Ngôn ngữ lập trình

Bài 8:
Đa Hình và Hàm Ảo

Giảng viên: Lê Nguyễn Tuấn Thành
Email: thanhlnt@tlu.edu.vn

Bộ Môn Công Nghệ Phần Mềm – Khoa CNTT
Trường Đại Học Thủy Lợi

Nội dung

1. Đa hình (Polymorphism)
2. Hàm ảo (Virtual function)
   - Gắn kết muộn (Late binding)
   - Cài đặt hàm ảo
   - Khi nào sử dụng hàm ảo?
   - Hàm ảo thuần (Pure Virtual Function) và Lớp trừu tượng (Abstract Class)
3. Hàm ảo và Con trỏ
   - Mở rộng tương thích kiểu
   - Ép kiểu lên (Upcasting)
   - Ép kiểu xuống (Downcasting)

Bài giảng có sử dụng hình vẽ trong cuốn sách "*Absolute C++*. W. Savitch, Addison Wesley, 2002"

Đa hình (Polymorphism)

- Xét ví dụ: với cùng là thông điệp "*nhảy*", một con kangaroo và một con cóc sẽ nhảy hai kiểu khác nhau.
  - Chúng có cùng hành vi "*nhảy*" nhưng nội dung của hành vi này là khác nhau
- *Đa hình (Polymorphism)* là hiện tượng các đối tượng thuộc các lớp khác nhau hiểu cùng một thông điệp theo các cách khác nhau
- Đa hình là một trong ba trụ cột quan trọng trong OOP

