sống.
 - Rau.Lặt - Cá.Ướp - Rau.Luộc
 - Cá.Kho - Cơm.Nấu

Các bước nấu ăn	
Verb	Object
Lặt	Rau
Ướp	Cá
Nấu	Cơm
Kho	Cá
Luộc	Rau

**Thay đổi
tư duy
lập trình!!**



OBJECT & CLASS

- Object (Đối tượng):

- Chương trình là “cỗ máy” phức tạp.
- Cấu thành từ nhiều loại “vật liệu”.
- Vật liệu cơ bản: hàm, cấu trúc.
- **Đã đủ tạo ra chương trình tốt?**

→ Vật liệu mới: **Đối tượng**

→ Viết một chương trình hướng đối tượng nghĩa là đang xây dựng một mô hình của một vài bộ phận trong thế giới thực.





OBJECT & CLASS

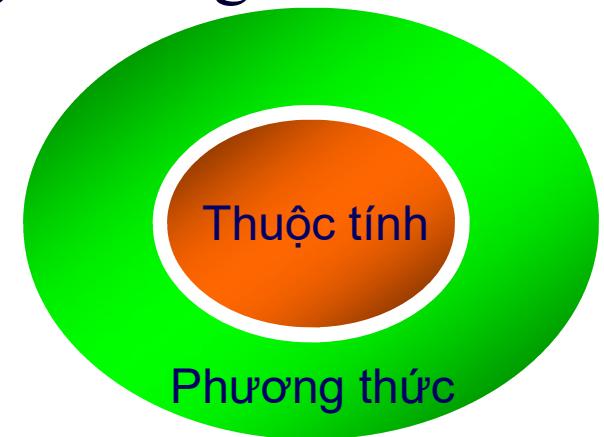
- Object (Đối tượng):

- Đặc trưng:

- Đóng gói cả dữ liệu và xử lý;
 - Thuộc tính (attribute): dữ liệu của đối tượng;
 - Phương thức (method): xử lý của đối tượng.

- Cấu trúc:

- Hộp đen: thuộc tính trong, phương thức ngoài.
 - Bốn nhóm phương thức:
 - Nhóm tạo hủy.
 - Nhóm truy xuất thông tin.
 - Nhóm xử lý nghiệp vụ.
 - Nhóm toán tử.





OBJECT & CLASS

- Object (Đối tượng): là một thực thể đang tồn tại trong hệ thống và được xác định bằng ba yếu tố:
 - *Định danh đối tượng*: xác định duy nhất cho mỗi đối tượng trong hệ thống, nhằm phân biệt các đối tượng với nhau;
 - *Trạng thái của đối tượng*: sự tổ hợp của các giá trị của các thuộc tính mà đối tượng đang có;
 - *Hoạt động của đối tượng*: là các hành động mà đối tượng có khả năng thực hiện được.

	Trạng thái	Hành động
Chiếc xe	Nhãn hiệu: “Ford” Màu sơn: Trắng Giá bán: 5000\$	Khởi động Dừng lại Chạy



OBJECT & CLASS

- Object (Đối tượng):
 - Một đối tượng gồm:
 - Định danh;
 - Thuộc tính (dữ liệu);
 - Hành vi (phương thức).
 - Mỗi đối tượng bất kể đang ở trạng thái nào đều có định danh và được đối xử như một thực thể riêng biệt:
 - Mỗi đối tượng có một handle (trong C++ là địa chỉ);
 - Hai đối tượng có thể có giá trị giống nhau nhưng handle khác nhau.



OBJECT & CLASS

- Class (Lớp):
 - Đối tượng là một thực thể cụ thể, tồn tại trong hệ thống;
 - Lớp là một khái niệm trừu tượng, dùng để chỉ một tập hợp các đối tượng có mặt trong hệ thống.
 - Ví dụ:
 - Mỗi chiếc xe có trong cửa hàng là một đối tượng, nhưng khái niệm “xe hơi” là một lớp đối tượng dùng để chỉ tất cả các loại xe có trong cửa hàng.
- ❖ Lưu ý:
 - Lớp là một khái niệm, mang tính trừu tượng, dùng để biểu diễn một tập các đối tượng;
 - Đối tượng là một thể hiện cụ thể của lớp, là một thực thể tồn tại trong hệ thống.

OBJECT & CLASS

Person1:

- Name: Peter.
- Age: 25.
- Hair Color: Brown.
- Eye Color: Brown.
- Job: Worker.



Person2:

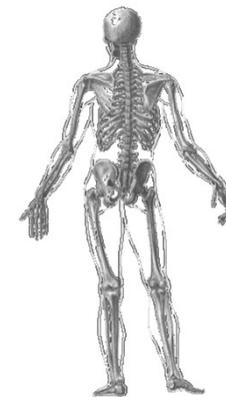
- Name: Thomas.
- Age: 50.
- Hair Color: White.
- Eye Color: Blue.
- Job: Teacher.



Tập hợp đối tượng có cùng
thuộc tính và phương thức

Human:

- Name.
- Age.
- Hair Color.
- Eye Color.
- Job.



Bản mô tả đối tượng
Kiểu của đối tượng



OBJECT & CLASS

- Class (Lớp):

- Một lớp có thể có một trong các khả năng sau:

- Hoặc chỉ có thuộc tính, không có phương thức;
 - Hoặc chỉ có phương thức, không có thuộc tính;
 - Hoặc có cả thuộc tính, phương thức (phổ biến);
 - Đặc biệt, lớp không có thuộc tính, phương thức nào, gọi là lớp trừu tượng, các lớp này không có đối tượng.

- Lớp và đối tượng mặc dù có mối liên hệ tương ứng lẫn nhau, nhưng lại khác nhau về bản chất.

Lớp	Đối tượng
Sự trừu tượng hóa của đối tượng	Là thể hiện của lớp
Là một khái niệm trừu tượng, chỉ tồn tại ở dạng khái niệm mô tả đặc tính chung của một số đối tượng	Là một thực thể cụ thể, có thực, tồn tại trong bộ nhớ
Là nguyên mẫu cho các đối tượng, xác định các hành vi và các thuộc tính cần thiết cho một nhóm các đối tượng cụ thể	Tất cả các đối tượng thuộc cùng một lớp có cùng các thuộc tính và hành động



OBJECT & CLASS

- Class (Lớp):
 - Trùu tượng hóa theo chức năng:
 - Mô hình hóa các phương thức của lớp dựa trên hành động của đối tượng.
 - Trùu tượng hóa theo dữ liệu:
 - Mô hình hóa các thuộc tính của lớp dựa trên thuộc tính của các đối tượng tương ứng.
 - Thuộc tính (attribute) là dữ liệu trình bày các đặc điểm về một đối tượng;
 - Phương thức (method): có liên quan tới những thứ mà đối tượng có thể làm. Một phương thức đáp ứng một chức năng tác động lên dữ liệu của đối tượng (thuộc tính).



OBJECT & CLASS

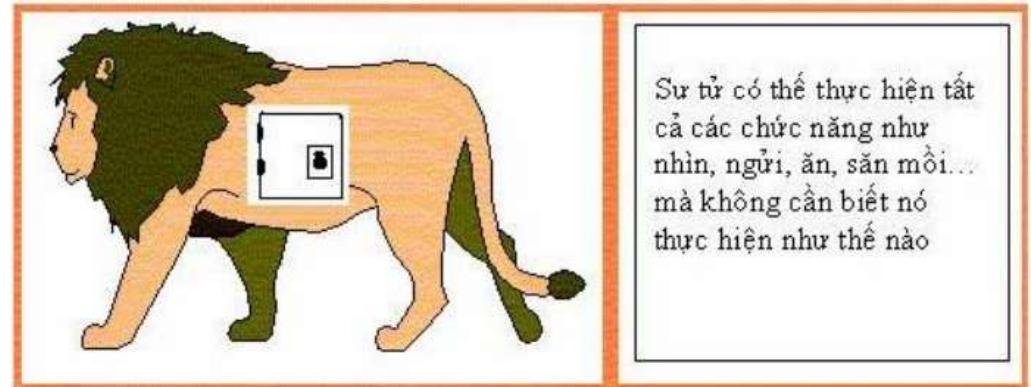
- Thuộc tính (attribute):
 - Là dữ liệu trình bày các đặc điểm về một đối tượng;
 - Bao gồm: Hằng, biến; Tham số nội tại.
 - Kiểu thuộc tính: Kiểu cổ điển; Kiểu do người dùng định nghĩa.
- Phương thức (method):
 - Có liên quan tới những thứ mà đối tượng có thể làm;
 - Một phương thức đáp ứng một chức năng tác động lên dữ liệu của đối tượng (thuộc tính);
 - Hàm nội tại của đối tượng (**hàm thành viên**);
 - Có kiểu trả về.



OBJECT & CLASS

- Thông điệp (message):
 - Là phương tiện để đối tượng này chuyển yêu cầu tới đối tượng khác, bao gồm:
 - Đối tượng nhận thông điệp;
 - Tên của phương thức thực hiện;
 - Các tham số mà phương thức cần.
- Truyền thông điệp:
 - Là cách một đối tượng triệu gọi một hay nhiều phương thức của đối tượng khác để yêu cầu thông tin;
 - Hệ thống yêu cầu đối tượng thực hiện phương thức:
 - Gửi thông báo và tham số cho đối tượng;
 - Kiểm tra tính hợp lệ của thông báo;
 - Gọi thực hiện hàm tương ứng phương thức.

- Tính đóng gói (encapsulation):
 - Là tiến trình che giấu việc thực thi chi tiết của một đối tượng;
 - Khái niệm: là cơ chế ràng buộc dữ liệu và các thao tác trên dữ liệu thành thể thống nhất;
 - Đóng gói gồm:
 - Bao gói: người dùng giao tiếp với hệ thống qua giao diện;
 - Che dấu: ngăn chặn các thao tác không được phép từ bên ngoài.
 - Ưu điểm:
 - Quản lý sự thay đổi;
 - Bảo vệ dữ liệu.





OBJECT & CLASS

- Tính đóng gói (encapsulation):
 - Đóng gói → Thuộc tính được lưu trữ hay phương thức được cài đặt như thế nào → được che giấu đi từ các đối tượng khác;
 - Việc che giấu những chi tiết thiết kế và cài đặt từ những đối tượng khác được gọi là **ẩn thông tin**.





OBJECT & CLASS

- Tính đóng gói (encapsulation):
 - Ví dụ: bài toán quản lý nhân viên văn phòng với lớp **Nhân viên**:
 - Cách tính lương nhân viên là khác nhau với mỗi người: Tiền lương = Hệ số lương * lương cơ bản * Tỉ lệ phần trăm
 - Việc gọi phương thức tính tiền lương là giống nhau cho mọi đối tượng Nhân viên;
 - Sự giống nhau về cách sử dụng phương thức cho các đối tượng của cùng một lớp, nhưng cách thực hiện phương thức lại khác nhau với các đối tượng khác nhau gọi là sự đóng gói dữ liệu của lập trình hướng đối tượng.

Nhân viên	
Tên	
Ngày sinh	
Giới tính	
Phòng ban	
Hệ số lương	
Tính lương nhân viên ()	



OBJECT & CLASS

- Tính đóng gói (encapsulation):
 - Cho phép che dấu sự cài đặt chi tiết bên trong:
 - Chỉ cần gọi các phương thức theo một cách thống nhất;
 - Phương thức có thể cài đặt khác nhau cho các trường hợp khác nhau.
 - Cho phép che dấu dữ liệu bên trong đối tượng:
 - Khi sử dụng, không biết thực sự bên trong đối tượng có những gì;
 - Chỉ thấy được những gì đối tượng cho phép truy nhập vào.
 - Cho phép tối đa hạn chế việc sửa lại mã chương trình.



OBJECT & CLASS

- Tính thừa kế (inheritance):
 - Hệ thống hướng đối tượng cho phép các lớp được định nghĩa kế thừa từ các lớp khác;
 - Khái niệm:
 - Khả năng cho phép xây dựng lớp mới được thừa hưởng các thuộc tính của lớp đã có;
 - Các phương thức & thuộc tính được định nghĩa trong một lớp có thể được sử dụng lại bởi lớp khác.
 - Đặc điểm:
 - Lớp nhận được có thể bổ sung các thành phần;
 - Hoặc định nghĩa là các thuộc tính của lớp cha.
 - Các loại thừa kế: Đơn thừa kế & Đa thừa kế.



OBJECT & CLASS

- Tính thừa kế (inheritance):
 - Ví dụ:

Lớp Nhân viên	Lớp Sinh viên
Thuộc tính: Tên Ngày sinh Giới tính Lương	Thuộc tính: Tên Ngày sinh Giới tính Lớp
Phương thức: Nhập/Xem tên Nhập/Xem ngày sinh Nhập /Xem giới tính Nhập/Xem lương	Phương thức: Nhập/Xem tên Nhập/Xem ngày sinh Nhập /Xem giới tính Nhập/Xem lớp



OBJECT & CLASS

- Tính thừa kế (inheritance):
 - Hai lớp có một số thuộc tính và phương thức chung: tên, ngày sinh, giới tính:
 - Không thể loại bỏ thuộc tính cá biệt để gộp lại thành một lớp;
 - Thuộc tính lương và lớp là cần thiết cho việc quản lý nhân viên, sinh viên.
 - Vấn đề này sinh:
 - Lặp lại việc viết mã cho một số phương thức;
 - Phải lặp lại việc sửa mã chương trình nếu có sự thay đổi về kiểu dữ liệu.



OBJECT & CLASS

- Tính thừa kế (inheritance):

Lớp Người

Thuộc tính:

Tên

Ngày sinh

Giới tính

Phương thức:

Nhập/Xem tên

Nhập/Xem ngày sinh

Nhập /Xem giới tính

Lớp Nhân viên kế thừa từ lớp Người

Thuộc tính:

Lương

Phương thức:

Nhập/Xem lương

Lớp Sinh viên kế thừa từ lớp Người

Thuộc tính:

Lớp

Phương thức:

Nhập/Xem lớp



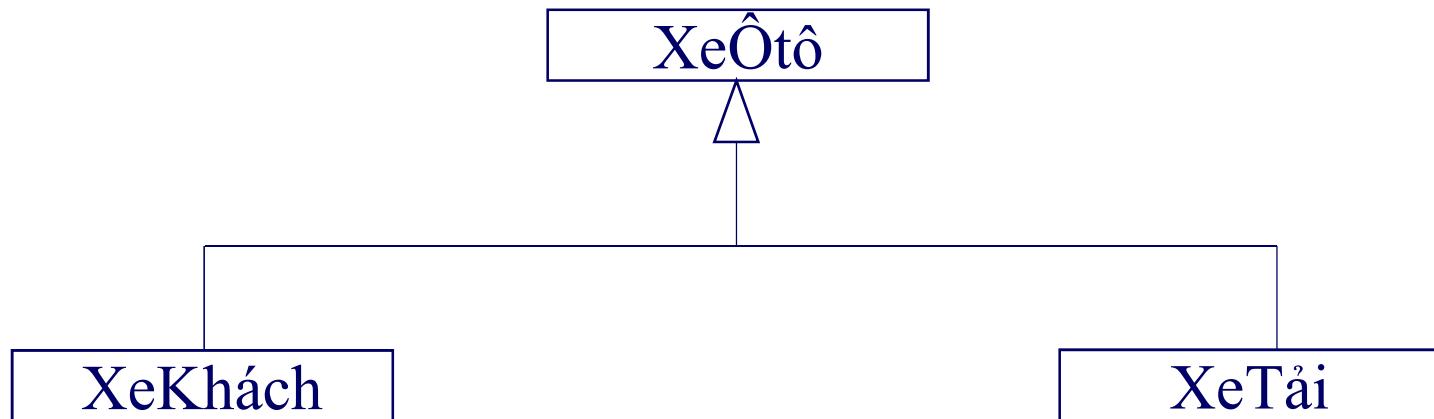
OBJECT & CLASS

- Tính thừa kế (inheritance):
 - Cho phép lớp dẫn xuất có thể sử dụng các thuộc tính và phương thức của lớp cơ sở tương tự như sử dụng thuộc tính và phương thức của mình;
 - Cho phép chỉ cần thay đổi phương thức của lớp cơ sở, có thể sử dụng được ở tất cả các lớp dẫn xuất;
 - Tránh sự cài đặt trùng lặp mã nguồn chương trình;
 - Chỉ cần thay đổi mã nguồn một lần khi thay đổi dữ liệu của các lớp.



OBJECT & CLASS

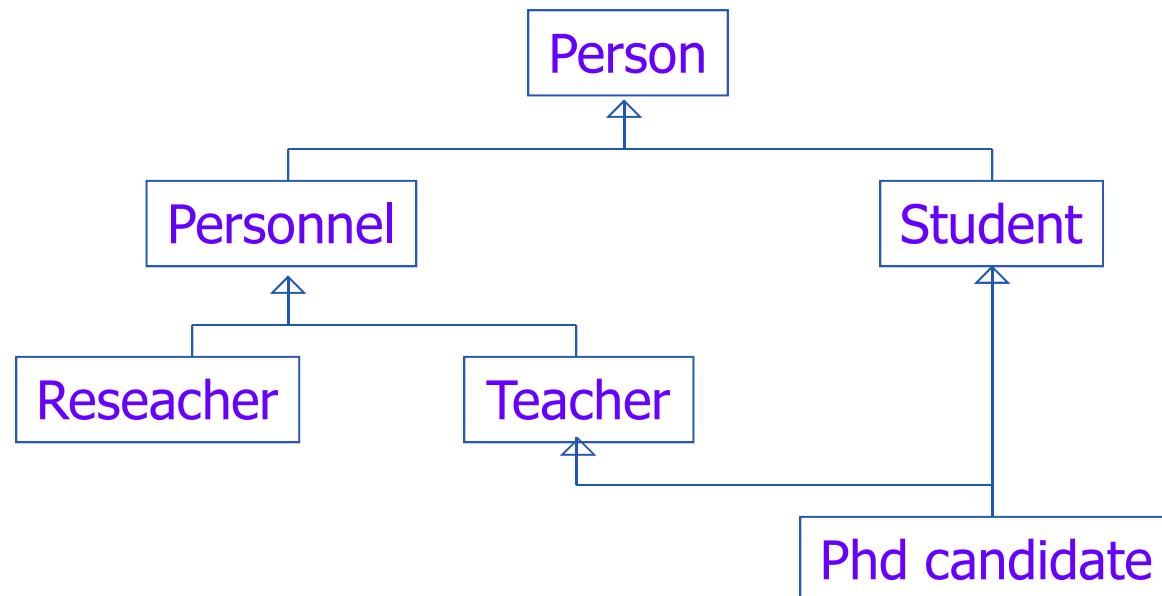
- Tính thừa kế (inheritance):
 - Đơn thừa kế: một lớp con chỉ thừa kế từ một lớp cha duy nhất
 - Ví dụ:
 - Lớp trừu tượng hay lớp chung: XeÔtô;
 - Lớp cụ thể hay lớp chuyên biệt: XeKhách, XeTải;
 - Lớp chuyên biệt có thể thay thế lớp chung trong tất cả các ứng dụng.





OBJECT & CLASS

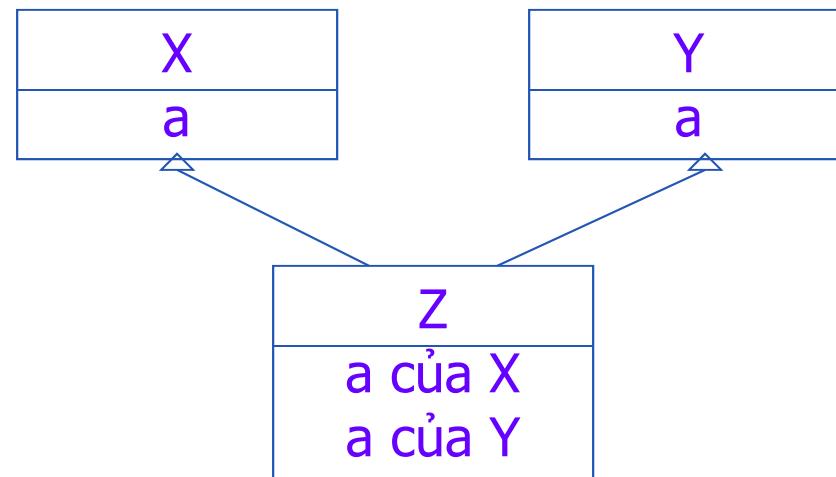
- Tính thừa kế (inheritance):
 - Đa thừa kế: một lớp con thừa kế từ nhiều lớp cha khác nhau
 - Ví dụ:





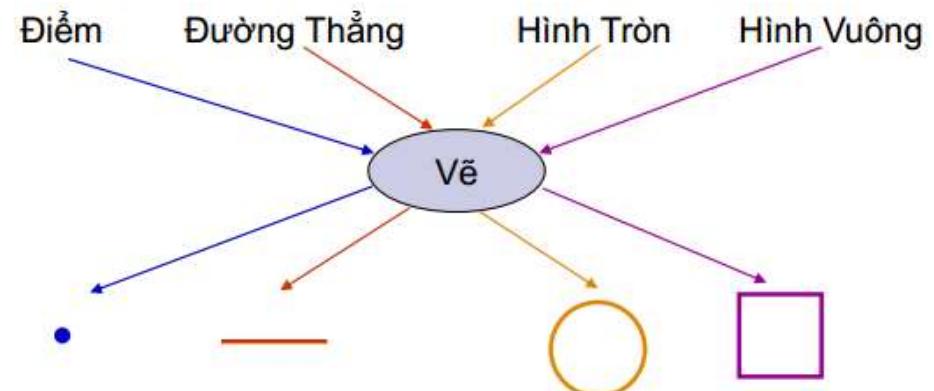
OBJECT & CLASS

- Tính thừa kế (inheritance):
 - Đa thừa kế:
 - Độ rộng tên các thuộc tính:



- Đa thừa kế không được chấp nhận bởi một số ngôn ngữ: Java, C#, ...

- Tính đa hình (polymorphism):
 - Đa hình: “nhiều hình thức”, hành động cùng tên có thể được thực hiện khác nhau đối với các đối tượng/các lớp khác nhau.
 - Ngũ cảnh khác → kết quả khác;
 - Khái niệm:
 - Khả năng đưa một phương thức có cùng tên trong các lớp con.
 - Thực hiện bởi:
 - Định nghĩa lại;
 - Nạp chồng.
 - Cơ chế dựa trên sự gán:
 - Kết gán sớm;
 - Kết gán muộn.





OBJECT & CLASS

- Tính đa hình (polymorphism):
 - Gọi phương thức show() từ đối tượng của lớp Người sẽ hiển thị tên, tuổi của người đó;
 - Gọi phương thức show() từ đối tượng của lớp Nhân viên sẽ hiển thị số lương của nhân viên;
 - Gọi phương thức show() từ đối tượng của lớp Sinh viên sẽ biết sinh viên đó học lớp nào.

Lớp *Người*

Thuộc tính:

Tên

Ngày sinh

Giới tính

Phương thức:

Nhập/Xem tên

Nhập/Xem ngày sinh

Nhập /Xem giới tính

Show

Lớp *Nhân viên* kế thừa từ lớp *Người*

Thuộc tính:

Lương

Phương thức:

Nhập/Xem lương

Show

Lớp *Sinh viên* kế thừa từ lớp *Người*

Thuộc tính:

Lớp

Phương thức:

Nhập/Xem lớp

Show



OBJECT & CLASS

- Tính đa hình (polymorphism):
 - Cho phép các lớp định nghĩa các phương thức trùng nhau: cùng tên, cùng tham số, cùng kiểu trả về:
 - Sự nạp chồng phương thức.
 - Khi gọi các phương thức trùng tên, dựa vào đối tượng đang gọi mà chương trình sẽ thực hiện phương thức của lớp tương ứng:
 - Kết quả sẽ khác nhau.



OBJECT & CLASS

- Tính đa hình (polymorphism):
 - Đa hình hàm - Functional polymorphism
 - Cơ chế cho phép một tên thao tác hoặc thuộc tính có thể được định nghĩa tại nhiều lớp và có thể có nhiều cài đặt khác nhau tại mỗi lớp trong các lớp đó;
 - Ví dụ: lớp Date cài 2 phương thức setDate(), một nhận tham số là một đối tượng Date, phương thức kia nhận 3 tham số day, month, year.
 - Đa hình đối tượng - Object polymorphism
 - Các đối tượng thuộc các lớp khác nhau có khả năng hiểu cùng một thông điệp theo các cách khác nhau;
 - Ví dụ: khi nhận được cùng một thông điệp draw(), các đối tượng Rectangle và Triangle hiểu và thực hiện các thao tác khác nhau.



CHƯƠNG 3

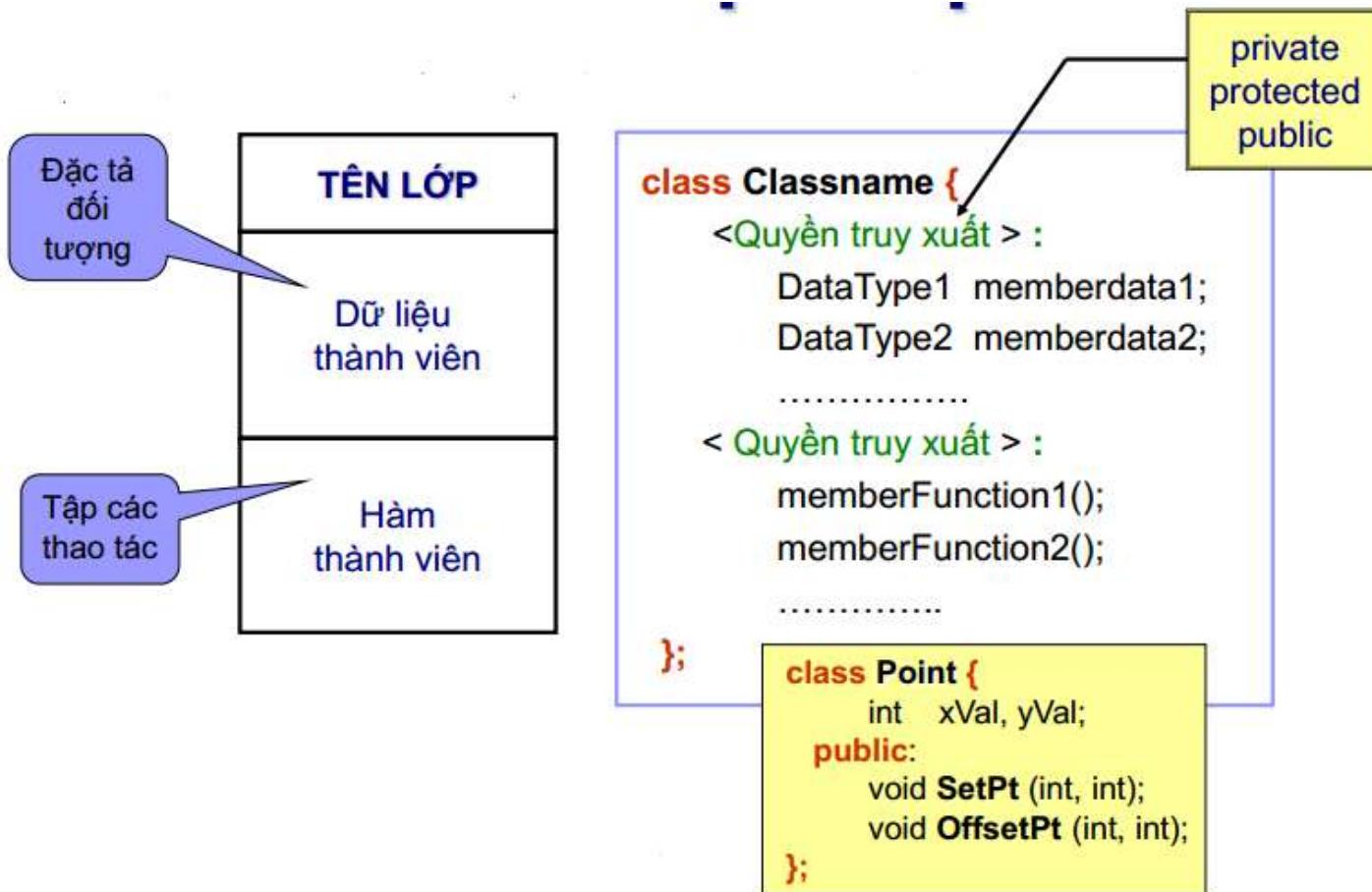
LỚP & ĐỐI TƯỢNG (C++)





KHÁI NIỆM LỚP

- Lớp: kiểu dữ liệu trừu tượng





CÂU TRÚC LỚP (C++)

- 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

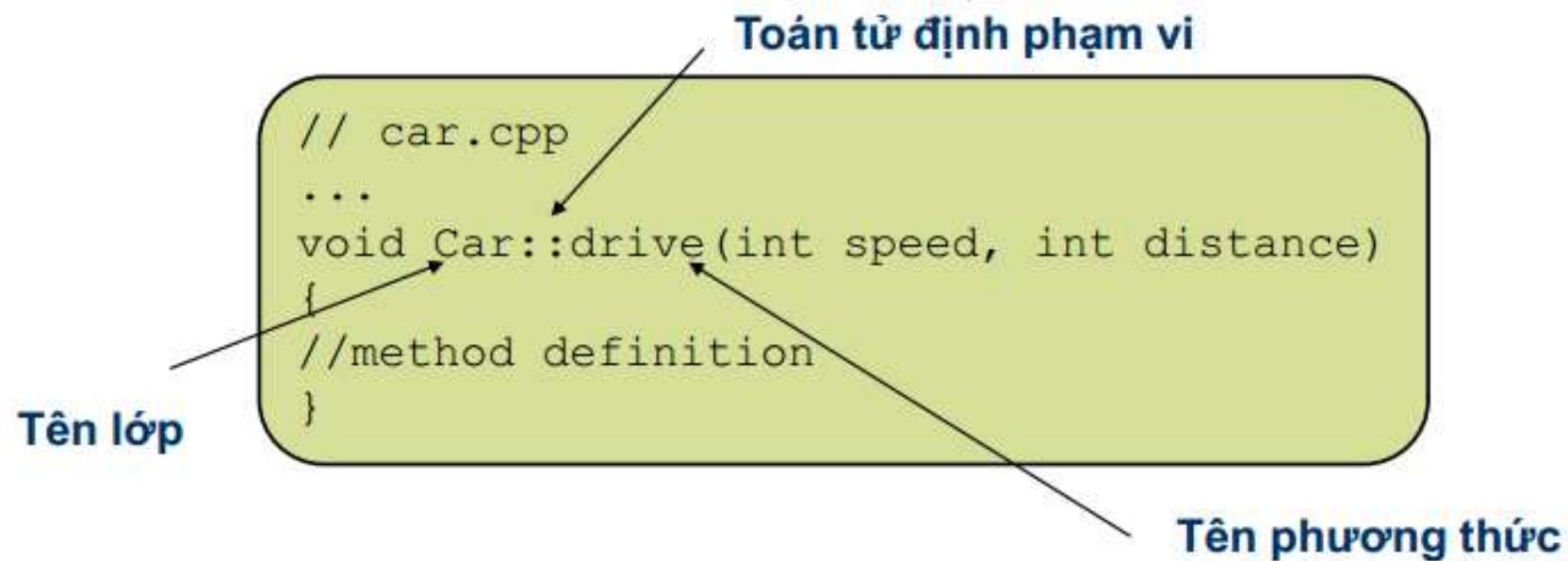
- Point.cpp

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



HÀM THÀNH VIÊN

- 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:





HÀM THÀNH VIÊN

- 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;
}
```

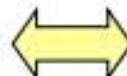


CON TRỎ THIS

- Con trả *this:

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

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



```
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 trả ***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 TRỎ THIS

- 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 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); \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;
}
```



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!

PhanSo

- Tỷ 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;
}
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.



COPY CONSTRUCTOR

- Hàm dựng sao chép (copy constructor):

- Ví dụ:

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



từ khóa 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:

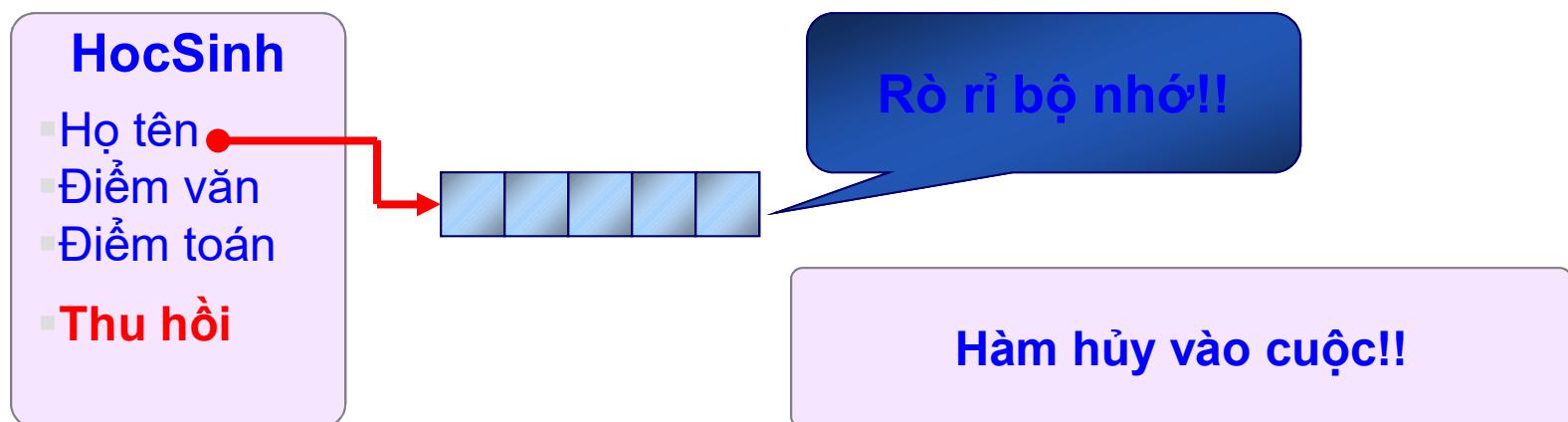
- Hàm dựng mặc định.
 - Hàm dựng có đầy đủ tham số.
 - Hàm dựng sao chép.





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.





DESTRUCTOR

- 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).



DESTRUCTOR

- 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 `~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;
}
```



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
nhiều lần
hàm hủy
được gọi?

