Chương 6: Đa hình

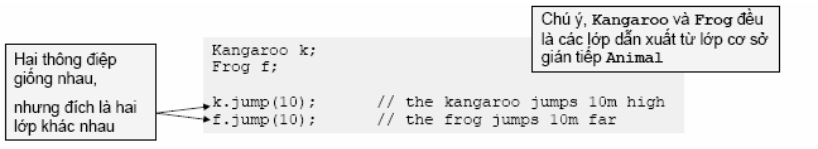
Cao Tuấn Dũng
Huỳnh Quyết Thắng
Bộ môn CNPM

Khái niệm đa hình

- *"Trong lập trình đối tượng đa hình là khả năng các đối tượng khác nhau có thể trả lời cùng một thông điệp theo cách riêng của chúng"*
- Khả năng của một đối tượng của các lớp khác nhau có thể đáp ứng thực hiện các hàm khác nhau của các lớp khác nhau cho cùng một giao diện gọi hàm

Overloading và Overriding

- Đa hình liên quan đến một khái niệm: method overriding (định nghĩa lại phương thức)
  - "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.



Overloading vs Overriding

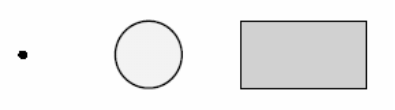
- Function overloading - Hàm chồng: dùng *một* tên hàm cho nhiều định nghĩa hàm, khác nhau ở danh sách tham số
- Method overloading – Phương thức chồng: tương tự

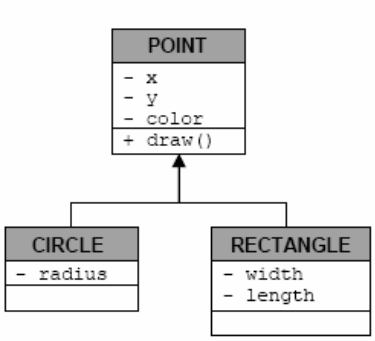
```
void jump(int howHigh);
void jump(int howHigh, int howFar);
```

  - 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*.

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

Định nghĩa lại phương thức

- 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);
...
```

```
class B:public A() {
   void foo();
   //no 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

Định nghĩa lại hành vi Draw (C++)

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

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

Khai báo lại tại lớp kế thừa

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

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

Cung cấp định nghĩa mới

Định nghĩa lại Draw()

- 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(); // Vẽ điểm Trắng tại (0,0)
c.draw(); // Vẽ hình tròn xanh dương bán kính 50 tại (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

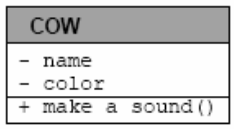
```
Circle *pc = new Circle(100,100,"blue",50);
Point* pp = pc;
```

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

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

Ví dụ về đa hình: Bò điên

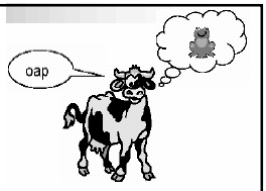
Một con bò – Kêu Moo



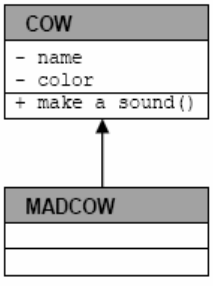

```
class Cow {
public:
   Cow(string name, string color);
   ~Cow();
   void makeASound();
private:
   string name;
   string color;
};
```

```
...
void Cow::makeASound();
{
   cout << "Moo" << endl;
}
```

Bò điên



- Bò điên – cũng là bò: nhưng tưởng mình là ếch kêu oap



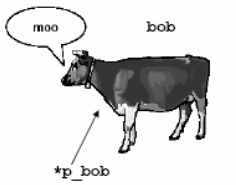

```
class MadCow : public Cow {
public:
   MadCow(string name, string color);
   ~MadCow();

   void makeASound();
};
```

```
...
void MadCow::makeASound();
{
   cout << "oap" << endl;
}
```

Bò điên

- Như vậy, nếu tạo một đối tượng **`Cow`**, và gọi hành vi **`makeASound()`** của nó, nó sẽ kêu "moo"

```
Cow bob("Bob", "brown");
bob.makeASound();
```



