

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (C++)

Giảng viên:

Đặng Hoài Phương

Bộ môn:

Công nghệ phần mềm

Khoa:

Công nghệ Thông tin

Trường Đại học Bách Khoa

Đại học Đà Nẵng





CHƯƠNG 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 Astraction):
 - Không quan tâm đến cấu trúc hàm;
 - Chỉ cần biết kết quả thực hiện của hàm.
- Nền tảng của 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.

- 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.

- 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 trừu tượng hóa đối tượng từ thấp lên cao:
 - Thu thập các thuộc tính của mỗi đối tượng, gán 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 trừu 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.

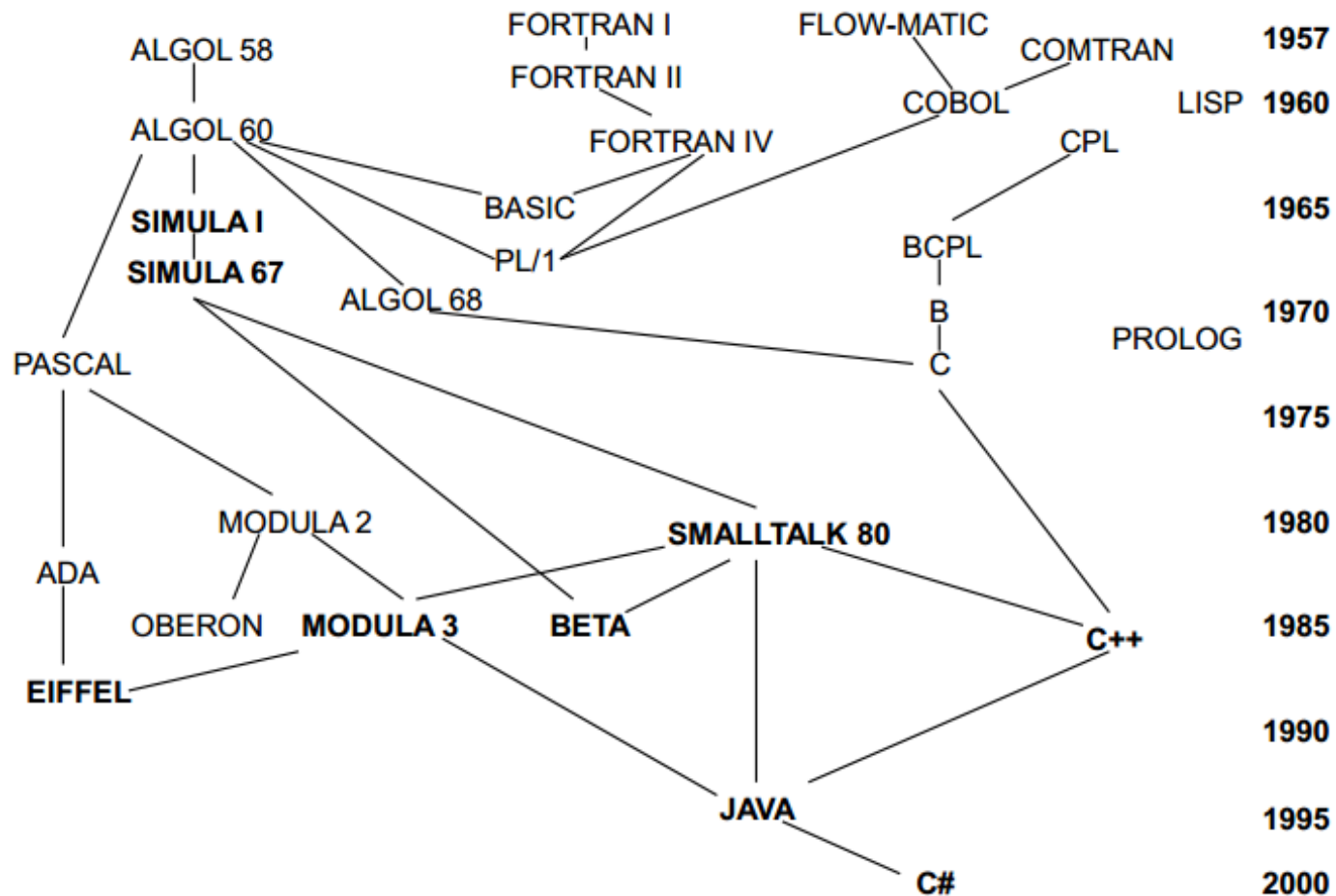


LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- 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 hẫ 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 kết theo cách từ dưới lên (bottom-up).