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;  
};
```

Tập Các
Số Thực

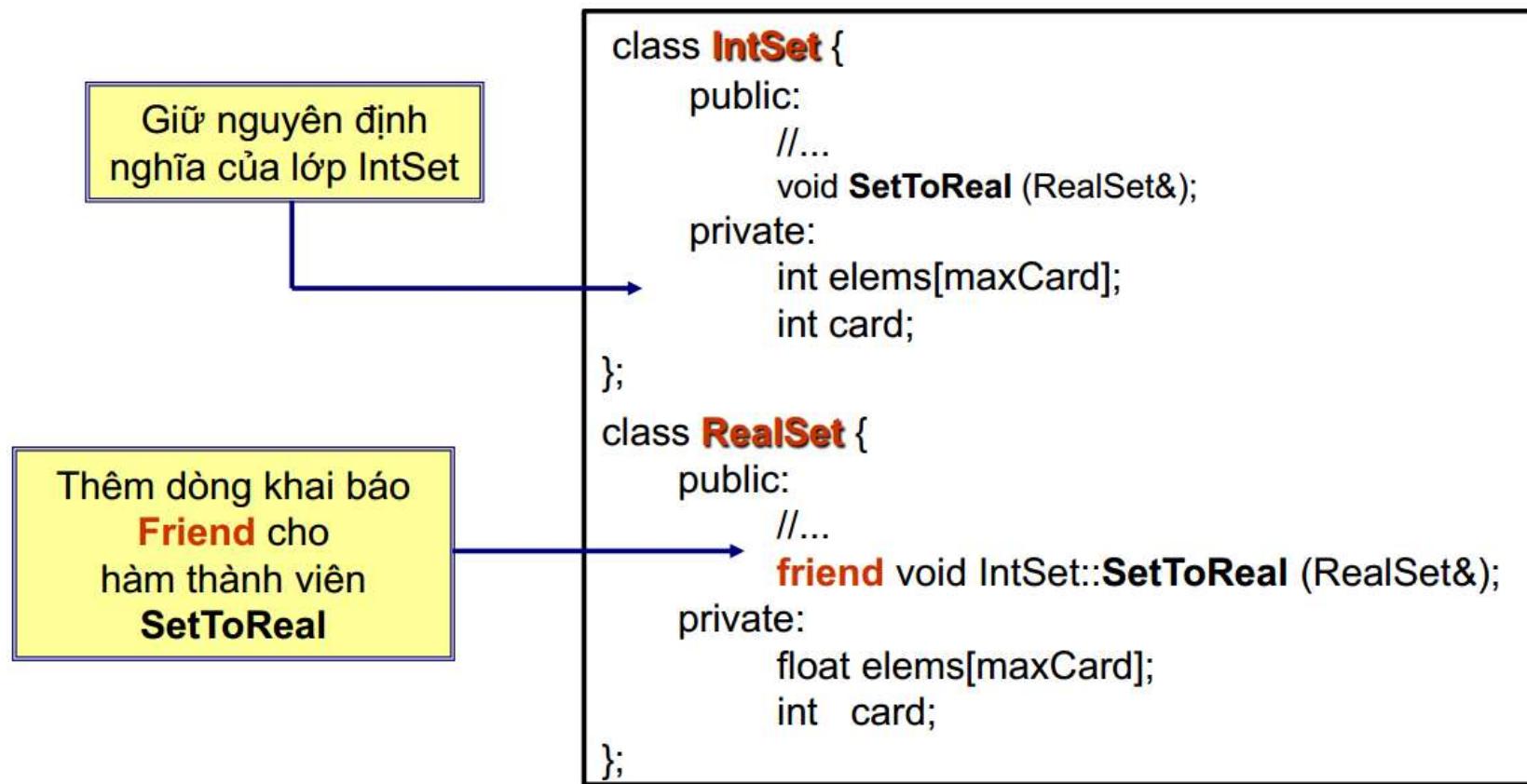
Hàm SetToReal
dùng để chuyển
tập số nguyên
thành tập số thực

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





- 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.

```

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.



- 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 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**



FRIEND

- 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);
};
```



FRIEND

- 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;
}
```



KHỞI TẠO THÀNH VIÊN DỮ LIỆU

- 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

- 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;
}
```



DANH SÁCH KHỞI TẠO THÀNH VIÊN

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

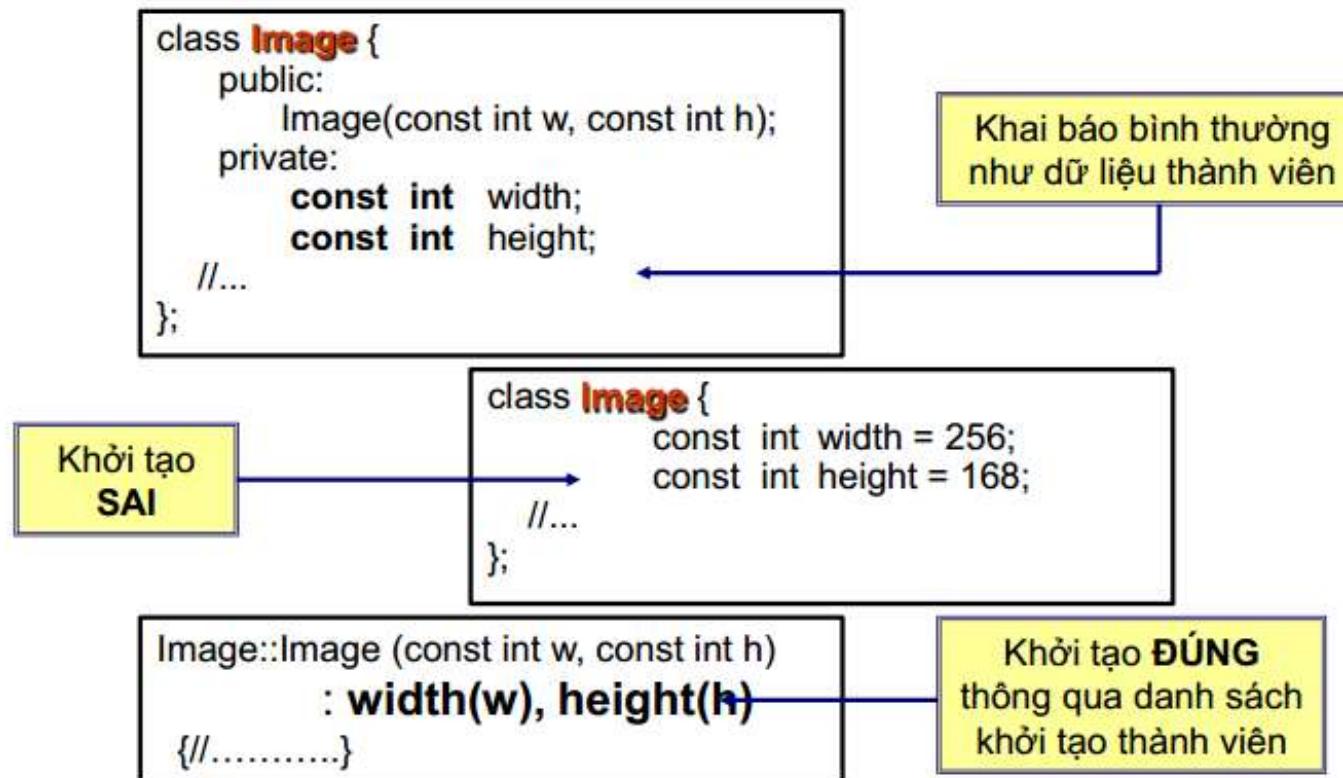
```
class Point {  
    int xVal, yVal;  
public:  
    Point (int x, int y) {  
        xVal = x;  
        yVal = y;  
    }  
    // .....  
};
```

```
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.





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 báo với trình biên dịch, vừa để tự gợi nhớ.



THÀNH VIÊN HẰNG

- 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
}
```



THÀNH VIÊN HẰNG

- 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);}
};
```



THÀNH VIÊN HẰNG

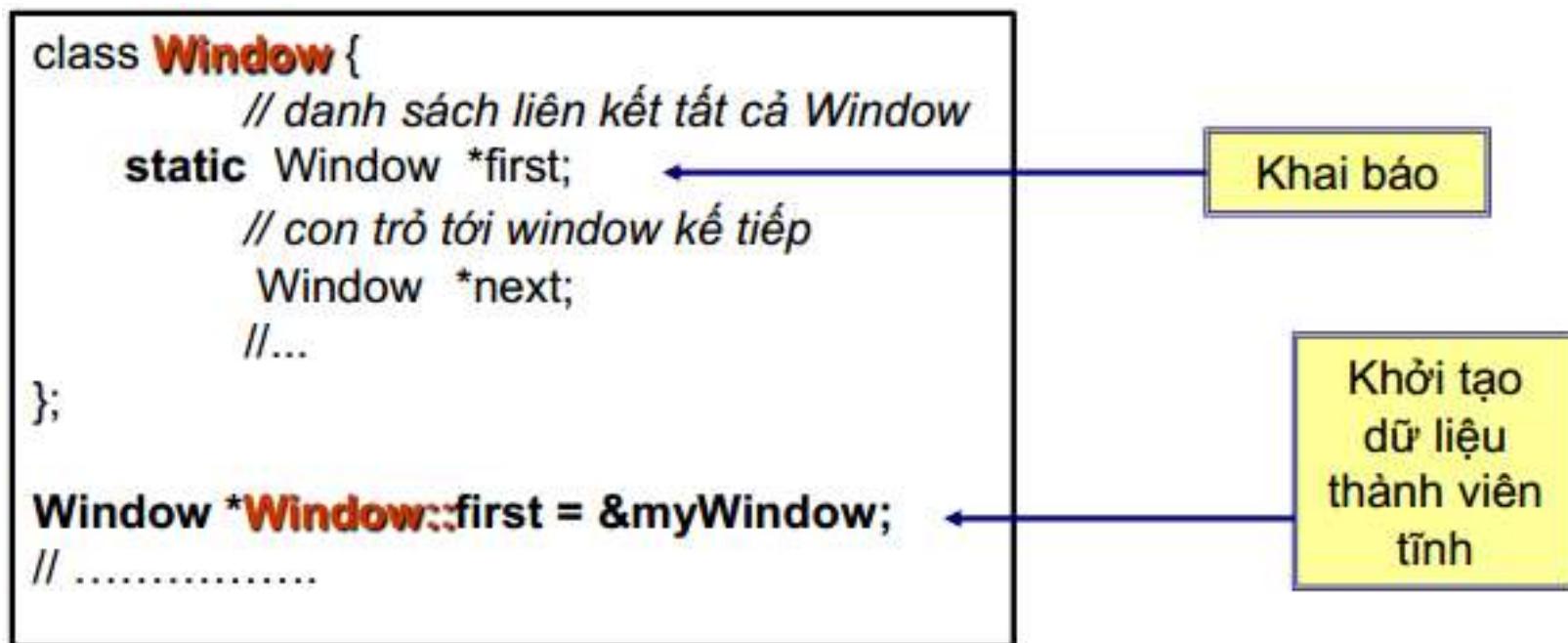
- Ví dụ:

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



THÀNH VIÊN 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 TÍNH

- 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;

};
```



THÀNH VIÊN TÍNH

- 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.



THÀNH VIÊN TĨNH

- **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;
```

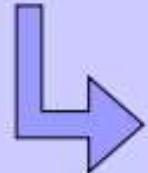
- Ta thường định nghĩa các thành viên tĩnh trong file chứa định nghĩa các phương thức;

- 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.



THÀNH VIÊN HẰNG TÍNH

- 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

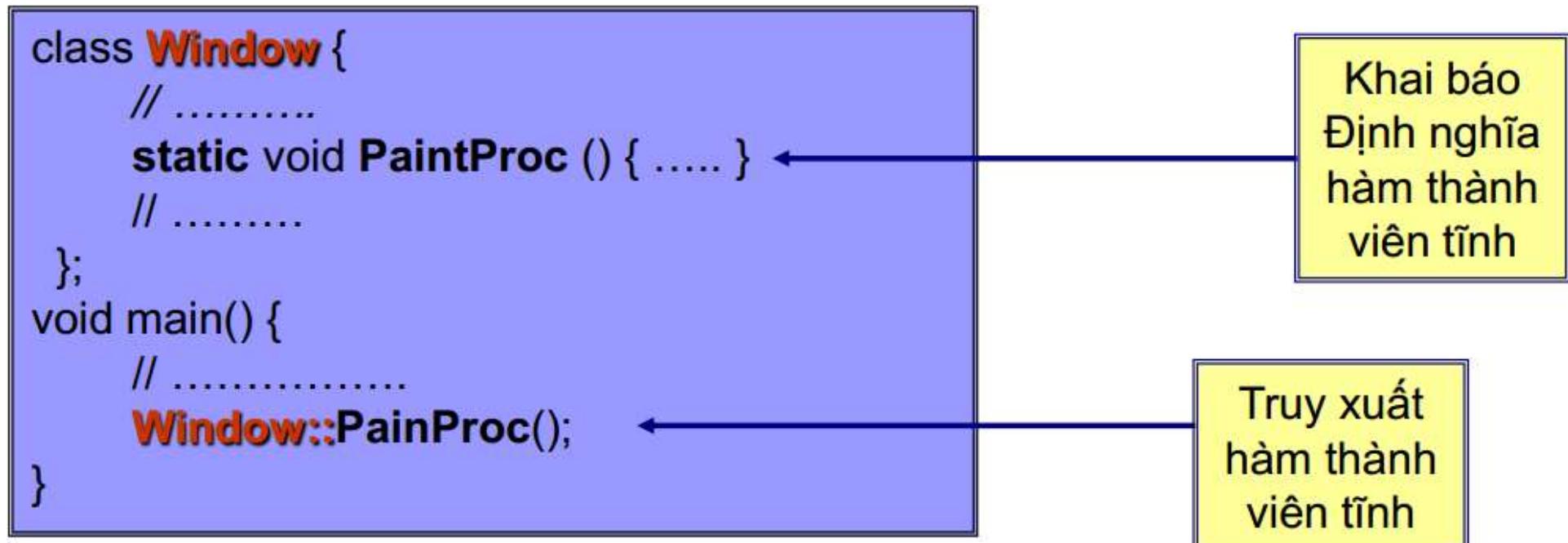
Tóm lại, ta nên khai báo:

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





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.



HÀM THÀNH VIÊN TĨNH

- 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;} }
```



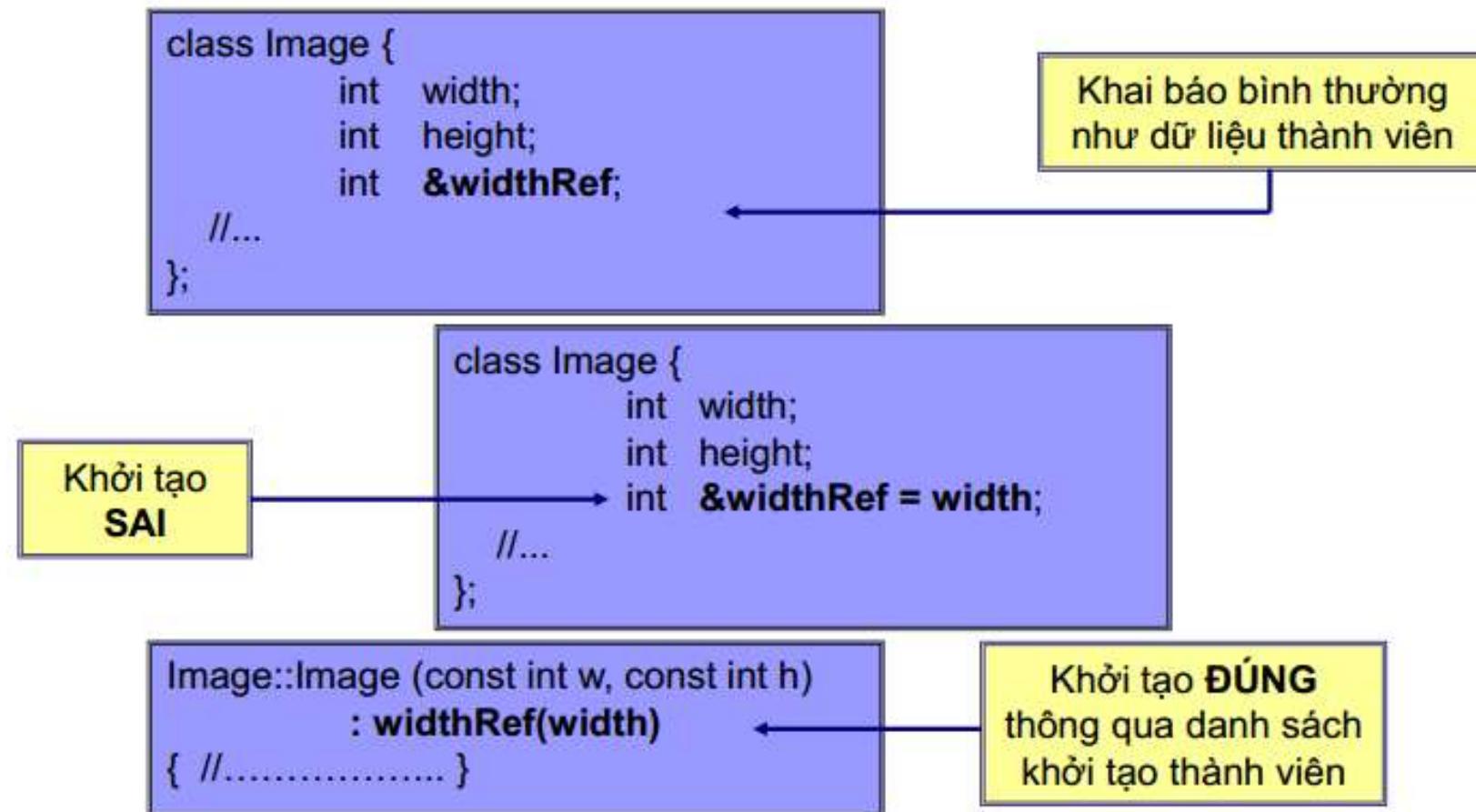
HÀM THÀNH VIÊN TĨNH

- 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;
}
```



THÀNH VIÊN THAM CHIẾU





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; //Error
```



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), (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).
 - 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) };



MẢNG CÁC ĐỐI TƯỢNG

- 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
```

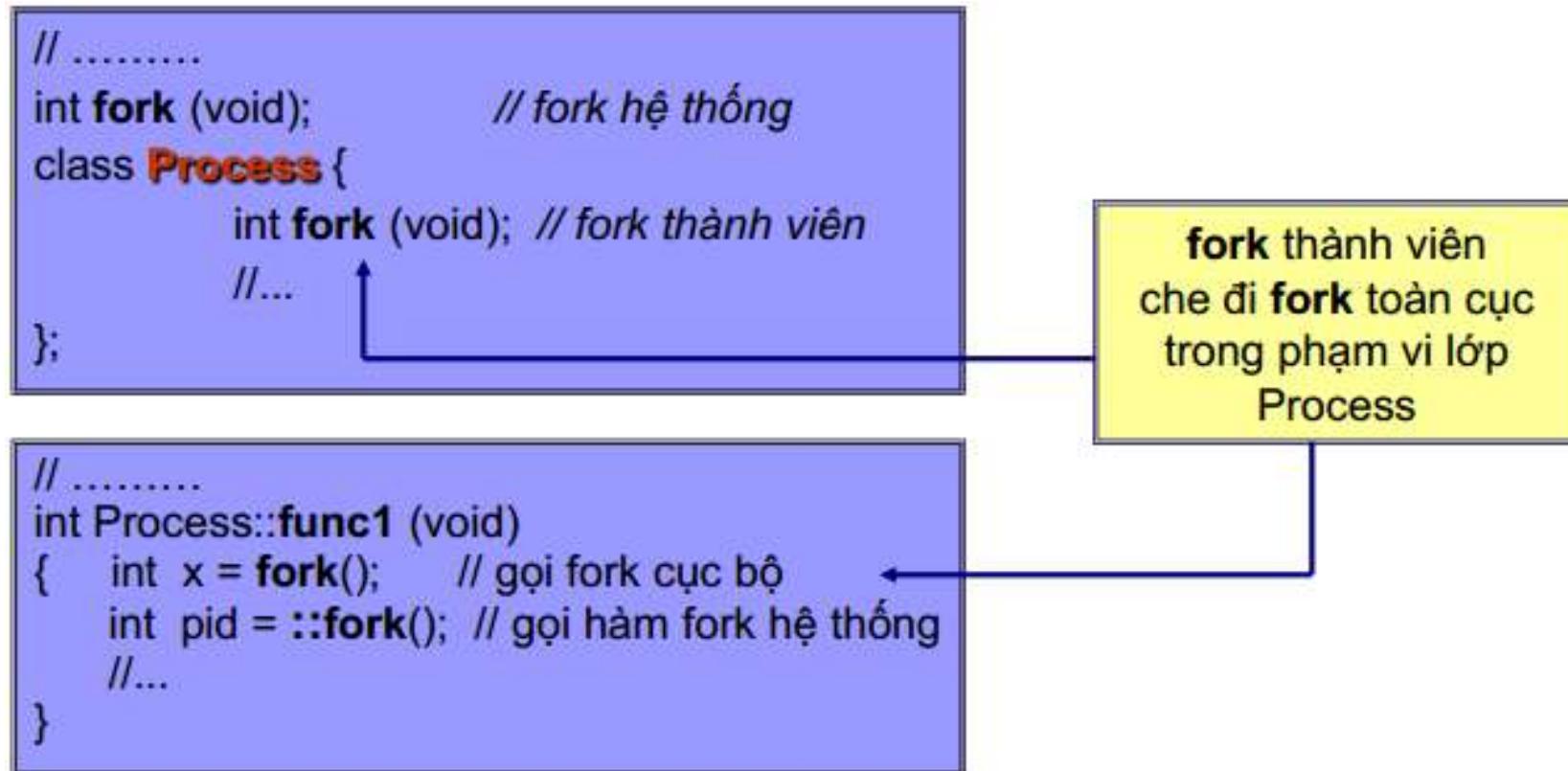
```
class Polygon {
public:
    //...
private:
    Point *vertices; // các đỉnh
    int nVertices; // số các đỉnh
};
```

Không cần biết kích thước mảng.



PHẠM VI LỚP

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





PHẠM VI LỚP

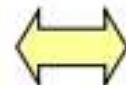
- Lớp toàn cục: đại đa số lớp trong C++;
- Lớp lồng nhau: lớp chứa đựng lớp;
- Lớp cục bộ: trong 1 hàm hoặc 1 khối.

```
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
```

- 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

- 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

- 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

- 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

- 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

- 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];
}
```

- $u = 0xDDCCBBAA; // Số hệ 16 \rightarrow b[4] ???$



CHƯƠNG 3

ĐA NĂNG HÓA





ĐA NĂNG HÓA HÀM

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



ĐA NĂNG HÓA HÀM

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



ĐA NĂNG HÓA TOÁN TỬ

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

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

- Các phép toán không thể tái định nghĩa:

.

.

*

::

?:

sizeof



ĐA NĂNG HÓA TOÁN TỬ

- Giới hạn của đa năng hóa toán tử:

- Toán tử gọi hàm **0** – là một toán tử nhiều ngôi;
- Thứ tự ưu tiên của một toán tử không thể được thay đổi bởi đa năng hóa;
- Tính kết hợp của một toán tử không thể được thay đổi bởi đa năng hóa. Các tham số mặc định không thể sử dụng với một toán tử đa năng hóa;
- Không thể thay đổi số các toán hạng mà một toán tử yêu cầu;
- Không thể thay đổi ý nghĩa của một toán tử làm việc trên các kiểu có sẵn.
- Không thể dùng đối số mặc định.



ĐA NĂNG HÓA TOÁN TỬ

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

aa@bb

@aa

aa@

aa.operator@(bb)

aa.operator@()

aa.operator@(int)

hoặc **operator@(aa,bb)**

hoặc **operator@(aa)**

hoặc **operator@(aa,int)**

Là phương thức của lớp

Là hàm toàn cục



ĐA NĂNG HÓA TOÁN TỬ

- Ví dụ: Sử dụng toán tử "+" để cộng hai đối tượng lớp Complex và trả về kết quả là một Complex.
 - Ta có thể khai báo hàm toàn cục sau:
const Complex operator+(const Complex& num1, const Complex& num2);
 - “x+y” sẽ được hiểu là “operator+(x,y)”;
 - Dùng từ khoá const để đảm bảo các toán hạng gốc không bị thay đổi.
 - Hoặc khai báo toán tử dưới dạng thành viên của Complex:
const Complex operator+(const Complex& num);
 - Đối tượng chủ của phương thức được hiểu là toán hạng thứ nhất của toán tử;
 - “x+y” sẽ được hiểu là “x.operator+(y)”.



ĐA NĂNG HÓA TOÁN TỬ

- Đa năng hóa toán tử **bằng hàm thành viên**:

- Khi đa năng hóa **0**, **[]**, **->** hoặc **=**, hàm đa năng hóa toán tử phải được khai báo như một thành viên lớp;
- Toán tử một ngôi hàm không có tham số, toán tử 2 ngôi hàm sẽ có 1 tham số.

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

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

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



ĐA NĂNG HÓA TOÁN TỬ

- Đa năng hóa toán tử **bằng hàm thành viên**:
 - $a @ b$: $a.\text{operator } @ (b)$
 - $x @ b$: với x là thuộc kiểu float, int, ... không thuộc kiểu lớp đang định nghĩa
 - Operator $@ (x, b)$



ĐA NĂNG HÓA TOÁN TỬ

- Đa năng hóa toán tử **bằng hàm toàn cục**: nếu toán hạng cực trái của toán tử là đối tượng thuộc lớp khác hoặc thuộc kiểu dữ liệu có sẵn:
 - Thường khai báo **friend**.

