KẾ THỪA & ĐA HÌNH

TÁI SỬ DỤNG

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

TÁI SỬ DỤNG

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

TÁI SỬ DỤNG

- Ví dụ: has_a

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

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

4.1 KHÁI NIỆM

- Ví dụ: has_a

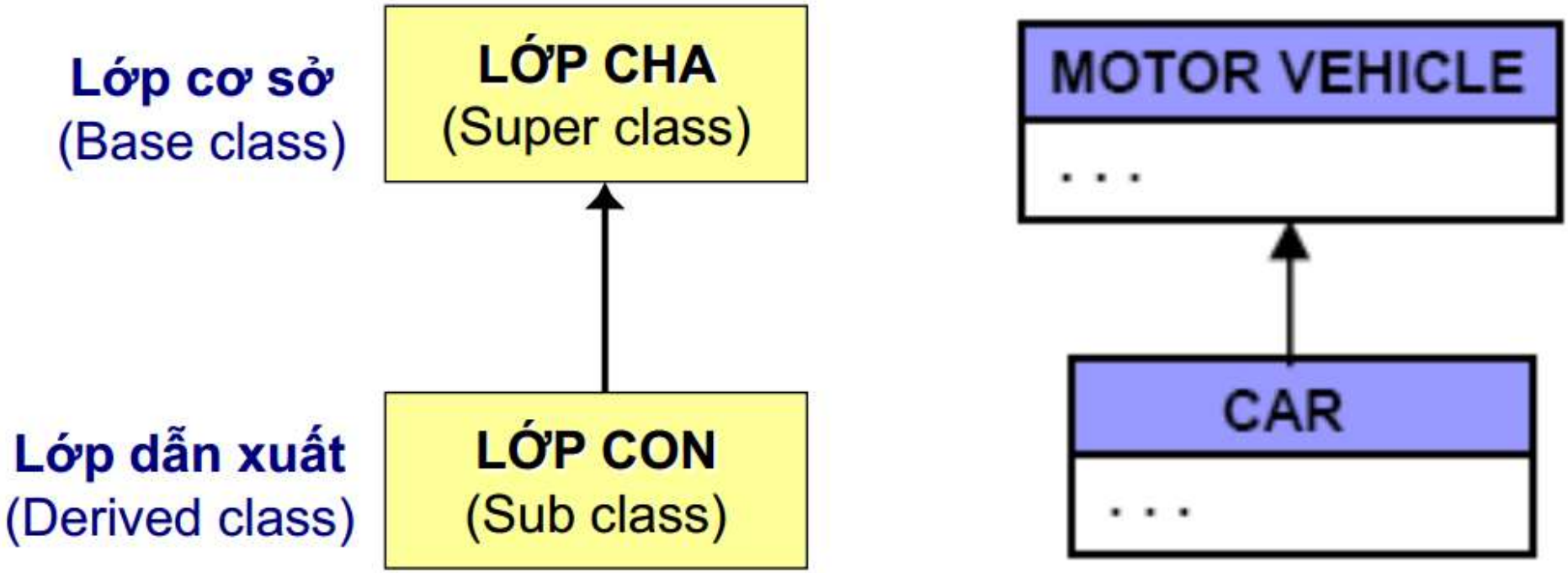
```
class Manager
{
    private:
        Employee me;
        Employee assistant;
    public:
        setAssistant(Employee e) {...}
        //...
};
//...
Manager junior;
Manager senior;
senior.setAssistant(junior); //error
```

KẾ THỪA

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

KẾ THỪA

- Kế thừa:

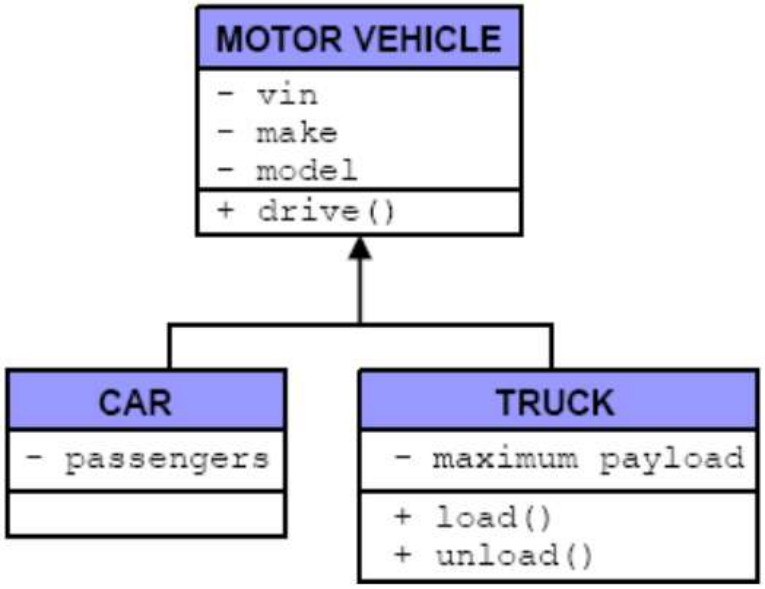


KẾ THỪA

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

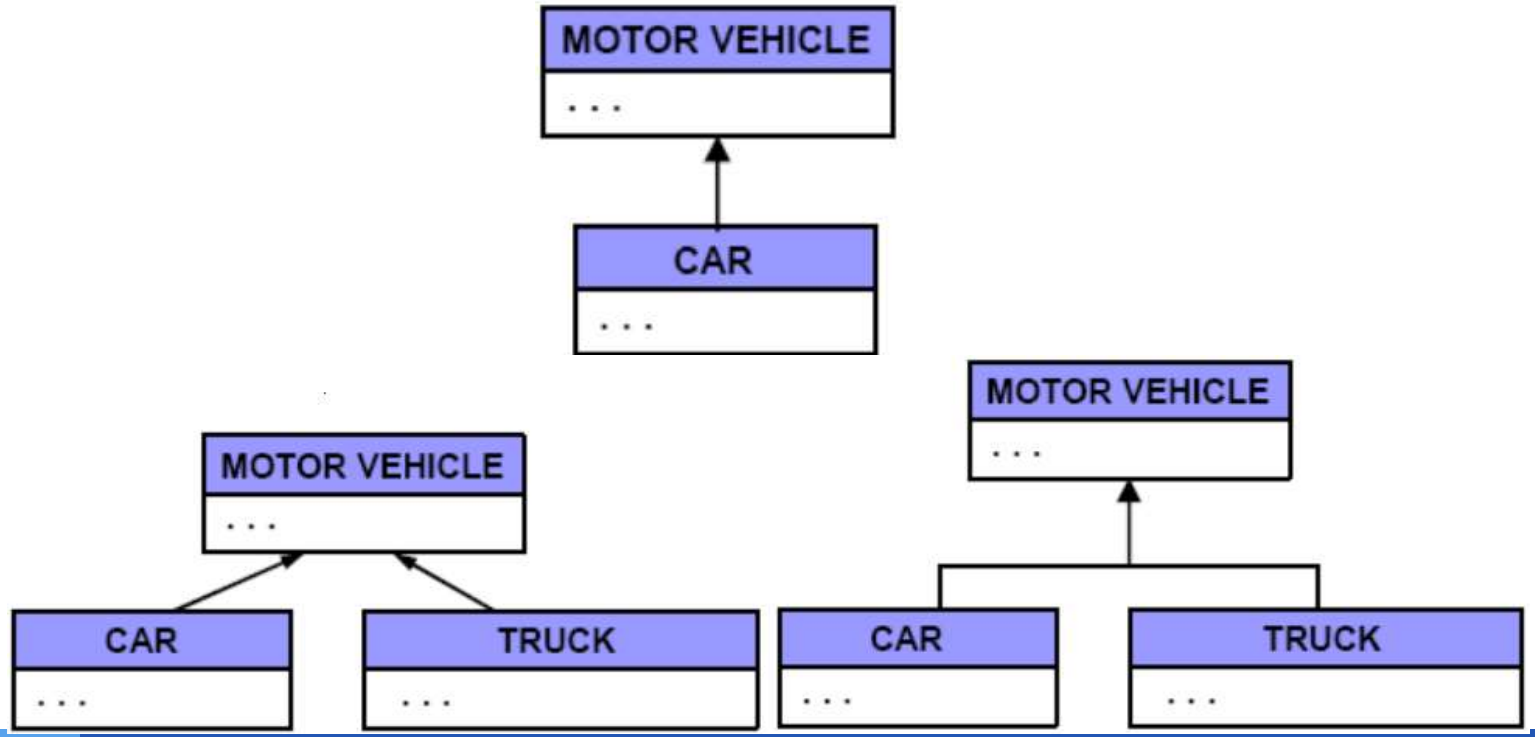
OBJECT RELATIONSHIP DIAGRAM – ORD

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



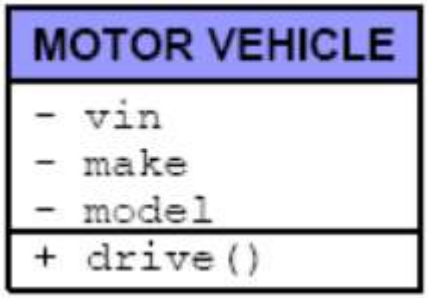
OBJECT RELATIONSHIP DIAGRAM – ORD

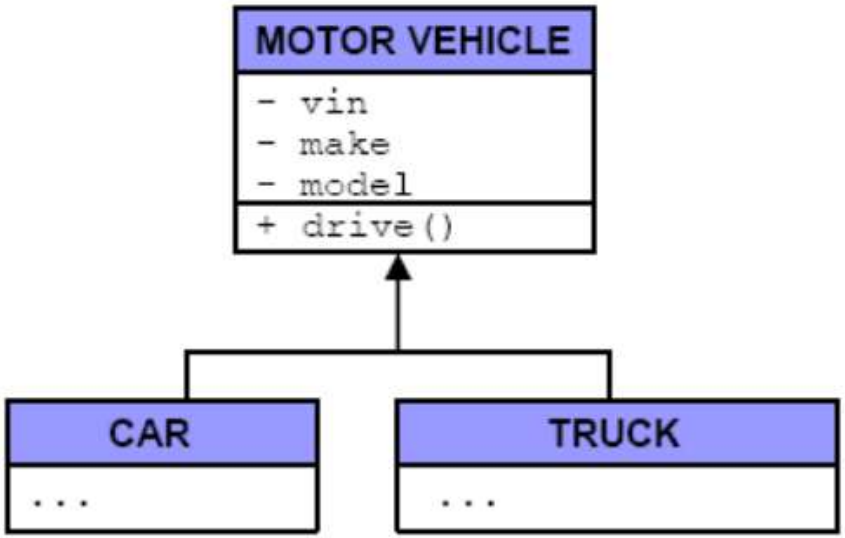
- Biểu diễn một quan hệ thừa kế giữa hai lớp bằng một mũi tên trỏ từ lớp con đến lớp cha;
- Có thể biểu diễn quan hệ với nhiều lớp con theo một trong hai kiểu sau:



OBJECT RELATIONSHIP DIAGRAM – ORD

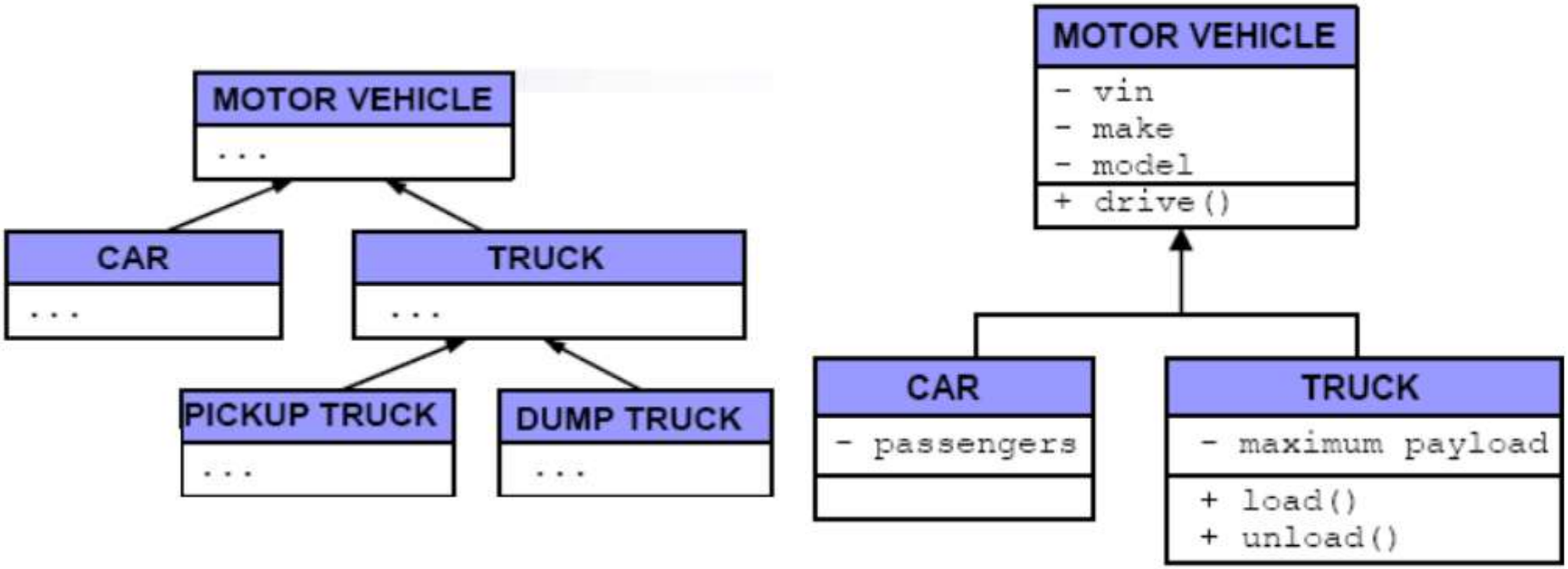
- Biểu diễn sơ đồ quan hệ:
  - Biểu diễn các thuộc tính và hành vi
    - Giả sử lớp MotorVehicle có các thuộc tính vin (số đăng ký xe), make (hãng), model (kiểu), và hành vi drive (lái).
  - Ta có sơ đồ quan hệ:
    - Mọi xe tải, xe ca đều có các thuộc tính vin, make, model, và hành vi drive.





SƠ ĐỒ QUAN HỆ ĐỐI TƯỢNG

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



CÂY THỪA KẾ

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

ĐỊNH NGHĨA LỚP DẪN XUẤT

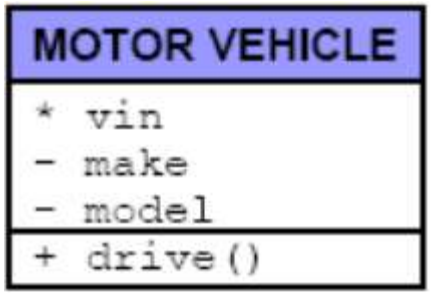
- Định nghĩa lớp con:

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

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

ĐỊNH NGHĨA LỚP DẪN XUẤT

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




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

ĐỊNH NGHĨA LỚP DẪN XUẤT

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

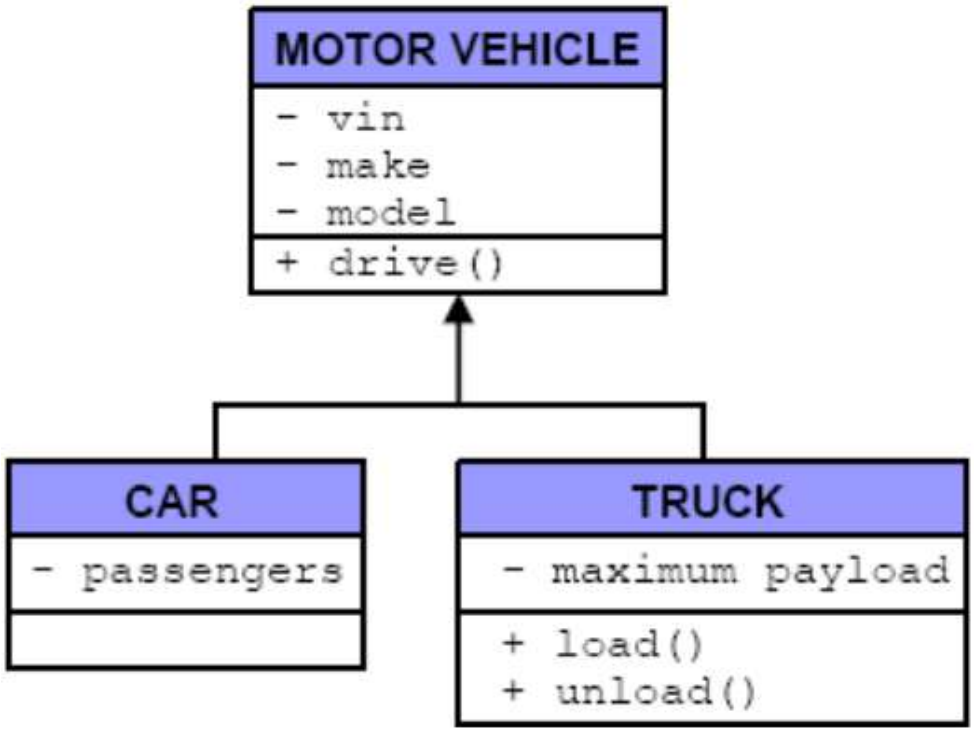
```
MotorVehicle::MotorVehicle(int vin, string make, string model)
{
    this->vin = vin;
    this->make = make;
    this->model = model;
}

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

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

ĐỊNH NGHĨA LỚP DẪN XUẤT

- Ví dụ:
  - Tạo lớp con Car



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

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

ĐỊNH NGHĨA LỚP DẪN XUẤT

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

MotorVehicle Car (int passengers);

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

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

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

ĐỊNH NGHĨA LỚP DẪN XUẤT

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

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

ĐỊNH NGHĨA LỚP DẪN XUẤT

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

ĐỊNH NGHĨA LỚP DẪN XUẤT

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

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

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

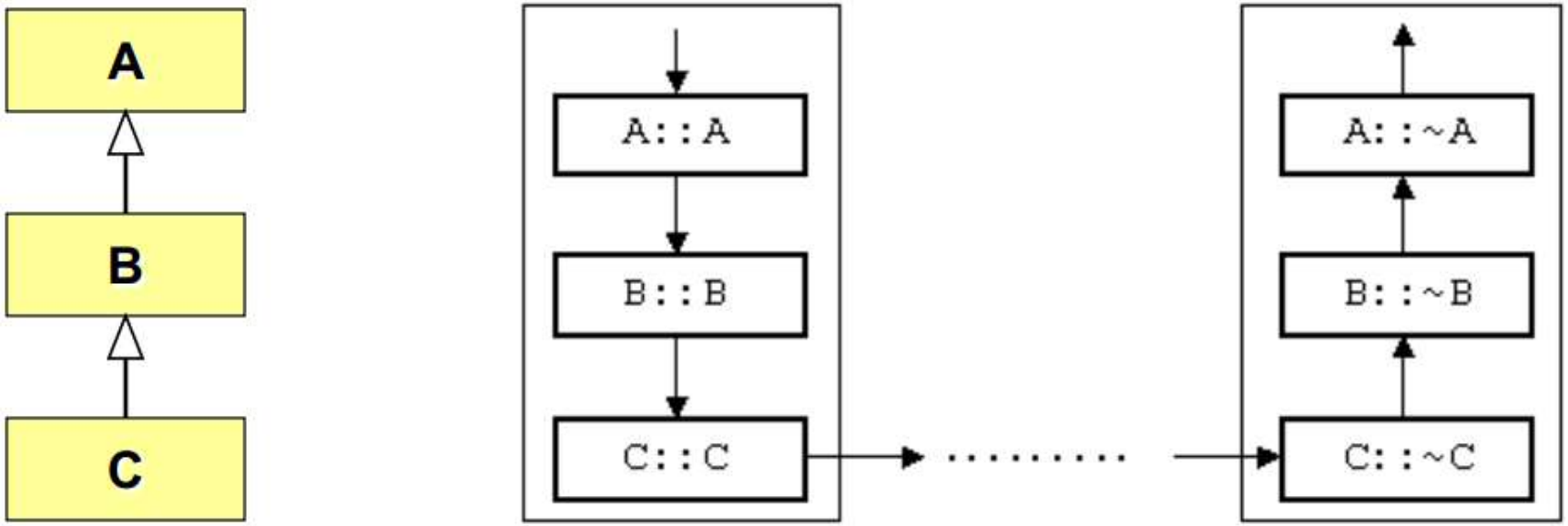
Gọi constructor của **MotorVehicle** với các tham số **vin**, **make**, **model**

ĐỊNH NGHĨA LỚP DẪN XUẤT

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

HÀM DỰNG VÀ HÀM HỦY

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



HÀM DỰNG VÀ HÀM HỦY

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