- 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 (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?**

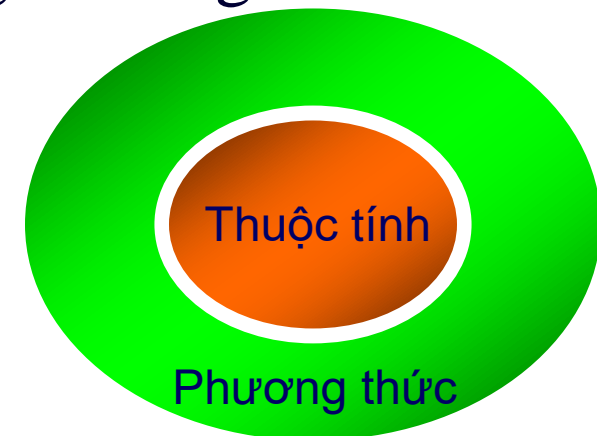


→ Vật liệu mới: **Đối tượng**

→ 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 (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 (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:

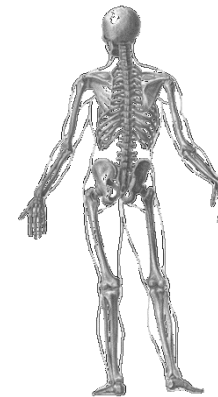
- Name: Thomas.
- Age: 50.
- Hair Color: White.
- Eye Color: Blue.
- Job: Teacher.



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**

- 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):
 - Trừu tượng hóa theo chức năng:
 - Mô hình hóa các phương thức của lớp dựa trên hành động của đối tượng.
 - Trừu 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

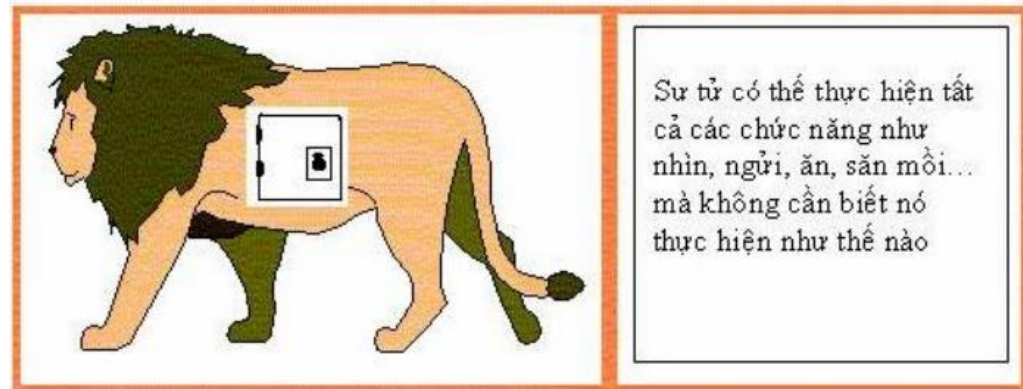
- Thuộc tính (attribute):
 - Là dữ liệu trình bày các đặc điểm về một đối tượng;
 - Bao gồm: Hằng, biến; Tham số nội tại.
 - Kiểu thuộc tính: Kiểu cố định; Kiểu do người dùng định nghĩa.
- Phương thức (method):
 - Có liên quan tới những thứ mà đối tượng có thể làm;
 - Một phương thức đáp ứng một chức 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.



- 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**.



- 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 * Tỷ 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 ()

- 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.

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

Lớp <i>Nhânviên</i>	Lớp <i>Sinhviên</i>
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

- 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 nảy 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.

- 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:

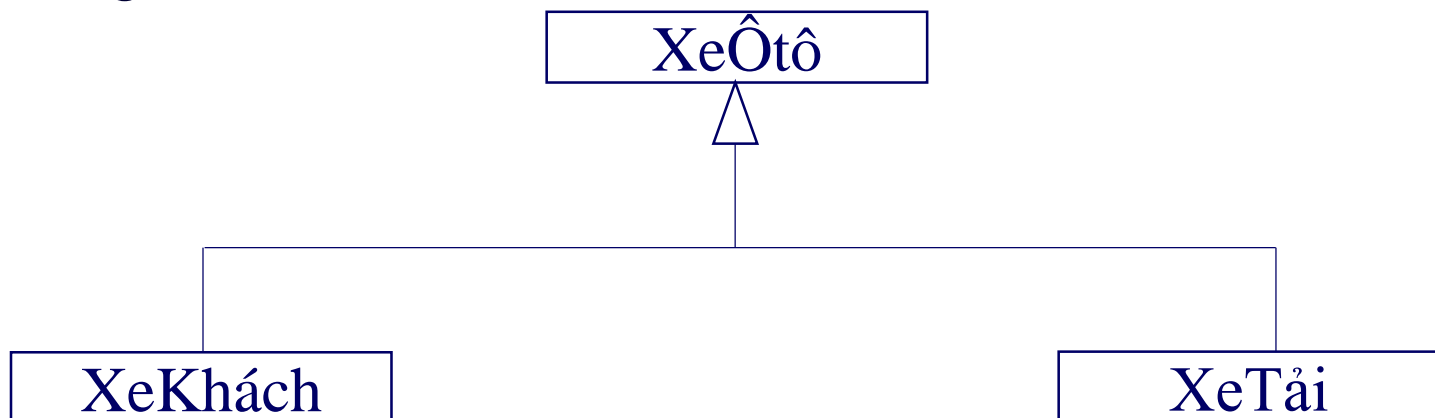
Lớp

Phương thức:

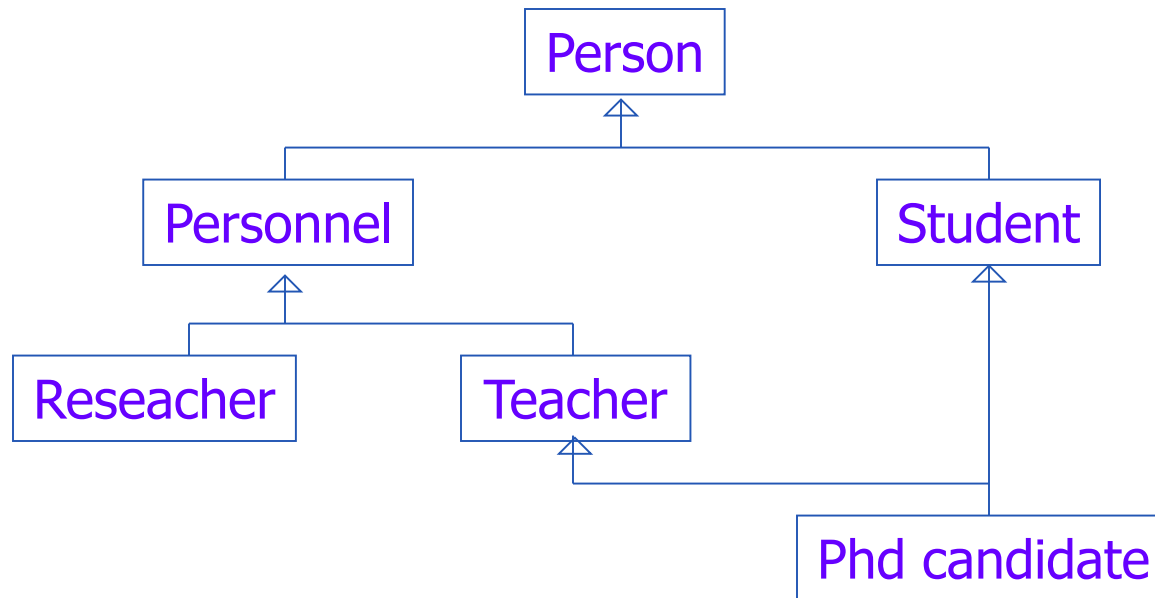
Nhập/Xem lớp

- 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.

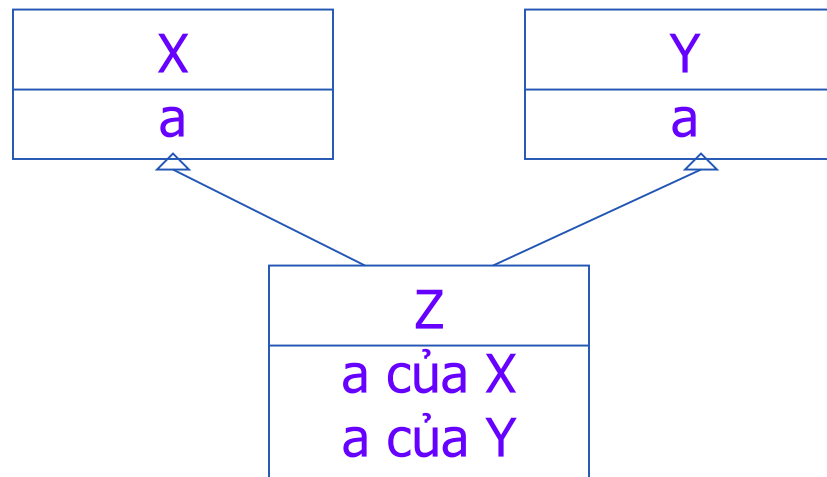
- Tính thừa kế (inheritance):
 - Đơn thừa kế: một lớp con chỉ thừa kế từ một lớp cha duy nhất
 - Ví dụ:
 - Lớp trừu tượng hay lớp chung: XeÔtô;
 - Lớp cụ thể hay lớp chuyên biệt: XeKhách, XeTải;
 - Lớp chuyên biệt có thể thay thế lớp chung trong tất cả các ứng dụng.



