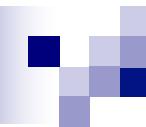


CHƯƠNG 6:

Đa hình (Polymorphism)

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



Nội dung

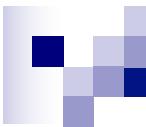
- Đa hình là gì?
- Đa hình hàm – function polymorphism
- Đa hình và Hướng đối tượng
- So sánh Overloading và Overriding
- Method Overidding
- Đa hình tĩnh – static polymorphism là gì?
- Liên kết động – dynamic binding
- Đa hình động – dynamic polymorphism
- Hàm ảo – virtual function
- Destructor ảo
- Lớp trừu tượng – Abstract class

Đa hình là gì?

- “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
- Có 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 và Hướng đối tượng

- Trong phạm vi 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 dẫn xuất
- Còn nhớ: 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 và 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.



chú ý rằng kangaroo và cóc thuộc hai nhánh trong cây phả hệ động vật

Overloading vs. Overriding

- Function overloading - Hàm chồng: dùng *một* tên hàm cho nhiều định nghĩa hàm, khác nhau ở danh sách tham số
- Method overloading – Phương thức chồng: 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 vs. Overriding

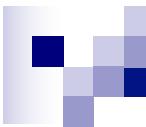
- Đa hình được cài đặt bởi một khái niệm tương tự nhưng hơi khác: method overriding
 - “override” có nghĩa “vượt quyền”
- Method overriding: nếu một phương thức của lớp cơ sở được định nghĩa lại tại lớp dẫn xuất thì định nghĩa tại lớp cơ sở có thể bị “che” bởi định nghĩa tại lớp dẫn xuất.
- Với method 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

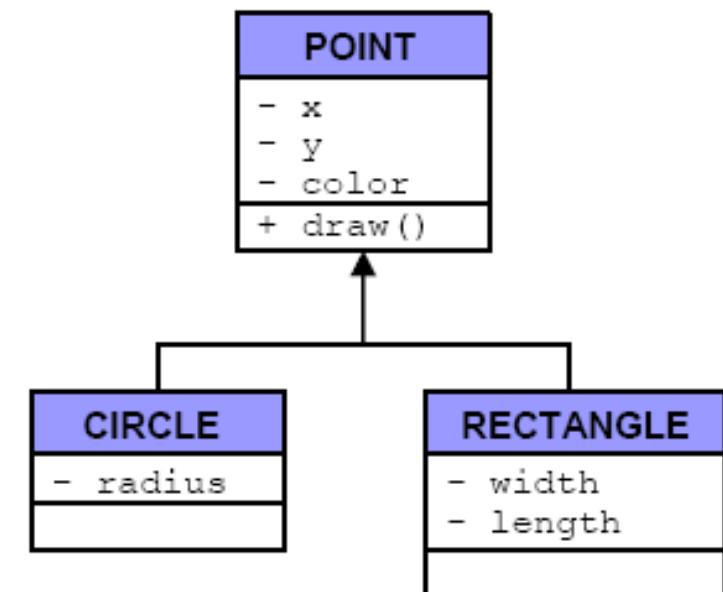
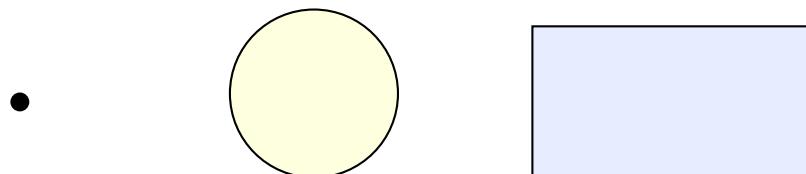
Chú ý, Kangaroo và Frog đều là các lớp
dẫn xuất từ lớp cơ sở gián tiếp Animal



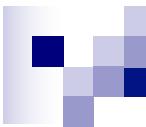
Method Overriding

Xét hành vi “draw” của các lớp trong cây phả hệ bên.

- thông điệp “draw” gửi cho một thể hiện của mỗi lớp trên sẽ yêu cầu thể hiện đó tự vẽ chính nó.