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

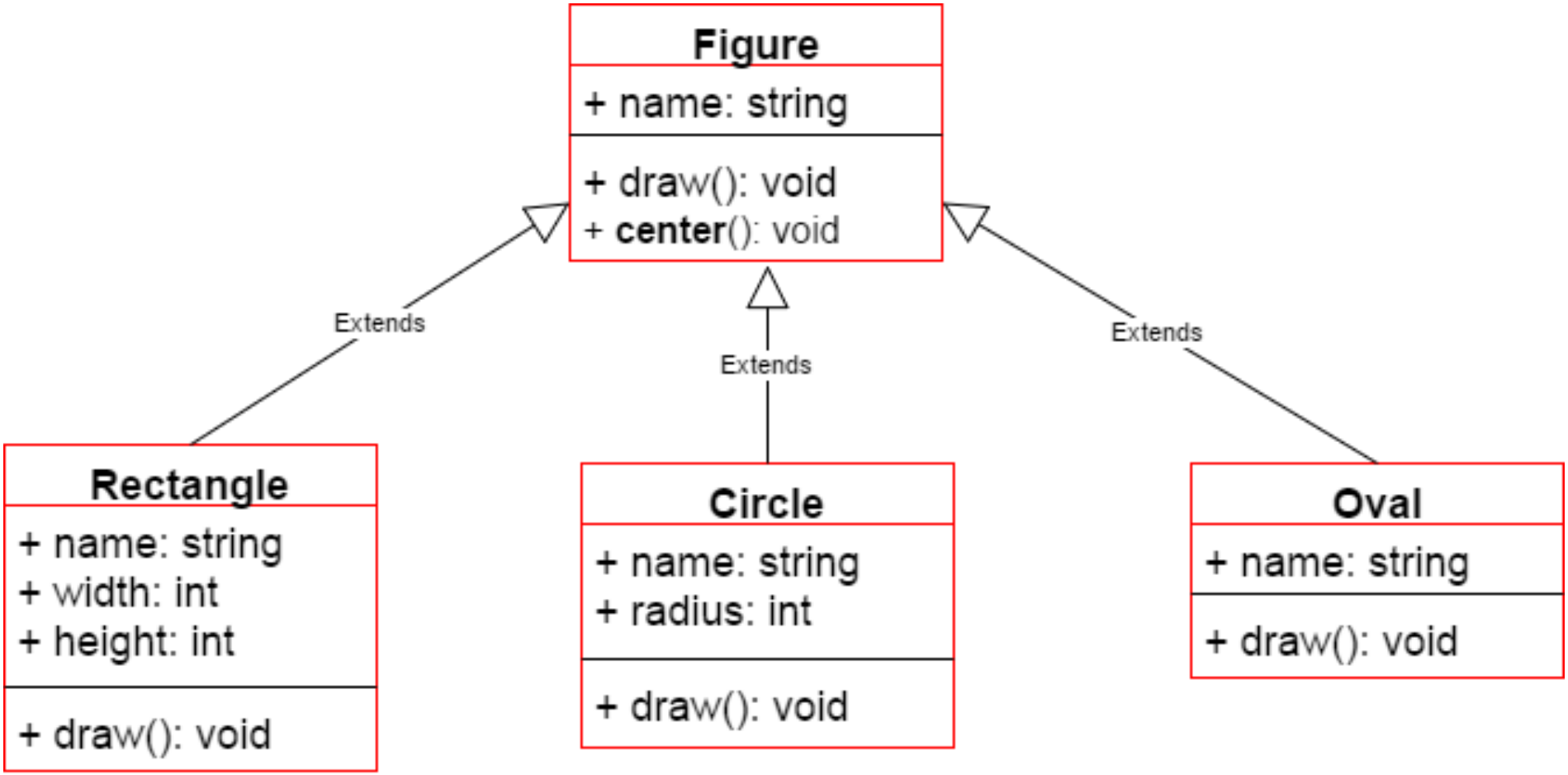
Hàm ảo

- Hàm ảo cung cấp khả năng đa hình này
- Hàm có thể được "*sử dụng*" trước khi thực sự được định nghĩa

Ví dụ 1: Các lớp mô tả hình vẽ (1/5) Hàm thành viên **draw**()

- Xây dựng các lớp cho nhiều kiểu hình vẽ khác nhau
  - Ví dụ: Hình chữ nhật (Rectangle), hình tròn (Circle), hình oval (Oval)…
  - Mỗi hình cụ thể là đối tượng của những lớp này
    - Dữ liệu cho hình chữ nhật: chiều cao, chiều rộng
    - Dữ liệu cho hình tròn: tâm, bán kính
  - Tất cả các lớp này đều kế thừa từ một lớp cha: *Figure*
- Các lớp này đều có hàm *draw()*
  - Mục đích là vẽ một hình cụ thể trên màn hình
  - Mỗi lớp có cài đặt khác nhau tương ứng với mỗi loại hình vẽ

Lớp Figure và các lớp con



Ví dụ 1: Các lớp mô tả hình vẽ (2/5)
Sử dụng hàm thành viên **draw**()

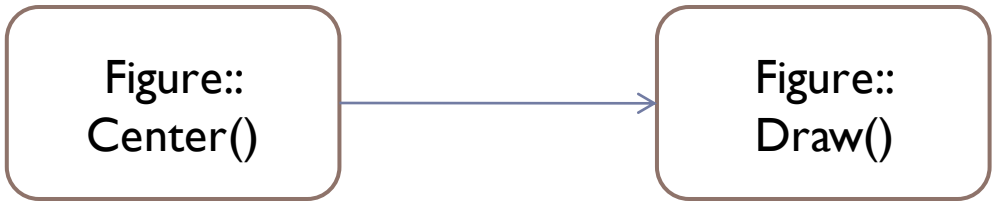
- Mỗi lớp con cần định nghĩa hàm *draw()* riêng
- Có thể gọi hàm *draw()* của mỗi lớp, ví dụ:

```
Rectangle r;
Circle c;
r.draw();   // Gọi hàm draw của lớp Rectangle
c.draw();  // Gọi hàm draw của lớp Circle
```

- Điều này là bình thường, chưa có gì đặc biệt ở đây!