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

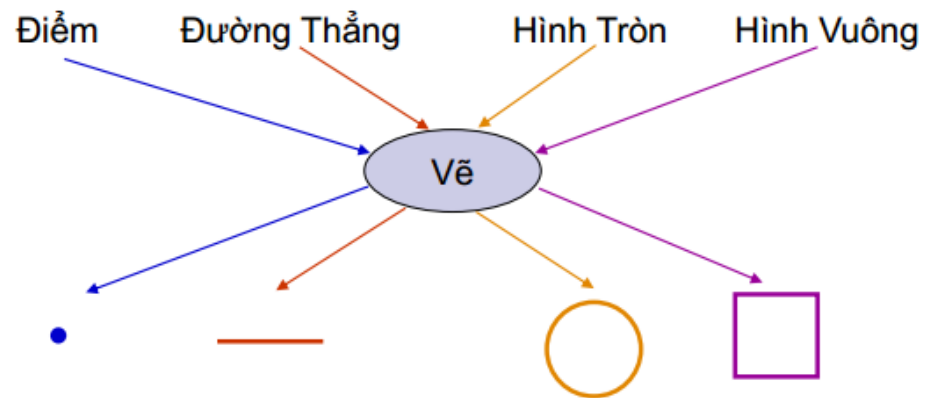


- Tính thừa kế (inheritance):
 - Đa thừa kế:
 - Đụng độ tên các thuộc tính:



- 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.
 - Ngữ 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.



- 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

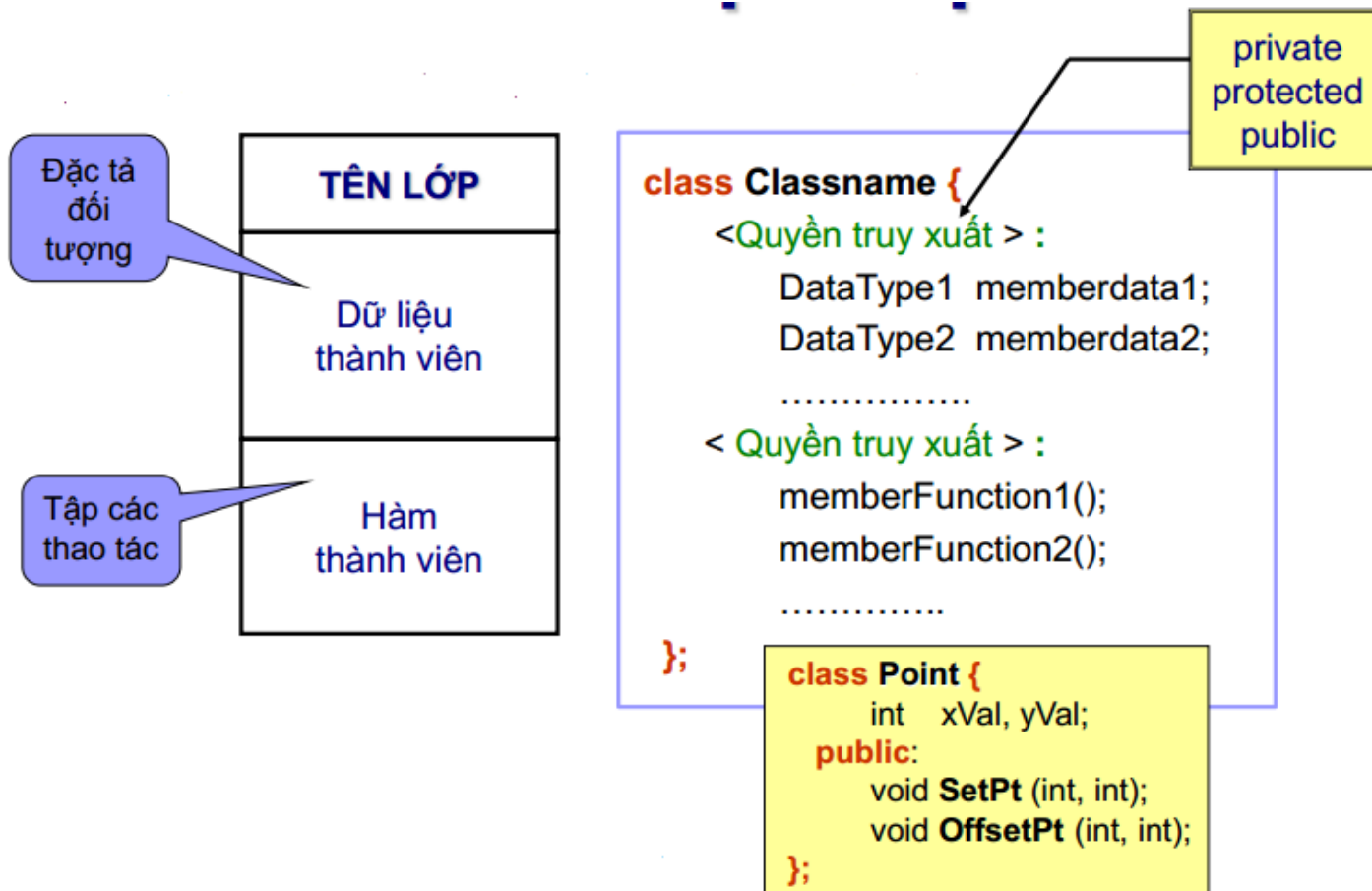
- Tính đa hình (polymorphism):
 - Đa hình hàm - Functional polymorphism
 - Cơ chế cho phép một tên thao tác hoặc thuộc tính có thể được định nghĩa tại nhiều lớp và có thể có nhiều cài đặt khác nhau tại mỗi lớp trong các lớp đó;
 - Ví dụ: lớp Date cài 2 phương thức setDate(), một nhận tham số là một đối 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++)