```
class Point {  
public:  
    Point (int x, int y) { Point::x = x; Point::y = y; }  
    friend Point operator + (Point &p, Point &q);  
    friend Point operator - (Point &p, Point &q);  
private:  
    int x, y;  
};  
Point operator + (Point &p, Point &q)  
{return Point(p.x + q.x,p.y + q.y); }  
Point operator - (Point &p, Point &q)  
{return Point(p.x - q.x,p.y - q.y); }  
  
void main() {  
    Point p1(10,20), p2(10,20);  
    Point p3 = p1 + p2;           Point p4 = p1 - p2;  
    Point p5 =operator + (p3, p4); Point p6 = operator - (p3, p4);  
};
```

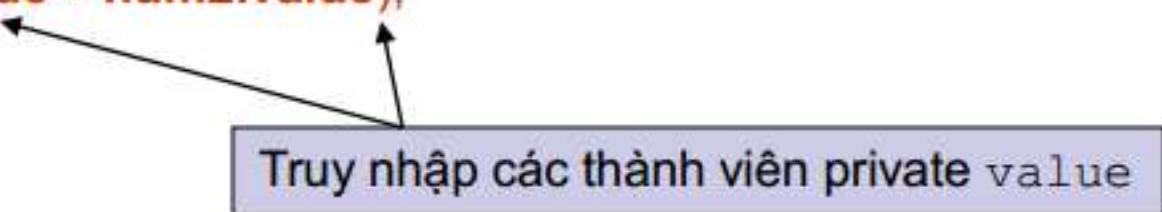
Có 2 tham số
(Nếu là toán tử hai ngôi)



ĐA NĂNG HÓA TOÁN TỬ

- Toán tử là hàm toàn cục
 - Quay lại với ví dụ về phép cộng cho Complex, ta có thể khai báo hàm định nghĩa phép cộng tại mức toàn cục:
const Complex operator+(const Complex& num1, const Complex& num2);
 - Khi đó, ta có thể định nghĩa toán tử đó như sau:

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



Truy nhập các thành viên private value
- Giải pháp: dùng hàm friend
 - **friend** cho phép một lớp cấp quyền truy nhập tới các phần nội bộ của lớp đó cho một số cấu trúc được chọn.



ĐA NĂNG HÓA TOÁN TỬ

- Toán tử là hàm toàn cục

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

```
class Complex
{
public:
    Complex(int value = 0);
    ~Complex();

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

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



ĐA NĂNG HÓA TOÁN TỬ

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

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



ĐA NĂNG HÓA TOÁN TỬ

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



ĐA NĂNG HÓA TOÁN TỬ XUẤT <<

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



ĐA NĂNG HÓA TOÁN TỬ XUẤT <<

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

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

- Định nghĩa toán tử:

```
ostream& operator<<( ostream& out, const Complex& num )
{
    out << "(" << num.R << ", " << num.I << ")";
    return out;
};
```



ĐA NĂNG HÓA TOÁN TỬ NHẬP >>

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



ĐA NĂNG HÓA TOÁN TỬ NHẬP >>

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

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

- Định nghĩa toán tử:

```
istream& operator>>(istream& in, Complex& num)
{
    cout<<"Nhập phần thực: "; in >> num.R;
    cout<<"Nhập phần ảo: "; in >> num.I;
    return in;
};
```



ĐA NĂNG HÓA TOÁN TỬ []

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

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



ĐA NĂNG HÓA TOÁN TỬ []

- Ví dụ:

```
class Vector
{
    private:    int Size; int *Data;
public:
    Vector(int S=2,int V=0);
    ~Vector();
    void Print() const;
    int & operator [] (int I);
};

Int& Vector::operator [] (int I)
{
    static int tam=0;
    return (i>0)&&(i<Size)?Data[I]:tam;
}
```



ĐA NĂNG HÓA TOÁN TỬ []

- Ví dụ:

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



ĐA NĂNG HÓA TỐC TỬ 0

- Định nghĩa là hàm thành viên.

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

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



CHUYỂN KIỂU

- Muốn thực hiện các phép cộng:

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

→ Có thể định nghĩa thêm 2 toán tử:

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



CHUYỂN KIỂU

- Chuyển đổi kiểu: ngôn ngữ định nghĩa sẵn.

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

→ Định nghĩa phép chuyển đổi kiểu

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

Chuyển kiểu
5 ⇌ Point(5)



CHUYỂN KIỂU

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



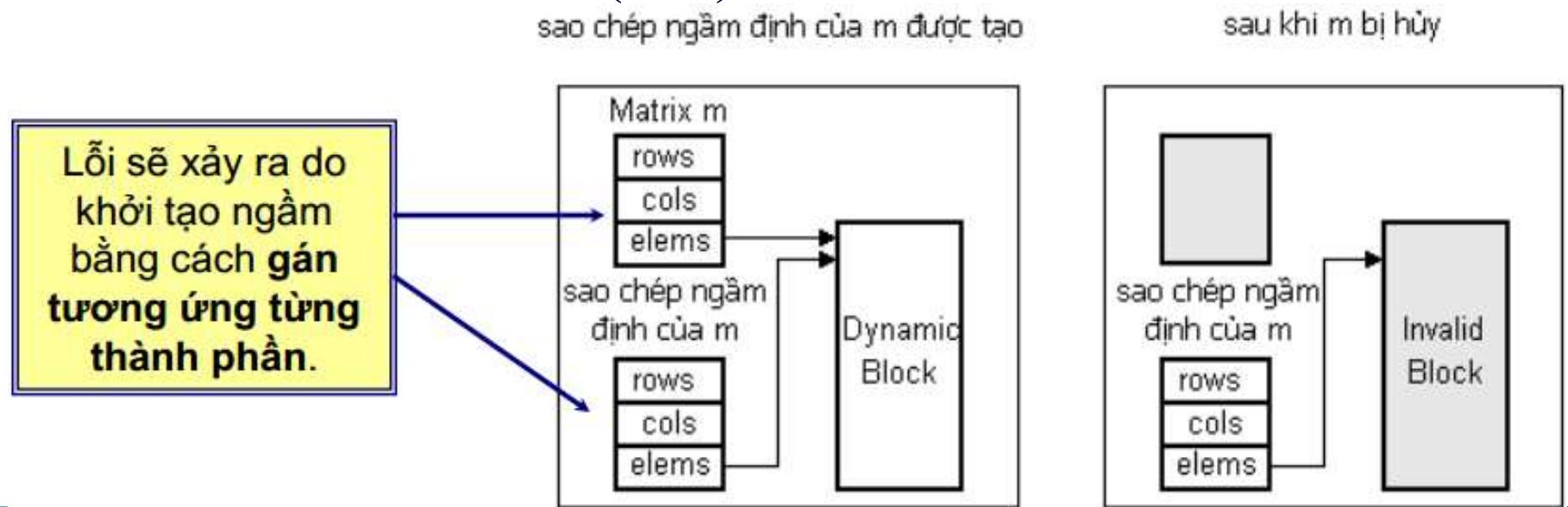
CHUYỂN KIỂU

- Ví dụ:

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

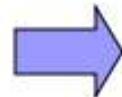
KHỞI TẠO NGẦM ĐỊNH

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





KHỞI TẠO NGÀM ĐỊNH



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

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

```
class Matrix {
    //....
    Matrix(const Matrix&);
};

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



GÁN NGÀM ĐỊNH

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

Hàm
thành
viên

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



PHÉP GÁN =

- Một trong những toán tử hay được overload nhất;
 - Cho phép gán cho đối tượng này một giá trị dựa trên một đối tượng khác;
 - Copy constructor cũng thực hiện việc tương tự → định nghĩa toán tử gán gần như giống hệt định nghĩa của copy constructor.
- Ta có thể khai báo phép gán cho lớp **MyNumber** như sau:

const MyNumber& operator=(const MyNumber& num);

- Phép gán nên luôn trả về một tham chiếu tới đối tượng đích (đối tượng được gán trị cho);
- Tham chiếu được trả về phải là const để tránh trường hợp a bị thay đổi bằng lệnh " $(a = b) = c;$ " (lệnh đó không tương thích với định nghĩa gốc của phép gán).



PHÉP GÁN =

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

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



PHÉP GÁN =

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



ĐA NĂNG HÓA TOÁN TỬ ++ & --

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



ĐA NĂNG HÓA TOÁN TỬ ++ & --

- Phép tăng ++
 - Giá trị trả về:
 - Tăng trước **++num**:
 - Trả về tham chiếu (**MyNumber &**);
 - Giá trị trái - lvalue (có thể được gán trị).
 - Tăng sau **num++**:
 - Trả về giá trị (giá trị cũ trước khi tăng);
 - Trả về đối tượng tạm thời chưa giá trị cũ;
 - Giá trị phải - rvalue (không thể làm đích của phép gán).
 - prototype
 - Tăng trước:
`MyNumber& MyNumber::operator++();`
 - Tăng sau:
`const MyNumber MyNumber::operator++(int).`



ĐA NĂNG HÓA TOÁN TỬ ++ & --

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

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

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

Không thể trả về tham chiếu



THAM SỐ & KIỂU TRẢ VỀ

- Cũng như khi overload các hàm khác, khi overload một toán tử, ta cũng có nhiều lựa chọn về việc truyền tham số và kiểu trả về:
 - Chỉ có hạn chế rằng ít nhất một trong các tham số phải thuộc kiểu người dùng tự định nghĩa
- Ở đây, ta có một số lời khuyên về các lựa chọn
- Các toán hạng:
 - Nên sử dụng tham chiếu mỗi khi có thể (đặc biệt là khi làm việc với các đối tượng lớn)
 - Luôn luôn sử dụng tham số là hằng tham chiếu khi đối số sẽ không bị sửa đổi:

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

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



THAM SỐ & KIỂU TRẢ VỀ

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



THAM SỐ & KIỂU TRẢ VỀ

- Giá trị trả về:

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

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



THAM SỐ & KIỂU TRẢ VỀ

- Cách đã dùng để trả về kết quả của toán tử:

```
const MyNumber MyNumber::operator+(const MyNumber& num)
```

```
{
```

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

```
    return result;
```

```
}
```

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

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

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

- C++ cung cấp một cách hiệu quả hơn:

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



THAM SỐ & KIỂU TRẢ VỀ

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

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



ĐA NĂNG HÓA TOÁN TỬ NEW & DELETE

- Hàm new và delete mặc định của ngôn ngữ:
 - Nếu đối tượng kích thước nhỏ, có thể sẽ gây ra quá nhiều khôi nhỏ → chậm;
 - Không đáng kể khi đối tượng có kích thước lớn.
→ Toán tử new và delete ít được tái định nghĩa.
- Có 2 cách đa năng hóa toán tử new và delete:
 - Có thể đa năng hóa một cách toàn cục nghĩa là thay thế hẳn các toán tử **new** và **delete** mặc định;
 - Đa năng hóa các toán tử **new** và **delete** với tư cách là hàm thành viên của lớp nếu muốn các toán tử **new** và **delete** áp dụng đối với lớp đó.
 - Khi chúng ta dùng **new** và **delete** đối với lớp nào đó, trình biên dịch sẽ kiểm tra xem **new** và **delete** có được định nghĩa riêng cho lớp đó hay không; nếu không thì dùng **new** và **delete** toàn cục (có thể đã được đa năng hóa).



ĐA NĂNG HÓA TOÁN TỬ NEW & DELETE

- Hàm toán tử của toán tử new và delete có prototype như sau:
void * operator new(size_t size);
void operator delete(void * ptr);
 - Trong đó tham số kiểu size_t được trình biên dịch hiểu là kích thước của kiểu dữ liệu được trao cho toán tử new.
- Ví dụ:
 - Đa năng hóa toán tử new và delete toàn cục;
 - Đa năng hóa toán tử new và delete cho một lớp.



ĐA NĂNG HÓA COMMA (,



ĐA NĂNG HÓA TOÁN TỬ

- Bài tập:
 - Cài đặt lớp String với các phép toán cơ bản =, +=, so sánh, [], (), >>, <<, ...
 - Cài đặt lớp Date với các phép toán ++, --, +=, >>, <<, ...



CHƯƠNG 5

THÙA KẾ





TÁI SỬ DỤNG

- Sử dụng lại:
 - Tồn tại nhiều loại đối tượng có các thuộc tính và hành vi tương tự hoặc liên quan đến nhau:
 - Person, Student, Manager, ...
 - Xuất hiện nhu cầu sử dụng lại các mã nguồn đã viết
 - Sử dụng lại thông qua *copy*;
 - Sử dụng lại thông qua quan hệ *has_a*;
 - Sử dụng lại thông qua cơ chế “**kế thừa**”;



TÁI SỬ DỤNG

- Sử dụng lại:
 - Copy mã nguồn:
 - Tốn công, dễ nhầm;
 - Khó sửa lỗi do tồn tại nhiều phiên bản.
 - Quan hệ has_a:
 - Sử dụng lớp cũ như là thành phần của lớp mới
 - Sử dụng lại cài đặt với giao diện mới:
 - Phải viết lại giao diện;
 - Chưa đủ mềm dẻo.



TÁI SỬ DỤNG

- Ví dụ: has_a

```
class Person
{
    private:
        String name;
        Date birthday;
    public:
        String getName() { return name; }
        //...
};

class Employee
{
    private:
        Person me;
        double salary;
    public:
        String getName() { return me.getName(); }
        //...
};
```



4.1 KHÁI NIỆM

- Ví dụ: has_a

```
class Manager
{
    private:
        Employee me;
        Employee assistant;
    public:
        setAssistant(Employee e) {...}
        //...
};

//...
Manager junior;
Manager senior;
senior.setAssistant(junior); //error
```



KẾ THỪA

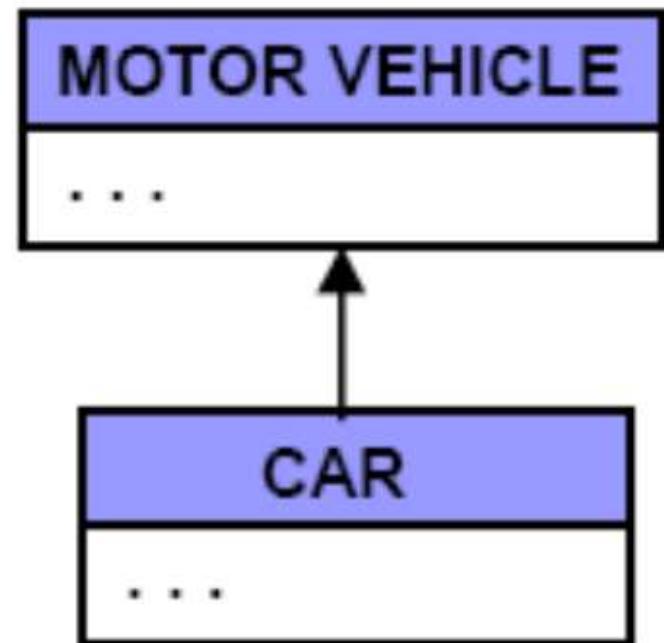
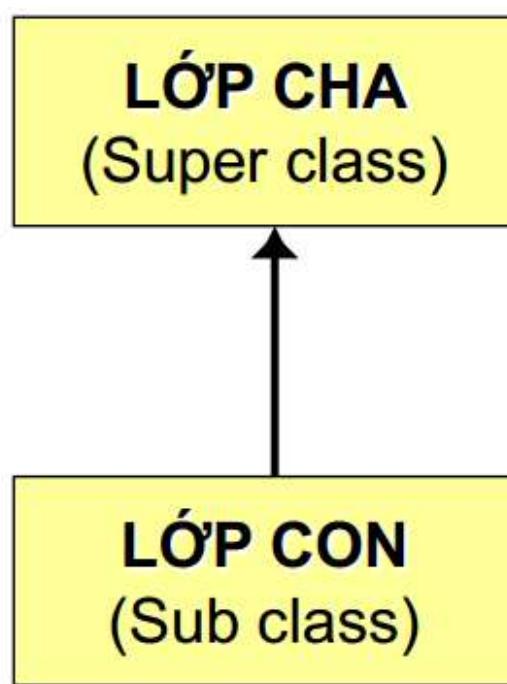
- Dựa trên quan hệ `is_a`;
- Thừa hưởng lại các thuộc tính và phương thức đã có;
- Chi tiết hóa cho phù hợp với mục đích sử dụng mới;
 - Thêm các thuộc tính mới;
 - Thêm hoặc hiệu chỉnh các phương thức.
- Ích lợi: có thể tận dụng lại
 - Các thuộc tính chung;
 - Các hàm có thao tác tương tự.
- Có 2 loại kế thừa: Đơn thừa kế & Đa thừa kế.



KẾ THỪA

- Kế thừa:

Lớp cơ sở
(Base class)





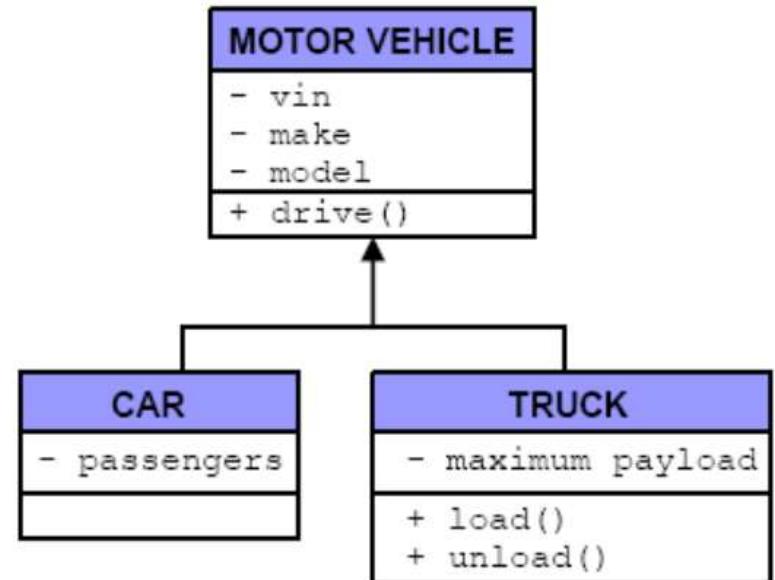
KẾ THỪA

- Lớp cha – superclass (hoặc lớp cơ sở - base class)
 - Lớp tổng quát hơn trong một quan hệ “là”;
 - Các đối tượng thuộc lớp cha có cùng tập thuộc tính và hành vi S.
- Lớp con – subclass (hoặc lớp dẫn xuất – derived class)
 - Lớp cụ thể hơn trong một quan hệ “là”;
 - Các đối tượng thuộc lớp con có cùng tập thuộc tính và hành vi S (do thừa kế từ lớp cha), kèm thêm tập thuộc tính và hành vi S' của riêng lớp con.
- Quan hệ thừa kế - Inheritance hay còn gọi là quan hệ “là”;
- Ta nói rằng lớp con “thừa kế từ” lớp cha, hoặc lớp con “được dẫn xuất từ” lớp cha.



OBJECT RELATIONSHIP DIAGRAM – ORD

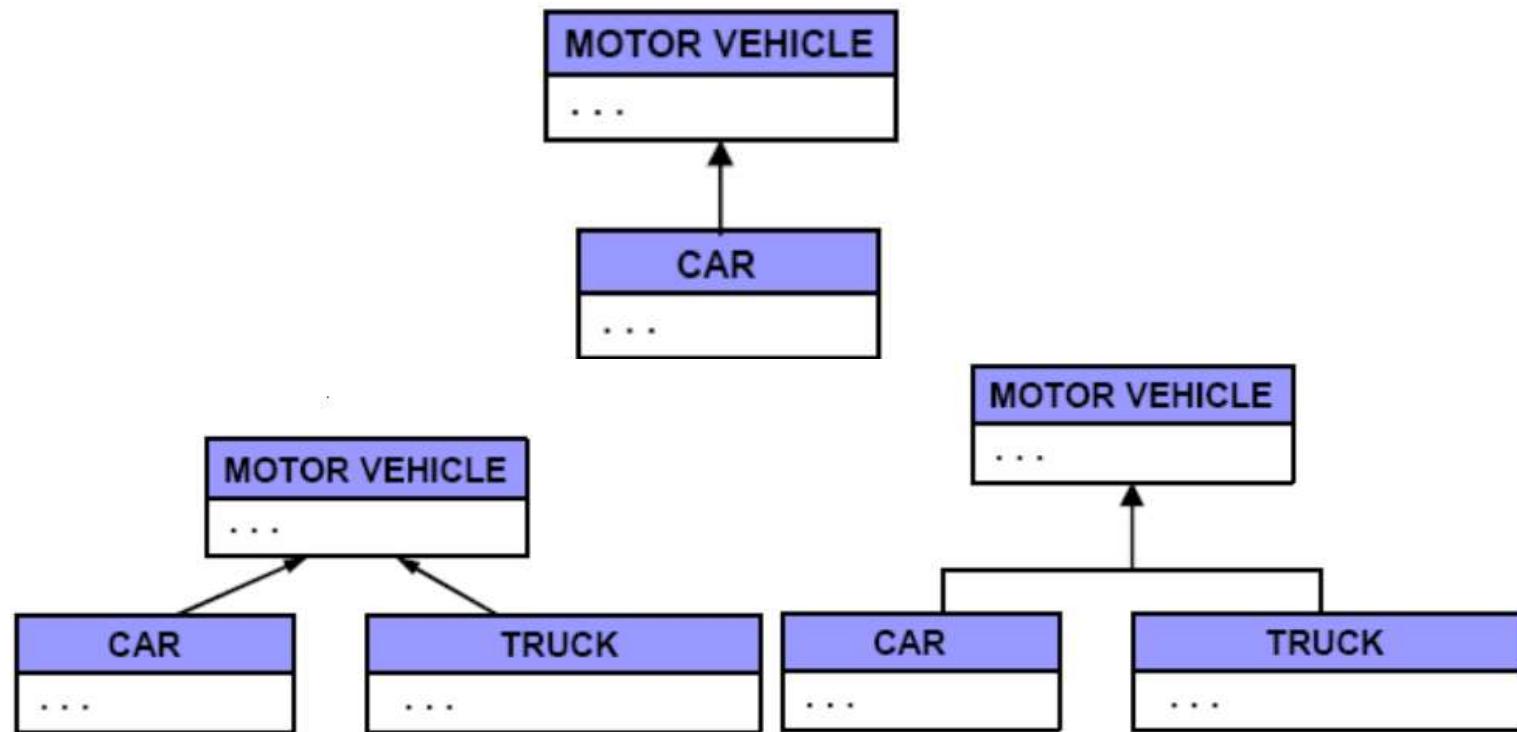
- Sơ đồ quan hệ đối tượng (Object Relationship Diagram – ORD)
 - Khi mô tả các quan hệ thừa kế giữa các lớp trong ORD, mục đích là để chỉ rõ sự khác biệt giữa các lớp tham gia quan hệ đó:
 - Một lớp con khác lớp cha của nó ở chỗ nào?
 - Các lớp con khác nhau ở chỗ nào?





OBJECT RELATIONSHIP DIAGRAM – ORD

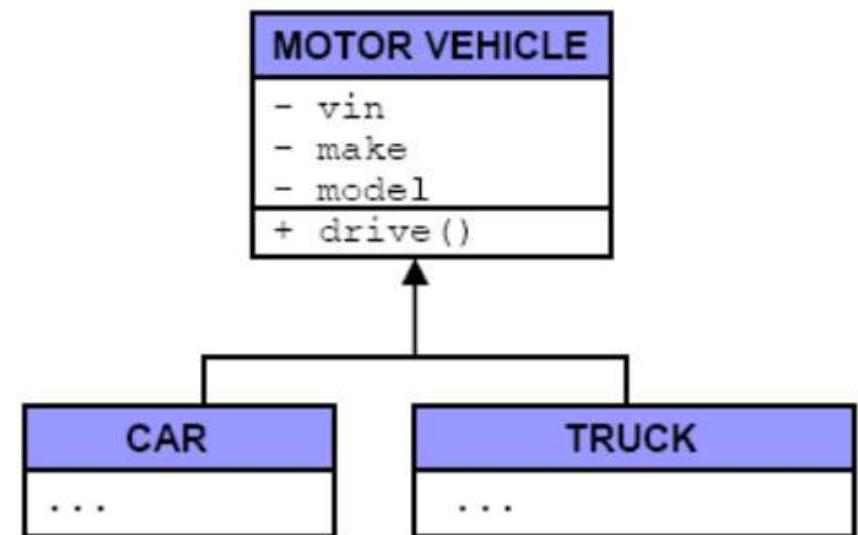
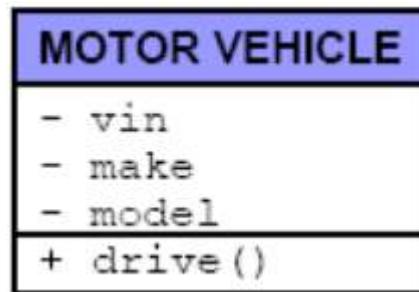
- Biểu diễn một quan hệ thừa kế giữa hai lớp bằng một mũi tên trỏ từ lớp con đến lớp cha;
- Có thể biểu diễn quan hệ với nhiều lớp con theo một trong hai kiểu sau:





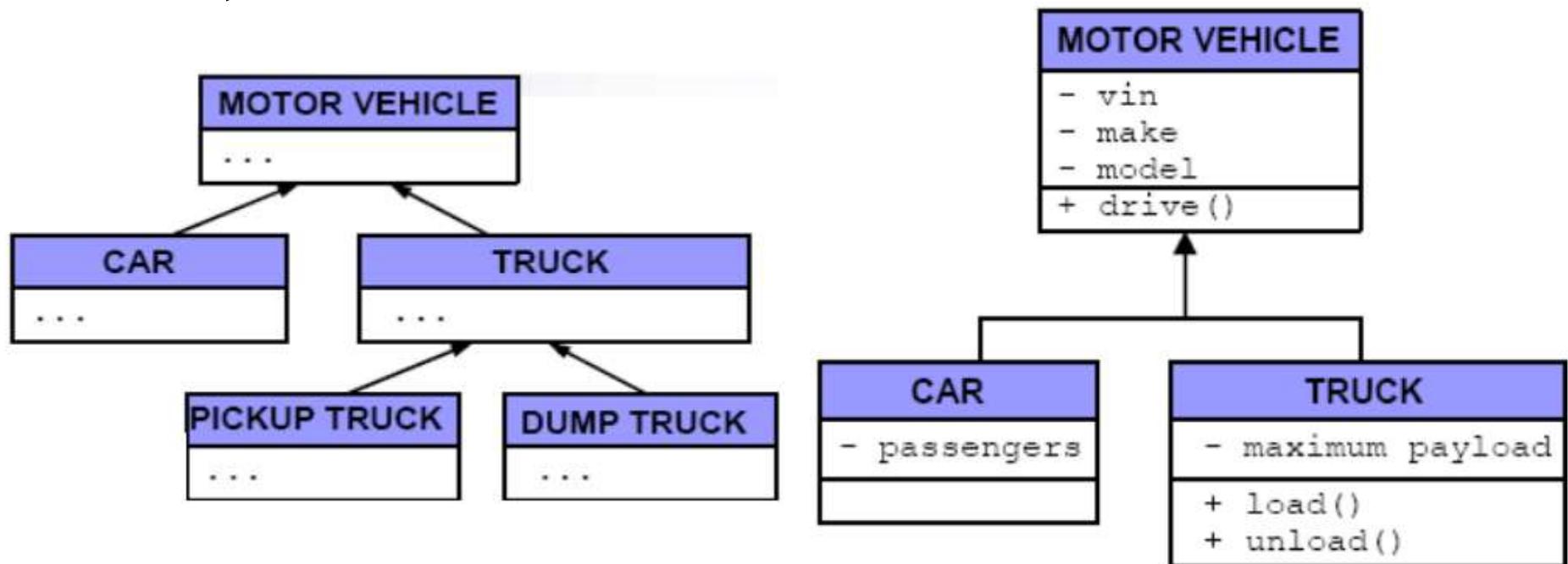
OBJECT RELATIONSHIP DIAGRAM – ORD

- Biểu diễn sơ đồ quan hệ:
 - Biểu diễn các thuộc tính và hành vi
 - Giả sử lớp MotorVehicle có các thuộc tính vin (số đăng ký xe), make (hãng), model (kiểu), và hành vi drive (lái).
 - Ta có sơ đồ quan hệ:
 - Mọi xe tải, xe ca đều có các thuộc tính vin, make, model, và hành vi drive.



SƠ ĐỒ QUAN HỆ ĐỐI TƯỢNG

- Mỗi xe ca đều có các thuộc tính vin, make, model, và hành vi drive, kèm theo thuộc tính passengers;
- Mỗi xe tải đều có các thuộc tính vin, make, model, và hành vi drive, kèm theo thuộc tính maximum payload và các hành vi load, unload.





CÂY THÙA KẾ

- Cây thừa kế:
 - Các quan hệ thừa kế luôn được biểu diễn với các lớp con đặt dưới lớp cha để nhấn mạnh bản chất phả hệ của quan hệ;
 - Ta cũng có thể có nhiều tầng thừa kế, tại mỗi tầng, các lớp con tiếp tục thừa kế từ lớp cha:
 - Một xe chở rác (dump truck) là xe tải, và cũng là xe chạy bằng động cơ
 - Nghĩa là các lớp con được thừa kế các thuộc tính và hành vi của mọi lớp cơ sở bên trên nó:
 - Một xe chở rác có mọi thuộc tính và hành vi của xe động cơ, kèm theo mọi thuộc tính và hành vi của xe tải, kèm theo các thuộc tính và hành vi của riêng xe rác.



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Định nghĩa lớp con:

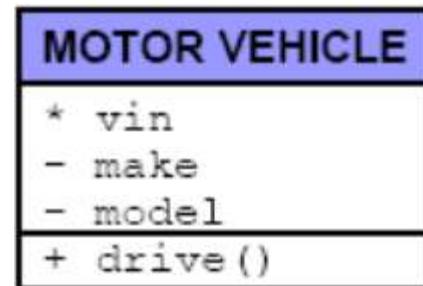
```
class MyDerivedClass :<keyword> MyBaseClass {  
    ...  
};
```

- Mô tả một lớp con cũng giống như biểu diễn nó trong ORD, ta chỉ tập trung vào những điểm khác với lớp cha:
- Ích lợi:
 - Đơn giản hóa khai báo lớp;
 - Hỗ trợ nguyên lý đóng gói của hướng đối tượng;
 - Hỗ trợ tái sử dụng code (sử dụng lại định nghĩa của các thành viên dữ liệu và phương thức);
 - Việc che dấu thông tin cũng có thể có vai trò trong việc tạo cây thừa kế.



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Ví dụ:
 - Bắt đầu bằng định nghĩa lớp cơ sở, **MotorVehicle**