- một thể hiện của Point phải vẽ một điểm, một thể hiện của Circle phải vẽ một đường tròn, và một thể hiện của Rectangle phải vẽ một hình chữ nhật



Method Overriding

- 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  
}
```

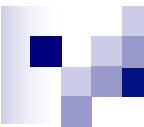
```
class Circle : public Point {  
public:  
    Circle (int x, int y,  
            string color,int radius);  
    ~Circle();  
    void draw();  
private:  
    int radius;
```

```
}
```

1. khai báo lại tại lớp dẫn xuất

2. khai báo lại tại lớp dẫn xuất

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



Method Overriding

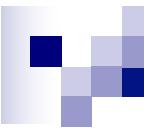
- Kết quả: khi tạo các thể hiện của các lớp khác nhau và gửi thông điệp “draw”, các phương thức thích hợp sẽ được gọi.

```
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 Overriding

■ 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);  
    ...  
}
```

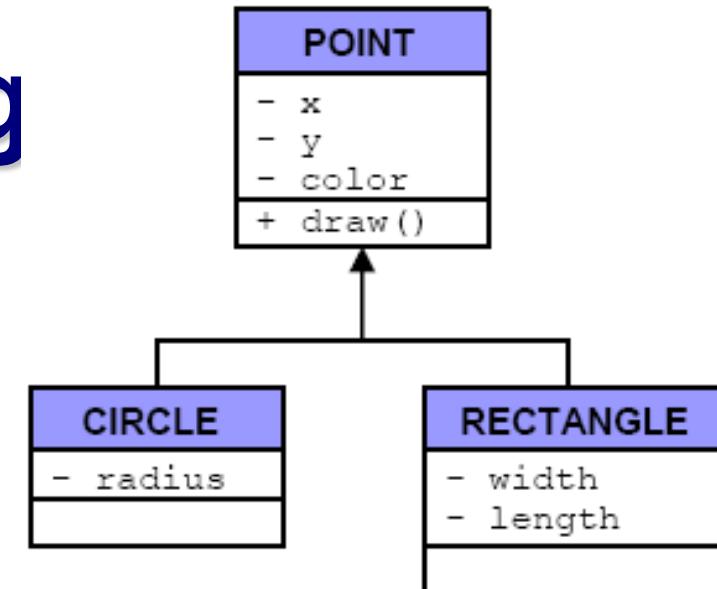
B override **foo()**, không override **foo(x)**
⇒ tại B, **foo(x)** bị che, nếu gọi **foo(x)** từ
một đối tượng B sẽ gây lỗi biên dịch

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

Method Overriding

- 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);
```



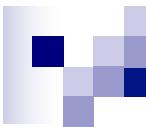
```
p.draw(); //Draws a white point at (0,0)  
c.draw(); //Draws a blue circle of radius 50  
at(100,100)
```

- 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;
```

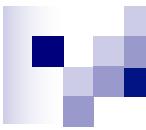
- Nhưng nếu có đa hình, lời gọi sau sẽ làm gì?

```
pp->draw(); // Draw what???
```



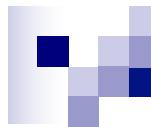
Function call binding (Liên kết lời gọi hàm)

- 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

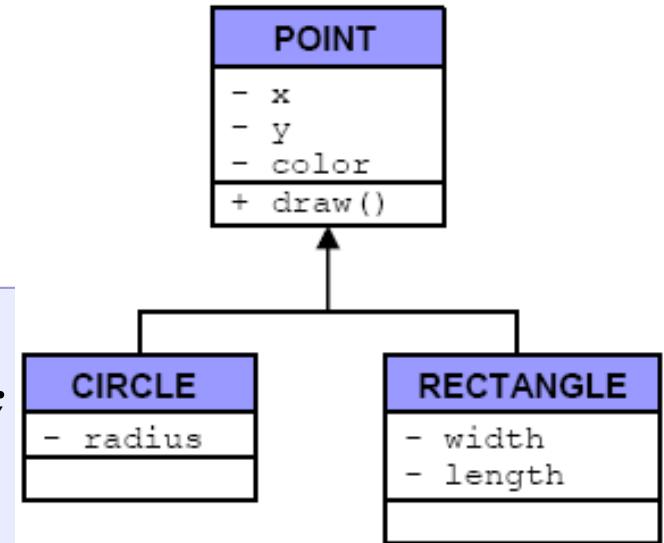


Function call binding (Liên kết lời gọi hàm)

- **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



Static binding