- Ta còn có thể tạo một con trỏ tới đối tượng **`Cow`**, và làm cho nó kêu "moo"

```
p_bob = &bob;
p_bob->makeASound();
```


Nhập nhằng

- Nếu coi nó là bò điên (**MadCow**), nó sẽ hành động đúng là bò điên

```
MadCow maddy("Maddy", "white");
maddy.makeASound(); // Go "oap"
MadCow * maddy = new MadCow("Maddy", "white");
maddy->makeASound(); // Go "oap"
```

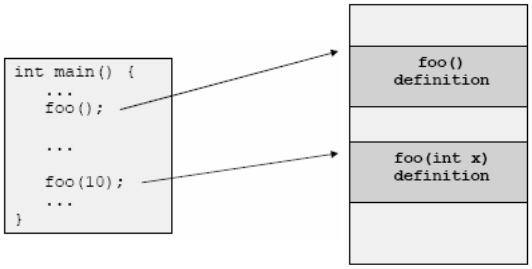
- Còn khi tưởng nó chỉ là bò thường (Cow), nó lại có vẻ bình thường

```
MadCow* maddy = new madCow("Maddy", "white");
Cow* cow = maddy; // Upcasting
cow->makeASound(); // Go "moo" ???
```

- Làm thế nào để bò điên lúc nào cũng điên (kêu "oap")?

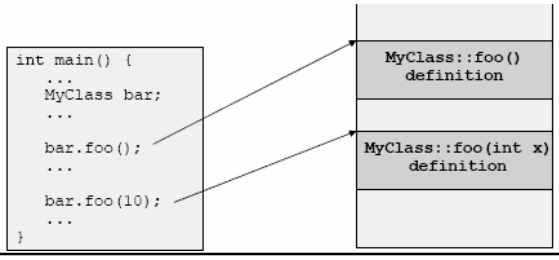
Liên kết lời gọi hàm function call binding

- Function call binding là quy trình xác định khối mã hàm cần chạy khi một lời gọi hàm được thực hiện
- C: đơn giản vì mỗi hàm có duy nhất một tên
- C++: chồng hàm, phân tích chữ ký kiểm tra danh sách tham số.



Ngôn ngữ HĐT method call binding

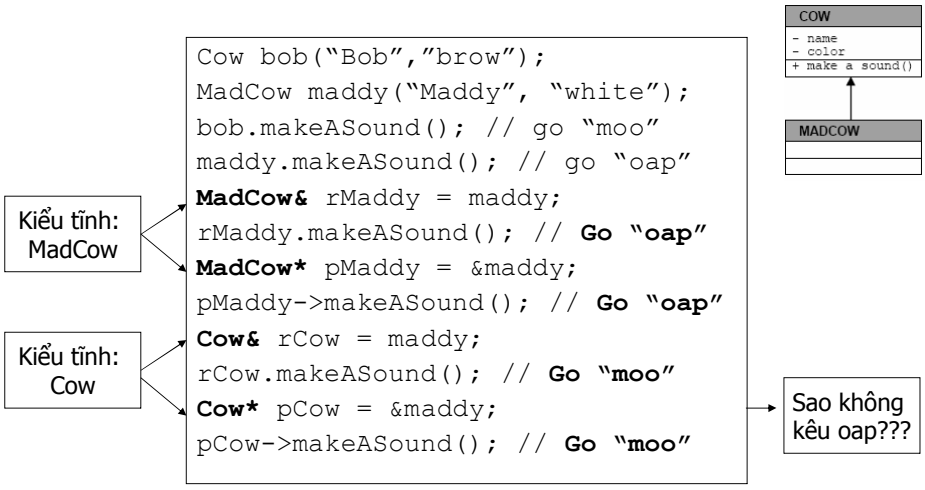
- Liên kết lời gọi phương thức
- Đối với các lớp độc lập (không thuộc cây thừa kế nào), quy trình này gần như không khác với function call binding
  - so sánh tên phương thức, danh sách tham số để tìm định nghĩa tương ứng
  - một trong số các tham số là tham số ẩn: con trỏ **this**