```
class MotorVehicle
{
public:
    MotorVehicle(int vin, string make, string model);
    ~MotorVehicle();
    void drive(int speed, int distance);
private:
    int vin;
    string make;
    string model;
};
```



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Ví dụ:

- Ta định nghĩa constructor, destructor, và hàm drive() (ở đây, ta chỉ định nghĩa tạm drive())

```
MotorVehicle::MotorVehicle(int vin, string make, string model)
{
    this->vin = vin;
    this->make = make;
    this->model = model;
}

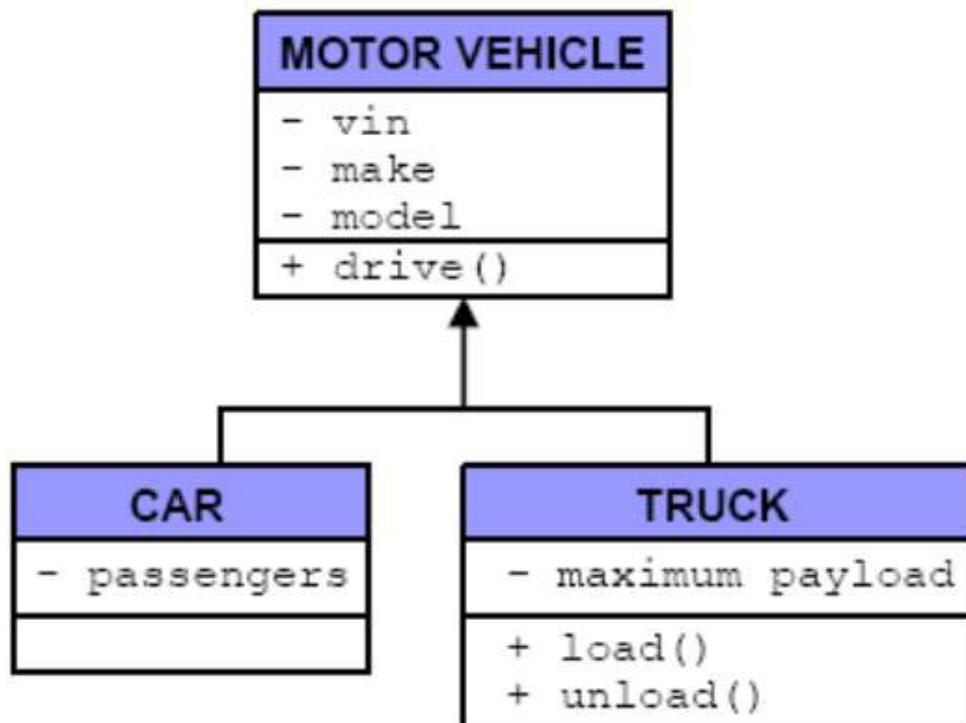
// We could actually use // the default destructor
MotorVehicle::~MotorVehicle() { /*...*/ }

void MotorVehicle::drive(int speed, int distance)
{
    cout << "Dummy drive() of MotorVehicle." << endl;
}
```



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Ví dụ:
 - Tạo lớp con Car



Chỉ rõ quan hệ giữa lớp con **Car** và lớp cha **MotorVehicle**

```
class Car : public MotorVehicle
{
public:
    Car (int passengers);
    ~Car();
private:
    int passengers;
};
```



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Định nghĩa lớp con:

- Hiện giờ constructor của lớp Car chỉ nhận 1 tham số passengers, trong khi các đối tượng Car cũng có tất cả các thành viên được thừa kế từ

MotorVehicle Car (int passengers);

- Do vậy, trừ khi ta muốn dùng giá trị mặc định cho các thành viên được thừa kế, ta nên truyền thêm tham số cho constructor để khởi tạo vin, make, model.

```
class Car : public MotorVehicle {  
public:  
    Car (int vin, string make, string model, int passengers);  
    ~Car();  
private:  
    int passengers;  
};  
Car a(...);
```

Quy ước: đặt các tham số
cho lớp cha lên đầu danh sách.



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Tối thiểu, ta sẽ định nghĩa constructor và (có thể cả) destructor:
 - Các lớp con không thừa kế constructor và destructor của lớp cha, do việc khởi tạo và huỷ các lớp khác nhau là khác nhau.
- Phiên bản constructor đầu tiên mà ta có thể nghĩ tới:

```
Car::Car(int vin, string make, string model, int passengers)
{
    this->vin = vin;
    this->make = make;
    this->model = model;
    this->passengers = passengers;
}
Car::~Car() {}
```



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Nhược điểm:
 - Trực tiếp truy nhập các thành viên dữ liệu của lớp cơ sở:
 - Thiếu tính đóng gói: phải biết sâu về chi tiết lớp cơ sở và phải can thiệp sâu;
 - Không tái sử dụng mã khởi tạo của lớp cơ sở;
 - Không thể khởi tạo các thành viên private của lớp cơ sở do không có quyền truy nhập.
- Nguyên tắc: một đối tượng thuộc lớp con bao gồm một đối tượng lớp cha cộng thêm các tính năng bổ sung của lớp con:
 - Một thể hiện của lớp cơ sở sẽ được tạo trước, sau đó "gắn" thêm các tính năng bổ sung của lớp dẫn xuất.
- Vậy, ta sẽ sử dụng constructor của lớp cơ sở.



ĐỊNH NGHĨA LỚP DẪN XUẤT

- Để sử dụng constructor của lớp cơ sở, ta dùng danh sách khởi tạo của constructor (tương tự như khi khởi tạo các hằng thành viên):
 - Cách duy nhất để tạo phần thuộc về thể hiện của lớp cha tạo trước nhất.
- Ta sửa định nghĩa constructor như sau:

```
Car::Car(int vin, string make, string model, int passengers)
        :MotorVehicle(vin,make,model)
```

```
{  
    this->passengers = passengers;  
}
```

Ta không cần khởi tạo các thành viên **vin**, **make**, **model** từ bên trong constructor của **Car** nữa

Gọi constructor của **MotorVehicle** với các tham số **vin**, **make**, **model**



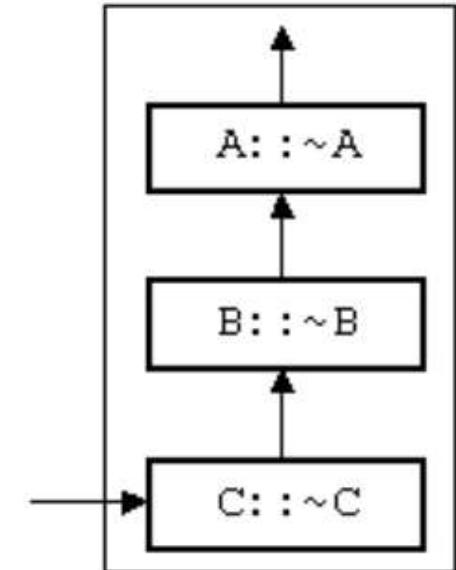
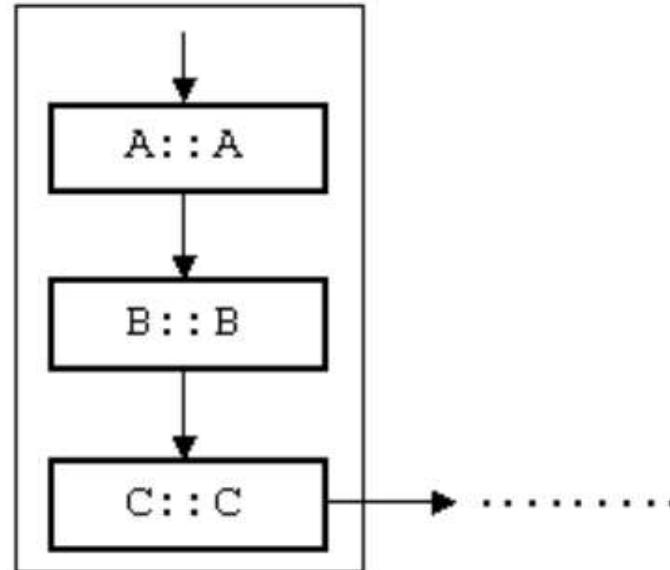
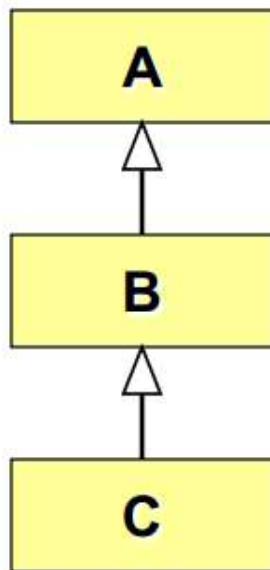
ĐỊNH NGHĨA LỚP DẪN XUẤT

- Để đảm bảo rằng một thể hiện của lớp cơ sở luôn được tạo trước, nếu ta bỏ qua lời gọi constructor lớp cơ sở tại danh sách khởi tạo của lớp dẫn xuất, trình biên dịch sẽ tự động chèn thêm lời gọi constructor mặc định của lớp cơ sở;
- Tuy ta cần gọi constructor của lớp cơ sở một cách tường minh, tại destructor của lớp dẫn xuất, lời gọi tương tự cho destructor của lớp cơ sở là không cần thiết;
 - Việc này được thực hiện tự động.



HÀM DỰNG VÀ HÀM HỦY

- Trong thừa kế, khi khởi tạo đối tượng:
 - Hàm xây dựng của lớp cha sẽ được gọi trước;
 - Sau đó mới là hàm xây dựng của lớp con.
- Trong thừa kế, khi hủy bỏ đối tượng:
 - Hàm hủy của lớp con sẽ được gọi trước;
 - Sau đó mới là hàm hủy của lớp cha.





HÀM DỰNG VÀ HÀM HỦY

- Nếu hàm dựng của lớp cơ sở yêu cầu phải cung cấp tham số để khởi tạo đối tượng → lớp con cũng phải có hàm dựng để cung cấp các tham số đó.

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

class HinhTron:Diem
{
    double r;
public:
    void Ve(int color) const;
};
```

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

class HinhTron:Diem
{
    double r;
public:
    HinhTron(double tx, double ty, double rr):
        Diem(tx,ty),r(rr){/*...*/}
    void Ve(int color) const;
    void TinhTien(double dx, double dy) const;
};
```

- HinhTron r;
- HinhTron t(200, 200, 50);



QUYỀN TRUY CẬP

- Ảnh hưởng của kế thừa đến phạm vi truy xuất các thành phần của lớp:
 - **Truy xuất theo chiều dọc:** Hàm thành phần của lớp con có quyền truy xuất các thành phần riêng tư của lớp cha hay không?
 - **Truy xuất theo chiều ngang:** Các đối tượng bên ngoài có quyền truy xuất đến các thành phần kế thừa ở lớp con thông qua các đối tượng của lớp con hay không?



QUYỀN TRUY CẬP

- Truy xuất theo chiều dọc:
 - **Thuộc tính truy xuất:** là đặc tính của một thành phần của lớp cho biết những nơi nào có quyền truy xuất thành phần đó
 - **public:** thành phần nào có thuộc tính này thì có thể được truy xuất từ bất cứ nơi nào;
 - **private:** thành phần nào có thuộc tính này thì nó là riêng tư của lớp đó (chỉ có các *hàm thành phần của lớp* hoặc là các *hàm bạn của lớp* mới được phép truy xuất);
 - **protected:** thành phần nào mang thuộc tính này thì chỉ có các lớp con mới có quyền truy cập



QUYỀN TRUY CẬP

```
class Nguoi
{
    char *HoTen;
    int NamSinh;
public:
    //...
};

class SinhVien : public Nguoi
{
    char *MaSo;
public:
    //...
    void Xuat() const;
    //Nguoi::HoTen va Nguoi::NamSinh
};

void SinhVien::Xuat() const
{
    cout << "Sinh vien, ma so: " << MaSo
        << ", ho ten: " << HoTen;
}
```

```
class Nguoi
{
    char *HoTen;
    int NamSinh;
public:
    //...
};

class SinhVien : public Nguoi
{
protected:    char *MaSo;
public:
    SinhVien(char *ht, char *ms, int ns):
        Nguoi(ht,ns)
    { MaSo = strdup(ms); }
    ~SinhVien() {delete [] MaSo; }
    void Xuat() const;
};

void SinhVien::Xuat() const
{
    cout << "Sinh vien, ma so: " << MaSo
        << ", ho ten: " << HoTen;
}
```



QUYỀN TRUY CẬP

- Truy xuất theo chiều ngang: phụ thuộc vào **thuộc tính kế thừa** của lớp con
 - **Kế thừa public:**
 - *protected* của cha thành *protected* của con;
 - *public* của cha thành *public* của con;
 - ❖ Nên sử dụng *khi và chỉ khi* có *quan hệ là một* từ lớp con đến lớp cha.
 - **Kế thừa private:**
 - *protected & public* cha thành *private* của con;
 - **Kế thừa protected:** thành phần nào mang thuộc tính này thì chỉ có các lớp con mới có quyền truy cập



QUYỀN TRUY CẬP

- Các quyền truy nhập có vai trò gì trong quan hệ thừa kế?
- Có hai kiểu quyền truy nhập cho các thành viên dữ liệu và phương thức:
 - **public** - thành viên/phương thức có thể được truy nhập từ mọi đối tượng C++ thuộc phạm vi;
 - **private** - thành viên/phương thức chỉ có thể được truy nhập từ bên trong chính lớp đó.
- Ta có thể sử dụng các từ khoá quyền truy nhập trong khai báo lớp để chỉ kiểu thừa kế

```
class Car : public MotorVehicle { /*...*/ };
```

```
class Car : public MotorVehicle
{
    public:
        Car(...);
        ~Car();
    private:
        int passengers;
};
```



QUYỀN TRUY CẬP

- Từ khoá được dùng để chỉ rõ “kiểu” thừa kế được sử dụng:
 - Nó quy định những ai có thể "nhìn thấy" quan hệ thừa kế đó;
- Thừa kế public (loại thông dụng nhất): mọi đối tượng C++ khi tương tác với một thể hiện của lớp con đều có thể coi nó như một thể hiện của lớp cha:
 - Mọi thành viên/phương thức public của lớp cha cũng là public trong lớp con.



QUYỀN TRUY CẬP

- Thùa kế **private**: chỉ có chính thể hiện đó biết nó được thừa kế từ lớp cha
 - Các đối tượng bên ngoài không thể tương tác với lớp con như với lớp cha, vì mọi thành viên được thừa kế đều trở thành private trong lớp con.
- Có thể dùng thừa kế **private** để tạo lớp con có mọi chức năng của lớp cha nhưng lại không cho bên ngoài biết về các chức năng đó.



QUYỀN TRUY CẬP

- Quay lại cây thừa kế với MotorVehicle là lớp cơ sở
 - Mọi thành viên dữ liệu đều được khai báo private, do đó chúng chỉ có thể được truy nhập từ các thể hiện của MotorVehicle;
 - Tất cả các lớp dẫn xuất không có quyền truy nhập các thành viên private của MotorVehicle.
- Vậy, đoạn mã sau sẽ có lỗi:
 - Lớp Truck không có quyền truy nhập thành viên private make của lớp cơ sở MotorVehicle.

```
class MotorVehicle
{
    //...
private:
    int vin;
    string make;
    string model;
};

void Truck::Load()
{
    if (this->make == "Ford") { /*...*/ }
}
```



QUYỀN TRUY CẬP

- Giả sử ta muốn các lớp con của MotorVehicle có thể truy nhập dữ liệu của nó
 - Thay từ khoá private bằng protected ta có khai báo:
 - Vậy, đoạn mã sau sẽ không có lỗi
 - Tuy nhiên truy nhập từ bên ngoài vẫn sẽ bị cấm:
 - Lớp Truck có quyền truy nhập thành viên protected Make của lớp cơ sở MotorVehicle.

```
class MotorVehicle
{
    //...
protected:
    int vin;
    string make;
    string model;
};

void Truck::Load()
{
    if (this->make == "Ford") { /*...*/ }
}
```



QUYỀN TRUY CẬP

Lớp cơ sở	Thừa kế public	Thừa kế private	Thừa kế protected
private	—	—	—
public	public	private	protected
protected	protected	private	protected

```
class A {  
    private:  
        int x;  
        void Fx (void);  
    public:  
        int y;  
        void Fy (void);  
    protected:  
        int z;  
        void Fz (void);  
};
```

```
class B : A { // Thừa kế dạng private  
    .....  
};  
class C : private A { // A là lớp cơ sở riêng của B  
    .....  
};  
class D : public A { // A là lớp cơ sở chung của C  
    .....  
};  
class E : protected A { // A: lớp cơ sở được bảo vệ  
    .....  
};
```



OVERIDING

- Lớp dẫn xuất có thể định nghĩa lại một hàm thành viên của lớp cơ sở mà nó được thừa kế.
- Khi đó nếu tên hàm được gọi đến trong lớp dẫn xuất thì trình biên dịch sẽ tự động gọi đến phiên bản hàm của lớp dẫn xuất.
- Muốn truy cập đến phiên bản hàm của lớp cơ sở từ lớp dẫn xuất thì sử dụng toán tử định phạm vi và tên lớp cơ sở trước tên hàm.



OVERIDING

- Ví dụ:

```
class DaGiac {  
    // ...  
    void Ve() const;  
    void ToMau() const;  
};  
class HCN :public DaGiac{  
    void ToMau() const;  
    ...  
};
```

```
class Ellipse{  
    //...  
public:  
    //...  
    void rotate(double rotangle){ //...}  
};  
class Circle:public Ellipse {  
public:  
    //...  
    void rotate(double rotangle)/* do nothing */  
};
```

```
class SinhVien : public Nguoi{  
    char *MaSo;  
public:  
    //...  
    void Xuat() const;  
};  
void SinhVien::Xuat() const {  
    cout << "Sinh vien, ma so: " << MaSo  
        << ", ho ten: " << HoTen;  
}
```



CÁC ĐỐI TƯỢNG ĐƯỢC KÉ THƯA TRONG C++

- Khi làm quen với thùng kê, ta thường nhắc đến khái niệm rằng một thể hiện của lớp dẫn xuất có thể được đối xử như thể nó là một thể hiện của lớp cơ sở;
 - Ví dụ, ta có thể coi một thể hiện của Car như là một thể hiện của MotorVehicle
- Như vậy chính xác nghĩa là gì? Trong C++, ta làm việc đó như thế nào?



CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THUẨA TRONG C++

- Ta đã nói rằng tư tưởng của thừa kế là khai báo các lớp con có mọi thuộc tính và hành vi của lớp cha.
- Nghĩa là, các tuyên bố sau là đúng:
 - Mọi đối tượng Car đều là MotorVehicle;
 - Mọi đối tượng Truck đều là MotorVehicle.
- Nhưng các tuyên bố ngược lại thì không đúng:
 - Mọi đối tượng MotorVehicle đều là Car;
 - Mọi đối tượng MotorVehicle đều là Truck.
- Ví dụ, trong một số trường hợp, ta có thể chỉ tạo các xe chạy bằng máy là xe ca. Nhưng trong cây thừa kế của ta, không có gì đòi hỏi rằng mọi xe chạy bằng máy đều là xe ca.



CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THUẨA TRONG C++

- Tư tưởng đó thể hiện rất rõ ràng trong cách ta định nghĩa một lớp con
 - Một lớp con trên cây thừa kế có mọi thuộc tính và hành vi của lớp cha;
 - Cộng thêm các thuộc tính và hành vi của riêng lớp con đó.
- Theo ngôn ngữ của C++: một thể hiện của một lớp con có thể truy nhập tới:
 - Mọi thuộc tính và hành vi (không phải private) của lớp cha;
 - Và các thành viên được định nghĩa riêng cho lớp con đó.



CÁC ĐỐI TƯỢNG ĐƯỢC KÉ THƯA TRONG C++

- Như vậy, một thể hiện của Car có quyền truy nhập các thuộc tính và hành vi sau:
 - Thành viên dữ liệu: vin, make, model, passengers;
 - Phương thức: drive().
- Ngược lại, không có lý gì một thể hiện của lớp cha lại có quyền truy nhập tới thuộc tính/hành vi chỉ được định nghĩa trong lớp con
 - MotorVehicle không thể truy nhập passengers của Car
- Tương tự, các lớp anh-chị-em không thể truy nhập các thuộc tính/hành vi của nhau:
 - Một đối tượng Car không thể có phương thức Load() và UnLoad(), cũng như một đối tượng Truck không thể có passengers.
- C++ đảm bảo các yêu cầu đó như thế nào?



CÁC ĐỐI TƯỢNG ĐƯỢC KÉ THƯA TRONG C++

- Trong cây thừa kế MotorVehicle, giả sử mọi thành viên dữ liệu đều được khai báo protected, và ta sử dụng kiểu thừa kế public;
- Lớp cơ sở MotorVehicle:

```
class MotorVehicle
{
public:
    MotorVehicle(int vin, string make, string model);
    ~MotorVehicle();
    void Drive(int speed, int distance);
protected:
    int vin;
    string make;
    string model;
};
```



CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THUẨA TRONG C++

- Các lớp dẫn xuất Car và Truck:

```
class Car: public MotorVehicle
{
    public:
        Car(int vin, string make, string model, int passengers);
        ~Car();
    protected: int passengers;
};

class Truck: public MotorVehicle
{
    public:
        Truck(int vin, string make, string model, int maxPayload);
        ~Truck();
        void Load();
        void Unload();
    protected: int maxPayload;
};
```



CÁC ĐỐI TƯỢNG ĐƯỢC KÉ THƯA TRONG C++

- Ta có thể khai báo các thể hiện của các lớp đó và sử dụng chúng như thế nào?
 - Ví dụ, khai báo các con trỏ tới 3 lớp:

```
MotorVehicle* mvPointer;  
Car* cPointer;  
Truck* tPointer;
```

- Sử dụng các con trỏ để khai báo các đối tượng thuộc các lớp tương ứng

```
...  
mvPointer = new MotorVehicle(10, "Honda", "S2000");  
cPointer = new Car(10, "Honda", "S2000", 2);  
tPointer = new Truck(10, "Toyota", "Tacoma", 5000);  
...
```



CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

- Trong cả ba trường hợp, ta có thể truy nhập các phương thức của lớp cha, do ta đã sử dụng kiểu thừa kế public

```
mvPointer->Drive(); // Method defined by this class  
cPointer->Drive(); // Method defined by base class  
tPointer->Drive(); // Method defined by base class
```

- Tuy nhiên, các phương thức định nghĩa tại một lớp dẫn xuất chỉ có thể được truy nhập bởi lớp đó
 - Xét phương thức Load() của lớp Truck:

```
mvPointer->Load(); // Error  
cPointer->Load(); // Error  
tPointer->Load(); // Method defined by this class
```



CON TRỎ VÀ KẾ THỪA

- Con trỏ trả đến đối tượng thuộc lớp cơ sở thì có thể trỏ đến các đối tượng thuộc lớp con;
- Ngược lại, con trỏ trả đến đối tượng thuộc lớp con thì không thể trỏ đến các đối tượng thuộc lớp cơ sở;
 - ❖ Có thể sử dụng ép kiểu trong trường hợp này → không nên dùng.



CON TRỎ VÀ KẾ THỪA

```
void main()
{
    Nguoi n("Nguyen Van Nhan", 1970);
    SinhVien s("Vo Vien Sinh", "200002541", 1984);
    Nguoi *pn;
    SinhVien *ps;
    pn = &n;
    ps = &s;
    pn = &s;
    ps = &n; // Sai
    ps = pn; // Sai
    ps = (SinhVien *)&n; // Sai logic
    ps = (SinhVien *)pn;
}
```



UPCAST

- Các thể hiện của lớp con thừa kế public có thể được đối xử như thể nó là thể hiện của lớp cha.
 - Từ một thể hiện của lớp con, ta có quyền truy nhập các thành viên và phương thức public mà ta có thể truy nhập trên một thể hiện của lớp cha.
- Do đó, C++ cho phép dùng con trỏ được khai báo thuộc loại con trỏ tới lớp cơ sở để chỉ tới thể hiện của lớp dẫn xuất:
 - Ta có thể thực hiện các lệnh sau:

```
MotorVehicle* mvPointer2;  
mvPointer2 = mvPointer; // Point to another MotorVehicle  
mvPointer2 = cPointer; // Point to a Car  
mvPointer2 = tPointer; // Point to a Truck
```



UPCAST

- Điều đáng lưu ý là ta thực hiện tất cả các lệnh gán đó mà **không cần** đổi kiểu tường minh:
 - Do mọi lớp con của MotorVehicle đều chắc chắn có mọi thành viên và phương thức có trong một MotorVehicle, việc tương tác với thể hiện của các lớp này như thể chúng là MotorVehicle không có chút rủi ro nào;
 - Ví dụ, lệnh sau đây là hợp lệ, bất kể mvPointer2 đang trỏ tới một MotorVehicle, một Car, hay một Truck:

mvPointer2->Drive();



- **Upcast** là quá trình tương tác với thể hiện của lớp dẫn xuất như thể nó là thể hiện của lớp cơ sở.
- Cụ thể, đây là việc đổi một con trỏ (hoặc tham chiếu) tới lớp dẫn xuất thành một con trỏ (hoặc tham chiếu) tới lớp cơ sở:
 - Ta đã thấy ví dụ về upcast đối với con trỏ:
MotorVehicle* mvPointer2 = cPointer;
 - Ví dụ về upcast đối với tham chiếu:

```
// Refer to the instance pointed to by cPointer
MotorVehicle& mvReference = *cPointer;
// Refer to an automatically-allocated instance c
Car c(10, "Honda", "S2000", 2);
MotorVehicle& mvReference2 = c;
```



UPCAST

- Upcast thường gặp tại các định nghĩa hàm, khi một con trỏ/tham chiếu đến lớp cơ sở được yêu cầu, nhưng con trỏ/tham chiếu đến lớp dẫn xuất cũng được chấp nhận
 - Xét hàm sau:
void sellMyVehicle(MotorVehicle& myVehicle) {/*...*/}
 - Có thể gọi sellMyVehicle một cách hợp lệ với tham số là một tham chiếu tới một MotorVehicle, một Car, hoặc một Truck.



UPCAST

- Nếu ta dùng một con trỏ tới lớp cơ sở để trỏ tới một thể hiện của lớp dẫn xuất, trình biên dịch sẽ chỉ cho ta coi đối tượng như thể nó thuộc lớp cơ sở:
 - Như vậy, ta không thể làm như sau:
mvPointer2->Load(); // Error
- Đó là vì trình biên dịch không thể đảm bảo rằng con trỏ thực ra đang trỏ tới một thể hiện của Truck.



- Chú ý rằng khi gắn một con trỏ/tham chiếu lớp cơ sở với một thể hiện của lớp dẫn xuất, ta không hề thay đổi bản chất của đối tượng được trỏ tới:
 - Ví dụ:

```
MotorVehicle* mvPointer2 = tPointer;
mvPointer2->Load(); //error
```
- Không làm một thể hiện của **Truck** suy giảm thành một **MotorVehicle**, nó chỉ cho ta một cách nhìn khác đối với đối tượng **Truck** và tương tác với đối tượng đó.
 - Do vậy, ta vẫn có thể truy nhập tới các thành viên và phương thức của lớp dẫn xuất ngay cả sau khi gán con trỏ lớp cơ sở tới nó:

```
tPointer = new Truck(...);
mvPointer2 = tPointer;    //Point to a Truck
tPointer->Load();        //We can still do this
mvPointer2->Load();      //Even though we can't do this (error)
```



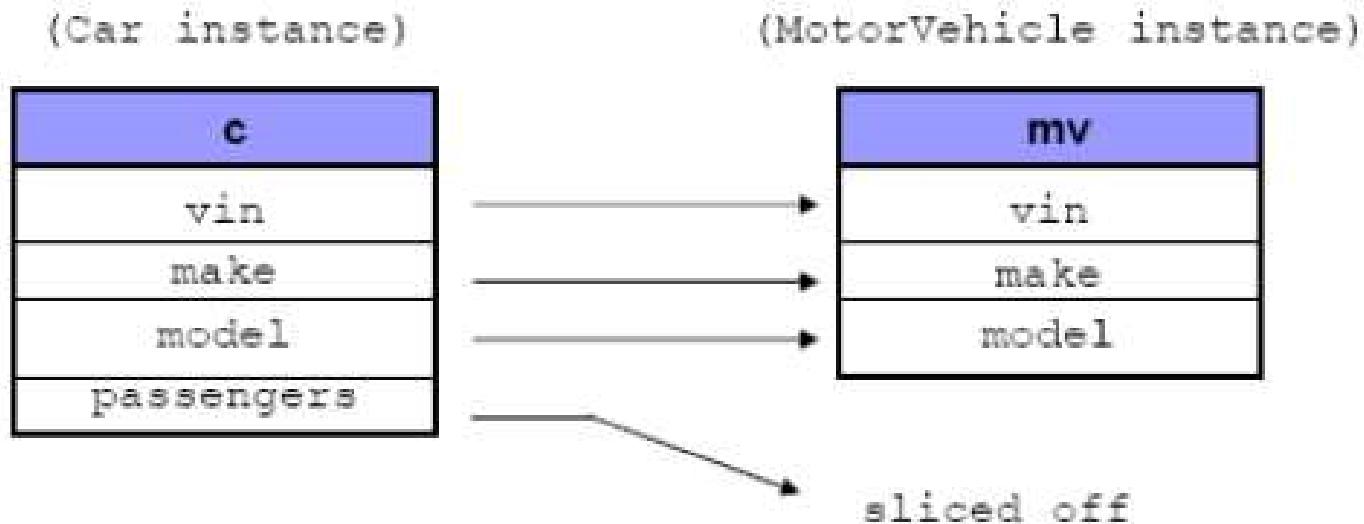
SLICING

- Đôi khi ta muốn đổi hẳn kiểu
 - Ví dụ, ta muốn tạo một thể hiện của MotorVehicle dựa trên một thể hiện của Car (sử dụng copy constructor cho MotorVehicle)
- Slicing là quá trình chuyển một thể hiện của lớp dẫn xuất thành một thể hiện của lớp cơ sở:
 - Hợp lệ vì một thể hiện của lớp dẫn xuất có tất cả các thành viên và phương thức của lớp cơ sở của nó.
- Quy trình này gọi là “slicing” vì thực chất ta cắt bớt (slice off) những thành viên dữ liệu và phương thức được định nghĩa trong lớp dẫn xuất.



SLICING

- Ví dụ:
 - Car c(10, “Honda”, “S2000”, 2);
 - MotorVehicle mv(c); //mv=c
 - Ở đây, một thể hiện của MotorVehicle được tạo bởi copy constructor chỉ giữ lại những thành viên của Car mà có trong MotorVehicle





SLICING

- Thực ra, quy trình này cũng giống hệt như khi ta ngầm đổi giữa các kiểu dữ liệu có sẵn và bị mất bớt dữ liệu (chẳng hạn khi đổi một số chấm động sang số nguyên)
- Slicing còn xảy ra khi ta dùng phép gán
Car c(10, “Honda”, “S2000”, 2);
MotorVehicle mv;
mv = c;



DOWNCAST

- Upcast là đổi con trỏ/tham chiếu tới lớp dẫn xuất thành con trỏ/tham chiếu tới lớp cơ sở.
- **Downcast** là quy trình ngược lại: đổi kiểu con trỏ/tham chiếu tới lớp cơ sở thành con trỏ/tham chiếu tới lớp dẫn xuất.
- Downcast là quy trình rắc rối hơn và có nhiều điểm không an toàn.



- Trước hết, downcast không phải là một quy trình tự động – nó luôn đòi hỏi đổi kiểu tường minh (explicit type cast)
- Điều đó là hợp lý:
 - Nhớ lại rằng: không phải “mọi xe chạy bằng máy đều là xe tải”
 - Do đó, rắc rối sẽ nảy sinh nếu trình biên dịch cho ta đổi một con trỏ bất kỳ tới **MotorVehicle** thành một con trỏ tới **Truck**, trong khi thực ra con trỏ đó đang trỏ tới một đối tượng **Car**.
- Ví dụ, đoạn mã sau sẽ gây lỗi biên dịch:

```
MotorVehicle* mvPointer3;  
...  
Car* cPointer2 = mvPointer3; // Error  
Truck* tPointer2 = mvPointer3; // Error  
MotorCycle mcPointer2 = mvPointer3; // Error
```



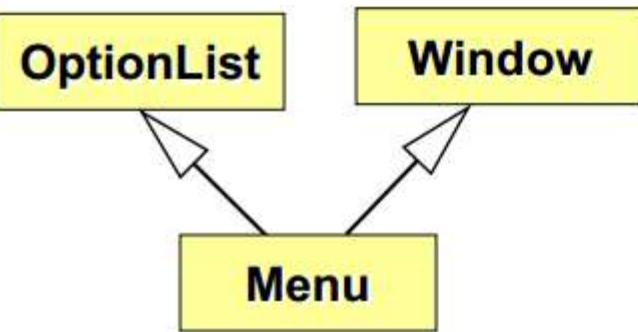
DOWNCAST

```
Car* cPointer = new Car(10,"Honda","S2000",2);
MotorVehicle *mv=cPointer;//Upcast
Truck* tPointer = static_cast<Truck*>(mv);
tPointer->Load();
```

- Đoạn mã trên hoàn toàn hợp lệ và sẽ được trình biên dịch chấp nhận;
- Tuy nhiên, nếu chạy đoạn trình trên, chương trình có thể bị đỗ vỡ (thường là khi lần đầu truy nhập đến thành viên/phương thức được định nghĩa của lớp dẫn xuất mà ta đổi mới).

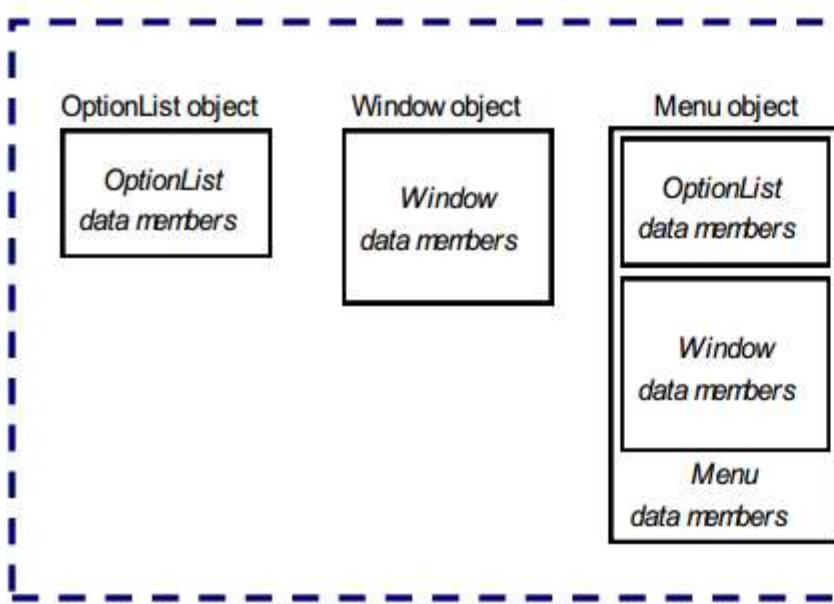


ĐA THÙA KẾ



```
class OptionList {
public:
    OptionList (int n);
    ~OptionList ();
    //...
};
```

```
class Window {
public:
    Window (Rect &);
    ~Window (void);
    //...
};
```



```
class Menu
    : public OptionList, public Window {
public:
    Menu (int n, Rect &bounds);
    ~Menu (void);
    //...
};

Menu::Menu (int n, Rect &bounds) :
    OptionList(n), Window(bounds)
{ /* ... */ }
```



SỰ MƠ HỒ TRONG ĐA THÙA KẾ

```
class OptionList {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Window {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Menu : public OptionList,  
           public Window  
{ ..... };
```

Hàm cùng tên

Chỉ rõ hàm
của lớp nào

Gọi
hàm
của lớp
nào?

```
void main() {  
    Menu m1(...);  
    m1.Highlight(10);  
    ....  
}
```

xử lý

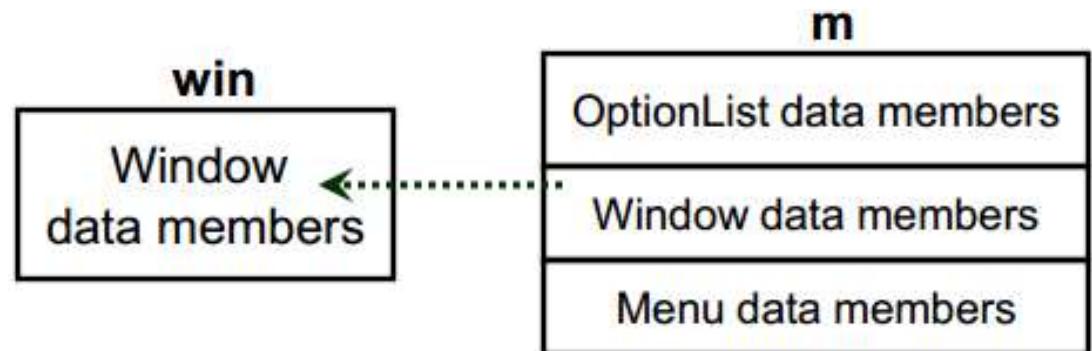
```
void main() {  
    Menu m1(...);  
    m1.OptionList::Highlight(10);  
    m1.Window::Highlight(20);  
    ....  
}
```



CHUYỂN KIỂU

- Có sẵn 1 phép chuyển kiểu không tường minh:
 - Đối tượng lớp cha = Đối tượng lớp con;
 - Áp dụng cho cả đối tượng, tham chiếu và con trỏ.

```
Menu m(n, bounds);  
Window win = m;  
Window &wRef = m;  
Window *wPtr = &menu;
```



- Không được thực hiện phép gán ngược:
 - Đối tượng lớp con = Đối tượng lớp cha; // SAI

Nếu muốn thực hiện
phải tự định nghĩa
phép ép kiểu

```
class Menu : public OptionList, public Window {  
public:  
    //...  
    Menu (Window&);  
};
```

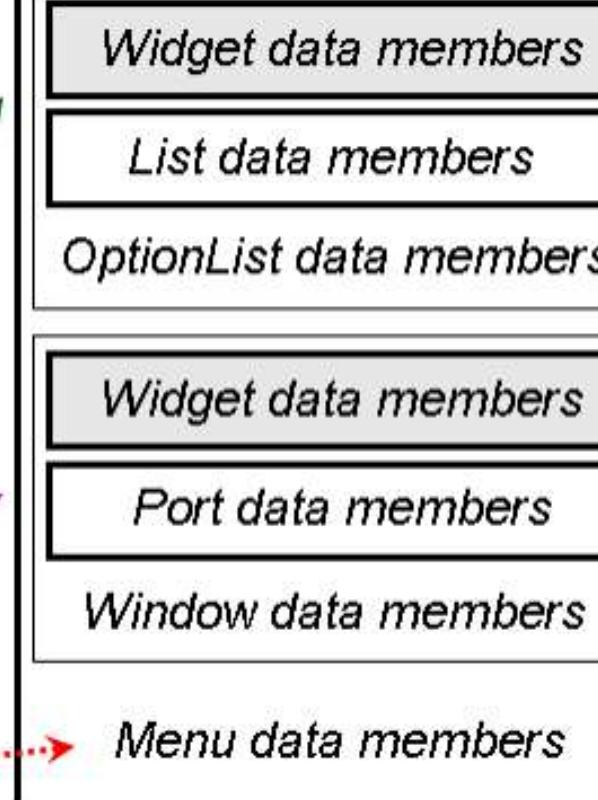


LỚP CƠ SỞ ẢO

```
class OptionList
    : public Widget, List
{ /*...*/ };

class Window
    : public Widget, Port
{ /*...*/ };

class Menu
    : public OptionList,
      public Window
{ /*...*/ };
```





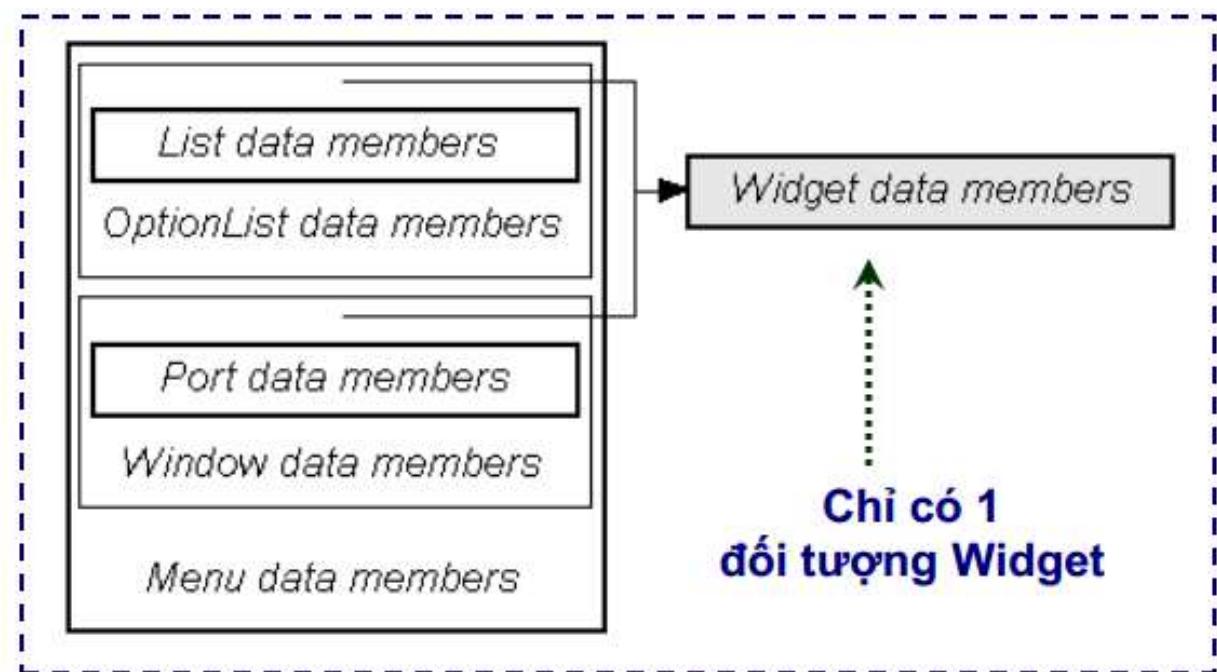
LỚP CƠ SỞ ẢO

- Cách xử lý: dùng lớp cơ sở ảo

```
class OptionList
    : virtual public Widget,
      public List
{
    /*...*/};

class Window
    : virtual public Widget,
      public Port
{
    /*...*/};

class Menu
    : public OptionList,
      public Window
{
    /*...*/};
```



```
Menu::Menu (int n, Rect &bounds) :
    Widget(bounds), OptionList(n), Window(bounds)
{ //... }
```



CÁC TOÁN TỬ ĐƯỢC TÁI ĐỊNH NGHĨA

- Tương tự như tái định nghĩa hàm thành viên:
 - Che giấu đi toán tử của lớp cơ sở;
 - Hàm dựng sao chép: **Y::Y (const Y&)**
 - Phép gán: **Y& Y::operator = (const Y&)**
- Nếu không định nghĩa, sẽ tự động có hàm dựng sao chép và phép gán do ngôn ngữ tạo ra:
 - SAI khi có con trỏ thành viên.
- Cảnh thận với toán tử **new** và **delete**.



CHƯƠNG 6

ĐA HÌNH





ĐA HÌNH

- “Polymorphism” có nghĩa “nhiều hình thức”, hay “nhiều dạng sống”;
- Một vật có tính đa hình (polymorphic) là vật có thể xuất hiện dưới nhiều dạng;
- Ta đã gặp một dạng của đa hình từ trước khi làm quen với khái niệm hướng đối tượng: Đa hình hàm
 - Đa hình hàm – function polymorphism, hay còn gọi là hàm chồng – function overloading, trong đó, một hàm có nhiều dạng

```
myFunction() {...}  
myFunction(int x) {...}  
myFunction(int x, int y) {...}
```
- Đó là khi một tên hàm có thể được dùng để chỉ các định nghĩa hàm khác nhau dựa trên danh sách tham số.



ĐA HÌNH & HƯỚNG ĐỐI TƯỢNG

- Trong phạm vi mô hình hướng đối tượng, thuật ngữ đa hình có ý nghĩa cụ thể và mạnh hơn;
- Ta đã thấy một chút về đa hình trong phần về thừa kế:
 - Lấy một con trỏ tới lớp cơ sở và dùng nó để truy nhập các đối tượng của lớp dân xuất.
- Ta liên lạc với các đối tượng bằng các thông điệp (các lời gọi hàm) để yêu cầu đối tượng thực hiện một hành vi nào đó;
- Việc hiểu các thông điệp đó như thế nào chính là nền tảng cho tính chất đa hình của hướng đối tượng.



ĐA HÌNH & HƯỚNG ĐỐI TƯỢNG

- Định nghĩa: Đa hình là hiện tượng các đối tượng thuộc các lớp *khác nhau* có khả năng hiểu *cùng một* thông điệp theo các cách *khác nhau*.
- Ta có thể có định nghĩa tương tự cho đa hình hàm: một thông điệp (lời gọi hàm) được hiểu theo các cách khác nhau tùy theo danh sách tham số của thông điệp.
- Ví dụ: nhận được cùng một thông điệp “nhảy”, một con kangaroo và một con cóc nhảy theo hai kiểu khác nhau: chúng cùng có hành vi “nhảy” nhưng các hành vi này có nội dung khác nhau.



OVERLOADING & OVERRIDING

- Function overloading - Hàm chòng: dùng một tên hàm cho nhiều định nghĩa hàm, khác nhau ở danh sách tham số
- Method overloading – Phương thức chòng: tương tự
 - void jump(int howHigh);
 - void jump(int howHigh, int howFar);
 - hai phương thức **jump** trùng tên nhưng có danh sách tham số khác nhau.
- Tuy nhiên, đây không phải đa hình hướng đối tượng mà ta đã định nghĩa, vì đây thực sự là hai thông điệp **jump khác nhau**.



OVERLOADING & OVERRIDING

- Đa hình được cài đặt bởi một khái niệm tương tự nhưng hơi khác: method overriding:
 - “override” có nghĩa “vượt quyền”.
- Method overriding: nếu một phương thức của lớp cơ sở được định nghĩa lại tại lớp dẫn xuất thì định nghĩa tại lớp cơ sở có thể bị “che” bởi định nghĩa tại lớp dẫn xuất;
- Với method overriding, toàn bộ thông điệp (cả tên và tham số) là *hoàn toàn giống nhau* – điểm khác nhau là lớp đối tượng được nhận thông điệp.

Hai thông điệp
giống nhau,
nhưng đích là hai
lớp khác nhau

Kangaroo k;
Frog f;

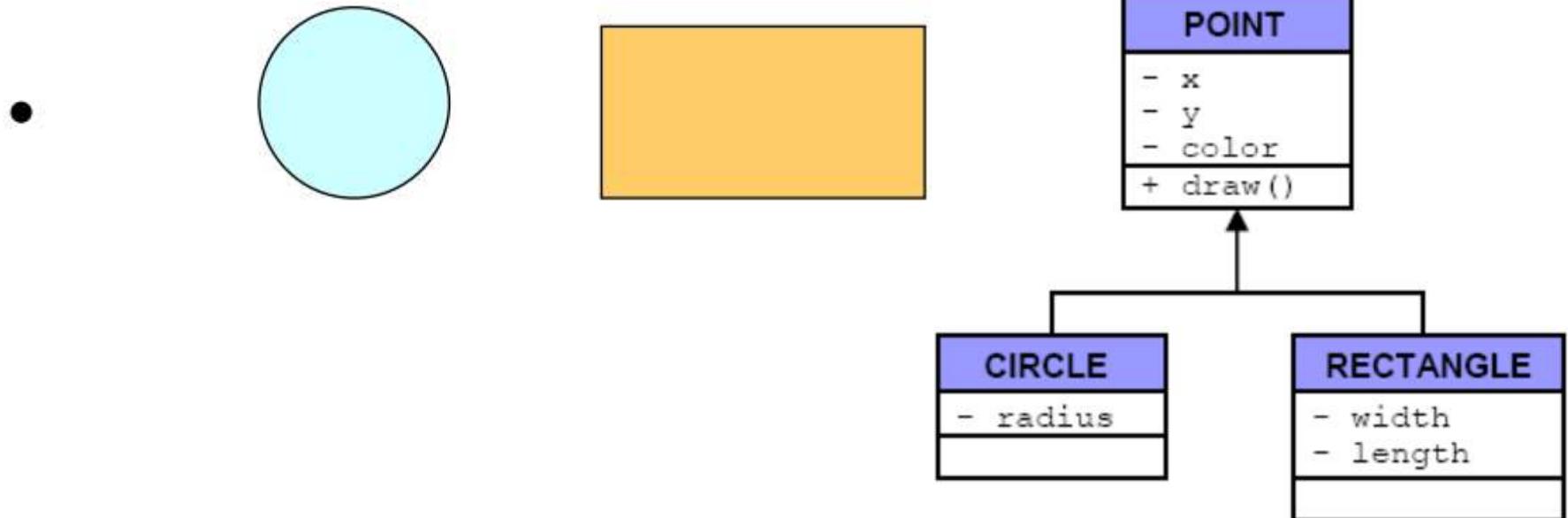
k.jump(10); // the kangaroo jumps 10m high
f.jump(10); // the frog jumps 10m far

Chú ý, Kangaroo và Frog đều là các lớp
dẫn xuất từ lớp cơ sở gián tiếp Animal



METHOD OVERLOADING

- Xét hành vi “draw” của các lớp trong cây phả hệ
 - Thông điệp “draw” gửi cho một thẻ hiện của mỗi lớp trên sẽ yêu cầu thẻ hiện đó tự vẽ chính nó;
 - Một thẻ hiện của Point phải vẽ một điểm, một thẻ hiện của Circle phải vẽ một đường tròn, và một thẻ hiện của Rectangle phải vẽ một hình chữ nhật.





METHOD OVERLOADING

- Với đặc điểm đa hình của method overriding, ta sẽ có được điều trên.
 - Định nghĩa lại hành vi draw tại các lớp dẫn xuất:

```
class Point {  
public:  
    Point(int x,int y,string color);  
    ~Point();  
    void draw();  
protected:  
    int x;  
    int y;  
    string color;  
};
```

```
...  
void Point::draw() {  
    // Draw a point  
}
```

```
class Circle : public Point {  
public:  
    Circle (int x, int y,  
            string color,int radius);  
    ~Circle();  
    void draw();  
private:  
    int radius;
```

1. khai báo lại tại lớp dẫn xuất

```
...  
void Circle::draw() {  
    // Draw a circle  
}
```

2. khai báo lại tại lớp dẫn xuất



METHOD OVERLOADING

- Kết quả: khi tạo các thể hiện của các lớp khác nhau và gửi thông điệp “draw”, các phương thức thích hợp sẽ được gọi.

```
Point p(0,0,"white");
Circle c(100,100,"blue",50);
p.draw(); // Draws a white point at (0,0)
c.draw(); // Draws a blue circle of radius 50 at (100,100)
```



METHOD OVERLOADING

- Lưu ý :

- Để override một phương thức của một lớp cơ sở, phương thức tại lớp dẫn xuất phải có cùng tên, cùng danh sách tham số, cùng kiểu giá trị trả về, cùng là const hoặc cùng không là const;
- Nếu lớp cơ sở có nhiều phiên bản overload của cùng một phương thức, việc override một trong các phương thức đó sẽ che tất cả các phương thức còn lại.

```
class A() {  
    void foo();  
    void foo(int x);  
...}
```

B override **foo()**, không override **foo(x)**
⇒ tại B, **foo(x)** bị che, nếu gọi **foo(x)** từ
một đối tượng B sẽ gây lỗi biên dịch

```
class B:public A(){  
    void foo(); ←  
    //no foo(int x);  
...}
```



METHOD OVERLOADING

- Khi tạo thể hiện của các lớp khác nhau và gọi cùng một hành vi, phương thức thích hợp sẽ được gọi:

```
Point p(0,0,"white");
Circle c(100,100,"blue",50);
p.draw(); //Draws a white point at (0,0)
c.draw(); //Draws a blue circle of radius 50 at (100,100)
```

- Ta cũng có thể tương tác với các thể hiện lớp dẫn xuất như thể chúng là thể hiện của lớp cơ sở
 - Circle *pc = new Circle(100,100,"blue",50);
 - Point* pp = pc;
- Nhưng nếu có đa hình, lời gọi sau sẽ làm gì?
 - pp->draw(); // Draw what???



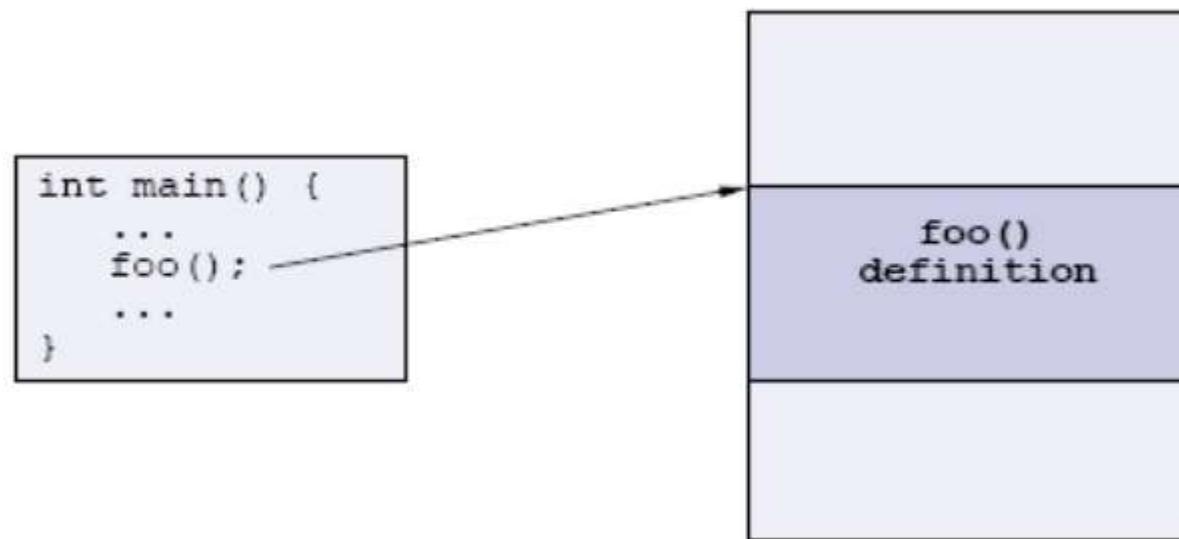
FUNCTION CALL BINDING

- Function call binding – Liên kết lời gọi hàm;
- C++ làm thế nào để tìm đúng hàm cần chạy mỗi khi ta có một lời gọi hàm hoặc gọi phương thức?
- Function call binding là quy trình xác định khối mã hàm cần chạy khi một lời gọi hàm được thực hiện;
- Xem xét:
 - function call binding
 - C, C++
 - method call binding
 - Static binding;
 - Dynamic binding.



FUNCTION CALL BINDING

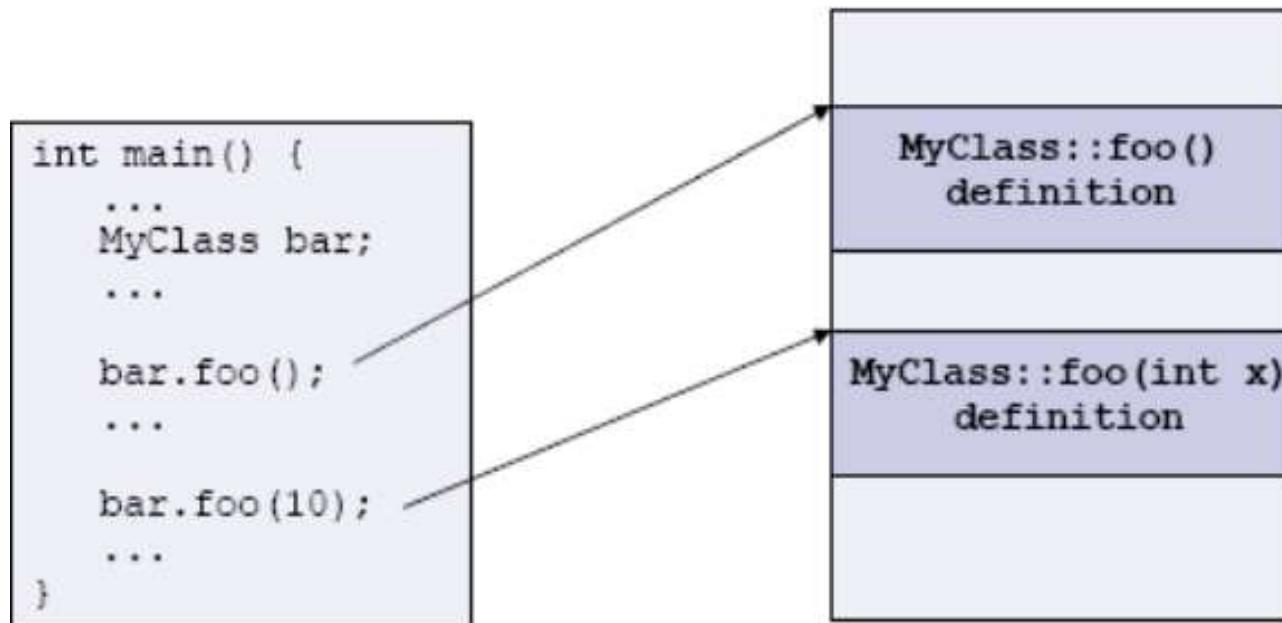
- Trong C, việc này rất dễ dàng vì mỗi tên hàm chỉ có thể dùng cho đúng một định nghĩa hàm:
 - Do đó, khi có một lời gọi tới hàm `foo()`, chỉ có thể có đúng một định nghĩa khớp với tên hàm đó.





METHOD CALL BINDING

- Method call binding – liên kết lời gọi phương thức;
- Đối với các lớp độc lập (không thuộc cây thừa kế nào), quy trình này gần như không khác với function call binding
 - So sánh tên phương thức, danh sách tham số để tìm định nghĩa tương ứng;
 - Một trong số các tham số là tham số ẩn: con trỏ **this**.





STATIC BINDING

- Static function call binding (hoặc static binding – liên kết tĩnh) là quy trình liên kết một lời gọi hàm với một định nghĩa hàm tại thời điểm biên dịch;
 - do đó còn gọi là “compile-time binding” – liên kết khi biên dịch, hoặc “early binding” – liên kết sớm.
- Hiểu rõ liên kết tĩnh – khi nào/tại sao nó xảy ra, khi nào/tại sao nó gây rắc rối – là chìa khoá để:
 - Tìm được cách làm cho bò điên lúc nào cũng điên;
 - Và quan trọng hơn, làm thế nào để có đa hình.



STATIC BINDING

- Với liên kết tĩnh, quyết định “định nghĩa hàm nào được chạy” được đưa ra tại thời điểm biên dịch - rất lâu trước khi chương trình chạy;
- Thích hợp cho các lời gọi hàm thông thường:
 - Mỗi lời gọi hàm chỉ xác định duy nhất một định nghĩa hàm, kể cả trường hợp hàm chồng.
- Phù hợp với các lớp độc lập không thuộc cây thừa kế nào:
 - Mỗi lời gọi phương thức từ một đối tượng của lớp hay từ con trỏ đến đối tượng đều xác định duy nhất một phương thức.