```
Point P(10,20,"brown");
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
```

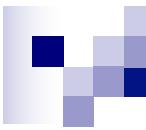
```
Circle* pC = &C;
pC->draw(); //hình tròn
```

Kiểu tĩnh :
Point

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

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

Ta muốn
in
hình tròn



Static binding

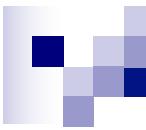
- Ta muốn:

`rP.draw()` và `pP->draw()`

sẽ gọi `Point::draw()` hay `Circle::draw()`

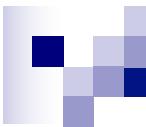
tùy theo `rP` và `pP` đang trỏ tới đối tượng `Point` hay `Circle`

- 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()`.



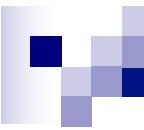
Static binding

- 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



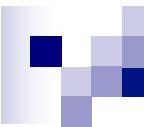
Static polymorphism

- 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, hoặc
 - 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**

```
Cirlce 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**

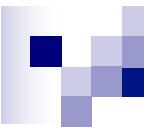
Dynamic Binding – liên kết động

- 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 ảo

- **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 khoá **virtual**) tại lớp cơ sở

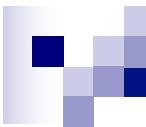
Hàm ảo

```
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 về Circle, nên
Circle::draw() được gọi

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



Hàm ảo

- 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

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

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

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

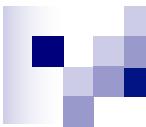
Không cần
nhưng nên có

Hàm ảo

- 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:
 1. 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
 2. 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.

Hàm ảo – constructor và destructor

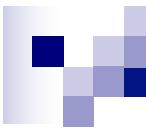
- Không thể khai báo các constructor ảo
 - constructor không được thừa kế, bao giờ cũng phải định nghĩa lại nên chuyện ảo không có nghĩa
- Có thể (và rất nên) khai báo destructor là hàm ảo.



Destructor ảo

- Nhớ lại 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

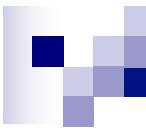
```
Car* c = new Car(10, "Suzuki", "RSX-R1000", 4);  
...  
delete c; // This works correctly - it will trigger  
           // the Car destructor and then works  
           // up to the MotorVehicle destructor
```



Destructor ảo

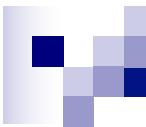
- Còn nếu ta dùng một con trỏ tới MotorVehicle thay cho con trỏ tới Car, chỉ có destructor của MotorVehicle được gọi.