```
class Diem
{
    double x,y;
    public:
        Diem(double x, double y):x(xx),y(yy){}
        //...
};
class HinhTron:Diem
{
    double r;
    public:
        void Ve(int color) const;
};
```

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

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

QUYỀN TRUY CẬP

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

QUYỀN TRUY CẬP

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

QUYỀN TRUY CẬP

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

```
class Nguoi
{
    char *HoTen;
    int NamSinh;
    public:
    //...
};
class SinhVien : public Nguoi
{
    protected:  char *MaSo;
    public:
        SinhVien(char *ht, char *ms, int ns):
            Nguoi(ht,ns)
            { MaSo = strdup(ms);}
        ~SinhVien() {delete [] MaSo;}
        void Xuat() const;
};
void SinhVien::Xuat() const
{
    cout << "Sinh vien, ma so: " << MaSo
         << ", ho ten: " << HoTen;
}
```

QUYỀN TRUY CẬP

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

QUYỀN TRUY CẬP

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

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

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

QUYỀN TRUY CẬP

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

QUYỀN TRUY CẬP

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

QUYỀN TRUY CẬP

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

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

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

QUYỀN TRUY CẬP

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

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

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

QUYỀN TRUY CẬP

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

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

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

OVERIDING

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

OVERIDING

- Ví dụ:

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

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

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

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

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

- Các lớp dẫn xuất Car và Truck:

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

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

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

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

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

CÁC ĐỐI TƯỢNG ĐƯỢC KẾ THỪA TRONG C++

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

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

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

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

CON TRỎ VÀ KẾ THỪA

- Con trỏ trỏ đến đối tượng thuộc lớp cơ sở thì có thể trỏ đến các đối tượng thuộc lớp con;
- Ngược lại, con trỏ trỏ đến đối tượng thuộc lớp con thì không thể trỏ đến các đối tượng thuộc lớp cơ sở;

❖ Có thể sử dụng ép kiểu trong trường hợp này → không nên dùng.

CON TRỎ VÀ KẾ THỪA

```
void main()
{

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

UPCAST

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

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

UPCAST

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

mvPointer2->Drive();

UPCAST

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

    MotorVehicle* mvPointer2 = cPointer;
  - Ví dụ về upcast đối với tham chiếu:

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

UPCAST

- Upcast thường gặp tại các định nghĩa hàm, khi một con trỏ/tham chiếu đến lớp cơ sở được yêu cầu, nhưng con trỏ/tham chiếu đến lớp dẫn xuất cũng được chấp nhận
  - Xét hàm sau:

    void sellMyVehicle(MotorVehicle& myVehicle) {/*...*/}
  - Có thể gọi sellMyVehicle một cách hợp lệ với tham số là một tham chiếu tới một MotorVehicle, một Car, hoặc một Truck.

UPCAST

- Nếu ta dùng một con trỏ tới lớp cơ sở để trỏ tới một thể hiện của lớp dẫn xuất, trình biên dịch sẽ chỉ cho ta coi đối tượng như thể nó thuộc lớp cơ sở:
  - Như vậy, ta không thể làm như sau:

mvPointer2->Load(); // Error

- Đó là vì trình biên dịch không thể đảm bảo rằng con trỏ thực ra đang trỏ tới một thể hiện của Truck.

UPCAST

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

```
MotorVehicle* mvPointer2 = tPointer;
mvPointer2->Load(); //error
```

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

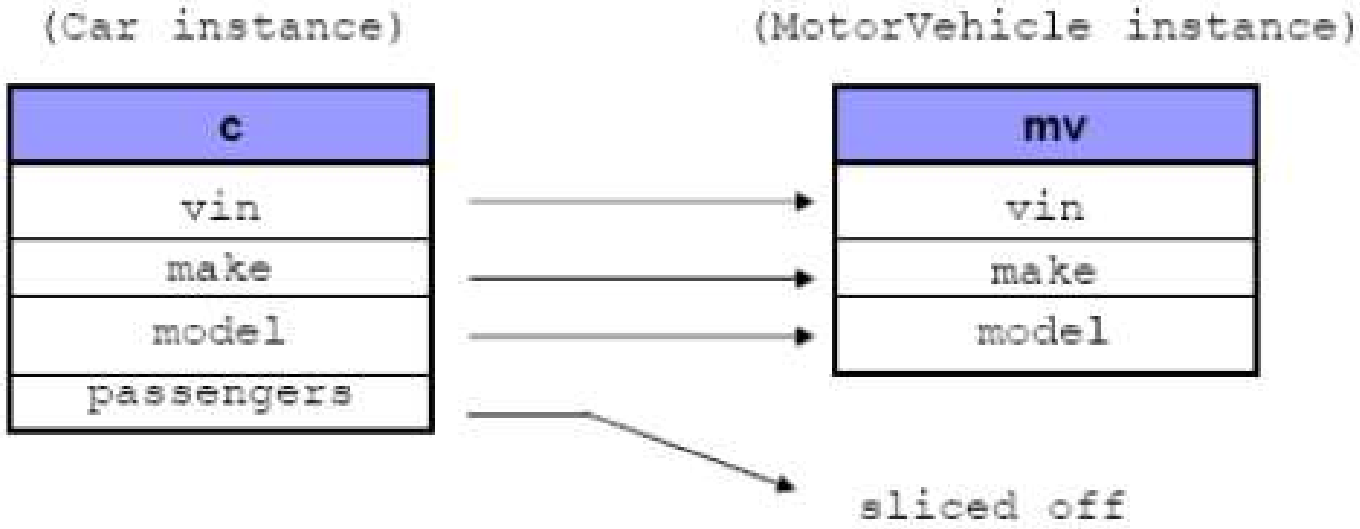
```
tPointer = new Truck(…);
mvPointer2 = tPointer;   //Point to a Truck
tPointer->Load();        //We can still do this
mvPointer2->Load();      //Even though we can't do this (error)
```

SLICING

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

SLICING

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



SLICING

- Thực ra, quy trình này cũng giống hệt như khi ta ngầm đổi giữa các kiểu dữ liệu có sẵn và bị mất bớt dữ liệu (chẳng hạn khi đổi một số chấm động sang số nguyên)
- Slicing còn xảy ra khi ta dùng phép gán

```
Car c(10, “Honda”, “S2000”, 2);
MotorVehicle mv;
mv = c;
```