```
int main() {  
    MyClass* bar;  
    ...  
    bar->foo();  
    ...  
}
```

```
class MyClass {  
public:  
    void foo();  
    ...  
};
```

```
int main() {  
    MyClass bar;  
    ...  
    bar.foo();  
    ...  
}
```



STATIC BINDING

- Trường hợp có cây thừa kế:

```
Point P(10,20,"brow");  
Circle C(5,10,2.0, "yellow"  
P.draw(); //in điểm  
C.draw(); //in hình tròn
```

Kiểu tĩnh :
Circle

```
Circle& rC = C;  
rC.draw(); //hình tròn
```

```
Circle* pC = &C;  
pC->draw(); //hình tròn
```

Kiểu tĩnh :
Point

```
Point& rP = C;  
rP.draw(); // in điểm
```

```
Point* pP = &C;  
pP->draw(); // in điểm
```



STATIC BINDING

- Ta muốn:
rP.draw() và **pP->draw();**
 - Sẽ gọi **Point::draw()** hay **Circle::draw()**
 - Tuỳ theo **rP** và **pP** đang trỏ tới đối tượng **Point** hay **Circle**.
- Thực tế xảy ra: với liên kết tĩnh, khi trình biên dịch sinh lời gọi hàm, nó thấy kiểu tĩnh của **rP** là **Point&**, nên gọi **Point::draw()** và kiểu tĩnh của **pP** là **Point*** nên gọi **Point::draw()**.



STATIC BINDING

- Liên kết tĩnh – Static binding: liên kết một tên hàm với phương thức thích hợp được thực hiện bởi việc phân tích tĩnh mã chương trình tại thời gian dịch dựa vào *kiểu tĩnh (được khai báo)* của đối tượng được dùng trong lời gọi hàm:
 - Liên kết tĩnh không quan tâm đến chuyện con trả (hoặc tham chiếu) có thể trả tới một đối tượng của một lớp dẫn xuất.
- Với liên kết tĩnh, địa chỉ đoạn mã cần chạy cho một lời gọi hàm cụ thể là không đổi trong suốt thời gian chương trình chạy.



STATIC POLYMORPHISM

- **Static polymorphism** – **đa hình tĩnh** là kiểu đa hình được cài đặt bởi liên kết tĩnh;
- Đối với đa hình tĩnh, trình biên dịch xác định trước định nghĩa hàm/phương thức nào sẽ được thực thi cho một lời gọi hàm/phương thức nào.



STATIC POLYMORPHISM

- Đa hình tĩnh thích hợp cho các phương thức:
 - Được định nghĩa tại một lớp, và được gọi từ một thể hiện của chính lớp đó (trực tiếp hoặc gián tiếp qua con trỏ);
 - Được định nghĩa tại một lớp cơ sở và được thừa kế public nhưng không bị override tại lớp dẫn xuất, và được gọi từ một thể hiện của lớp dẫn xuất đó:
 - Trực tiếp hoặc gián tiếp qua con trỏ tới lớp dẫn xuất;
 - Qua một con trỏ tới lớp cơ sở.
 - Được định nghĩa tại một lớp cơ sở và được thừa kế public và bị override tại lớp dẫn xuất, và được gọi từ một thể hiện của lớp dẫn xuất đó (trực tiếp hoặc gián tiếp qua con trỏ tới lớp dẫn xuất).



STATIC POLYMORPHISM

- Ta gặp rắc rối với đa hình tĩnh (như trong trường hợp Circle), khi ta muốn gọi định nghĩa đã được override tại lớp dẫn xuất của một phương thức qua con trỏ tới lớp cơ sở:

Với trình biên dịch,
pP là con trỏ tới **Point**

```
Circle C(5,10,2.0, "yellow");  
...  
Point* pP = &C;  
pP->draw(); // in điểm
```

- Ta muốn trình biên dịch hoãn quyết định gọi phương thức nào cho lời gọi hàm trên cho đến khi chương trình thực sự chạy;
- Cần một cơ chế cho phép xác định kiểu động tại thời gian chạy (tại thời gian chạy, chương trình có thể xác định con trỏ đang thực sự trỏ đến cái gì);
- Vậy, ta cần đa hình động – dynamic polymorphism.



DYNAMIC BINDING

- Dynamic function call binding (dynamic binding – liên kết động) là quy trình liên kết một lời gọi hàm với một định nghĩa hàm tại thời gian chạy:
 - Còn gọi là “run-time” binding hoặc “late binding”.
- Với liên kết động, quyết định chạy định nghĩa hàm nào được đưa ra tại thời gian chạy, khi ta biết chắc con trỏ đang trỏ đến đối tượng thuộc lớp nào:

Khi chạy đến đây,
chương trình nhận ra
pP đang trỏ tới
Circle, nên gọi
Cirlce::draw()

```
Cirlce C(5,10,2.0, "yellow");  
...  
Point* pP = &C;  
pP->draw(); // in hình tròn
```

- Đa hình động (dynamic polymorphism) là loại đa hình được cài đặt bởi liên kết động.



VIRTUAL FUNCTION

- **Hàm/phương thức ảo – virtual function/method** là cơ chế của C++ cho phép cài đặt đa hình động;
- Nếu khai báo một hàm thành viên (phương thức) là **virtual**, trình biên dịch sẽ đẩy lùi việc liên kết các lời gọi phương thức đó với định nghĩa hàm cho đến khi chương trình chạy;
 - nghĩa là, ta bảo trình biên dịch sử dụng liên kết động thay cho liên kết tĩnh đối với phương thức đó.
- Để một phương thức được liên kết tại thời gian chạy, nó phải khai báo là phương thức ảo (từ khóa **virtual**) tại *lớp cơ sở*.



VIRTUAL FUNCTION

- Ví dụ:

```
class Point {  
public:  
    Point(int x,int y,string color);  
    ~Point();  
    virtual void draw();  
protected:  
    int x;  
    int y;  
    string color;  
};
```

Khai báo hàm ảo,
yêu cầu trình biên dịch
dùng liên kết động

draw đã được khai báo là hàm ảo,
Khi chạy, pP trả tới Circle, nên
Circle::draw() được gọi

```
int main() {  
    Cirlce C(5,10,2.0, "yellow");  
    ...  
    Point* pP = &C;  
    pP->draw(); // in hình tròn  
    ...  
}
```



VIRTUAL FUNCTION

- Hàm ảo là phương thức được khai báo với từ khoá virtual trong định nghĩa lớp (nhưng không cần tại định nghĩa hàm, nếu định nghĩa hàm nằm ngoài định nghĩa lớp);
- Một khi một phương thức được khai báo là hàm ảo tại lớp cơ sở, nó sẽ *tự động* là hàm ảo tại mọi lớp dẫn xuất trực tiếp hoặc gián tiếp;
- Tuy *không cần* tiếp tục dùng từ khoá virtual trong các lớp dẫn xuất, nhưng vẫn **nên dùng** để tăng tính dễ đọc của các file header.
 - Nhắc ta và những người dùng lớp dẫn xuất của ta rằng phương thức đó sử dụng liên kết động.



VIRTUAL FUNCTION

- Ví dụ:

```
class Point
public:
...
virtual void draw();
...
};

...
void Point::draw()
{...}

class Circle: public Point {
public:
...
virtual void draw();
...
};

...
void Circle::draw()
{...}
```

Các hàm `draw()`
của `Point` và các lớp dẫn
xuất đều là hàm ảo

Không cần
nhưng nên có



VIRTUAL FUNCTION

- Đa hình động dễ cài và cần thiết. Vậy tại sao lại cần đa hình tĩnh?
- Trong Java, mọi phương thức đều mặc định là phương thức ảo, tại sao C++ không như vậy?
- Có hai lý do:
 - Tính hiệu quả
 - C++ cần chạy nhanh, trong khi liên kết động tốn thêm phần xử lý phụ, do vậy làm giảm tốc độ;
 - Ngay cả những hàm ảo không được override cũng cần xử lý phụ → vi phạm nguyên tắc của C++: chương trình không phải chạy chậm vì những tính năng không dùng đến.
 - Tính đóng gói
 - Khi khai báo một phương thức là phương thức không ảo, ta có ý rằng ta không định để cho phương thức đó bị override.



VIRTUAL FUNCTION

- Constructor: không thể khai báo các constructor ảo
 - Constructor không được thừa kế, bao giờ cũng phải định nghĩa lại nên chuyện ảo không có nghĩa.
- Destructor: có thể (và rất nên) khai báo destructor là hàm ảo.



VIRTUAL FUNCTION

- Destructor:

- Cây thừa kế MotorVehicle, nếu ta tạo và huỷ một đối tượng Car qua một con trỏ tới Car, mọi việc đều xảy ra như mong đợi:

```
Car* c = new Car(10, "Suzuki", "RSX-R1000", 4);
//...
delete c;
// This works correctly - it will trigger
// the Car destructor and then works
// up to the MotorVehicle destructor
```



VIRTUAL FUNCTION

- Destructor:
 - Còn nếu ta dùng một con trỏ tới MotorVehicle thay cho con trỏ tới Car, chỉ có destructor của MotorVehicle được gọi:

```
Car* c = new Car(10, "Suzuki", "RSX-R1000", 4);
MotorVehicle* mv = c; // Upcasting
delete mv;
// With static polymorphism, the compiler thinks
// that this is referring to an instance of
// MotorVehicle, so only the base class
// (MotorVehicle) destructor is invoked
```



VIRTUAL FUNCTION

- Destructor:
 - Chú ý: việc gọi nhầm destructor không ảnh hưởng đến việc thu hồi bộ nhớ
 - Trong mọi trường hợp, phần bộ nhớ của đối tượng sẽ được thu hồi chính xác;
 - Trong ví dụ trước, kể cả nếu chỉ có destructor của **MotorVehicle** được gọi, phần bộ nhớ của toàn bộ đối tượng **Car** vẫn được thu hồi.
 - Tuy nhiên, nếu không gọi đúng destructor, các đoạn mã dọn dẹp quan trọng có thể bị bỏ qua:
 - Chẳng hạn xoá các thành viên được cấp phát động.



VIRTUAL FUNCTION

- Destructor:

- Quy tắc chung: mỗi khi tạo một lớp để được dùng làm lớp cơ sở, ta nên khai báo destructor là hàm ảo:

- Kể cả khi destructor của lớp cơ sở rỗng (không làm gì);
 - Vậy ta sẽ sửa lại lớp MotorVehicle như sau:

```
class MotorVehicle
{
public:
    MotorVehicle(int vin, string make, string model);
    virtual ~MotorVehicle();
};
```



ABSTRACT CLASS

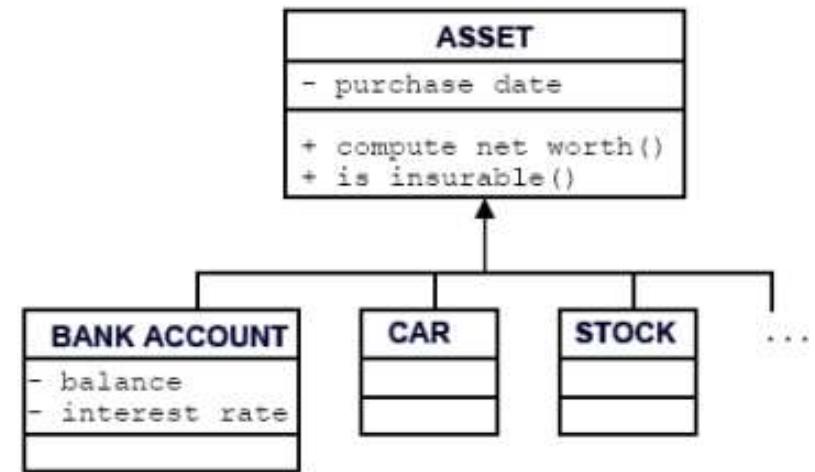
- Tạo thể hiện của lớp:

- Khi mới giới thiệu khái niệm đối tượng, ta nói rằng chúng có thể được nhóm lại thành các lớp, trong đó mỗi lớp là một tập các đối tượng có cùng thuộc tính và hành vi;
- Ta cũng nói theo chiều ngược lại rằng ta có thể định nghĩa một đối tượng như là một thể hiện của một lớp;
- Nghĩa là: lớp có thể *tạo đối tượng*;
- Hầu hết các lớp ta đã gặp đều tạo được thể hiện:
 - Ta có thể tạo thể hiện của Point hay Circle;
 - Ta có thể tạo thể hiện của MotorVehicle hay Car



ABSTRACT CLASS

- Tạo thể hiện của lớp:
 - Tuy nhiên, với một số lớp, có thể không hợp lý khi nghĩ đến chuyện tạo thể hiện của các lớp đó;
 - Ví dụ, một hệ thống quản lý tài sản (asset): chứng khoán (stock), ngân khoản (bank account), bất động sản (real estate), ô tô (car), ...
 - Một đối tượng tài sản chính xác là cái gì?
 - Phương thức computeNetWorth() (tính giá trị) sẽ tính theo kiểu gì nếu không biết đó là ngân khoản, chứng khoán, hay ô tô?
 - Ta có thể nói rằng, một đối tượng ngân khoản là một thể hiện của tài sản, nhưng thực ra không phải, nó là một thể hiện của lớp dẫn xuất của tài sản.





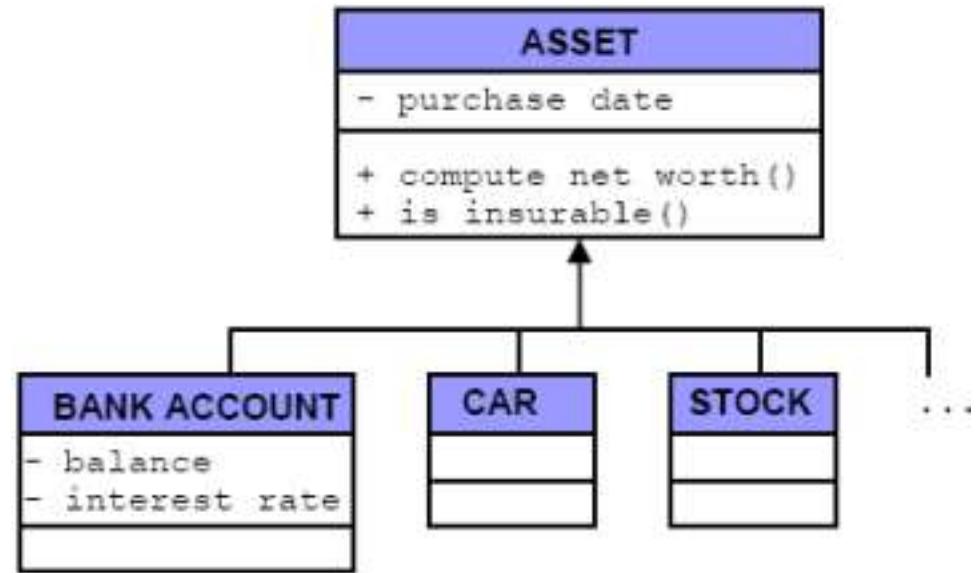
ABSTRACT CLASS

- Chúng ta có thể tạo ra các lớp cơ sở để tái sử dụng mà không muốn tạo ra đối tượng thực của lớp:
 - Các lớp Point, Circle, Rectangle chung nhau khái niệm cùng là hình vẽ Shape.
- Giải pháp là khái báo lớp trừu tượng (Abstract Base Class – ABC) là một lớp không thể tạo thể hiện;
- Thực tế, ta thường phân nhóm các đối tượng theo kiểu này:
 - Chim và ếch đều là động vật, nhưng một con động vật là con gì?
 - Bia và rượu đều là đồ uống, nhưng một thứ đồ uống chính xác là cái gì?
- Có thể xác định xem một lớp có phải là lớp trừu tượng hay không khi ta không thể tìm được một thể hiện của lớp này mà lại không phải là thể hiện của một lớp con
 - Có con động vật nào không thuộc một nhóm nhỏ hơn không?
 - Có đồ uống nào không thuộc một loại cụ thể hơn không?



ABSTRACT CLASS

- Giả sử Asset là lớp trừu tượng, nó có các ích lợi gì?
- Mọi thuộc tính và hành vi của lớp cơ sở (Asset) có mặt trong mỗi thể hiện của các lớp dẫn xuất (BankAccount, Car, Stock,...);
- Ta vẫn có thể nói về quan hệ anh chị em giữa các thể hiện của các lớp dẫn xuất qua lớp cơ sở;
- Nói về một cái ngân khoản cụ thể và một cái xe cụ thể nào đó như đang nói về tài sản
 - Ta có thể tạo một hàm tính tổng giá trị tài sản của một người, trong đó tính đa hình được thể hiện khi gọi hành vi “compute net worth” của các đối tượng tài sản.





ABSTRACT CLASS

- Các lớp trừu tượng chỉ có ích khi chúng là các lớp cơ sở trong cây thừa kế:
 - Ví dụ, nếu ta cho BankAccount là lớp trừu tượng, và nó không có lớp dẫn xuất nào, vậy nó để làm gì?
- Các lớp cơ sở trừu tượng có thể được đặt tại các tầng khác nhau của một cây thừa kế:
 - Có thể coi BankAccount là lớp trừu tượng với các lớp con (chẳng hạn tài khoản tiết kiệm có kỳ hạn và tài khoản thường...);
 - Cây phả hệ động vật.
- Yêu cầu duy nhất là mỗi lớp trừu tượng phải có ít nhất một lớp dẫn xuất không trừu tượng.



THIẾT KẾ THÙA KẾ

- Các cây thừa kế có thể rất phức tạp và có nhiều tầng;
- Vấn đề là: phức tạp đến đâu thì vừa?
- Ta sẽ bàn về thiết kế các cây thừa kế nói chung, trong đó có một số điểm cụ thể về ứng dụng của các lớp trừu tượng.
- Đối với một thiết kế hệ thống bất kỳ, khi thiết kế các cây thừa kế, điều quan trọng là phải chú trọng vào mục đích của cây thừa kế (các lớp/đối tượng này sẽ dùng để làm gì?)
- Khi quyết định nên biểu diễn một nhóm đối tượng bằng một lớp hay một cây thừa kế gồm nhiều lớp, hãy nghĩ xem liệu có đối tượng nào chứa các thuộc tính và hành vi mà các đối tượng khác không có hay không.
- Hơn nữa, ta cần nghĩ xem chính xác những thể hiện nào sẽ được sinh ra trong hệ thống.



THIẾT KẾ THÙA KẾ

- Ví dụ: xét nhóm đối tượng đồ uống – “beverage”;
- Ta có thể nghĩ tới các thông tin sau trong việc tạo thể hiện:
 - Loại đồ uống (cà phê, chè, rượu vang, ...);
 - Các ly/cốc đồ uống cụ thể (cốc chè của tôi, ly rượu của anh, ...).
- Bước đầu tiên: xác định xem hệ thống phải xử lý những gì
 - Có thể hệ thống của ta quản lý các loại đồ uống bán tại một quán ăn. Do đó, ta sẽ tập trung vào dạng thông tin đầu tiên (loại đồ uống).



THIẾT KẾ THÙA KẾ

- Tiếp theo, ta cần làm theo hướng dẫn chung về thiết kế hướng đối tượng – *bắt đầu bằng dữ liệu*:
 - Để bắt đầu, ta có thể muốn quản lý loại đồ uống và giá bán;
 - Nếu chỉ có vậy, ta sẽ chỉ cần đến đúng một lớp đồ uống.
- Đây rõ ràng là một lớp có thể tạo thể hiện, ví dụ:
 - {coffee, \$1.50};
 - {tea, \$1.25};
 - {wine (glass), \$5.00};
 - {wine (bottle), \$20.00};
- Tuy nhiên, nếu ta muốn lưu trữ nhiều thông tin hơn thì sao?



THIẾT KẾ THÙA KẾ

- Tiếp theo, ta cần làm theo hướng dẫn chung về thiết kế hướng đối tượng – *bắt đầu bằng dữ liệu*:
 - Để bắt đầu, ta có thể muốn quản lý loại đồ uống và giá bán;
 - Nếu chỉ có vậy, ta sẽ chỉ cần đến đúng một lớp đồ uống.
- Đây rõ ràng là một lớp có thể tạo thể hiện, ví dụ:
 - {coffee, \$1.50};
 - {tea, \$1.25};
 - {wine (glass), \$5.00};
 - {wine (bottle), \$20.00};
- Tuy nhiên, nếu ta muốn lưu trữ nhiều thông tin hơn thì sao?

BEVERAGE
- type of drink
- price



THIẾT KẾ THÙA KẾ

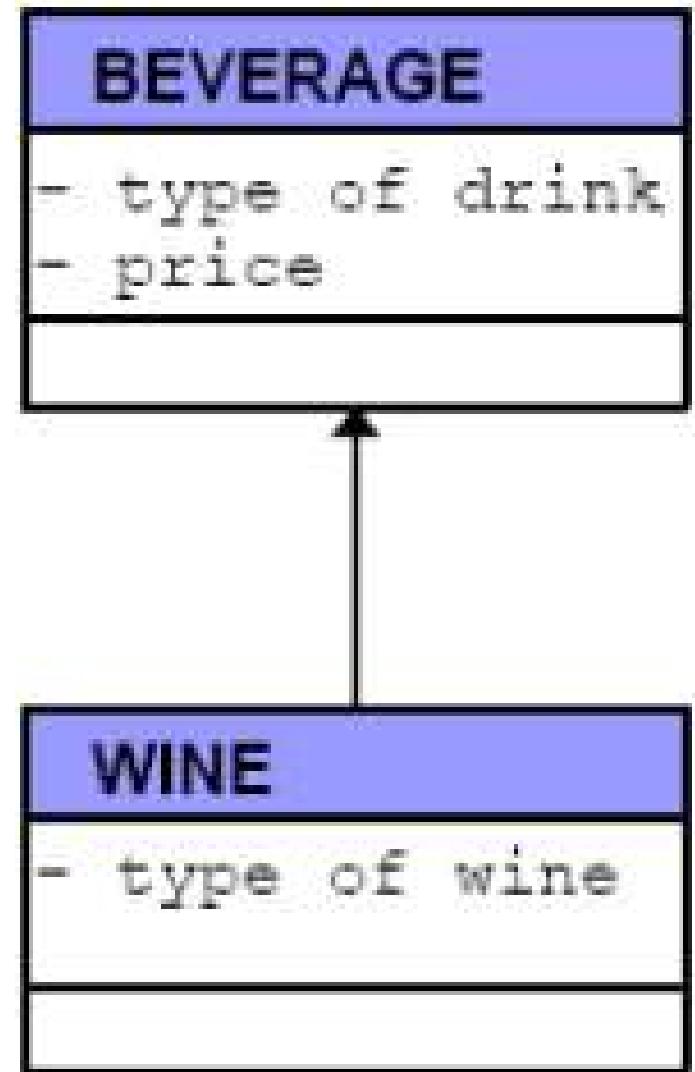
- Có thể, ta còn muốn lưu loại vang (trắng, đỏ, ...);
- Trong trường hợp đó, ta có hai lựa chọn:
 - Thêm một thuộc tính “wine type” vào mọi đồ uống, để trông thuộc tính đó đối với các đồ uống không phải rượu vang;
 - Thêm một lớp con “wine” để chứa thuộc tính bổ sung đó.
- Do đang tập sử dụng thiết kế hướng đối tượng tốt, ta sẽ chọn cách thứ hai.





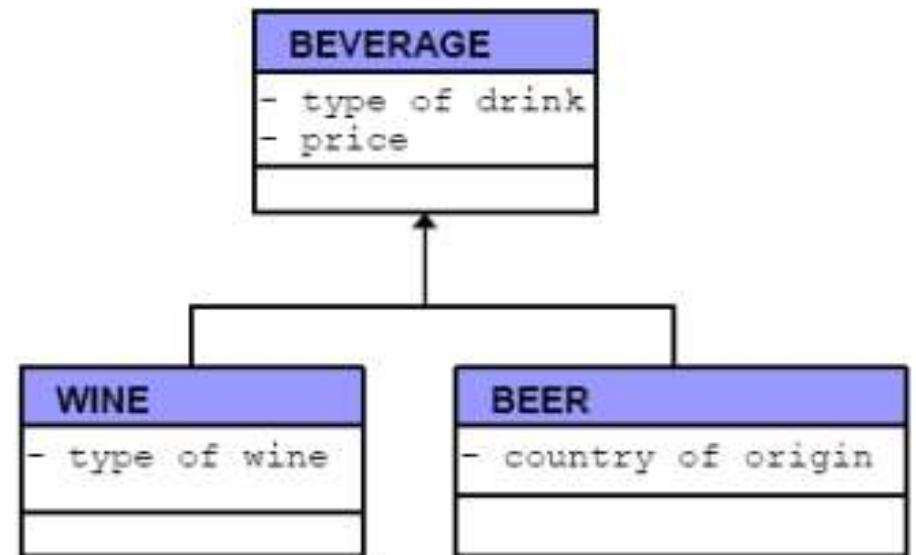
THIẾT KẾ THÙA KẾ

- Trong trường hợp này, việc tạo thể hiện từ cả hai lớp là hợp lý:
 - Lớp Beverage có thể có các thể hiện như: {coffee, \$1.50} và {tea, \$1.25};
 - Lớp Wine có thể có các thể hiện {Meridian (glass), \$5.00, chardonay} và {Lindeman's (bottle), \$20.00, sauvignon} (giá trị thứ ba biểu thị loại vang).
- Vậy, trong trường hợp này, ta không muốn lớp cơ sở là lớp trừu tượng.



THIẾT KẾ THÙA KẾ

- Nay, thay cho quán ăn, ta làm việc với một cửa hàng:
 - Chuyên bán bia và rượu, không bán loại đồ uống nào khác;
 - Cần theo dõi nhãn hiệu và giá;
 - Cần lưu loại vang (như ví dụ trước) cho rượu vang, và nước sản xuất đối với bia.
- Mọi loại rượu bia đều được tạo thể hiện từ hai lớp Wine và Beer
 - Chỉ quản lý rượu và bia;
 - Vậy, không có lý do để tạo thể hiện từ Beverage → ta coi Beverage là lớp trừu tượng.





THIẾT KẾ THÙA KẾ

- Ví dụ cho thấy một lớp cơ sở nên được khai báo là lớp trừu tượng nếu và chỉ nếu ta không bao giờ có lý do gì để tạo thể hiện từ lớp đó;
- Tuy nhiên, trong thiết kế hệ thống, ta chỉ xét trong phạm vi mà hệ thống quan tâm đến:
 - Có nhiều loại đồ uống không phải rượu hay bia (thí dụ chè, cà phê...), nhưng hệ thống của ta chỉ quan tâm đến rượu và bia;
 - Do đó, lớp Beverage là trừu tượng trong phạm vi của thiết kế của ta.
- Do vậy, khi quyết định có nên khai báo một lớp là trừu tượng hay không, ta không nên chú trọng đến nhu cầu tạo thể hiện từ lớp đó trong mọi tình huống, mà nên chú trọng vào nhu cầu tạo thể hiện từ lớp đó trong phạm vi hệ thống cụ thể của ta.



THIẾT KẾ THÙA KẾ

- Vậy, có hai bước để tạo một cây thừa kế:
 - Quyết định xem ta có thực sự cần một cây thừa kế hay không;
 - Xác định các thuộc tính/hành vi thuộc về các lớp cơ sở (lớp cơ sở là bất kỳ lớp nào có lớp dẫn xuất, không chỉ là lớp tại đỉnh cây).
- Nói chung, nếu có một thuộc tính/hành vi mà lớp dẫn xuất nào cũng dùng thì nó nên được đặt tại lớp cơ sở;
- Nói cách khác, nếu ta đang tạo một lớp cơ sở và ta muốn ép mọi lớp dẫn xuất của nó phải cài đặt hành vi nào, thì ta nên đặt hành vi đó vào lớp cơ sở đang tạo.



CÀI ĐẶT LỚP TRÙU TƯỢNG

- Để tạo các lớp trừu tượng, C++ đưa ra khái niệm hàm “thuần” ảo – pure virtual function;
- **Hàm thuần ảo** (hay **phương thức thuần ảo**) là một phương thức ảo được khai báo nhưng không được định nghĩa;
- Cú pháp hàm thuần ảo:
 - Đặt “= 0” vào cuối khai báo phương thức:
virtual void MyMethod(...) = 0;
 - Không cần định nghĩa phương thức;
 - Nếu không có “= 0” và không định nghĩa, trình biên dịch sẽ báo lỗi.



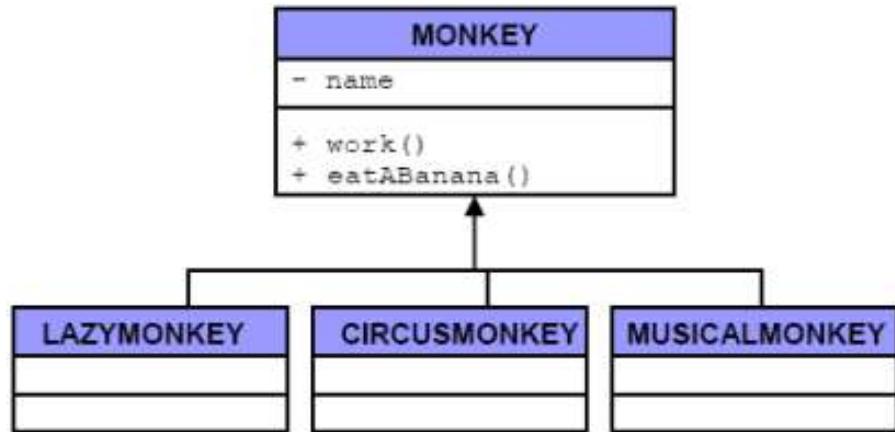
HÀM THUẦN ẢO

- Nếu một lớp có một phương thức không có định nghĩa, C++ sẽ không cho phép tạo thể hiện từ lớp đó, nhờ vậy, lớp đó trở thành lớp trùu tượng:
 - Các lớp dẫn xuất của lớp đó nếu cũng không cung cấp định nghĩa cho phương thức đó thì cũng trở thành lớp trùu tượng.
- Nếu một lớp dẫn xuất muốn tạo thể hiện, nó phải cung cấp định nghĩa cho mọi hàm thuần ảo mà nó thừa kế;
- Do vậy, bằng cách khai báo một số phương thức là thuần ảo, một lớp trùu tượng có thể “bắt buộc” các lớp con phải cài đặt các phương thức đó.



HÀM THUẦN ẢO

- Khai báo `work()` là hàm thuần ảo → **Monkey** trở thành lớp trừu tượng;