Ví dụ 1: Các lớp mô tả hình vẽ (3/5)
Hàm thành viên **center**()

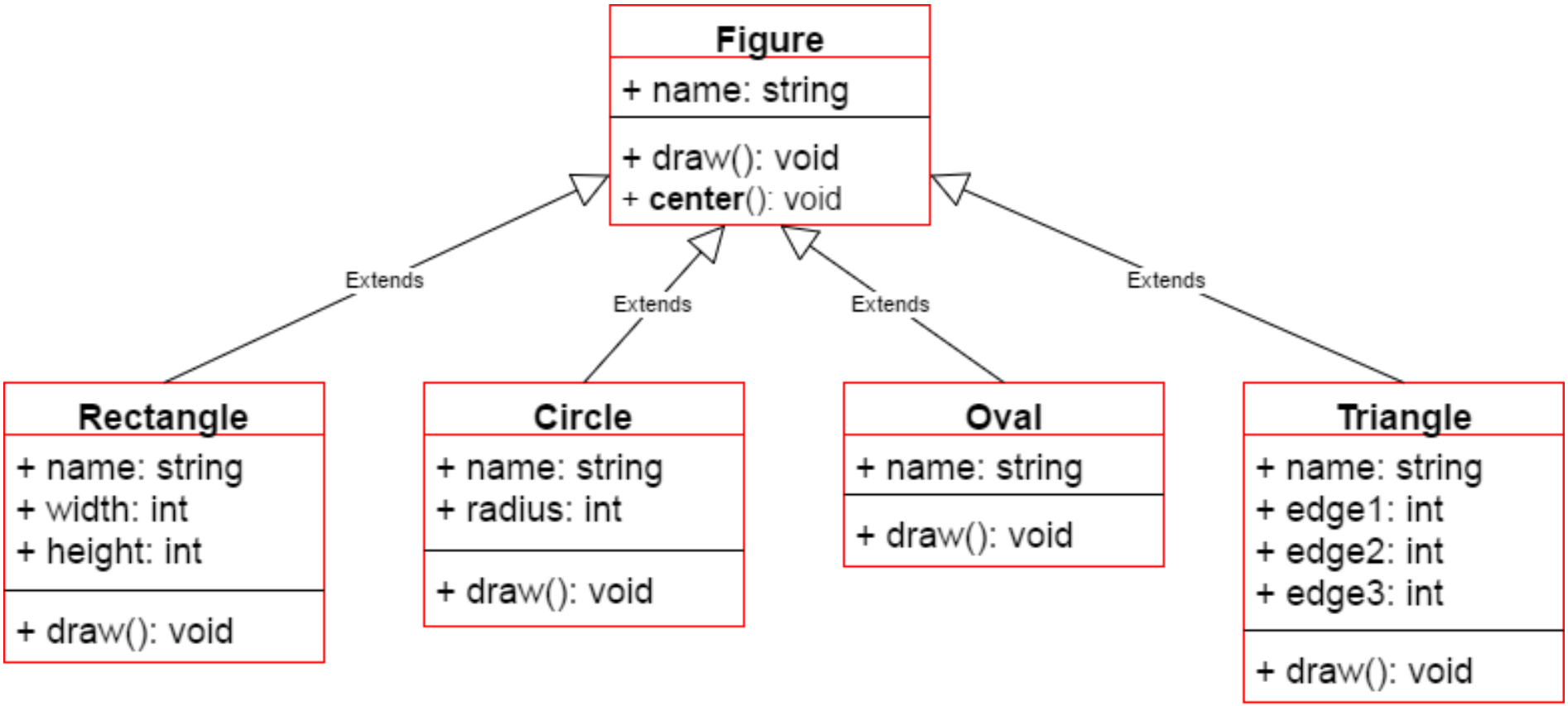
- Lớp cha *Figure* bao gồm những hàm có thể áp dụng cho "tất cả" hình vẽ
- Xét hàm *center*() để di chuyển một hình vẽ từ vị trí hiện tại tới vị trí trung tâm màn hình
  - Cách làm: xóa hình ở vị trí hiện tại, sau đó vẽ lại hình đó tại vị trí trung tâm màn hình
  - Hàm *Figure*::*center*() sẽ sử dụng (gọi) hàm *draw*() để vẽ lại hình
  - Câu hỏi:
    - Hàm *draw*() nào sẽ được gọi?
    - Từ lớp nào?