DOWNCAST

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

DOWNCAST

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

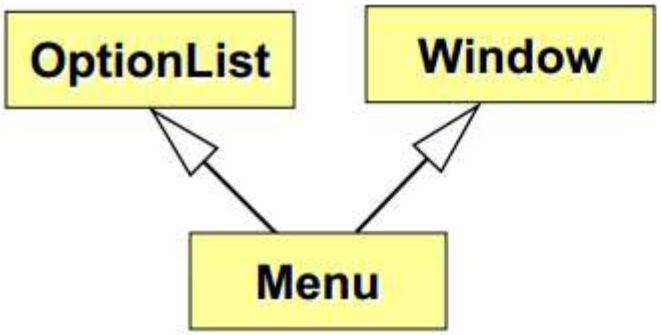
```
MotorVehicle* mvPointer3;
…
Car* cPointer2 = mvPointer3; // Error
Truck* tPointer2 = mvPointer3; // Error
MotorCycle mcPointer2 = mvPointer3; // Error
```

DOWNCAST

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

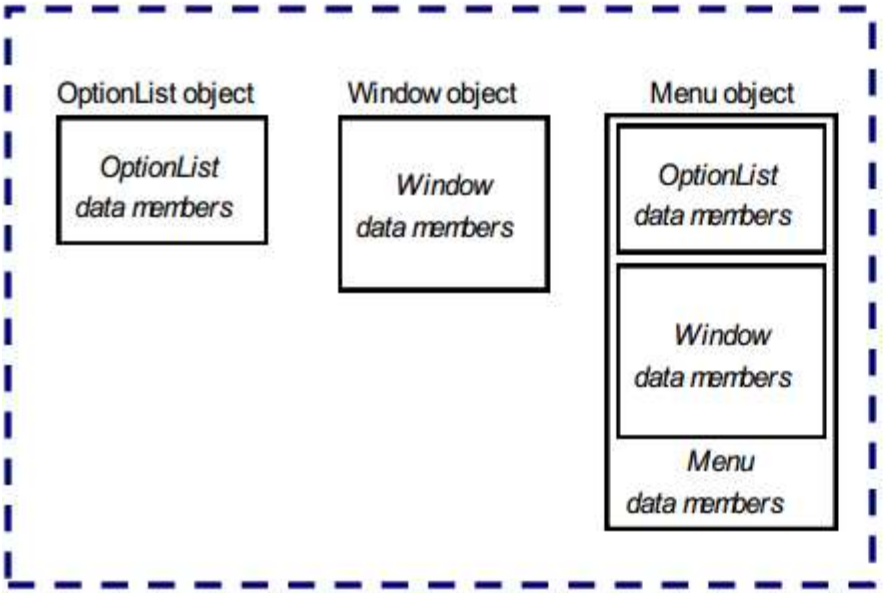
- Đoạn mã trên hoàn toàn hợp lệ và sẽ được trình biên dịch chấp nhận;
- Tuy nhiên, nếu chạy đoạn trình trên, chương trình có thể bị đổ vỡ (thường là khi lần đầu truy nhập đến thành viên/phương thức được định nghĩa của lớp dẫn xuất mà ta đổi tới).

ĐA THỪA KẾ




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

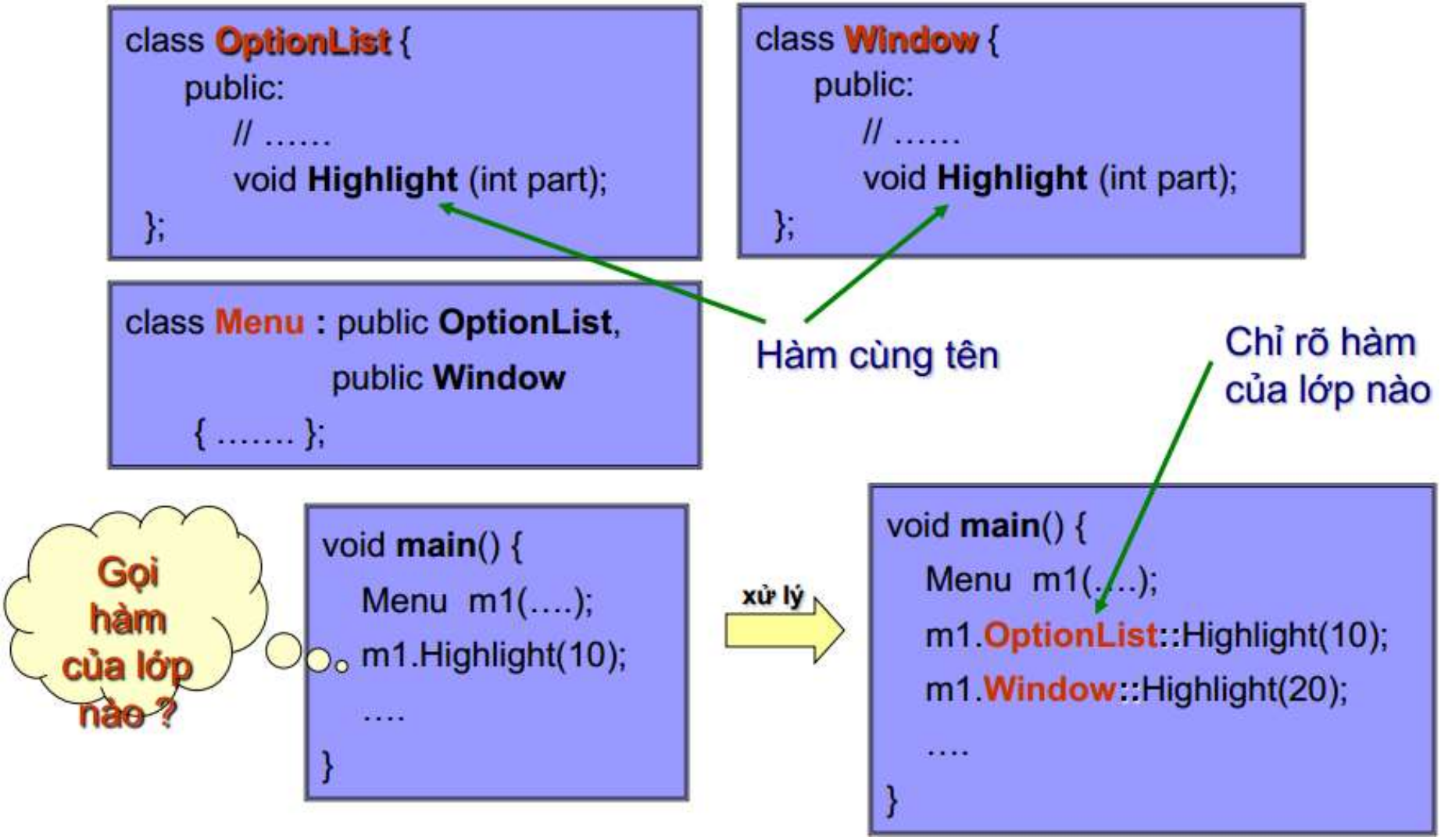
```
class Window {
    public:
        Window (Rect &);
        ~Window (void);
        //...
};
```