Liên kết tĩnh

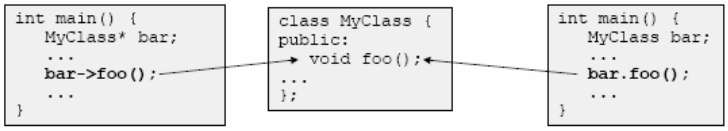
- C/C++ function call binding, và C++ method binding cơ bản đều là ví dụ của static function call binding
- Static function call binding (hoặc static binding – liên kết tĩnh) là quy trình liên kết một lời gọi hàm với một định nghĩa hàm tại *thời điểm biên dịch*.
  - do đó còn gọi là "compile-time binding" – liên kết khi biên dịch, hoặc "early binding" – liên kết sớm

Liên kết tĩnh: trường hợp có cây thừa kế



Liên kết tĩnh

- 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

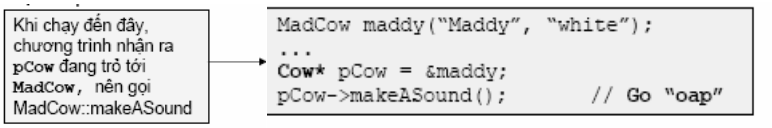


Đa hình tĩnh

- 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)

Đa hình 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.

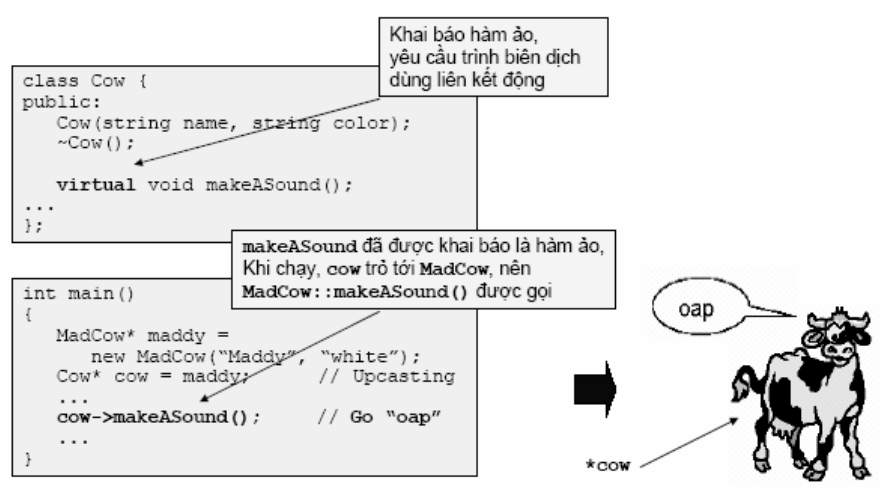


- *Đa hình động (dynamic polymorphisme) là loại đa hình cài đặt bằng liên kết động*

Hàm ảo

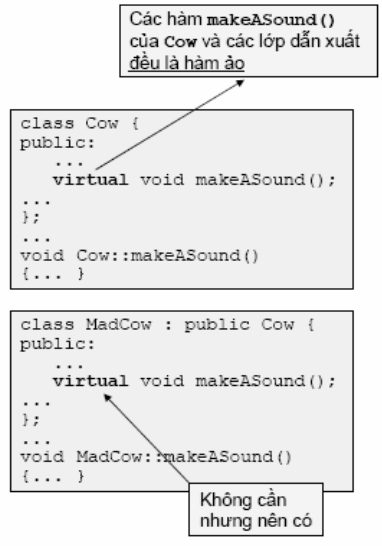
- **Hàm/phương thức ảo – virtual function/method** là cơ chế của C++ cho phép cài đặt đa hình động
- Nếu khai báo một hàm thành viên (phương thức) là virtual, trình biên dịch sẽ đẩy lùi việc liên kết các lời gọi phương thức đó với định nghĩa hàm cho đến khi chương trình chạy.
  - nghĩa là, ta bảo trình 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*

Quay lại ví dụ Bò điên



Hàm ảo

- Khai báo với từ khoá virtual trong định nghĩa lớp
- Không cần tại định nghĩa hàm, nếu định nghĩa hàm nằm ngoài định nghĩa lớp
- Một khi một phương thức được khai báolà hàm ảo tại lớp cơ sở, nó sẽ *tự động* là hàm ảo tại mọi lớp dẫn xuất trực tiếp hoặc gián tiếp.