- **Lớp:** kiểu dữ liệu trừu tượng



- Khai báo lớp: file.h (file trùng tên lớp)

- Point.h

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

- Cài đặt phương thức: file.cpp

- Point.cpp

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


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

Toán tử định phạm vi

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

Tên lớp

Tên phương thức

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

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

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

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

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

```
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 trỏ ***this**

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

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

hoặc

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

- Con trỏ this được các phương thức tự động sử dụng, nên việc ta có sử dụng nó một cách tường minh hay bỏ qua không ảnh hưởng đến tốc độ chạy chương trình;
- Nhiều lập trình viên sử dụng **this** một cách tường minh mỗi khi truy nhập các thành viên dữ liệu:
 - Để đảm bảo không có rắc rối về phạm vi;
 - Ngoài ra, còn để tự nhắc rằng mình đang truy nhập thành viên.

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

```
Point(int xVal, int yVal)
{
    Point::xVal = xVal;
    Point::yVal = yVal;
}
```


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

PhanSo

- Tử số??
- Mẫu số??

HocSinh

- Họ tên??
- Điểm văn??
- Điểm toán??

Hàm dựng ra đời!!



CONSTRUCTOR

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



DEFAULT CONSTRUCTOR

- Hàm dựng mặc định (default constructor):
 - Đối với constructor mặc định, nếu ta không cung cấp một phương thức constructor nào, C++ sẽ tự sinh constructor mặc định là một phương thức rỗng (không làm gì);
 - Mục đích để luôn có một constructor nào đó để gọi khi không có tham số nào
 - Tuy nhiên, nếu ta không định nghĩa constructor mặc định nhưng lại có các constructor khác, trình biên dịch sẽ báo lỗi không tìm thấy constructor mặc định nếu ta không cung cấp tham số khi tạo thể hiện.



DEFAULT CONSTRUCTOR

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

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

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

DEFAULT CONSTRUCTOR

- Hàm dựng mặc định với đối số mặc định

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

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

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



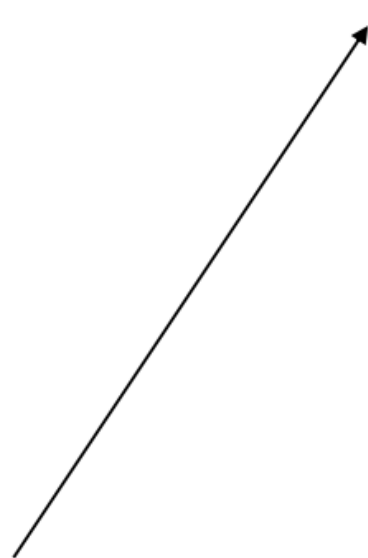
COPY CONSTRUCTOR

- Hàm dựng sao chép (copy constructor):
 - Copy constructor là constructor đặc biệt được gọi khi ta tạo đối tượng mới là bản sao của một đối tượng đã có sẵn
 - `MyClass x(5);`
 - `MyClass y = x;` hoặc `MyClass y(x);`
 - C++ cung cấp sẵn một copy constructor, nó chỉ đơn giản copy từng thành viên dữ liệu từ đối tượng cũ sang đối tượng mới;
 - Tuy nhiên, trong nhiều trường hợp, ta cần thực hiện các công việc Khởi tạo khác trong copy constructor → có thể định nghĩa lại copy constructor.

- Hàm dựng sao chép (copy constructor):

- Ví dụ:

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



Kiểu tham số là tham chiếu đến đối tượng kiểu Foo



tham số là đối tượng được sao chép

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



COPY CONSTRUCTOR

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

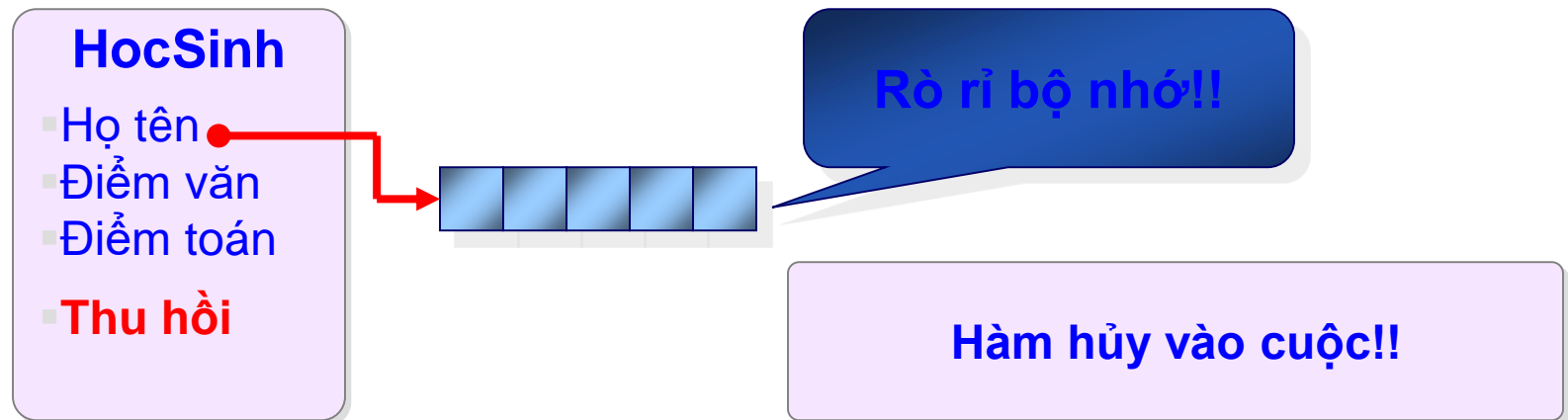
```
//Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;
Point::Point(int x, int y)
{
    this->xVal = x;
    this->yVal = y;
}
Point::Point(const Point &p)
{
    this->xVal = p.xVal;
    this->yVal = p.yVal;
}
Point::~~Point() { }
void Point::Show()
{
    cout << this->xVal
     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