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

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

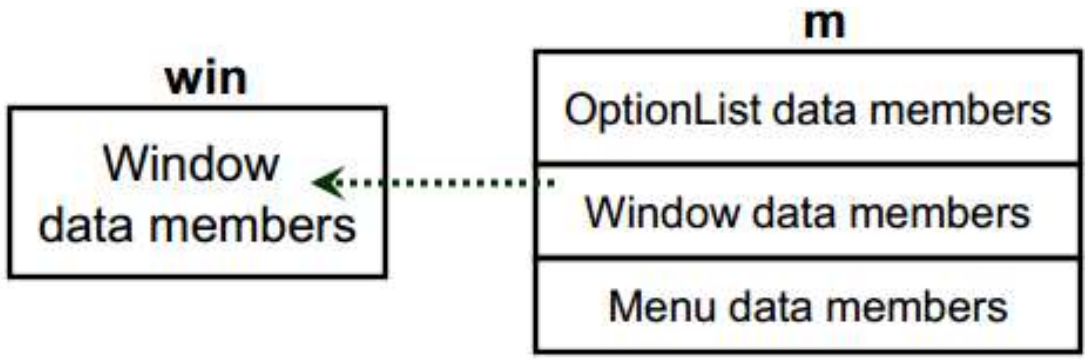
SỰ MƠ HỒ TRONG ĐA THỪA KẾ



CHUYỂN KIỂU

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

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



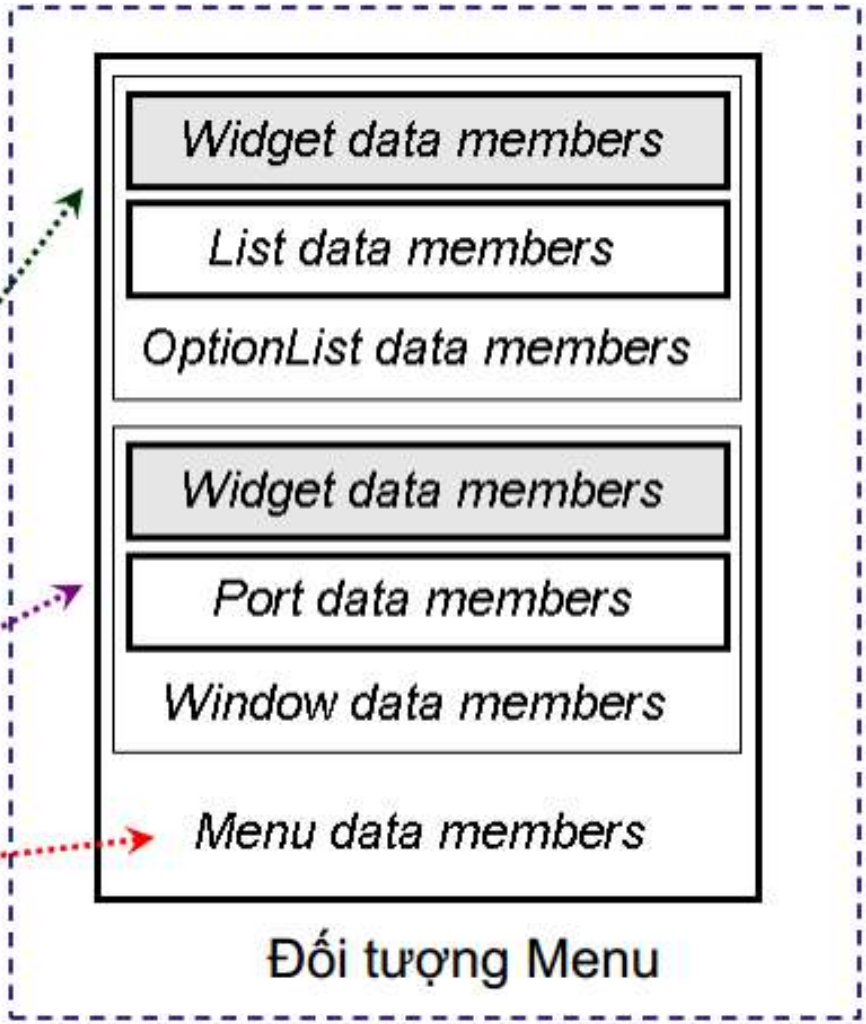
- Không được thực hiện phép gán ngược:
  - Đối tượng lớp con = Đối tượng lớp cha; // SAI

Nếu muốn thực hiện phải tự định nghĩa phép ép kiểu

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

LỚP CƠ SỞ ẢO

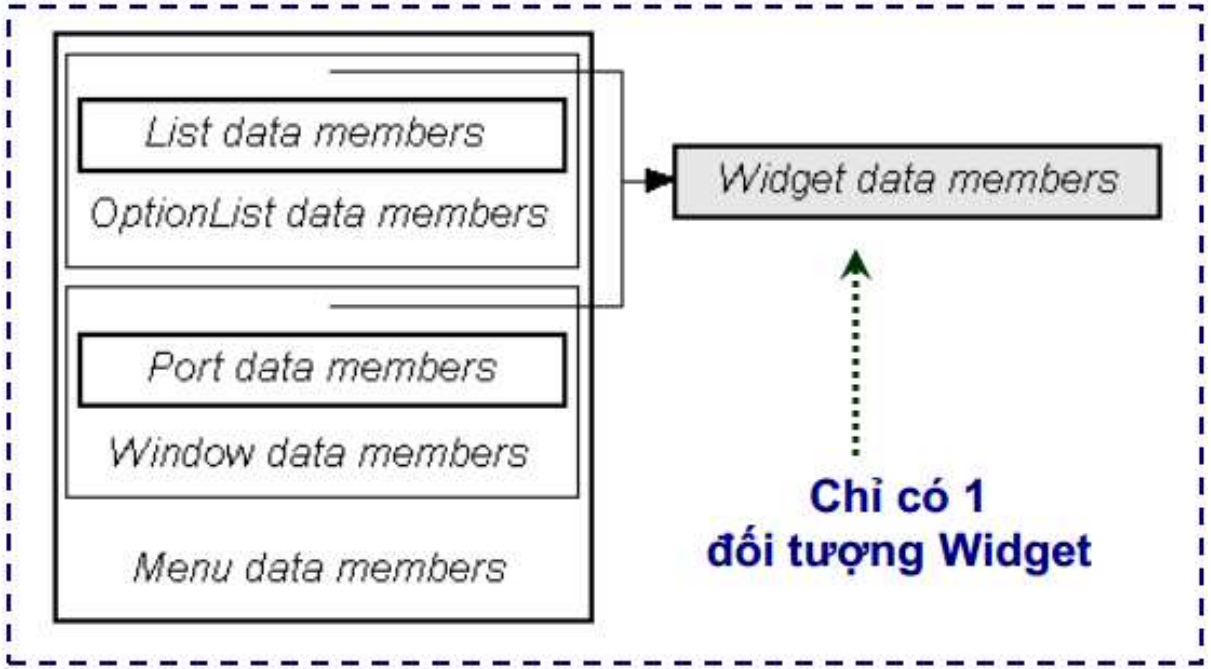
```
class OptionList
      : public Widget, List
 { /*...*/ };
class Window
      : public Widget, Port
 { /*...*/ };
class Menu
     : public OptionList,
       public Window
 { /*...*/ };
```



LỚP CƠ SỞ ẢO

- Cách xử lý: dùng lớp cơ sở ảo

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




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

CÁC TOÁN TỬ ĐƯỢC TÁI ĐỊNH NGHĨA

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

  → SAI khi có con trỏ thành viên.
- Cẩn thận với toán tử **new** và **delete**.

ĐA HÌNH

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

    myFunction() {... }

    myFunction(int x) {... }

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

ĐA HÌNH & HƯỚNG ĐỐI TƯỢNG

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

ĐA HÌNH & HƯỚNG ĐỐI TƯỢNG

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

OVERLOADING & OVERRIDING

- Function overloading - Hàm 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 & OVERRIDING

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

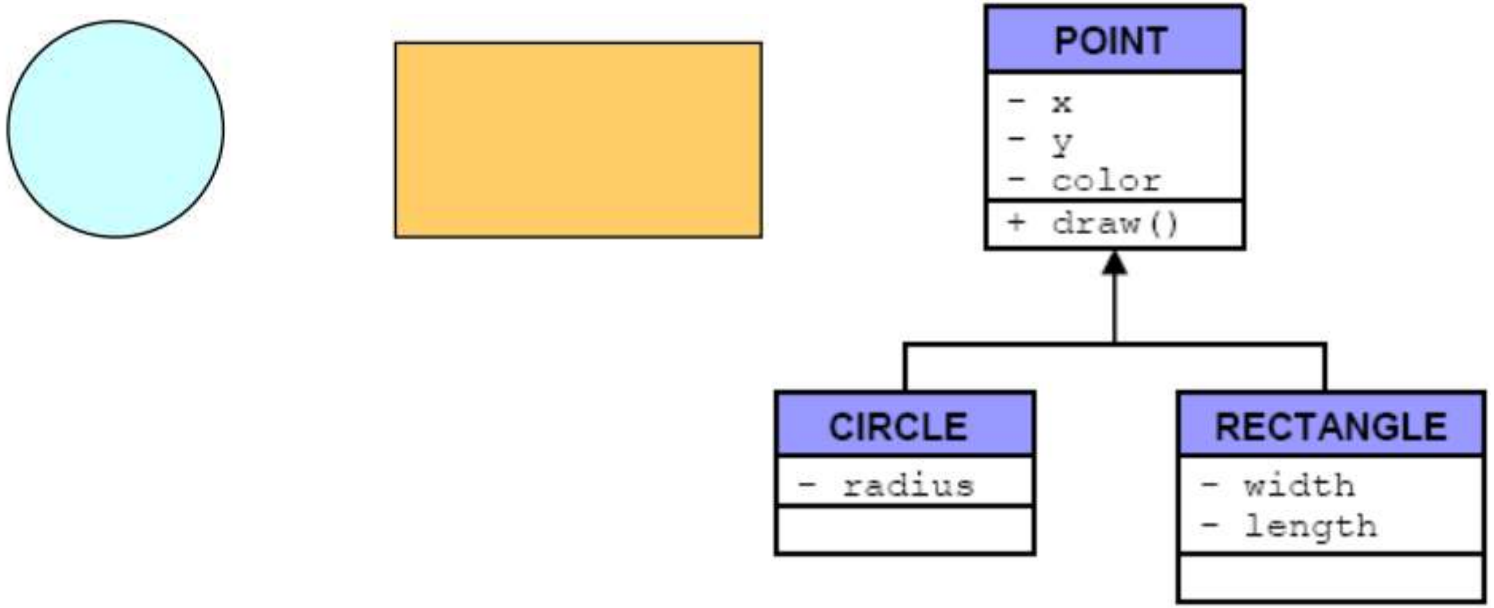
Hai thông điệp giống nhau, nhưng đích là hai lớp khác nhau

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

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

METHOD OVERLOADING

- Xét hành vi "draw" của các lớp trong cây phả hệ
  - Thông điệp "draw" gửi cho một 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 OVERLOADING