```
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
```



Destructor ảo

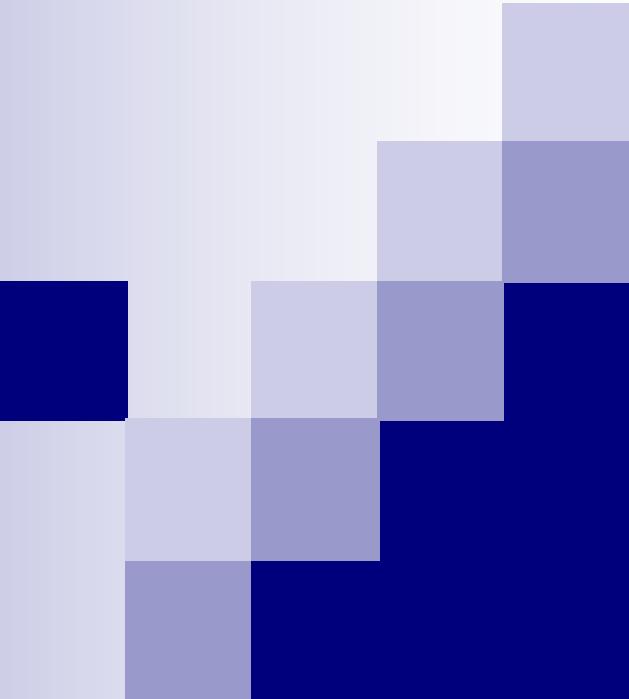
- Chú ý: việc gọi nhầm destructor *không ảnh hưởng* đến việc thu hồi bộ nhớ
 - trong mọi trường hợp, phần bộ nhớ của đối tượng sẽ được thu hồi chính xác
 - Trong ví dụ trước, kể cả nếu chỉ có destructor của **MotorVehicle** được gọi, phần bộ nhớ của toàn bộ đối tượng **Car** vẫn được thu hồi
- Tuy nhiên, 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



Destructor ảo

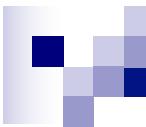
- Quy tắc chung: mỗi khi tạo một lớp để được dùng làm lớp cơ sở, ta nên khai báo destructor là hàm ảo.
 - kể cả khi destructor của lớp cơ sở rỗng (không làm gì)
 - Vậy ta sẽ sửa lại lớp **MotorVehicle** như sau:

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



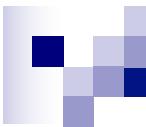
Lớp trừu tượng

- Abstract class



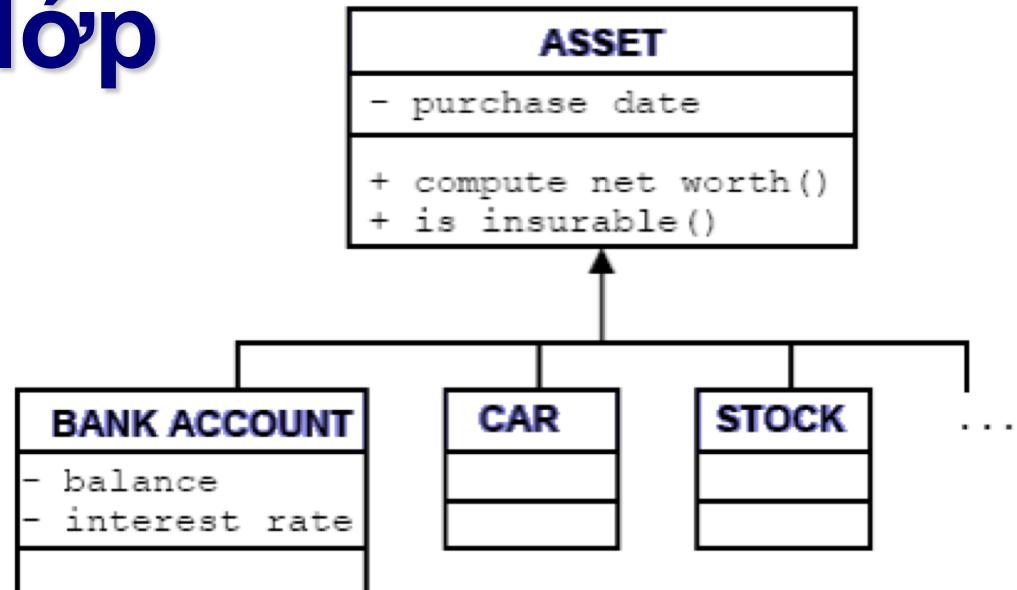
Tạo thể hiện của lớp

- 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



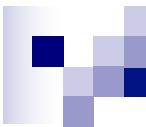
Tạo thể hiện của lớp

- 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), v.v..
 - 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



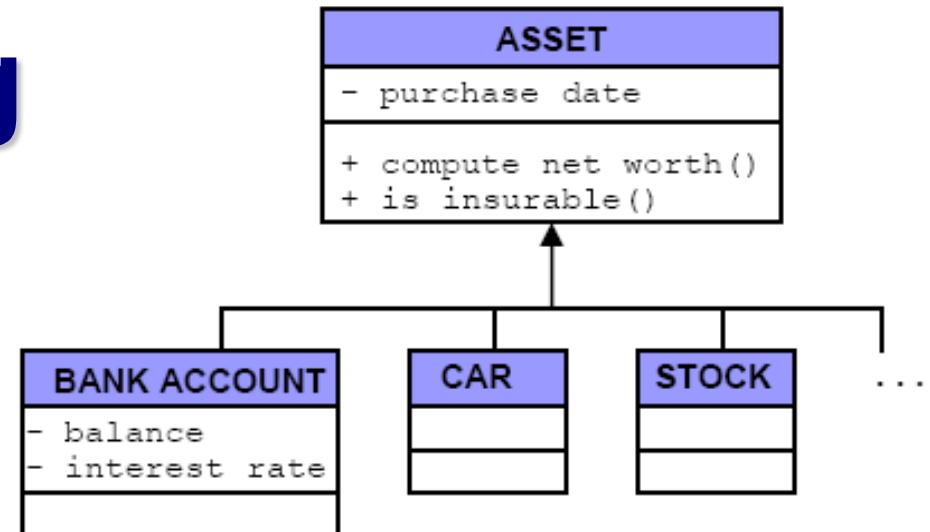
Lớp trừu tượng – abstract class

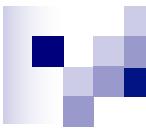
- Chúng ta có thể tạo ra các lớp cơ sở để tái sử dụng mà không muốn tạo ra đối tượng thực của lớp
 - các lớp Point, Circle, Rectangle chung nhau khái niệm cùng là hình vẽ *Shape*
- lớp trừu tượng (**Abstract Base Class – ABC**) là một lớp không thể tạo thể hiện
- Thực tế, ta thường phân nhóm các đối 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?



Lớp trừu tượng

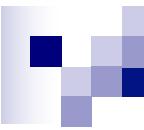
- Giả sử Asset là lớp trừu tượng, nó có các ích lợi gì?
- Mọi thuộc tính và hành vi của lớp cơ sở (Asset) có mặt trong mỗi thể hiện của các lớp dẫn xuất (BankAccount, Car, Stock,...)