Ví dụ 1: Các lớp mô tả hình vẽ (4/5) Định nghĩa một lớp hình vẽ mới

- Xét một lớp hình vẽ mới: lớp *Triangle* kế thừa từ lớp *Figure*
- Lớp *Triangle* kế thừa hàm *center*() từ lớp cha *Figure*
  - Chỉ định nghĩa lại hàm draw(), không định nghĩa lại hàm *center*() trong lớp Triangle
  - Liệu hàm *center()* này có hoạt động được với lớp *Triangle*?
  - Liệu hàm *center()* này có gọi hàm *draw*() riêng của lớp *Triangle*?
  - Nếu hàm này gọi hàm *draw()* của lớp *Figure* thì nghĩa là không hoạt động đúng với lớp *Triangle*!
- ***Chúng ta muốn***: lớp *Triangle* kế thừa hàm *center*() của lớp cha *Figure* và sẽ TỰ ĐỘNG gọi hàm *draw()* của lớp *Triangle* chứ KHÔNG PHẢI hàm *draw()* của lớp *Figure*
- ***Vấn đề:*** Hàm *center()* của lớp *Figure* được định nghĩa TRƯỚC KHI lớp *Triangle* định nghĩa nên nó không biết sự tồn tại lớp *Triangle*

Thêm một lớp con mới Triangle của lớp Figure



Ví dụ 1: Các lớp mô tả hình vẽ (5/5) Hàm ảo

- Hàm ảo là câu trả lời cho vấn đề trên
- Hàm ảo nói với trình biên dịch:
  - Không biết một hàm sẽ được cài đặt như thế nào
  - Đợi cho đến khi được sử dụng trong chương trình
  - Khi đó sẽ lấy phần cài đặt từ đối tượng cụ thể
- Cơ chế này được gọi là *gắn kết muộn (late binding)* hoặc *gắn kết động (dynamic binding)*
  - Những hàm ảo cài đặt cơ chế gắn kết muộn (late binding)

Ví dụ 2: Doanh số bán hàng (1/2)

- Xây dựng chương trình giúp lưu trữ hồ sơ cho một cửa hàng phụ tùng ô tô.
  - Mục đích: lưu trữ doanh số bán hàng (Sale)
  - Không lường trước hết tất cả loại doanh số bán hàng
  - Đầu tiên chỉ có doanh số bán lẻ thông thường
  - Sau đó có thể thêm: doanh số bán hàng giảm giá (DiscountSale), doanh số bán hàng qua thư điện tử, …
    - Phụ thuộc vào nhiều yếu tố như giá, thuế …

Ví dụ 2: Doanh số bán hàng (2/2)

- Chương trình phải:
  - Tính toán số lượng lớn bán hàng mỗi ngày
  - Tính toán lượng bán hàng lớn nhất, nhỏ nhất trong ngày
  - Có thể là lượng bán hàng trung bình trong ngày
- Tất cả các hàm tính toán này đều bắt nguồn từ những hóa đơn riêng lẻ
  - Nhưng sau này nhiều hàm để tính hóa đơn khác sẽ được thêm vào, khi những loại doanh số bán hàng khác nhau được thêm vào.
- Vì thế hàm để tính toán một hóa đơn sẽ là một hàm ảo!