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

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

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

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

METHOD OVERLOADING

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

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

METHOD OVERLOADING

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

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

```
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

METHOD OVERLOADING

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

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

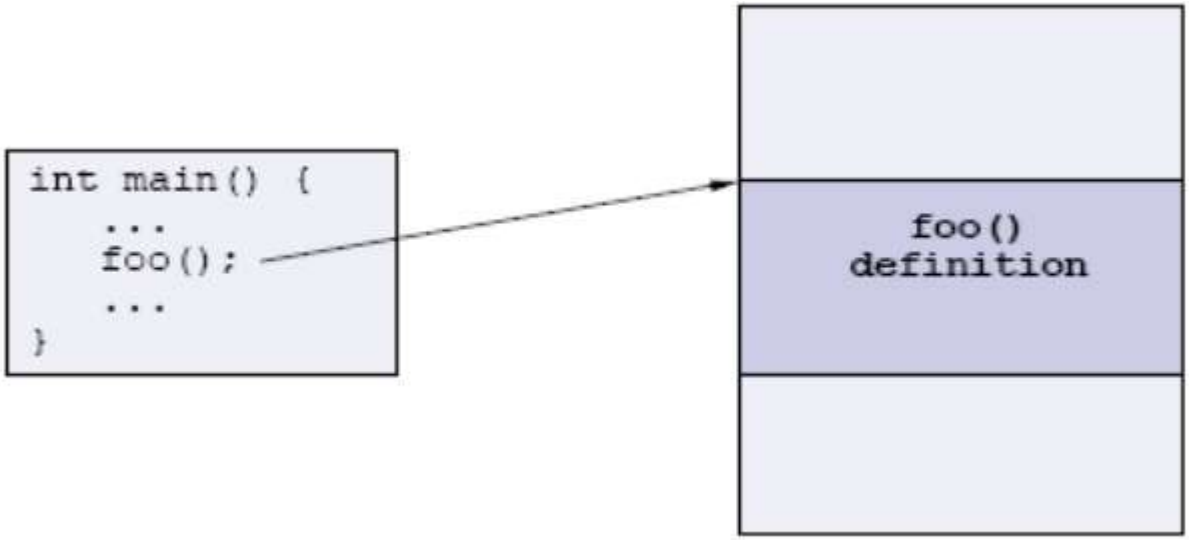
- Ta cũng có thể tương tác với các thể hiện lớp dẫn xuất như thể chúng là thể hiện của lớp cơ sở
  - Circle *pc = new Circle(100,100,"blue",50);
  - Point* pp = pc;
- Nhưng nếu có đa hình, lời gọi sau sẽ làm gì?
  - pp->draw(); // Draw what???

FUNCTION CALL BINDING

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

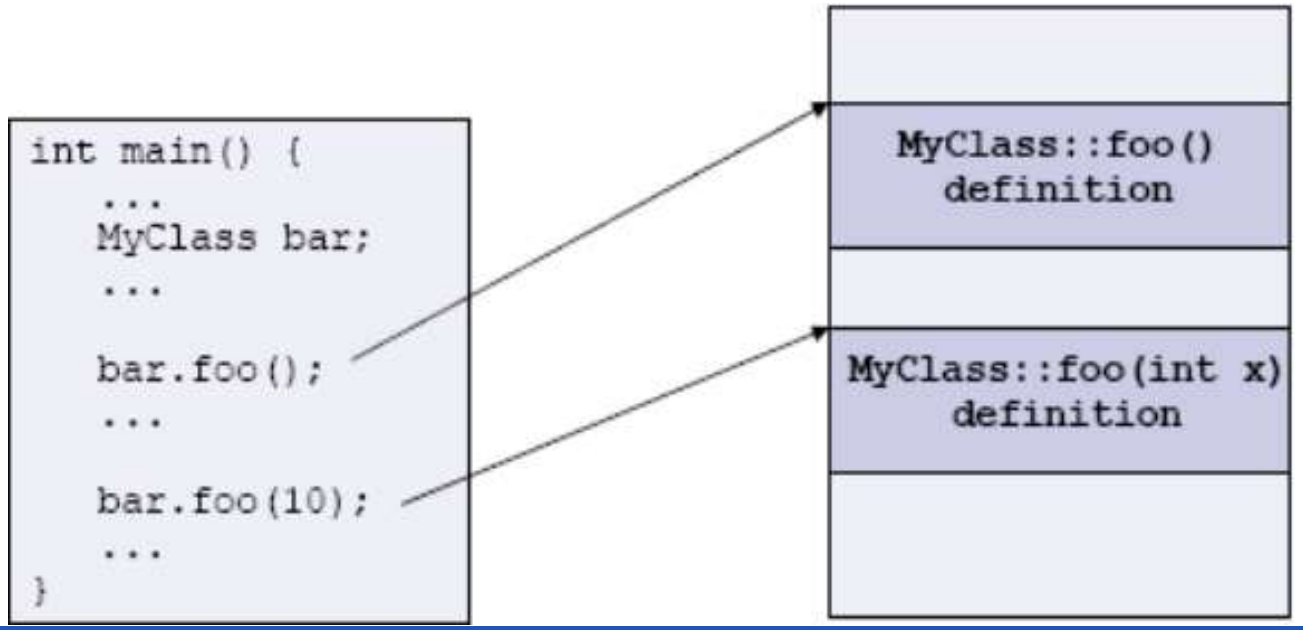
FUNCTION CALL BINDING

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



METHOD CALL BINDING

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

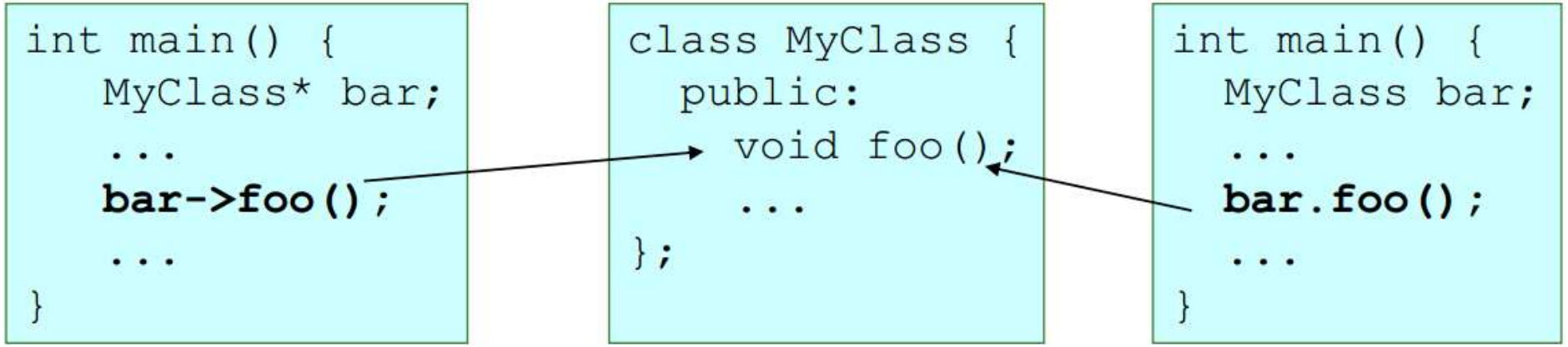


STATIC BINDING

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

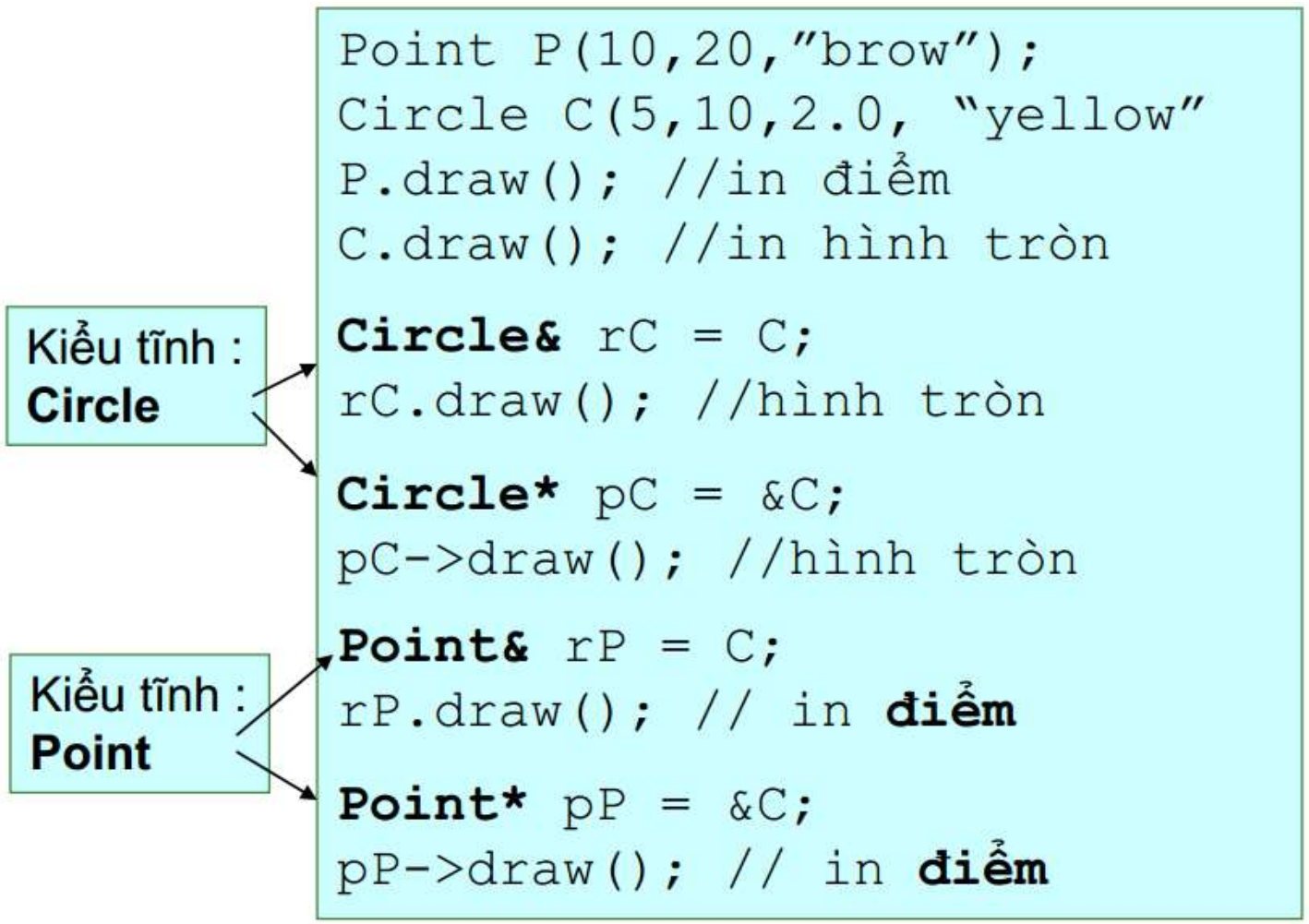
STATIC BINDING

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



STATIC BINDING

- Trường hợp có cây thừa kế:



STATIC BINDING

- Ta muốn:

rP.draw() và pP->draw();

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

STATIC BINDING

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

STATIC POLYMORPHISM

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

STATIC POLYMORPHISM

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

STATIC POLYMORPHISM

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

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