- Ta vẫn có thể nói về quan hệ anh chị em giữa các thể hiện của các lớp dẫn xuất qua lớp cơ sở.
- nói về một cái ngân 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





Lớp trùu tượng

- Các lớp trùu tượng chỉ có ích khi chúng là các lớp cơ sở trong cây thừa kế
 - Ví dụ, nếu ta cho BankAccount là lớp trùu tượng, và nó không có lớp dẫn xuất nào, vậy nó để làm gì?
- Các lớp cơ sở trùu tượng có thể được đặt tại các tầng khác nhau của một cây thừa kế.
 - có thể coi BankAccount là lớp trùu 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 trùu tượng phải có ít nhất một lớp dẫn xuất không trùu tượng

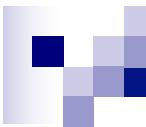


Cài đặt lớp trừu tượng

- 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
Ví dụ: `virtual void MyMethod() = 0;`
 - không cần định nghĩa phương thức
 - nếu không có “ = 0” và không định nghĩa, trình biên dịch sẽ báo lỗi

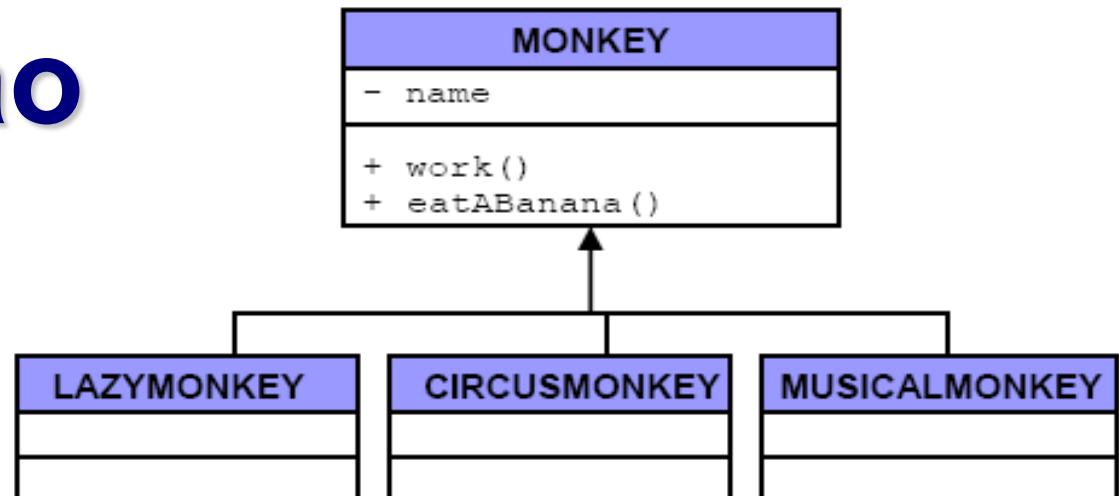
Hàm thuần ảo

- 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 trừu tượng
 - các lớp dẫn xuất của lớp đó nếu cũng không cung cấp định nghĩa cho phương thức đó thì cũng trở thành lớp trừu 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 trừu tượng có thể “bắt buộc” các lớp con phải cài đặt các phương thức đó



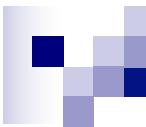
Hàm thuần ảo

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



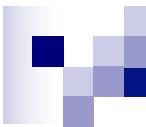
```
class Monkey{
public:
    Monkey(...);
    virtual ~Monkey();
    virtual void work()=0; //pthúc thuần ảo
    void eatABanana();
    ...
};
```

- 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 tương 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:

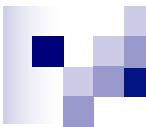
```
void Monkey::work()
{
    cout << "No." << endl;
}
```

- 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();
}
```