```
class Monkey
{
public: Monkey(...);
virtual ~Monkey();
virtual void work()=0;
void eatABanana();
};
```

- Như vậy, nếu các lớp **LazyMonkey**, **CircusMonkey**, **MusicalMonkey**, ... muốn tạo thể hiện thì phải tự cài đặt phương thức `work()` cho riêng mình.



HÀM THUẦN ẢO

- Cũng có thể cung cấp định nghĩa cho hàm thuần ảo;
- Điều này cho phép lớp cơ sở cung cấp mã mặc định cho phương thức, trong khi vẫn cầm lớp trừu tượng tạo thể hiện;
- Để sử dụng đoạn mã mặc định này, lớp con cần cung cấp một định nghĩa gọi trực tiếp phiên bản định nghĩa của lớp cơ sở một cách tường minh.



HÀM THUẦN ẢO

- Ví dụ, trong file chứa định nghĩa **Monkey** (.cpp), ta có thể có định nghĩa phương thức thuần ảo **work()** như sau:

```
void Monkey::work()
{ cout << "No." << endl; }
```
- Tuy nhiên, vì đây là phương thức thuần ảo nên lớp **Monkey** không thể tạo thể hiện;
- Nếu muốn lớp **LazyMonkey** tạo được thể hiện và sử dụng định nghĩa mặc định của **work()**, ta có thể định nghĩa phương thức **work()** của **LazyMonkey** như sau:

```
void LazyMonkey::work()
{ Monkey::work(); }
```



HÀM THUẦN ẢO

- Sử dụng hàm thuần ảo là một cách xây dựng chặt chẽ các cây thừa kế. Nó cho phép người tạo lớp cơ sở quy định chính xác những gì mà các lớp dẫn xuất cần làm, mặc dù ta chưa biết rõ chi tiết:
 - Ta có thể khai báo mọi phương thức mà ta muốn các lớp con cài đặt (quy định về các tham số và kiểu trả về), ngay cả khi ta không biết chính xác các phương thức này cần hoạt động như thế nào.
- Do làm việc với các hàm ảo, nên ta có đầy đủ ích lợi của đa hình động;
- Nên khai báo các lớp cơ sở là trừu tượng mỗi khi có thể, điều đó làm thiết kế rõ ràng mạch lạc hơn:
 - Đặc biệt là khi làm việc với các cây thừa kế lớn hoặc các cây thừa kế được thiết kế bởi nhiều người.



CHƯƠNG 7

TEMPLATE





GENETIC PROGRAMMING

- Lập trình tổng quát là phương pháp lập trình độc lập với chi tiết biểu diễn dữ liệu:
 - Tư tưởng là ta định nghĩa 1 khái niệm không phụ thuộc 1 biểu diễn cụ thể nào, và sau đó mới chỉ ra kiểu dữ liệu cụ thể nào, và sau đó mới chỉ ra kiểu dữ liệu thích hợp làm tham số.
- Qua các ví dụ, ta sẽ thấy đây là 1 phương pháp tự nhiên tuân theo khuôn mẫu hướng đối tượng theo nhiều kiểu.



GENETIC PROGRAMMING

- Ta đã quen với ý tưởng có 1 phương thức được định nghĩa sao cho khi sử dụng với các lớp khác nhau, nó sẽ đáp ứng 1 cách thích hợp:
 - Khi nói về đa hình, nếu phương thức “draw” được gọi cho 1 đối tượng bất kỳ trong cây kế thừa Shape, định nghĩa tương ứng sẽ được gọi để đối tượng được vẽ đúng;
 - Trong trường hợp này, mỗi hình đòi hỏi 1 định nghĩa phương thức hơi khác nhau để đảm bảo sẽ vẽ ra hình đúng.
- Nhưng nếu định nghĩa hàm cho các kiểu dữ liệu khác nhau nhưng không cần phải khác nhau thì sao?



GENETIC PROGRAMMING

- Ví dụ: xét hàm sau

```
void HV(int &a, int &b)
{
    int temp;
    temp = a; a = b; b = temp;
}
```

- Hàm trên chỉ cần hoán đổi giá trị chứa trong 2 biến int;
- Nếu ta muốn thực hiện việc tương tự cho 1 kiểu dữ liệu khác, chẳng hạn float?
- Có thực sự cần đến 2 phiên bản không?

```
void HV(float &a, float &b)
{
    float temp;
    temp = a; a = b; b = temp;
}
```



GENETIC PROGRAMMING

- Ví dụ: ta định nghĩa 1 lớp biểu diễn cấu trúc ngăn xếp cho kiểu int;
- Ta thấy khai báo & định nghĩa của Stack phụ thuộc tại 1 mức độ nào đó vào kiểu dữ liệu int:
 - Một số phương thức lấy tham số & trả về kiểu int;
 - Nếu ta muốn tạo ngăn xếp cho 1 kiểu dữ liệu khác thì sao?
 - Ta có nên định nghĩa lại hoàn toàn lớp Stack (kết quả sẽ tạo ra nhiều lớp chẳng hạn IntStack, FloatStack, ...) hay không?



GENETIC PROGRAMMING

- Ví dụ: ta định nghĩa 1 lớp biểu diễn cấu trúc ngăn xếp cho kiểu int;
- Ta thấy khai báo & định nghĩa của Stack phụ thuộc tại 1 mức độ nào đó vào kiểu dữ liệu int:
 - Một số phương thức lấy tham số & trả về kiểu int;
 - Nếu ta muốn tạo ngăn xếp cho 1 kiểu dữ liệu khác thì sao?
 - Ta có nên định nghĩa lại hoàn toàn lớp Stack (kết quả sẽ tạo ra nhiều lớp chẳng hạn IntStack, FloatStack, ...) hay không?

```
class Stack
{
public:
    Stack();
    ~Stack();
    void push(const int& i);
    void pop(int& i);
    bool isEmpty() const;
private:
    int *data;
```



GENETIC PROGRAMMING

- Ta thấy, trong 1 số trường hợp, đưa chi tiết về kiểu dữ liệu vào trong định nghĩa hàm hoặc lớp là điều không có lợi:
 - Trong khi ta cần các định nghĩa khác nhau cho “draw” của Point hay Circle, vẫn đề khác hẳn trường hợp 1 hàm chỉ có nhiệm vụ hoán đổi 2 giá trị.
- Thực ra, khái niệm lập trình tổng quát học theo sự sử dụng 1 phương pháp của lớp cơ sở cho các thể hiện của các lớp dẫn xuất:
 - Ví dụ: trong cây thừa kế, ta muốn cùng 1 phương thức eatBanana() được thực thi, bất kể con trỏ /tham chiếu đang chỉ tới 1 Monkey hay LazyMonkey.
- Với lập trình tổng quát, ta tìm cách mở rộng sự trừu tượng hóa ra ngoài địa hạt của các cây thừa kế.



GENETIC PROGRAMMING

- Sử dụng tiền xử lý của C:
 - Tiền xử lý thực hiện thay thế text trước khi dịch;
 - Do đó, có thể dùng #define để chỉ ra kiểu dữ liệu & thay đổi tại chỗ khi cần.

```
#define TYPE int
void swap(TYPE & a, TYPE & b) {
    TYPE temp;
    temp = a; a = b; b = temp;
}
```

Trình tiền xử lý sẽ thay mọi "TYPE" bằng "int" trước khi thực hiện biên dịch

- Hạn chế:
 - Nhảm chán & dễ lỗi;
 - Chỉ cho phép đúng 1 định nghĩa trong 1 chương trình.



C++ TEMPLATE

- Template là 1 cơ chế thay thế mã cho phép tạo các cấu trúc mà không phải chỉ rõ kiểu dữ liệu;
- Từ khóa template được dùng trong C++ để báo cho trình biên dịch rằng đoạn mã sau sẽ thao tác 1 hoặc nhiều kiểu dữ liệu chưa xác định:
 - Từ khóa template được sau bởi 1 cặp <> chứa tên của các kiểu dữ liệu tùy ý được cung cấp:

```
template <typename T>
//Declaration that makes reference to a data type "T"
template <typename T, typename U>
//Declaration that makes reference to a data type "T"
// and data type "U"
```

- Chú ý: 1 lệnh template chỉ có hiệu quả đối với khai báo **ngay sau** nó.



FUNCTION TEMPLATE

- Function template: khuôn mẫu hàm cho phép định nghĩa các hàm tổng quát dùng đến các kiểu dữ liệu tùy ý;
- Định nghĩa hàm HV() bằng khuôn mẫu hàm:

```
template <typename T>
void HV(T &a, T &b)
{
    T temp;
    temp = a; a = b; b = temp;
}
```

- Phiên bản trên khá giống với phiên bản swap() bằng C sử dụng #define, nhưng nó mạnh hơn nhiều.



FUNCTION TEMPLATE

- Thực chất, khi sử dụng template, ta đã định nghĩa 1 tập hợp vô hạn các hàm chồng nhau với tên **HV()**;
- Để gọi 1 trong các phiên bản này, ta chỉ cần gọi nó với kiểu dữ liệu tương ứng:

```
int x = 1; y = 2;  
float a = 1.1; b = 2.2;  
HV(x, y); //Invokes int version of HV()  
HV(a, b); //Invokes float version of HV()
```

- Khi biên dịch mã:
 - Sự thay thế “T” trong khai báo/định nghĩa hàm **HV()** không phải thay thế text đơn giản & cũng không được thực hiện bởi tiền xử lý;
 - Việc chuyển phiên bản mẫu **HV()** thành các cài đặt cụ thể cho **int** & **float** được thực hiện bởi trình biên dịch.



FUNCTION TEMPLATE

- Hoạt động của trình biên dịch khi gấp lời gọi hàm HV(int, int):
 - Trình biên dịch tìm xem có 1 hàm HV() được khai báo với 2 tham số kiểu int hay không:
 - Nó không tìm thấy 1 hàm thích hợp, nhưng tìm thấy 1 template có thể dùng được.
 - Tiếp theo, nó xem xét khai báo của template HV() để xem có thể khớp được với lời gọi hàm hay không:
 - Lời gọi hàm cung cấp 2 tham số thuộc cùng 1 kiểu int;
 - Trình biên dịch thấy template chỉ ra 2 tham số thuộc cùng kiểu T, nên nó kết luận rang T phải là kiểu int;
 - Do đó, trình biên dịch kết luận rang template khớp với lời gọi hàm.



FUNCTION TEMPLATE

- Khi đã xác định được template khớp với lời gọi hàm, trình biên dịch kiểm tra xem đã có 1 phiên bản của HV() với 2 tham số kiểu int được sinh ra từ template hay chưa:
 - Nếu đã có, lời gọi được liên kết (bind) với phiên bản đã được sinh (lưu ý: khái niệm liên kết này giống với khái niệm ta đã nói đến trong đa hình tĩnh);
 - Nếu không, trình biên dịch sẽ sinh 1 cài đặt HV() lấy 2 tham số kiểu int (thực ra là viết đoạn mã mà ta sẽ tạo nếu ta tự mình viết) – và liên kết lời gọi hàm với phiên bản vừa sinh.



FUNCTION TEMPLATE

- Vậy, đến cuối quy trình biên dịch đoạn mã trong ví dụ, sẽ có 2 phiên bản của HV() được tạo (1 cho 2 tham số kiểu int, 1 cho 2 tham số kiểu float) với các lời gọi hàm của ta được liên kết với phiên bản thích hợp:
 - Có chi phí phụ về thời gian biên dịch đối với việc sử dụng template;
 - Có chi phí phụ về không gian liên quan đến mỗi cài đặt của HV() được tạo trong khi biên dịch;
 - Tuy nhiên, tính hiệu quả của các cài đặt đó cũng không khác với khi ta tự cài đặt chúng.



FUNCTION TEMPLATE

- Vậy, đến cuối quy trình biên dịch đoạn mã trong ví dụ, sẽ có 2 phiên bản của HV() được tạo (1 cho 2 tham số kiểu int, 1 cho 2 tham số kiểu float) với các lời gọi hàm của ta được liên kết với phiên bản thích hợp:
 - Có chi phí phụ về thời gian biên dịch đối với việc sử dụng template;
 - Có chi phí phụ về không gian liên quan đến mỗi cài đặt của HV() được tạo trong khi biên dịch;
 - Tuy nhiên, tính hiệu quả của các cài đặt đó cũng không khác với khi ta tự cài đặt chúng.

```
int x = 1; y = 2;  
float a = 1.1; b = 2.2;  
HV<int>(x, y);      //Invokes int version of HV()  
HV<float>(a, b);    //Invokes float version of HV()
```



CLASS TEMPLATE

- Class template: khuôn mẫu lớp cho phép sử dụng các thể hiện của 1 hoặc nhiều kiểu dữ liệu tùy ý;
 - Có thể định nghĩa class template cho struct & union.
- Khai báo class template tương tự như function template:
 - Ví dụ: tao 1 struct Pair như sau:
 - Khai báo Pair cho 1 cặp giá trị kiểu int;
 - Ta có thể sửa khai báo trên thành 1 khuôn mẫu lấy kiểu tùy ý. Tuy nhiên 2 thành viên first & second phải thuộc cùng kiểu;
 - Hoặc ta có thể cho phép 2 thành viên nhận các kiểu dữ liệu khác nhau.



CLASS TEMPLATE

- Class template: khuôn mẫu lớp cho phép sử dụng các thê hiện của 1 hoặc nhiều kiểu dữ liệu tùy ý;
 - Ví dụ:

```
struct Pair
{
    int first;
    int second;
};

template <typename T>
struct Pair
{
    T first;
    T second;
};

template <typename T, typename U>
struct Pair
{
    T first;
    U second;
};
```



CLASS TEMPLATE

- Để tạo các thể hiện của template Pair, ta phải dùng ký hiệu cặp ngoặc nhọn:
 - Khác với function template khi ta có thể bỏ qua kiểu dữ liệu cho các tham số , đối với class template chúng phải được cung cấp tường minh.

```
Pair p;           //Not permitted
Pair<int, int> q; //Creates a pair of ints
Pair<int, float> r; //Creates a pair with an int & a float
```

- Tại sao đòi hỏi kiểu tường minh?
 - Cấp phát bộ nhớ cho đối tượng;
 - Nếu không biết các kiểu dữ liệu được sử dụng, trình biên dịch làm thế nào để biết cần đến bao nhiêu bộ nhớ?



CLASS TEMPLATE

- Cũng như function template, không có struct Pair mà chỉ có các struct có tên Pair<int, int>, Pair<int, float>, Pair<int, char>, ...
- Quy trình tạo các phiên bản struct Pair từ class template cũng giống như đối với function template;
- Khi trình biên dịch lần đầu gặp khai báo dùng Pair<int, int>, nó sẽ kiểm tra xem struct đó đã tồn tại chưa, nếu chưa, nó sinh ra 1 khai báo tương ứng:
 - Đối với các class template cho class, trình biên dịch sẽ sinh cả các định nghĩa phương thức cần thiết để khớp với khai báo class.



CLASS TEMPLATE

- Khi đã tạo được 1 thể hiện của 1 class template, ta có thể tương tác với nó như thể nó là thể hiện của 1 class (struct, union) thông thường:

```
Pair<int, int> q;  
Pair<int, float> r;  
q.first = 5;  
q.second = 10;  
r.first = 15;  
r.second = 2.5;
```

- Tiếp theo, ta sẽ tạo 1 class template cho lớp Stack:



CLASS TEMPLATE

- Khi thiết kế class template thường ta tạo nên 1 phiên bản cụ thể trước, sau đó mới chuyển nó thành 1 template:
 - Ví dụ: cài đặt hoàn chỉnh lớp Stack cho số nguyên.
- Điều đó cho phép phát hiện các vấn đề về khái niệm trước khi chuyển thành phiên bản tổng quát:
 - Khi đó, ta có thể test tương đối đầu đủ lớp Stack cho số nguyên để tìm các lỗi tổng quát mà không phải quan tâm đến các vấn đề liên quan đến template.



CLASS TEMPLATE

- Khai báo định nghĩa lớp Stack cho số nguyên:

```
class Stack
{
public:
    Stack();
    ~Stack();
    void push(const int& i) throw (logic_error);
    void pop(int& i) throw (logic_error);
    bool isEmpty() const;
    bool isFull() const;
private:
    static const int max = 10;
    int contents[max];
    int current;
};
```



CLASS TEMPLATE

- Khai báo định nghĩa lớp Stack cho số nguyên:

```
Stack::Stack()
{   this->current = 0;    }
Stack::~Stack()
{}
void Stack::push(const int& i) throw (logic_error)
{
    if(this->current < this->max)
        this->contents[this->current++] = i;
    else
        throw logic_error("Stack is full.");
}
void Stack::pop(int& i) throw (logic_error)
{
    if(this->current > 0)
        i = this->contents[--this->current];
    else
        throw logic_error("Stack is empty.");
}
bool Stack::isEmpty() const
{   return (this->current == 0);      }
bool Stack::isFull() const
{   return (this->current == this->max);      }
```



CLASS TEMPLATE

- Template Stack:

```
template <typename T>
class Stack
{
public:
    Stack();
    ~Stack();
    void push(const T& i) throw (logic_error);
    void pop(T& i) throw (logic_error);
    bool isEmpty() const;
    bool isFull() const;
private:
    static const int max = 10;
    T contents[max];
    int current;
};
```



CLASS TEMPLATE

- Template Stack:

```
template <typename T> ←  
Stack<T>::Stack() {  
    this->current = 0;  
}  
  
template <typename T>  
Stack<T>::~Stack() {}  
  
template <typename T>  
void Stack<T>::push(const T& i) {  
    if (this->current < this->max) {  
        this->contents[this->current++] = i;  
    }  
    else {  
        throw logic_error("Stack is full.");  
    }  
}
```

Mỗi phương thức cần một lệnh **template** đặt trước

Mỗi khi dùng toán tử phạm vi, cần một ký hiệu ngoặc nhọn kèm theo tên kiểu
Ta đang định nghĩa một lớp **Stack<type>**, chứ không định nghĩa lớp **Stack**

Thay thế kiểu của đối tượng được lưu trong ngăn xếp (trước là **int**) bằng kiểu tùy ý **T**



CLASS TEMPLATE

- Template Stack:

```
template <typename T>
void Stack<T>::pop(T& i) {
    if (this->current > 0) {
        i = this->contents[--this->current];
    }
    else {
        throw logic_error("Stack is empty.");
    }
}

template <typename T>
bool Stack<T>::isEmpty() const {
    return (this->current == 0);
}

template <typename T>
bool Stack<T>::isFull() const {
    return (this->current == this->max);
}
```



CLASS TEMPLATE

- Template Stack:

- Sau đó, ta có thể tạo & sử dụng các thê hiện của các lớp đã được định nghĩa bởi template:

```
int x = 5, y;  
char c = 'a', d;  
  
Stack<int> s;  
Stack<char> t;  
  
s.push(x);  
t.push(c);  
s.pop(y);  
t.pop(d);
```



CÁC THAM SỐ CỦA TEMPLATE

- Ta mới nói đến các lệnh template với tham số thuộc kiểu **typename**;
- Tuy nhiên, còn có 2 kiểu tham số khác:
 - Kiểu thực sự (int, float, ...);
 - Các template khác.



CÁC THAM SỐ CỦA TEMPLATE

- Trong cài đặt template Stack, ta có 1 hàng max quy định số lượng tối đa các đối tượng mà ngăn xếp có thể chứa:
 - Như vậy, mỗi thẻ hiện sẽ có cùng kích thước đối với mọi kiểu của đối tượng được chứa.
- Nếu ta không muốn đòi hỏi mọi Stack đều có kích thước tối đa như nhau?
- Ta có thể thêm 1 tham số vào lệnh template chỉ ra 1 số int (giá trị này sẽ được dùng để xác định giá trị cho max): template <typename T, **int I**>
- Lưu ý: ta khai báo tham số int giống như trong các khai báo khác



CÁC THAM SỐ CỦA TEMPLATE

- Sửa khai báo & định nghĩa template Stack:

```
template <typename T, int I>
class Stack {
    public:
        Stack();
        ~Stack();
        void push(const T& i) throw (logic_error);
        void pop(T& i) throw (logic_error);
        bool isEmpty() const;
        bool isFull() const;
    private:
        static const int max = I;
        T contents[max];
        int current;
};
```

Khai báo tham số mới

Sử dụng tham số mới để xác định giá trị **max** của một lớp thuộc một kiểu nào đó



CÁC THAM SỐ CỦA TEMPLATE

- Sửa khai báo & định nghĩa template Stack:

Sửa tên lớp dùng cho các toán tử phạm vi

```
template <typename T, int I>
Stack<T, I>::Stack() {
    this->current = 0;
}

template <typename T, int I>
Stack<T, I>::~Stack() {}

template <typename T, int I>
void Stack<T, I>::push(const T& i) {
    if (this->current < this->max) {
        this->contents[this->current++] = i;
    }
    else {
        throw logic_error("Stack is full.");
    }
}
```

Sửa các lệnh template



CÁC THAM SỐ CỦA TEMPLATE

- Ta có thể tạo các thể hiện của các lớp Stack với các kiểu dữ liệu & kích thước đa dạng:

```
Stack<int, 5> s; // Creates an instance of a Stack
                  // class of ints with max = 5
Stack<int, 10> t; // Creates an instance of a Stack
                  // class of ints with max = 10
Stack<char, 5> u; // Creates an instance of a Stack
                  // class of chars with max = 5
```

- Lưu ý rằng các lệnh trên tạo thể hiện của 3 lớp khác nhau.



CÁC THAM SỐ CỦA TEMPLATE

- Các ràng buộc khi sử dụng các kiểu thực sự làm tham số cho lệnh template:
 - Chỉ có thể dùng kiểu số nguyên, con trỏ hoặc tham chiếu;
 - Không được gán giá trị cho tham số hoặc lấy địa chỉ của tham số.
- Loại tham số cho template là 1 template khác:
 - Ví dụ: thiết kế template cho lớp Map (ánh xạ) ánh xạ các khóa tới các giá trị:
 - Lớp này cần lưu các ánh xạ từ khóa tới giá trị, nhưng ta không muốn chỉ ra kiểu của các đối tượng được lưu trữ ngay từ đầu;
 - Cần tạo Map là 1 class template sao cho có thể sử dụng các kiểu khác nhau cho khóa & giá trị;
 - Tuy nhiên, cần chỉ ra lớp chứa (container) là 1 class template, để nó có thể lưu trữ các khóa & giá trị là các kiểu tùy ý.



CÁC THAM SỐ CỦA TEMPLATE

- Khai báo lớp Map:

```
template <typename K, typename V,
          template <typename T> Container>
class Map
{
    private:
        Container<K> keys;
        Container<V> value;
};
```

- Tạo thể hiện của lớp Map:

Map<string, int, Stack> wordcount;

- Lệnh trên tạo 1 thể hiện của lớp Map<string, int, Stack> chứa các thành viên là 1 tập các string & 1 tập các int (giả sử còn có các đoạn mã thực hiện ánh xạ mỗi từ tới 1 số int biểu diễn số lần xuất hiện của từ đó):

- Ta đã dùng template Stack để làm Container lưu trữ các thông tin trên.



CÁC THAM SỐ CỦA TEMPLATE

- Như vậy, khi trình biên dịch sinh ra các khai báo & định nghĩa thực sự cho các lớp Map, nó sẽ đọc các tham số mô tả các thành viên dữ liệu;
- Khi đó, nó sẽ sử dụng template Stack để sinh mã cho 2 lớp **Stack<string>** & **Stack<int>**;
- Đến đây, ta phải hiểu rõ tại sao container phải là 1 template, nếu không, làm thế nào để có thể dùng nó để tạo các loại stack khác nhau.



CHƯƠNG 8

EXCEPTION





XỬ LÝ LỖI

- Chương trình nào cũng có khả năng gặp phải các tình huống không mong muốn
 - Người dùng nhập dữ liệu không hợp lệ;
 - Đĩa cứng bị đầy;
 - File cần mở bị khóa;
 - Đối số cho hàm không hợp lệ;
- Xử lý như thế nào?
 - Một chương trình không quan trọng có thể dừng lại;
 - Chương trình điều khiển không lưu? Điều khiển máy bay?



XỬ LÝ LỖI