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

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



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



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

- Tính chất hàm hủy (destructor):
 - Tự động gọi khi đối tượng bị hủy.
 - Mỗi lớp có duy nhất một hàm hủy.
 - Trong C++, hàm hủy có tên **~<Tên lớp>**.

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

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


- Ví dụ:

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

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

Tổng cộng
có bao
nhiều lần
hàm hủy
được gọi?

Tập Các
Số Nguyên

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

Tập Các
Số Thực

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

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

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



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

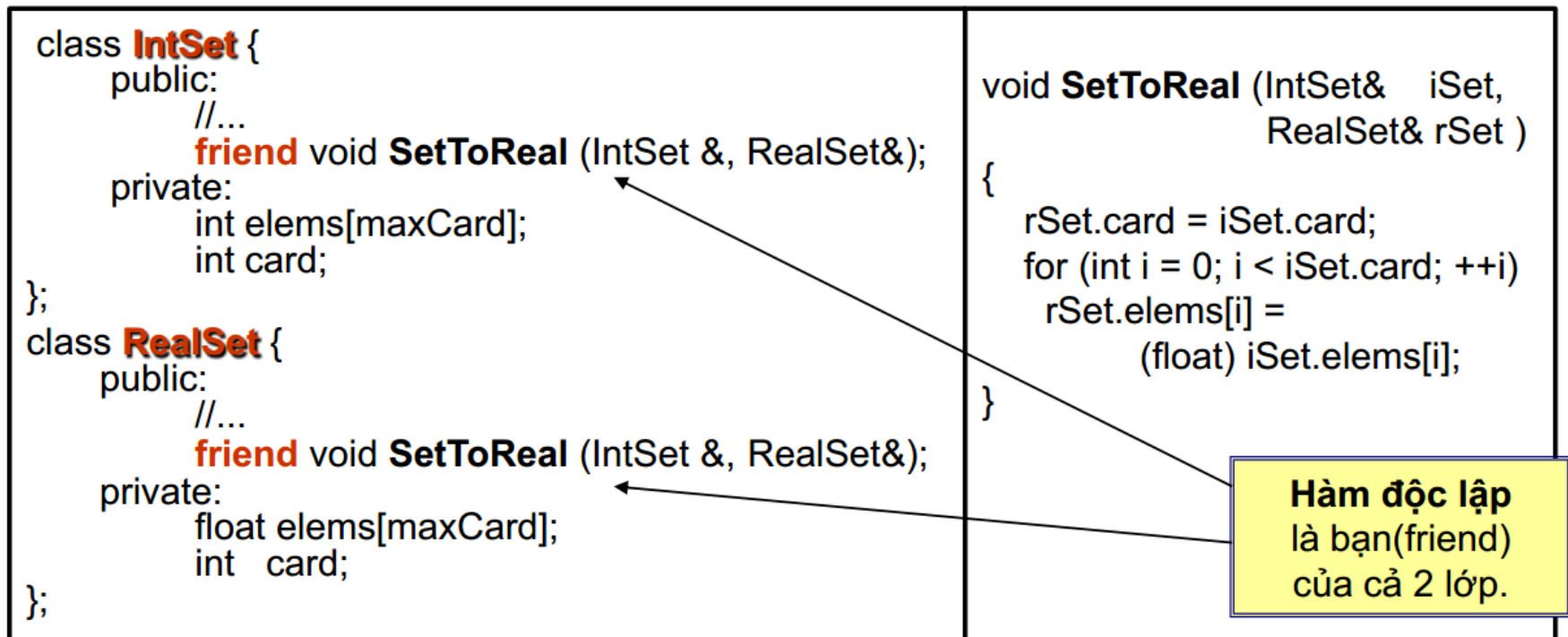
- Cách 1: Khai báo hàm thành viên của lớp IntSet là bạn (friend) của lớp RealSet.