Nhận xét

- 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



Tóm tắt

■ Hàm ảo:

- Chuyển lời gọi hàm đến đúng đối tượng.
- Chỉ có ý nghĩa khi gọi từ con trỏ.

■ Hàm thuần ảo:

- Hàm ảo chỉ có khai báo mà không có cài đặt.
- Lớp kế thừa đảm nhận việc cài đặt.
- Lớp có chứa hàm thuần ảo → lớp trừu tượng
- Lớp trừu tượng chỉ dùng để kế thừa.

■ Hàm hủy ảo:

- Hàm hủy phải luôn luôn là hàm ảo.

Bài tập

■ Bài tập 6.1:

Có 2 loại hình:

- Hình tam giác: biểu diễn bởi 3 đỉnh.
- Hình chữ nhật: biểu diễn bởi 2 điểm trên trái và dưới phải.

Giả sử có sẵn một danh sách các hình. xuất thông tin của từng hình trong danh sách đó.

Sau đó, giả sử có thêm loại hình mới là hình tròn.

- Hình tròn: biểu diễn bởi tâm và bán kính.

Khi đó, chương trình sẽ phải được chỉnh sửa như thế nào?

Bài tập

■ Bài tập 6.2:

Một công ty kinh doanh bất động sản sở hữu N miếng đất. Mỗi miếng đất có những thông tin sau:

- Mã số (ví dụ: MD001).
- Đơn giá một m².

Ngoài ra, mỗi miếng đất tùy theo hình dạng còn có những thông tin riêng.

Hiện có 3 loại hình dạng:

- Hình chữ nhật: diện tích = dài x rộng.
- Hình thang: diện tích = (đáy lớn + đáy nhỏ) * chiều cao / 2.
- Hình tam giác: diện tích = đáy * chiều cao / 2.

Những miếng đất hình thang và hình tam giác được công ty giảm giá 10%.

Viết chương trình:

- Nhập danh sách những miếng đất của công ty.
- Tính tổng diện tích các miếng đất.
- Tính tổng giá tiền của các miếng đất.