```
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

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

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

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

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

VIRTUAL FUNCTION

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

VIRTUAL FUNCTION

- Ví dụ:

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

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

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

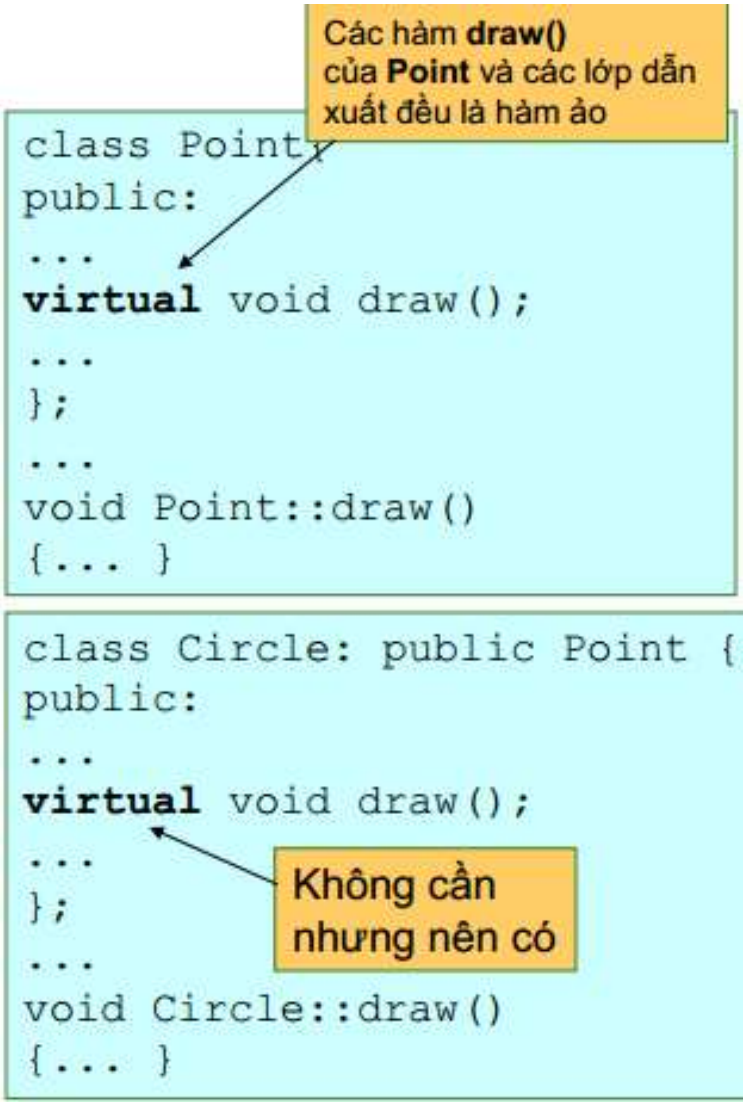
```
int main() {
    Cirlce C(5,10,2.0, "yellow");
    ...
    Point* pP = &C;
    pP->draw(); // in hình tròn
    ...
}
```

VIRTUAL FUNCTION

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

VIRTUAL FUNCTION

- Ví dụ:



VIRTUAL FUNCTION

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

VIRTUAL FUNCTION

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

VIRTUAL FUNCTION

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

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

VIRTUAL FUNCTION

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

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

VIRTUAL FUNCTION

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

VIRTUAL FUNCTION

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

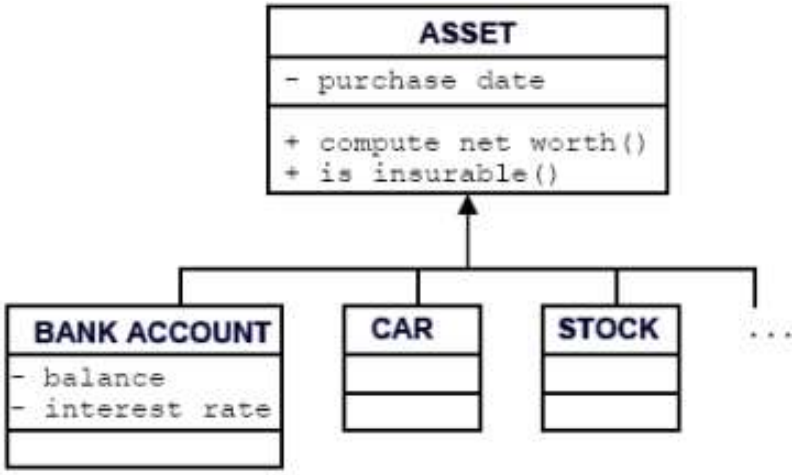
```
class MotorVehicle
{
    public:
        MotorVehicle(int vin, string make, string model);
        virtual ~MotorVehicle();
};
```