```
double MyDivide(double numerator, double denominator) {  
    if (denominator == 0.0) {  
        // Do something to indicate that an error occurred  
    }  
    else {  
        return numerator / denominator;  
    }  
}
```

xử lý lỗi
như thế nào
bây giờ?

- Mã thư viện (nơi gấp lỗi) thường không có đủ thông tin để xử lý lỗi:
 - Cần có cơ chế để mã thư viện báo cho mã ứng dụng rằng nó vì 1 lý do nào đó không thể tiếp tục chạy được, để mã ứng dụng xử lý tùy theo tình huống.



XỬ LÝ LỖI TRUYỀN THÔNG

- Xử lý lỗi truyền thông thường là mõi hàm lại thông báo trạng thái *thành công/thất bại* qua 1 mã lỗi:
 - Biến toàn cục (chẳng hạn **errno**);
 - Giá trị trả về:
 - int remove(const char *filename);
 - Tham số phụ là tham chiếu:
 - double MyDivide(double numerator, double denominator, int &status);



XỬ LÝ LỖI TRUYỀN THÔNG

- Hạn chế:
 - Phải có lệnh kiểm tra lỗi sau mỗi lời gọi hàm:
 - Code rối rắm, dài, khó đọc.
 - Lập trình viên ứng dụng quên kiểm tra hoặc cố tình bỏ qua:
 - Bản chất con người;
 - Lập trình viên ứng dụng thường không có kinh nghiệm bằng lập trình viên thư viện.
 - Rắc rối khi đầy thông báo lỗi từ hàm được gọi sang hàm gọi vì từ 1 hàm ta chỉ có thể trả về 1 kiểu thông báo lỗi.



C++ EXCEPTION

- Exception – ngoại lệ là cơ chế thông báo & xử lý lỗi giải quyết được các vấn đề kể trên;
- Tách được phần xử lý lỗi ra khỏi phần thuật toán chính; cho phép 1 hàm thông báo về nhiều loại ngoại lệ;
- Không phải hàm nào cũng phải xử lý lỗi:
 - Nếu có 1 số hàm gọi thành chuỗi, ngoại lệ chỉ cần được xử lý tại 1 hàm là đủ.
- Không thể bỏ qua ngoại lệ, nếu không chương trình sẽ kết thúc;
- Cơ chế ngoại lệ mềm dẻo hơn xử lý lỗi truyền thống.



CÁC KIỂU & CƠ CHẾ NGOẠI LỆ

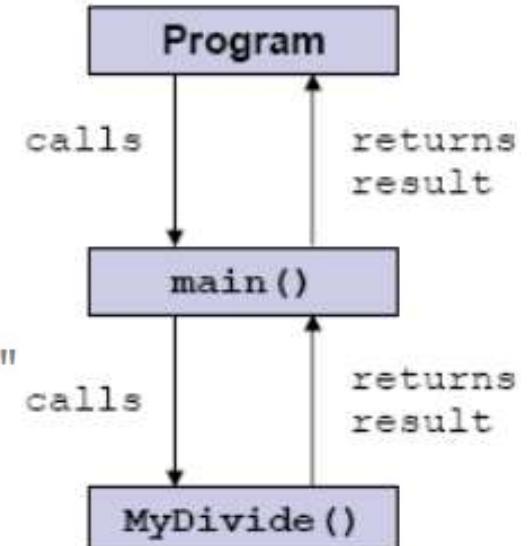
- Một ngoại lệ là 1 đối tượng chứa thông tin về 1 lỗi & được dùng để truyền thông tin đó tới cấp thực thi cao hơn;
- Ngoại lệ có thể thuộc kiểu dữ liệu bất kỳ của C++:
 - Có sẵn: int, char*, ...
 - Kiểu người dùng tự định nghĩa (thường dùng);
 - Các lớp ngoại lệ trong thư viện `<exception>`.
- Cơ chế ngoại lệ:
 - Quá trình truyền ngoại lệ từ ngữ cảnh thực thi hiện hành tới mức thực thi cao hơn gọi là **ném ngoại lệ** (throw an exception):
 - Vị trí trong mã của hàm nơi ngoại lệ được ném được gọi là **điểm ném** (throw point).
 - Khi 1 ngữ cảnh thực thi tiếp nhận & truy nhập 1 ngoại lệ, nó được coi là **bắt ngoại lệ** (catch the exception).



CƠ CHẾ NGOẠI LỆ

- Quy trình gọi hàm & trả về trong trường hợp bình thường:

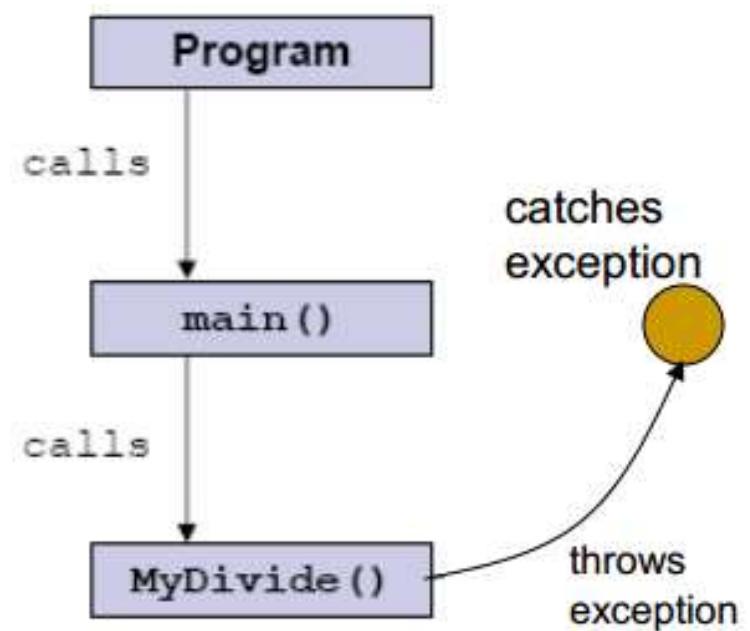
```
int main()
{
    int x, y;
    //Prompt user for two numbers
    cout << "Enter two integers, separated by";
    cin >> x >> y;
    cout << "The first number divided by the second is:" 
    |   | << MyDivide(x, y) << endl;
    return 0;
}
```





CƠ CHẾ NGOẠI LỆ

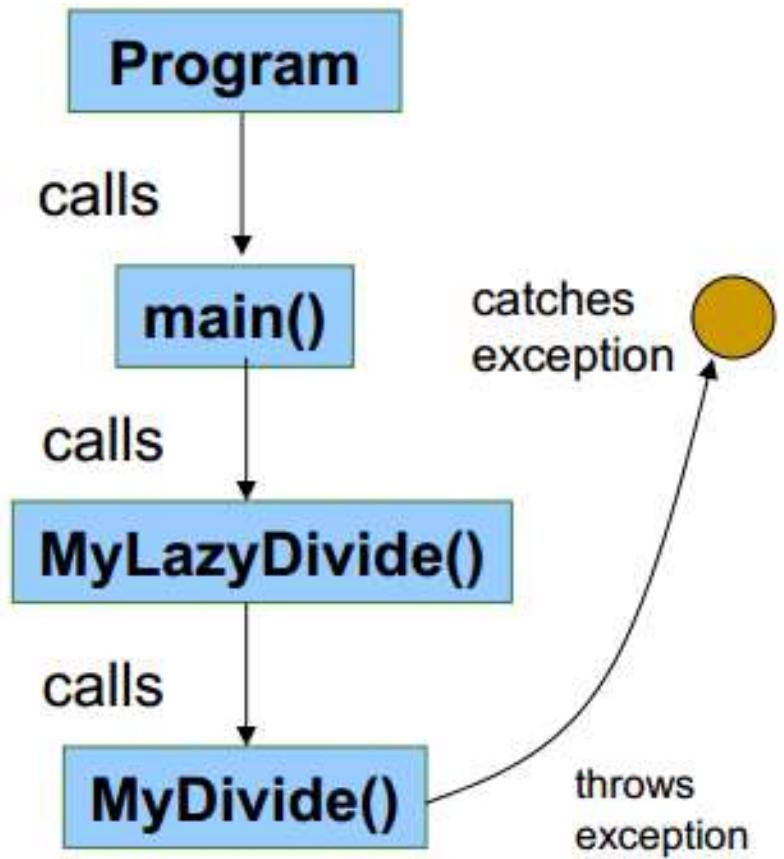
- Quy trình ném & bắt ngoại lệ:
 - Giả sử người dùng nhập mẫu số bằng 0;
 - Mã chương trình trong MyDivide() tạo 1 ngoại lệ (bằng cách nào đó) & ném;
 - Khi 1 hàm ném ngoại lệ, nó lập tức kết thúc thực thi & gửi ngoại lệ đó cho nơi gọi nó;
 - Nếu main() có thể xử lý ngoại lệ, nó sẽ bắt & giải quyết ngoại lệ.
- Chẳng hạn yêu cầu người dùng nhập lại mẫu số.





CƠ CHẾ NGOẠI LỆ

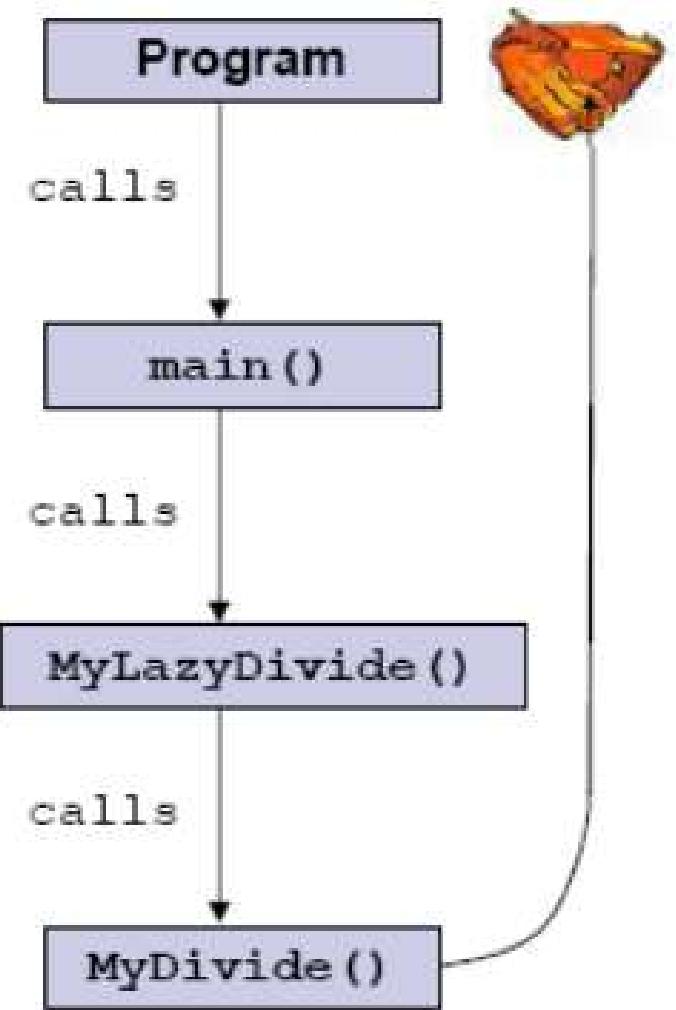
- Nếu 1 hàm không thể bắt ngoại lệ:
 - Giả sử hàm **MyLazyDivide()** không thể bắt ngoại lệ do **MyDivide()** ném.
- Không phải hàm nào bắt được ngoại lệ cũng có thể bắt được **mọi loại** ngoại lệ;
- Chẳng hạn hàm `f()` bắt được các ngoại lệ `E1` nhưng không bắt được các ngoại lệ `E2`;
- Ngoại lệ đó sẽ được chuyển lên mức trên cho `main()` bắt.





CƠ CHẾ NGOẠI LỆ

- Nếu không có hàm nào bắt được ngoại lệ:
 - Hiện giờ ví dụ của ta vẫn chưa có đoạn mã bắt ngoại lệ nào, nếu có 1 ngoại lệ được ném, nó sẽ được chuyển qua tất cả các hàm.
- Tại mức thực thi cao nhất, chương trình tổng (nơi gọi hàm **main()**) sẽ bắt mọi ngoại lệ còn sót lại mà nó nhìn thấy;
- Khi đó, chương trình lập tức kết thúc:
 - Quy trình này không hoàn toàn giống với các quy trình “bắt” thông thường & ta nên tránh không để chúng xảy ra.





CÚ PHÁP XỬ LÝ NGOẠI LỆ

- Ta đã có các khái niệm cơ bản về xử lý ngoại lệ, sử dụng cơ chế đó trong C++ như thế nào?
- Cơ chế xử lý ngoại lệ của C++ có 3 tính năng chính:
 - Khả năng tạo & ném ngoại lệ (sử dụng từ khóa **throw**);
 - Khả năng bắt & giải quyết ngoại lệ (sử dụng từ khóa **catch**);
 - Khả năng tách logic xử lý ngoại lệ trong 1 hàm ra khỏi phần còn lại của hàm (sử dụng từ khóa **try**).



THROW – NÉM NGOẠI LỆ

- Để ném 1 ngoại lệ, ta dùng từ khóa **throw**, kèm theo đối tượng mà ta định ném:

throw <object>;

- Ta có thể dùng **mọi thứ** làm ngoại lệ, kể cả giá trị thuộc kiểu có sẵn:
 - **throw 15;**
 - **throw MyObj();**
- Ví dụ: **MyDivide()** ném ngoại lệ là 1 **string**:

```
double MyDivide(double numerator, double denominator)
{
    if(denominator == 0.0)
        throw string("The denominator cannot be 0");
    else
        return numerator/denominator;
}
```



THROW – NÉM NGOẠI LỆ

- Trường hợp cần cung cấp nhiều thông tin hơn cho hàm gọi, ta tạo 1 class dành riêng cho các ngoại lệ;
- Ví dụ: ta cần cung cấp cho người dùng 2 số nguyên
 - Ta có thể tạo 1 lớp ngoại lệ:

```
class MyExceptionClass
{
    public:
        MyExceptionClass(int a, int b);
        ~MyExceptionClass();
        int getA();
        int getB();
    private:    int a, b;
};
```

- Sau đó, dùng thể hiện của lớp vừa tạo để làm ngoại lệ.

```
int x, y;
if(...) throw MyExceptionClass(x, y);
```



KHỐI TRY - CATCH

- Khối try – catch dùng để:
 - Tách phần giải quyết lỗi ra khỏi phần có thể sinh lỗi;
 - Quy định các loại ngoại lệ được bắt tại mức thực thi hiện hành.
- Cú pháp chung cho khối try – catch:

```
try {
    // Code that could generate an exception
}
catch (<Type of exception>) {
    // Code that resolves an exception of that type
};
```

- Mã liên quan đến thuật toán nằm trong khối try;
- Mã giải quyết lỗi đặt trong (các) khối catch.
- Tách mã → chương trình dễ hiểu, dễ bảo trì hơn.



KHỐI TRY - CATCH

- Có thể có nhiều khối catch, mỗi khối chứa mã để giải quyết 1 loại ngoại lệ cụ thể:

```
try {  
    // Code that could generate an exception  
}  
catch (<Exception type1>) {  
    // Code that resolves a type1 exception  
}  
catch (<Exception type2>) {  
    // Code that resolves a type2 exception  
}  
...  
catch (<Exception typeN>) {  
    // Code that resolves a typeN exception  
};
```

Dấu chấm phẩy đánh };
dấu kết thúc của
tổn khối try-catch



KHỐI TRY - CATCH

- Ta viết lại hàm **main()** ban đầu để sử dụng khối **try - catch**:

```
int main() {  
    int x, y;  
    double result;  
    // Prompt user for two numbers  
    cout << "Enter two integers, separated by a space: ";  
    cin >> x >> y;  
    try {  
        result = MyDivide(x, y); // Use temporary variable to separate call  
                               // to MyDivide() from output code  
    }  
    catch (string) {  
        // resolve error  
    };  
    cout << "The first number divided by the second is: " << result << endl;  
}
```

Chú ý: **MyDivide()** ném ngoại lệ là một **string**

Giải quyết lỗi như thế nào?



KHỐI TRY - CATCH

```
int main() {  
    int x, y;  
    double result;  
    bool success;  
    do {  
        success = true; // Set success to true  
        cout << "Enter two integers, separated by a space: "; cin >> x >> y;  
        try {  
            result = MyDivide(x, y);  
        }  
        catch (string& s) {  
            cout << s << endl;  
            success = false; // An exception occurred - set success to false  
        };  
    } while (success == false); // If success == false, repeat loop  
    cout<<"The first number divided by the second is:"<< result<<endl;  
}
```

Có thể dùng một biến bool làm cờ báo hiệu thành công hay thất bại và đưa khối **try-catch** vào trong một vòng **do..while** để thực hiện nhiệm vụ cho đến khi thành công



KHỒI TRY - CATCH

- Tại lệnh **catch**, có 2 thay đổi so với phiên bản trước:
 - Ta đã sửa kiểu thành tham chiếu đến **string** thay vì **string**. Cách này hiệu quả hơn & tránh được slicing nếu ta làm việc với các ngoại lệ là thể hiện của các lớp dẫn xuất.
- Ngoại lệ bắt được được đặt tên (“s”) để ta có thể truy nhập nó từ bên trong khối **catch**.



EXCEPTION MATCHING

- Khi 1 ngoại lệ được ném từ trong 1 khối **try**, hệ thống xử lý ngoại lệ sẽ kiểm tra các kiểu được liệt kê trong khối **catch** theo thứ tự liệt kê:
 - Khi tìm thấy kiểu ăn khớp, ngoại lệ được coi là được giải quyết, không cần tiếp tục tìm kiếm;
 - Nếu không tìm thấy, mức thực thi hiện hành bị kết thúc, ngoại lệ được chuyển lên mức cao hơn.
- Chú ý: khi tìm các kiểu dữ liệu khớp với ngoại lệ, trình biên dịch nói chung sẽ không thực hiện đổi kiểu tự động:
 - Nếu 1 ngoại lệ kiểu **float** được ném, nó sẽ không khớp với 1 khối **catch** cho ngoại lệ kiểu **int**.
- Tuy nhiên, 1 đối tượng hoặc tham chiếu kiểu dẫn xuất sẽ khớp với 1 lệnh **catch** dành cho kiểu cơ sở



EXCEPTION MATCHING

- Do vậy trong đoạn mã sau, mọi ngoại lệ là đối tượng được sinh từ cây **MotorVehicle** sẽ khớp lệnh **catch** đầu tiên (các lệnh còn lại sẽ không bao giờ chạy):

```
try {...}  
//Matches everything from the MotorVehicle hierarchy  
catch(MotorVehicle &mv) {...}  
catch(Car &c) {...}  
catch(Truck &t) {...};
```

- Nếu muốn bắt các ngoại lệ dẫn xuất tách khỏi ngoại lệ cơ sở, ta phải xếp lệnh **catch** cho lớp dẫn xuất lên trước:

```
try {...}  
catch(Car &c) {...}  
catch(Truck &t) {...}  
catch(MotorVehicle &mv) {...};
```



EXCEPTION MATCHING

- Nếu ta muốn bắt tất cả các ngoại lệ được ném (kể cả các ngoại lệ ta không thể giải quyết):
 - Ví dụ: ta có thể cần chạy 1 số đoạn mã dọn dẹp trước khi hàm kết thúc (chẳng hạn dọn dẹp các vùng bộ nhớ cấp phát động).
- Để có 1 lệnh **catch** bắt được mọi ngoại lệ, ta đặt dấu “...” bên trong lệnh **catch**:

catch(...){...};

- Do tham số “...” bắt được mọi ngoại lệ, ta chỉ sử dụng nó cho lệnh **catch** cuối cùng trong 1 khối **try-catch** (nếu không, nó sẽ vô hiệu hóa các lệnh **catch** đứng sau).

```
try {...}
catch(<exception type1>) {...}
catch(<exception type1>) {...}
//...
catch(<exception typeN>) {...}
catch(...) {...};
```



RE-THROWING EXCEPTION

- Chuyển tiếp ngoại lệ;
- Trong 1 số trường hợp, ta có thể muốn chuyển tiếp 1 ngoại lệ mà ta đã bắt – thường là khi ta không có đủ thông tin để giải quyết trọn vẹn ngoại lệ đó:
 - Thông thường, ta sẽ chuyển tiếp 1 ngoại lệ sau khi thực hiện 1 số công việc dọn dẹp bên trong 1 lệnh **catch “...”**;
 - Để chuyển tiếp, ta dùng từ khóa **throw**, không kèm tham số, từ bên trong lệnh **catch**.

```
catch(...)  
{  
    cout << "An exception was thrown." << endl;  
    //...  
    throw; //re-throw exception  
};
```



RE-THROWING EXCEPTION

- Khi 1 ngoại lệ được chuyển tiếp, mọi lệnh catch còn lại trong khối sẽ bị bỏ qua, ngoại lệ được chuyển lên mức thực thi tiếp theo (như ném 1 ngoại lệ mới);
- Do ngoại lệ được ném lại mà không bị sửa đổi, ta có thể thấy được tầm quan trọng của việc sử dụng tham chiếu
 - Nếu 1 lệnh catch cho lớp cơ sở với tham số không phải tham chiếu bắt được 1 ngoại lệ thuộc lớp dẫn xuất, nó sẽ thay đổi vĩnh viễn đối tượng để đối tượng đó chỉ còn chứa các thuộc tính của lớp cơ sở;
 - Bằng cách sử dụng tham chiếu, ta có thể đảm bảo rằng ngoại lệ được chuyển tiếp mà không bị thay đổi.



QUẢN LÝ BỘ NHỚ

- Ta đã biết: khi 1 ngoại lệ được ném, hàm thực hiện đang chạy sẽ kết thúc, điều khiển được trả về cho mức thực thi tiếp theo cao hơn cho đến khi gặp điểm bắt ngoại lệ. Stack sẽ được cuộn cho đến khi gặp điểm bắt ngoại lệ;
- Do đó quy trình dọn dẹp tự động xảy ra như khi hàm kết thúc bình thường, đó là:
 - Các đối tượng được cấp phát tự động bên trong hàm sẽ được thu hồi;
 - Trong các đối tượng trên, đối với đối tượng bất kỳ mà constructor của nó đã được thực hiện hoàn chỉnh, destructor của nó sẽ được gọi.
- Các đối tượng còn lại phải được hủy 1 cách tường minh.



QUẢN LÝ BỘ NHỚ

- Ví dụ:

```
bool FlightList::contains(Flight *f) const throw(char*) {
    Flight flight;
    Airport *a = new Airport("SFO", "San Francisco");
    ...
    throw "Out of Bounds";
    ...
}
...
try {
    if (flightList->contains(flight))
        ...
} catch(char* str) {
    // Handle the error
}
```

Khi ngoại lệ được ném,
destructor cho **flight**
được tự động gọi, nhưng
destructor của **a** thì không.



QUẢN LÝ BỘ NHỚ

- Nếu không tìm thấy lệnh **catch** tương ứng, sau khi mọi hàm đã kết thúc, 1 hàm thư viện đặc biệt **terminate()** sẽ được chạy (có thể coi đây là nơi bắt ngoại lệ cuối cùng);
- Trường hợp mặc định, **terminate()** gọi hàm **abort()**, hàm này sẽ lập tức kết thúc chương trình:
 - Trong trường hợp này, quy trình dọn dẹp không xảy ra, như vậy, destructor của các đối tượng tĩnh & toàn cục sẽ không được gọi;
 - Lưu ý rằng **terminate()** cũng được gọi ngay khi có 1 ngoại lệ được ném trong quy trình dọn dẹp (nghĩa là 1 destructor cho 1 đối tượng cấp phát tự động ném ngoại lệ trong quá trình unwind).
- Có thể thay đổi hoạt động của **terminate()** bằng cách sử dụng hàm **set_terminate()**.



LÓP EXCEPTION

- Để tích hợp hơn nữa các ngoại lệ vào ngôn ngữ C++, lớp **exception** đã được đưa vào thư viện chuẩn:
 - Sử dụng **#include <exception>**; và **using namespace std;**
- Sử dụng thư viện này, ta có thể ném các thê hiện của **exception** hoặc tạo các lớp dẫn xuất từ đó;
- Lớp **exception** có 1 hàm ảo **what()**, có thể định nghĩa lại **what()** để trả về 1 xâu ký tự:

```
try { . . . }
catch (exception e)
{     cout << e.what(); }
```



LỚP EXCEPTION

- Một số lớp ngoại lệ chuẩn khác được dẫn xuất từ lớp cơ sở **exception**;
- File header **<stdexcept>** (cũng thuộc thư viện chuẩn C++) chứa 1 số lớp ngoại lệ dẫn xuất từ **exception**:
 - File này cũng đã **#include <exception>** nên khi dùng không cần **#include** cả 2.
- Trong đó có 2 lớp quan trọng được dẫn xuất trực tiếp từ **exception**:
 - **runtime_error**;
 - **logic_error**.



LỚP EXCEPTION

- **runtime_error** dùng để đại diện cho các lỗi trong thời gian chạy (các lỗi là kết quả của các tình huống không mong đợi, chẳng hạn: hết bộ nhớ);
- **logic_error** dùng cho các lỗi logic chương trình (chẳng hạn truyền tham số không hợp lệ);
- Thông thường, ta sẽ dùng các lớp này (hoặc các lớp dẫn xuất của chúng) thay vì dùng trực tiếp **exception**:
 - Một lý do là cả 2 lớp này đều có constructor nhận tham số là 1 **string** mà nó sẽ là kết quả trả về của hàm **what()**.



LÓP EXCEPTION

- **runtime_error** có các lớp dẫn xuất sau:
 - *range_error*: điều kiện sau bị vi phạm;
 - *overflow_error*: xảy ra tràn số học;
 - *bad_alloc*: không thể cấp phát bộ nhớ.
- **logic_error** có các lớp dẫn xuất sau:
 - *domain_error*: điều kiện trước bị vi phạm;
 - *invalid_argument*: tham số không hợp lệ được truyền cho hàm;
 - *length_error*: tạo đối tượng lớn hơn độ dài cho phép;
 - *out_of_range*: tham số ngoài khoảng (chẳng hạn chỉ số không hợp lệ).



LỚP EXCEPTION

- Ta có thể viết lại hàm MyDivide() để sử dụng các ngoại lệ chuẩn tương ứng như sau:

```
double MyDivide(double numerator, double denominator)
{
    if(denominator == 0.0)
        throw invalid_argument("The denominator cannot be 0.");
    else
        return numerator/denominator;
}
```

- Ta sẽ phải sửa lệnh catch cũ để bắt được ngoại lệ kiểu **invalid_argument** (thay cho kiểu **string** trong phiên bản trước)



LÓP EXCEPTION

```
int main()
{
    int x, y;
    double result;
    do
    {
        cout << "Enter two integers, separated by a space:";
        cin >> x >> y;
        try
        {
            result = MyDivide(x, y);
        }
        catch(invalid_argument &e)
        {
            cout << e.what() << endl;
            continue;
        }
    }
    while(0);
    cout << "The first number divided by the second is:"
        << result << endl;
    return 0;
}
```



KHAI BÁO NGOẠI LỆ

- Làm thế nào để user biết được 1 hàm/phương thức có thể ném những ngoại lệ nào?
- Đọc chú thích, tài liệu?
 - Không phải lúc nào cũng có tài liệu & tài liệu đủ thông tin;
 - Không tiện nếu phải kiểm tra cho mọi hàm.
- C++ cho phép khai báo 1 hàm có thể ném những loại ngoại lệ nào hoặc sẽ không ném ngoại lệ:
 - Một phần của giao diện của hàm;
 - Ví dụ: hàm **MyDivide()** có thể ném ngoại lệ **invalid_argument**.



KHAI BÁO NGOẠI LỆ

- Cú pháp: từ khóa throw ở cuối lệnh khai báo hàm, tiếp theo là cặp ngoặc “**0**” chứa 1 hoặc nhiều tên kiểu (tách nhau bằng dấu “**,**”):
void MyFunction(...) **throw(type1, type2, ..., typeN)** {...}
 - Hàm không bao giờ ném ngoại lệ:
void MyFunction(...) **throw()** {...}
- Cú pháp tương tự đối với phương thức:
bool FlightList::contains(Flight *f) const **throw(char *)** {...}
- Nếu không có khai báo throw, hàm/phương thức có thể ném bất kỳ loại ngoại lệ nào.



KHAI BÁO NGOẠI LỆ

- Chuyện gì xảy ra nếu ta ném 1 ngoại lệ thuộc kiểu không có trong khai báo?
 - Nếu 1 hàm ném 1 ngoại lệ không thuộc các kiểu đã khai báo, hàm **unexcepted()** sẽ được gọi;
 - Theo mặc định, **unexcepted()** gọi hàm **terminate()** mà ta đã nói đến;
 - Tương tự **terminate()**, hoạt động của **unexcepted()** cũng có thể được thay đổi bằng cách sử dụng hàm **set_unexpected()**.



KHAI BÁO NGOẠI LỆ

- Ta phải đặc biệt cẩn trọng khi làm việc với các cây thừa kế & các khai báo ngoại lệ;
- Giả sử có lớp cơ sở **B** chứa 1 phương thức ảo **foo()**:
 - Khai báo **foo()** có thể ném 2 loại ngoại lệ **e1** & **e2**:

```
class B
{
    public:
        void foo() throw(e1, e2);
};
```

- Giả sử **D** là lớp dẫn xuất của **B**, **D** định nghĩa lại **foo()**:
 - Cần có những hạn chế nào đối với khả năng ném ngoại lệ của **D**?



KHAI BÁO NGOẠI LỆ

- Khai báo 1 phương thức về cốt yếu là để tuyên bố những gì người dùng có thể mong đợi từ phương thức đó:
 - Đưa ngoại lệ vào khai báo hàm/phương thức hạn chế các loại đối tượng có thể được ném từ hàm/phương thức.
- Khi có sự có mặt của thừa kế & đa hình, điều trên cũng phải áp dụng được;
- Do vậy, nếu 1 lớp dẫn xuất override 1 phương thức của lớp cơ sở, nó **không thể** bổ sung các kiểu ngoại lệ mới vào phương thức:
 - Nếu không, ai đó truy nhập phương thức qua 1 con trỏ trỏ tới lớp cơ sở có thể gặp phải 1 ngoại lệ mà họ không mong đợi (do nó không có trong khai báo của lớp cơ sở).
- Tuy nhiên, 1 lớp dẫn xuất được phép giảm bớt số loại ngoại lệ có thể ném.



KHAI BÁO NGOẠI LỆ

- Ví dụ: nếu phiên bản `foo()` của lớp **B** có thể ném ngoại lệ thuộc kiểu `e1 & e2`; phiên bản override của lớp **D** có thể chỉ được ném ngoại lệ thuộc loại `e1`:
 - Không vi phạm quy định của lớp cơ sở **B**.
- Tuy nhiên, **D** sẽ không thể bổ sung kiểu ngoại lệ mới `e3` cho các ngoại lệ mà `foo()` của **D** có thể ném:
 - Do việc đó vi phạm khảng định rằng thể hiện của **D** “là” thể hiện của **B**.



KHAI BÁO NGOẠI LỆ

- Các ràng buộc tương tự cũng áp dụng khi loại đối tượng được ném thuộc về 1 cây thừa kế:
 - Giả sử ta có cây thừa kế thứ hai gồm các lớp ngoại lệ, trong đó **BE** là lớp cơ sở và **DE** là lớp dẫn xuất.
- Nếu phiên bản **foo()** của **B** chỉ ném đối tượng thuộc lớp **DE** (lớp dẫn xuất), thì phiên bản **foo()** của lớp **D** không thể ném thể hiện của lớp **BE** (lớp cơ sở).



CONSTRUCTOR & NGOẠI LỆ

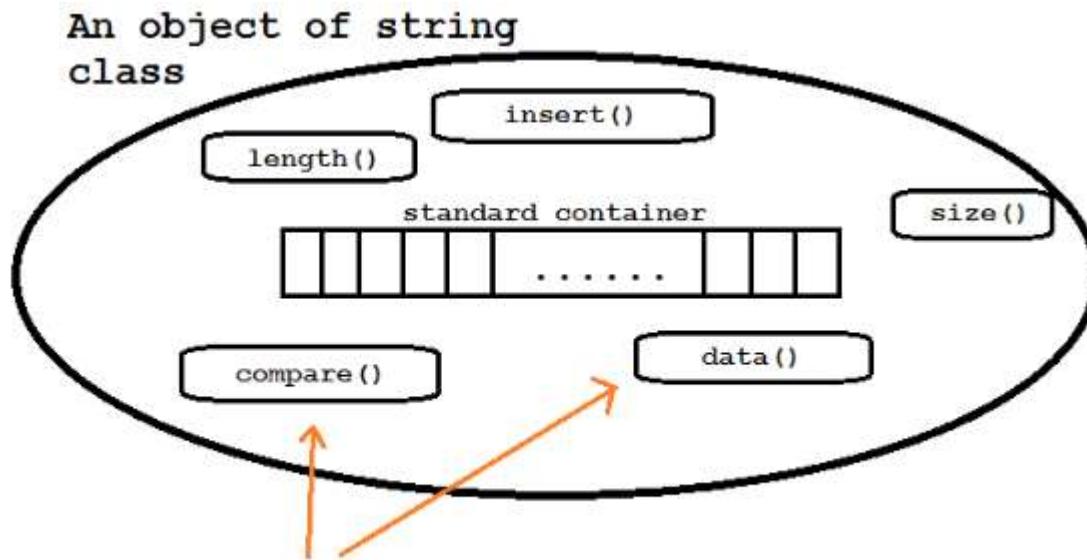
- Cách tốt nhất để thông báo việc khởi tạo không thành công:
 - **Constructor** không có giá trị trả về.
- Cần chú ý để đảm bảo **constructor** không bao giờ để 1 đối tượng ở trạng thái khởi tạo dở:
 - Dọn dẹp trước khi ném ngoại lệ.



DESTRUCTOR & NGOẠI LỆ

- Không nên để ngoại lệ được ném từ destructor;
- Nếu destructor trực tiếp hoặc gián tiếp ném ngoại lệ, chương trình sẽ kết thúc:
 - Hậu quả: chương trình nhiều lỗi có thể kết thúc bất ngờ mà ta không nhìn thấy được nguồn gốc có thể của lỗi.
- Vậy, destructor cần bắt tất cả các ngoại lệ có thể được ném từ các hàm được gọi từ đây.

- C: **cstring**
- C++: **string**
- **string** là một lớp chuẩn mô tả về chuỗi kí tự, nó cung cấp khả năng lưu trữ chuỗi kí tự gọi **là standard container**:
 - **standard container** có thể tự thay đổi kích thước vùng nhớ cho phù hợp với yêu cầu về mặt lưu trữ chuỗi kí tự.



The methods used for manipulating the standard container

Thank You !