Định nghĩa lớp **sale**

```
class Sale
   {
   public:
         Sale();
         Sale(double thePrice);
         double getPrice() const;
         virtual double bill() const;
         double savings(const Sale& other) const;
   private:
         double price;
   };
```

Hàm thành viên ***savings*** và hàm nạp chồng toán tử <

```
// Khoảng cách giữa 2 doanh số bán hàng
double Sale::savings(const Sale& other) const
   {
         return (bill() – other.bill());
   }

// So sánh 2 doanh số bán hàng
bool operator < (const Sale& first,
                 const Sale& second)
   {
         return (first.bill() < second.bill());
   }
```

Lưu ý: CẢ HAI hàm này đều gọi hàm *bill*()!

Định nghĩa lớp **sale**

- Lớp *sale* biểu diễn doanh số bán hàng cho mỗi mục đơn lẻ mà không tính tới yếu tố giảm giá hay phí tăng thêm
- Chú ý từ khóa *virtual* trong khai báo của hàm thành viên *bill()*
  - Tác dụng: sau đó, những lớp kế thừa của lớp *Sale* có thể định nghĩa những phiên bản hàm *bill()* của riêng chúng
  - Những hàm thành viên khác của lớp *Sale* sẽ sử dụng phiên bản hàm *bill*() dựa trên đối tượng của lớp con!
  - Chúng sẽ không tự động sử dụng phiên bản hàm *bill*() của lớp cha *Sale*!

Định nghĩa lớp con **DiscountSale**

```
// Lớp con DiscountSale biểu diễn doanh số bán hàng giảm giá
class DiscountSale : public Sale
   {
   public:
         DiscountSale();
         DiscountSale(double thePrice,
                        double theDiscount);
         double getDiscount() const;
         void setDiscount(double newDiscount);
         double bill() const; // Cài đặt lại hàm ảo bill cho lớp con này
   private:
          double discount;
   };
```

Cài đặt hàm *bill()* của lớp con DiscountSale

```
// Cài đặt hàm bill cho lớp con DiscountSale
double DiscountSale::bill() const
   {
         double fraction = discount/100;
         return (1 – fraction)*getPrice();
   }
```

- Từ khóa *virtual* không cần xuất hiện trong cài đặt thực tế của hàm ảo *bill()* của lớp con *DiscountSale*
  - Tự động là hàm ảo trong lớp con
  - Khai báo (trong giao diện) cũng không yêu cầu phải có từ khóa *virtual* (nhưng thường được sử dụng)
- Hàm ảo trong lớp cơ sở (lớp cha) sẽ tự động là hàm ảo trong lớp kế thừa (lớp con)

Lớp con DiscountSale

- Hàm thành viên *bill()* của lớp *DiscountSale* được cài đặt khác so với hàm này trong lớp cha *Sale*
  - Riêng biệt cho việc bán hàng giảm giá
- Hàm thành viên *savings* và toán tử < của lớp cha Sale
  - Sẽ gọi định nghĩa này của hàm *bill()* cho tất cả các đối tượng của lớp con *DiscountSale*!
  - Thay vì gọi phiên bản mặc định được định nghĩa trong lớp cha *Sale*!
- Nhớ lại: lớp *Sale* được viết trước lớp con *DiscountSale*
  - Hàm thành viên *savings* và toán tử < được biên dịch ngay cả trước khi có ý tưởng về việc tạo lớp con *DiscountSale*!
- *DiscountSale d1;*
  d1.savings(d2);
  - Lời gọi trong hàm *savings* này tới hàm *bill()* sẽ biết sử dụng định nghĩa hàm *bill()* từ lớp DiscountSale!

Thực thi hàm ảo bằng cách nào?