ABSTRACT CLASS

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

ABSTRACT CLASS

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

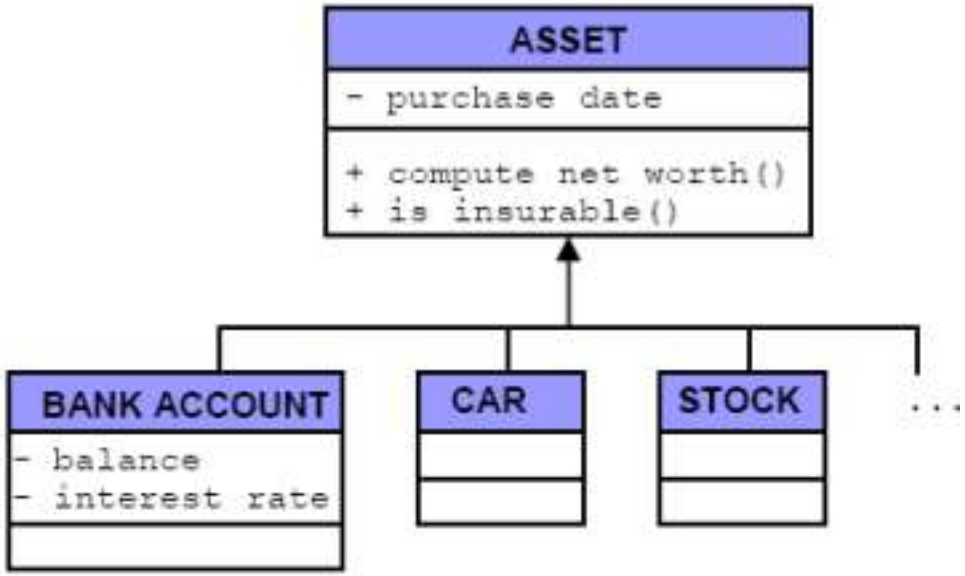


ABSTRACT CLASS

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

ABSTRACT CLASS

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



ABSTRACT CLASS

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

THIẾT KẾ THỪA KẾ

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

THIẾT KẾ THỪA KẾ

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

THIẾT KẾ THỪA KẾ

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

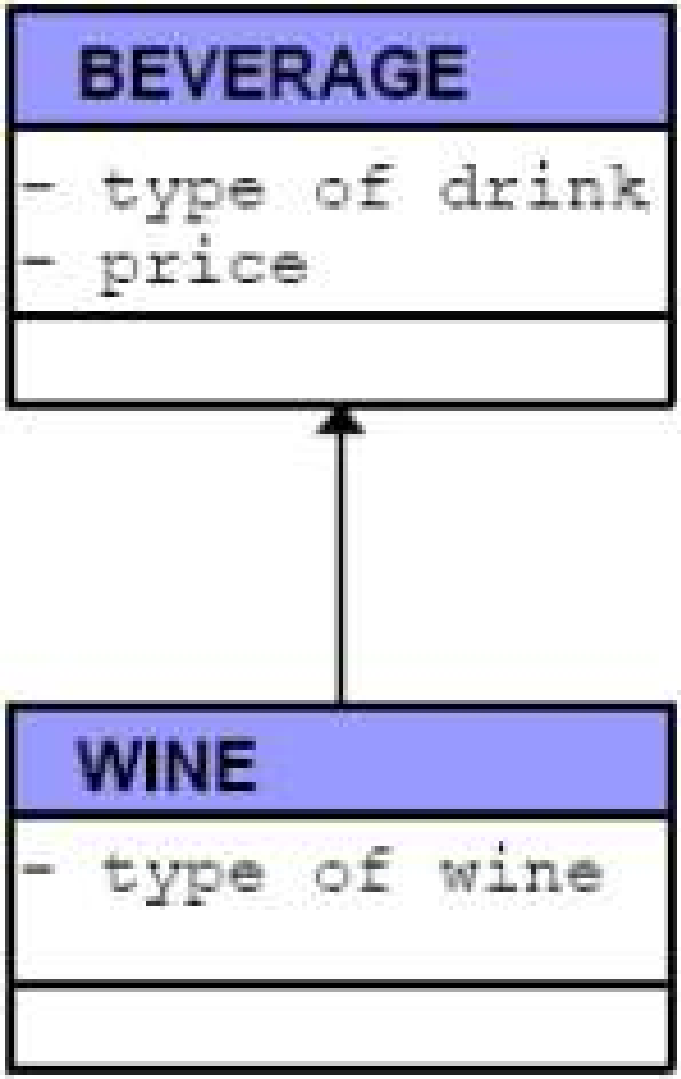
THIẾT KẾ THỪA KẾ

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

- Tuy nhiên, nếu ta muốn lưu trữ nhiều thông tin hơn thì sao?

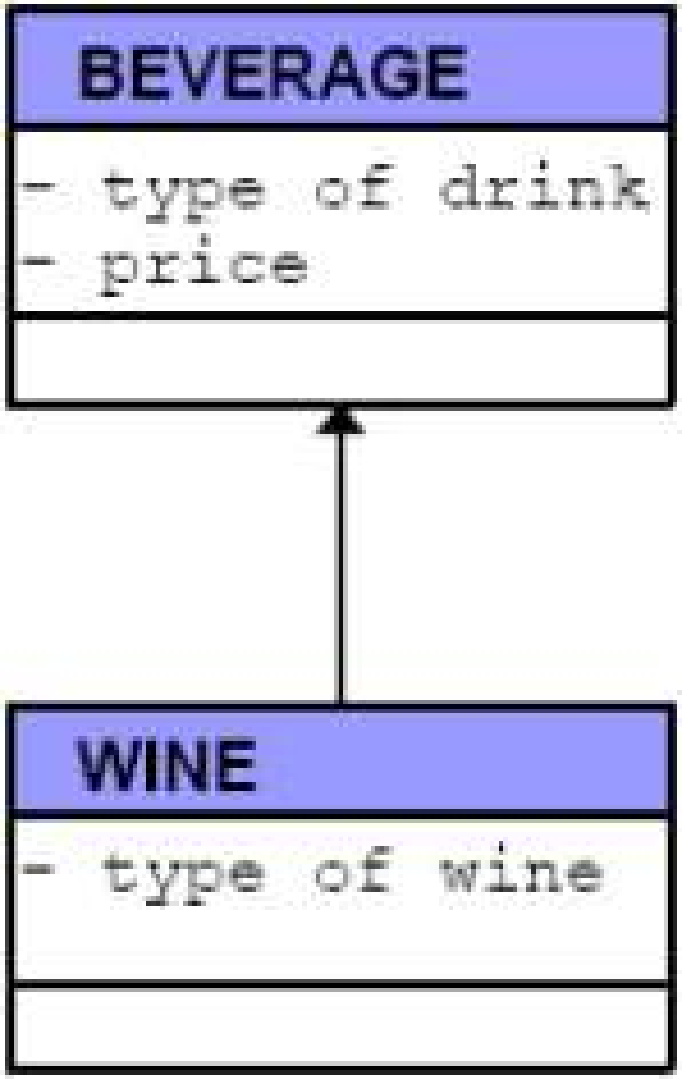
THIẾT KẾ THỪA KẾ

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



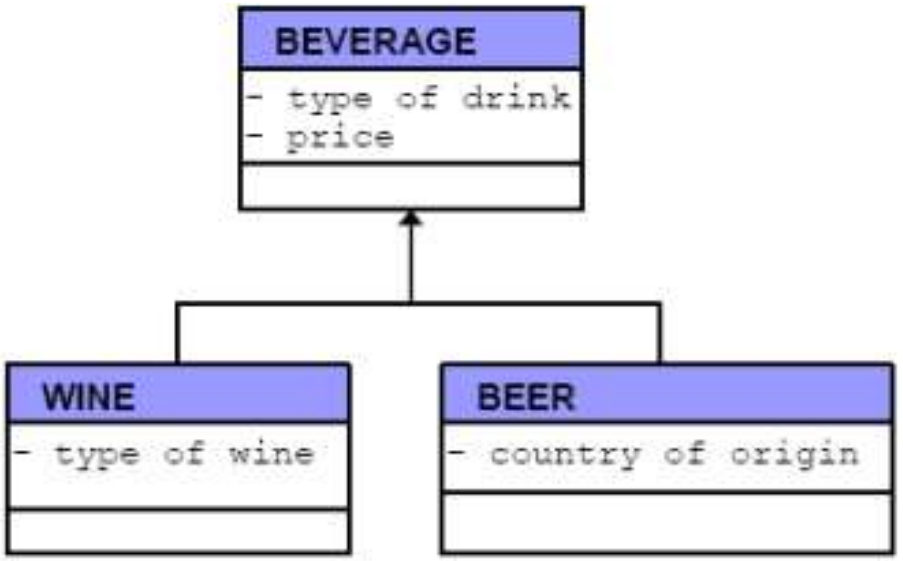
THIẾT KẾ THỪA KẾ

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



THIẾT KẾ THỪA KẾ

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



THIẾT KẾ THỪA KẾ

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

THIẾT KẾ THỪA KẾ

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

CÀI ĐẶT LỚP 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:

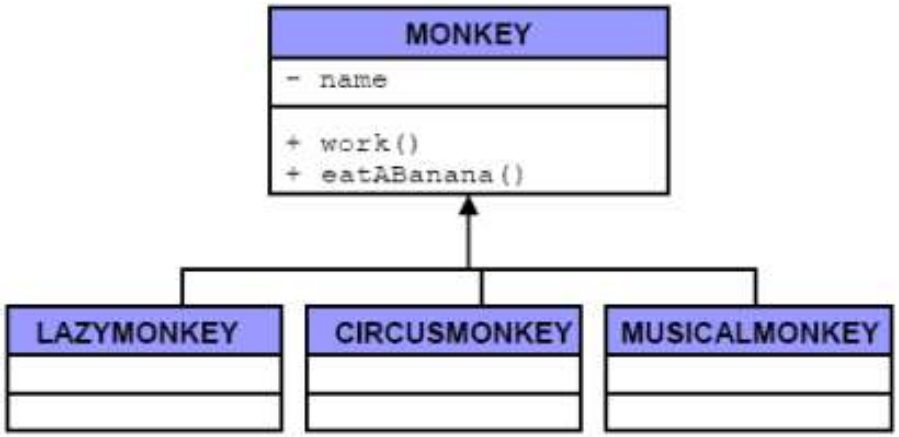
    virtual void MyMethod(...) = 0;
  - Không cần định nghĩa phương thức;
  - Nếu không có "**= 0**" và không định nghĩa, trình biên dịch sẽ báo lỗi.

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;
            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 cấm lớp trừu tượng tạo thể hiện;
- Để sử dụng đoạn mã mặc định này, lớp con cần cung cấp một định nghĩa gọi trực tiếp phiên bản định nghĩa của lớp cơ sở một cách tường minh.

HÀM THUẦN ẢO

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

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

- Tuy nhiên, vì đây là phương thức thuần ảo nên lớp **Monkey** không thể tạo thể hiện;
- Nếu muốn lớp **LazyMonkey** tạo được thể hiện và sử dụng định nghĩa mặc định của **work()**, ta có thể định nghĩa phương thức **work()** của **LazyMonkey** như sau:

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

HÀM THUẦN ẢO

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

Thank You !

IT FACULTY