Điều kiện để thực hiện đa hình trong C++

- Tồn tại sự kế thừa giữa các lớp theo một sơ đồ phân cấp nào đó
- Các lớp trong sơ đồ phân cấp này phải có hàm được khai báo với từ khoá virtual – phải có các hàm thành phần là virtual
- Để duyệt cây phân cấp phải khai báo và sử dụng con trỏ hoặc reference đến lớp cơ sở.
- Con trỏ hoặc reference được sử dụng để gọi các hàm thành phần ảo

Constructor và Destructor ảo

- Không thể khai báo các constructor ảo
- Có thể (và rất nên) khai báo destructor là hàm ảo.

```
class A {
public:
   A() {
        cout <<"A()"<< endl;
        p = new char[5];
   }
   virtual ~A() {
        cout <<"~A()"<< endl;
        delete [] p;
   }
private:
   char* p;
};
```

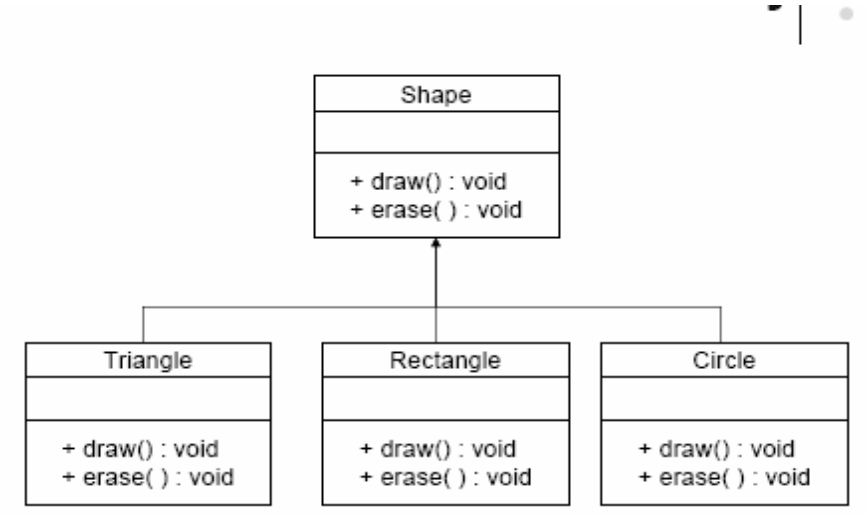
Destructor ảo

```
class Z : public A {
public:
  Z() {
     cout <<"Z()"<< endl;
     q = new char[5000];
  }
  ~Z() {
     cout <<"~Z()"<< endl;
     delete [] q;
  }
private:
  char* q;
};
```


Đa hình trong Java

- Không còn khái niệm con trỏ
- Trong Java liên kết lời gọi phương thức mặc nhiên là động
  - Mọi phương thức mặc định là phương thức ảo. Như vậy chúng mặc nhiên bị "định nghĩa lại" bởi các phương thức lớp kế thừa
  - Trừ các phương thức tĩnh (liên kết tĩnh)
- Thực hiện đơn giản hơn nhiều so với C++

Ví dụ với Java



Ví dụ với Java

```
...
public static void handleShapes(Shape[] shapes){
// Cac hinh ve se duoc ve theo cach rieng cua chung
for( int i = 0; i < shapes.length; ++i) {
shapes[i].draw();
}
...
//Ta don gian goi phuong thuc xoa khong quan tam den
// doi tuong thuoc lop nao
for( int i = 0; i < shapes.length; ++i) {
shapes[i].erase();
}
}
...
```

Tổng kết phần KTLTHDT

- Các đối tượng là các thành phần phần mềm có thể sử dụng lại, mô hình các thực thể trong thế giới thực
- Đối tượng có các thuộc tính biểu hiện trạng thái của nó, có các hành vi cho phép chúng chuyển đổi từ trạng thái này sang trạng thái khác

Tổng kết phần KTLTHDT

- Thiết kế HĐT là quá trình:
  - mô hình hóa thế giới thực
  - mô hình hóa sự tương tác giữa các đối tượng
  - đóng gói các thuộc tính và phương thức
  - che giấu thông tin giữa các đối tượng
  - đảm bảo tương tác hiệu quả thông qua một giao diện được định nghĩa tốt.