- Liên quan đến khái niệm *gắn kết muộn* (*late binding*)
  - Hàm ảo cài đặt *gắn kết muộn* (late binding)
  - Nói trình biên dịch đợi cho đến khi hàm được sử dụng trong chương trình
  - Quyết định phiên bản nào của hàm được sử dụng dựa trên đối tượng gọi

Ghi đè (Overriding)

- Định nghĩa của hàm ảo thay đổi trong một lớp kế thừa
  - Chúng ta gọi đó là "ghi đè" (overidden)
    - Khác với nạp chồng (overloading) như thế nào ?
- Tương tự như định nghĩa lại cho các hàm chuẩn
- Phân biệt:
  - Hàm ảo thay đổi: ghi đè (overidden)
  - Hàm bình thường thay đổi: định nghĩa lại (redefined)

Nhược điểm của việc sử dụng hàm ảo

- Bỏ qua tất cả những lợi ích của hàm ảo như chúng ta đã thấy
- Hàm ảo có một bất lợi lớn: phụ phí (overhead)!
  - Sử dụng nhiều bộ nhớ hơn
  - Gắn kết muộn khiến chương trình chạy chậm hơn
- Vì vậy nếu hàm ảo không thật cần thiết thì không nên sử dụng

Hàm ảo thuần (Pure virtual functions)

- Lớp cơ sở có thể định nghĩa một vài thành viên của nó!
  - Mục đích của nó đơn giản là để cho những lớp khác kế thừa
- Nhớ lại lớp *Figure*
  - Tất cả các hình vẽ là đối tượng của lớp kế thừa cụ thể. Ví dụ: *Rectangle, Circle, Triangle, …*
  - Lớp *Figure* không có ý niệm về việc bằng cách nào có thể vẽ được!
- Tạo một hàm ảo thuần: *virtual void draw() = 0;*
- Các hàm ảo thuần không yêu cầu định nghĩa
  - Bắt buộc các lớp kế thừa phải định nghĩa phiên bản hàm riêng của nó

Lớp cơ sở trừu tượng (Abstract base classes)

- Lớp có một hay nhiều hàm ảo thuần gọi là: *lớp cơ sở trừu tượng*
  - Chỉ có thể được sử dụng như lớp cơ sở
  - Không thể tạo đối tượng từ lớp trừu tượng này. Bởi vì nó không có định nghĩa hoàn thiện của tất cả các thành viên!
- Nếu lớp thừa kế không định nghĩa tất cả hàm ảo thuần thì nó cũng sẽ là một lớp cơ sở trừu tượng

Mở rộng tương thích kiểu (Type compatibility)

- Giả sử **D** là lớp kế thừa từ lớp cơ sở B
  - Đối tượng của lớp **D** có thể được gán cho đối tượng của lớp cơ sở **B**
  - Nhưng ngược lại thì không thể!
- Xét ví dụ trước:
  - Một đối tượng *DiscountSale* "là" một Sale, nhưng điều ngược lại không đúng

Tương thích kiểu – ví dụ

```
class Pet
   {
   public:
         string name;
         virtual void print() const;
   };
class Dog : public Pet
   {
   public:
         string breed;
         virtual void print() const;
   };
```

Sử dụng hai lớp **Pet** và **Dog**

- Xét khai báo sau:

  Dog vdog;
  Pet vpet;

- Chú ý các biến thành viên *name* và *breed* đều public! Chỉ nhằm mục đích minh họa
- Tất cả mọi thứ "là" *dog* thì đều "là" *pet*

  vdog.name = "Tiny";
  vdog.breed = "Great Dane";
  vpet = vdog;

- Có thể gán giá trị về kiểu của lớp cha, nhưng không có chiều ngược lại
  - Một *pet* "không là" một *dog*

Mất mát thông tin (Slicing)

- **Chú ý**: khi giá trị được gán về *vpet*, biến thành viên *breed* của nó bị mất đi
  - *cout << vpet.breed;* // sẽ tạo ra một thông báo lỗi
  - Được gọi là vấn đề mất mát thông tin
- Điều này là hợp lý
  - Khi đối tượng của lớp Dog chuyển thành đối tượng của lớp Pet, nó sẽ được đối xử như một Pet
    - Do đó không còn các thuộc tính của một Dog
- Vấn đề mất mát thông tin gây phiền toái
  - *vpet* vẫn là một *Greet Dane* có tên là *Tiny*
  - Chúng ta muốn tham chiếu đến biến thành viên *breed* của nó kể cả khi nó được đối xử như một Pet
  - Có thể làm thế với con trỏ trỏ đến những biến động