Giữ nguyên định nghĩa của lớp IntSet

Thêm dòng khai báo **Friend** cho hàm thành viên **SetToReal**

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

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



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

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

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


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

```
class RealSet
{
    public:
        friend void IntSet::SetToReal (RealSet&) ;
    private:
        //...
};
```
 - Khi xử lý, trình biên dịch cần phải biết là đã có lớp IntSet;
 - Tuy nhiên các phương thức của IntSet lại dùng đến RealSet nên phải có lớp RealSet trước khi định nghĩa IntSet.
- Cho nên ta không thể tạo IntSet khi chưa tạo RealSet và không thể tạo RealSet khi chưa tạo IntSet.



FRIEND – KHAI BÁO FORWARD

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

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

FRIEND – KHAI BÁO FORWARD

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

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

Trình biên dịch chưa biết **SetToReal**

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

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

- Bài tập:
 - Không sử dụng hàm bạn

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

- Bài tập:
 - Sử dụng hàm bạn

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

- Bài tập:
 - Sử dụng hàm bạn

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


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

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

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

Image::Image(const int w, const int h)
{
    width = w;
    height = h;
}
```

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

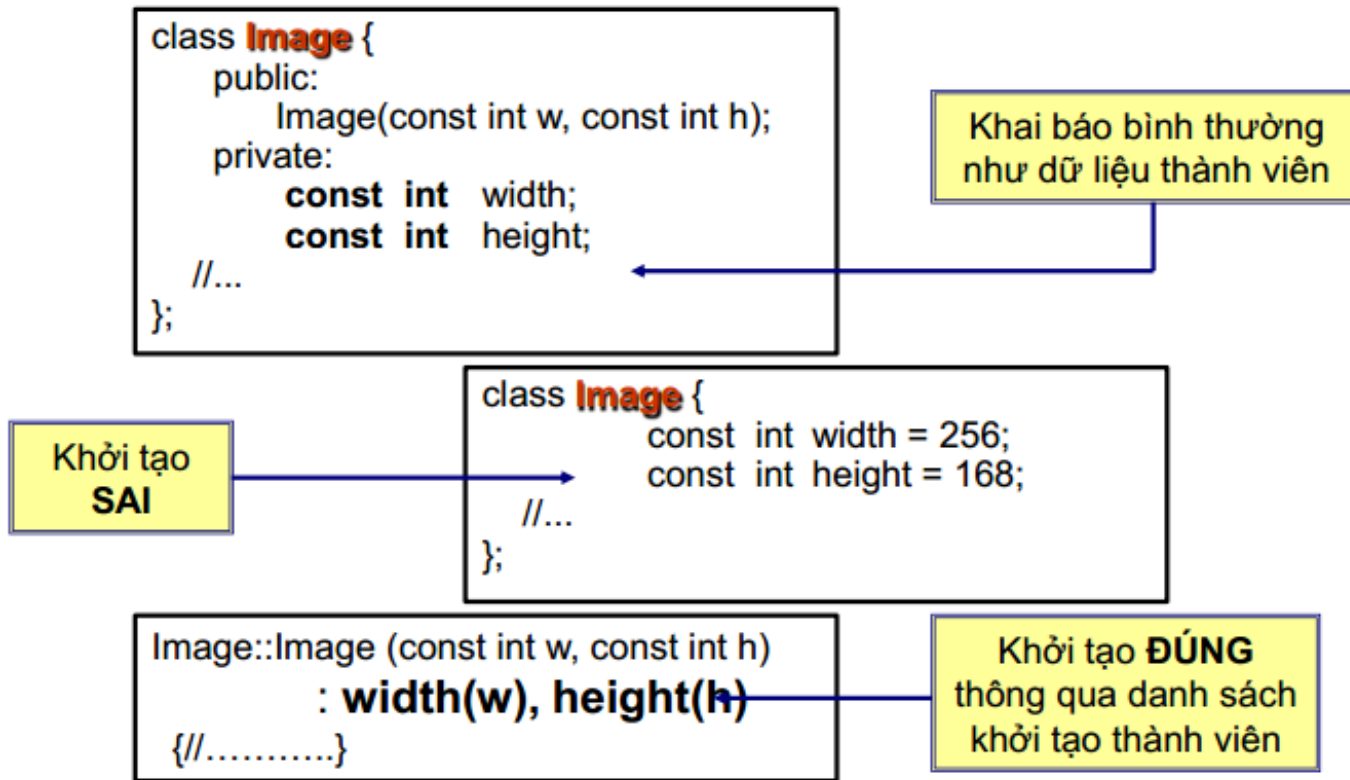
```
class Point {  
    int  xVal, yVal;  
public:  
    Point (int x, int y) {  
        xVal = x;  
        yVal = y;  
    }  
    // .....  
};
```

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

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

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

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



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

- Ví dụ:

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

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


- Ví dụ:

```
inline char *strdup(const char *s)
{   return strcpy(new char[strlen(s) + 1], s); }

class string
{
    char *p;
public: string(char *s = "") {p = strdup(s);}
    ~string() {delete [] p;}
    string(const string &s2) {p = strdup(s2.p);}
    void Output() const {cout << p;}
    void ToLower() {strlwr(p);}
};
```

- Ví dụ:

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

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

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

Khai báo

Khởi tạo
dữ liệu
thành viên
tĩnh

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

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

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

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

- ❖ Khởi tạo biến đếm bằng 0 vì ban đầu không có đối tượng nào.

- Định nghĩa & Khởi tạo:
 - Thành viên tĩnh được lưu trữ độc lập với các thể hiện của lớp. Do đó, các thành viên tĩnh phải được định nghĩa:


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

- Ta thường định nghĩa các thành viên tĩnh trong file chứa định nghĩa các phương thức;
- Nếu muốn khởi tạo giá trị cho thành viên tĩnh ta cho giá trị khởi tạo tại định nghĩa:

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


- Ví dụ:

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



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

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

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

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

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

Tóm lại, ta nên khai báo:

- **static:**
 - Đối với các thành viên dữ liệu ta muốn dùng chung cho mọi thể hiện (đối tượng) của một lớp.
- **const:**
 - Đối với các thành viên dữ liệu cần giữ nguyên giá trị trong suốt thời gian sống của một thể hiện.
- **static const:**
 - Đối với các thành viên dữ liệu cần giữ nguyên cùng một giá trị tại tất cả các đối tượng của một lớp.

HÀM THÀNH VIÊN TĨNH

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

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

Khai báo
Định nghĩa
hàm thành
viên tĩnh

Truy xuất
hàm thành
viên tĩnh

- Ví dụ:

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

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

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

- Ví dụ:

```
typedef int bool;
const bool false = 0, true = 1;
class CDate
{
    static int dayTab[13];
    int day, month, year;
public:
    CDate(int d=1, int m=1, int y=2010);
    static bool LeapYear(int y)
    {return y%400 == 0 || y%4==0 && y%100 != 0;}
    static int DayOfMonth(int m, int y);
    static bool ValidDate(int d, int m, int y);
    void Input();
};
int CDate::dayTab[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
CDate::CDate(int d=1, int m=1, int y=2010)
{    if (ValidDate(d,m,y)) {day=d;month=m;year=y;}    }
```


- Ví dụ:

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

THÀNH VIÊN THAM CHIẾU

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

Khai báo bình thường
như dữ liệu thành viên

Khởi tạo
SAI

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

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

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

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

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

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

- Ví dụ:

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

- Ví dụ:

```
class Diem
{
    double x,y;
public:
    Diem(double xx, double yy) {x = xx; y = yy;}
    //...
};
class TamGiac
{
    Diem A,B,C;
public:
    TamGiac(double xA, double yA, double xB, double yB,
double xC, double yC) :A(xA,yA) , (xB,yB) ,C(xC,yC) {}
    void Ve() const;
    //...
};
TamGiac t(100,100,200,400,300,300);
```

- Sử dụng hàm xây dựng không đối số (hàm xây dựng mặc nhiên - default constructor).
 - Ví dụ: `Point pentagon[5];`
- Sử dụng bộ khởi tạo mảng:
 - Ví dụ:
`Point triangle[3] = { Point(4,8), Point(10,20), Point(35,15) };`
 - Ngắn gọn:
`Set s[4] = { 10, 20, 30, 40 };`
 - tương đương với:
`Set s[4] = { Set(10), Set(20), Set(30), Set(40) };`

- Sử dụng dạng con trỏ:

- Cấp vùng nhớ:

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

- Thu hồi vùng nhớ:

`delete[] pentagon;`

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

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

Không cần biết kích
thước mảng.

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

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

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

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

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

```
class Rectangle { // Lớp lồng nhau
public:
    Rectangle (int, int, int, int);
    //..
private:
    class Point {
    public:
        Point(int a, int b) { ... }
    private:
        int x, y;
    };
    Point topLeft, botRight;
};
Rectangle::Point pt(1,1); // sd ở ngoài
```

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

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

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



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

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

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

- Struct:

- Giống như C:

```
struct Tên_kiểu_ct
```

```
{
```

```
    // Khai báo các thành phần của cấu trúc
```

```
};
```

- Khai báo biến (struct):

- C: **struct Tên_kiểu_ct** danh sách biến, mảng cấu trúc;
- C++: **Tên_kiểu_ct** danh sách biến, mảng cấu trúc;

- Struct:

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

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


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

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

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

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

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

- Union:
 - Giống như C:

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

- Khai báo biến (struct):
 - C: **union Tên_kiểu_hợp** danh sách biến, mảng kiểu **hợp**;
 - C++: **Tên_kiểu_ct** danh sách biến, mảng kiểu **hợp**;

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

union

{

// Khai báo các thành phần

} ;

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

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

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

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

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

Thank You !