Giải quyết vấn đề mất mát thông tin

- *Pet *ppet;*
  *Dog *pdog;*
  pdog = new Dog;
  pdog->name = "Tiny";
  pdog->breed = "Great Dane";
  ppet = pdog;
- Không thể truy cập trường *breed* của đối tượng được trỏ tới bởi pet:

  cout << ppet->breed; // Không hợp lệ!
- Phải sử dụng hàm ảo thành viên: *ppet->print();*
  - Gọi hàm thành viên *print()* trong lớp *Dog*!
    - Bởi vì nó là hàm ảo
  - C++ sẽ đợi để nhìn đối tượng con trỏ nào mà ppet thực sự trỏ tới trước khi lời gọi được gắn kết

Hàm hủy ảo (Virtual destructors)

- Hàm hủy cần giải phóng động dữ liệu được cấp phát
- Xét ví dụ:

  *Base *pBase = new Derived;*

  …

  delete pBase;

  - Sẽ gọi hàm hủy của lớp cơ sở mặc dù *pBase* đang trỏ tới đối tượng của lớp *Derived*!
  - Xây dựng hàm hủy ảo sẽ giải quyết vấn đề này!
- Cách tốt là định nghĩa tất cả hàm hủy là hàm ảo

Ép kiểu (Casting)

- Xét ví dụ:

```
Pet vpet;
Dog vdog;
…
vdog = static_cast<Dog>(vpet);     // Không hợp lệ!
```

- Không thể ép một pet thành một dog, nhưng:

```
vpet = vdog;                       // Hợp lệ!
vpet = static_cast<Pet>(vdog);     // Hợp lệ!
```

- Ép kiểu lên (upcasting) là hợp lệ
  - Ép từ kiểu con cháu lên kiểu tổ tiên

Ép kiểu xuống (Downcasting)

- Ép kiểu xuống rất nguy hiểm!
  - Ép từ kiểu tổ tiên thành kiểu con cháu
  - Giả sử thông tin được thêm vào
  - Có thể được thực hiện với *dynamic_cast*

```
Pet *ppet;
ppet = new Dog;
Dog *pdog = dynamic_cast<Dog*>(ppet);
```

    - Hợp lệ, nhưng nguy hiểm
- Ép kiểu xuống hiếm khi dùng do một số nhược điểm
  - Phải kiểm tra xem tất cả thông tin có được thêm vào hay không
  - Tất cả hàm thành viên phải là hàm ảo

Tóm tắt

- Gắn kết muộn (late binding) trì hoãn quyết định về việc hàm thành viên nào được gọi cho đến khi chạy chương trình
  - Trong C++, hàm ảo sử dụng cơ chế gắn kết muộn
- Hàm ảo thuần không có định nghĩa
  - Một lớp với ít nhất một hàm ảo thuần gọi là lớp trừu tượng
  - Không thể tạo đối tượng từ lớp trừu tượng
- Đối tượng của lớp kế thừa có thể được gán cho đối tượng của lớp cơ sở
  - Có thể một vài thông tin của lớp kế thừa bị mất
  - Gán con trỏ và đối tượng động cho phép giải quyết vấn đề mất mát thông tin (slicing)
- Nên định nghĩa tất cả hàm hủy là hàm ảo
  - Đảm bảo bộ nhớ được giải phóng đúng cách

Giáo trình Tham khảo

- **Giáo trình chính: W. Savitch, *Absolute C++*, Addison Wesley, 2002**
- Tham khảo:
  - A. Ford and T. Teorey, *Practical Debugging in C++*, Prentice Hall, 2002
  - Nguyễn Thanh Thủy, *Kĩ thuật lập trình C++*, NXB Khoa học và Kĩ Thuật, 2006