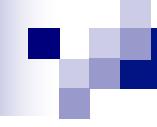


CHƯƠNG 5:

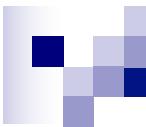
THÙA KẾ (INHERITANCE)

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

- Khái niệm
- Lớp dẫn xuất đơn giản
- Ký hiệu các thứ bậc
- Hàm xây dựng và hàm hủy
- Quyền truy nhập
- Các đối tượng được thừa kế
 - Downcast
 - Upcast
- Đa thừa kế - Sự mơ hồ



Sử dụng lại

■ Vấn đề trùng lắp thông tin:

- Nhiều lớp có thông tin giống nhau.

- Có 2 dạng:

- Dạng chia sẻ: $A \cap B \neq \emptyset$.
 - Dạng mở rộng: $B = A + \varepsilon$.

- Nhược điểm:

- Xây dựng tốn kém.
 - Dung lượng lưu trữ lớn.
 - Thay đổi phần chung khó khăn.

A
x, y, z
a, b, c

B
x, y, z
u, v, w

A
x, y, z

B
x, y, z
a, b, c

Giải quyết: tái sử dụng!!

Sử dụng lại

- Thực tế, 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 tái 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”

Sử dụng lại

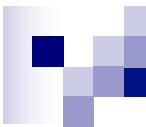
- Copy mã nguồn
 - Tốn công, dễ nhầm
 - Khó sửa lỗi do tồn tại nhiều phiên bản
- Quan hệ *has_a*
 - Sử dụng lớp cũ như là thành phần của lớp mới
 - Sử dụng lại cài đặt với giao diện mới
 - Phải viết lại giao diện
 - Chưa đủ mềm dẻo

Sử dụng lại

- Ví dụ: quan hệ *has_a*

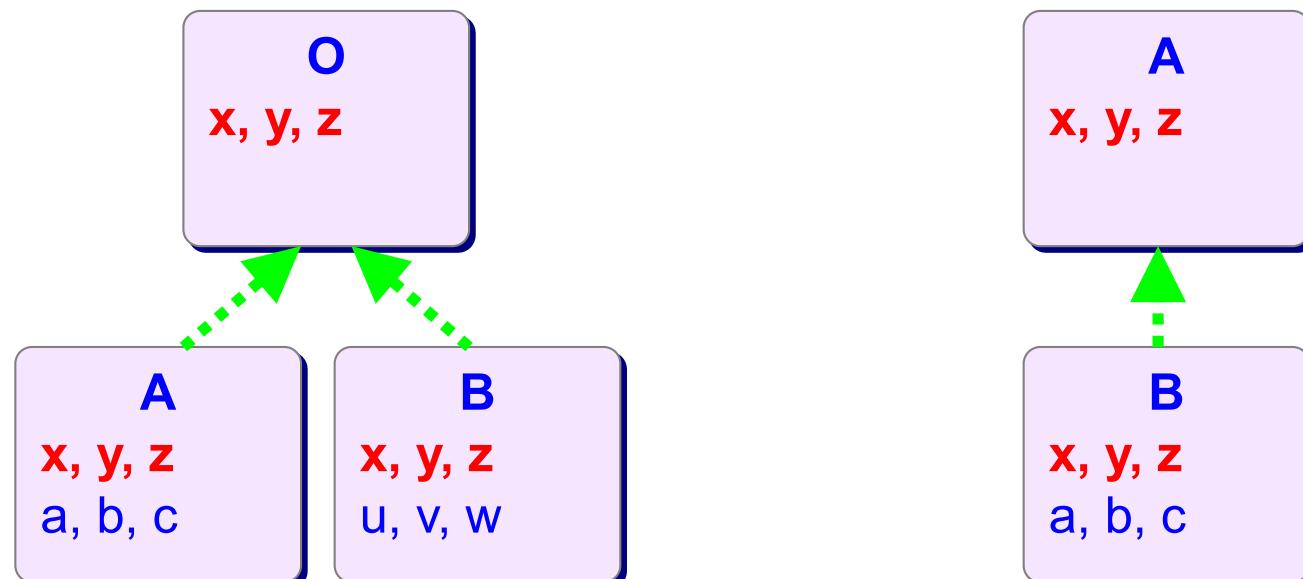
```
class Person
{
    private:
        String name;
        Date bithday;
    public:
        String getName() { return name; }
        //...
};

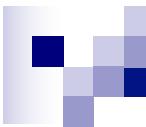
class Employee
{
    private:
        Person me;
        double salary;
    public:
        String getName() { return me.getName(); }
        //...
};
```



Khái niệm kế thừa

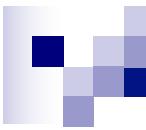
- Định nghĩa lớp mới dựa trên những lớp đã có.
- **Lớp cơ sở:** lớp dùng để định nghĩa lớp mới.
- **Lớp kế thừa:** lớp được định nghĩa từ lớp đã có.
- Lớp kế thừa thừa hưởng **TẤT CẢ** từ lớp cơ sở.





Khái niệm 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.



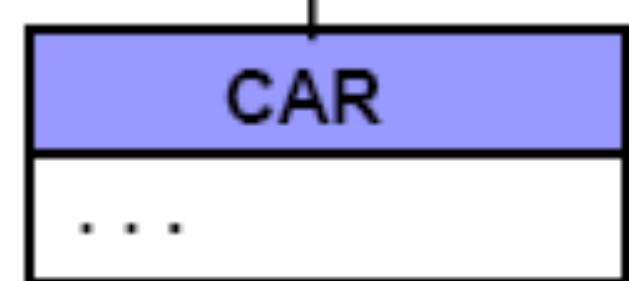
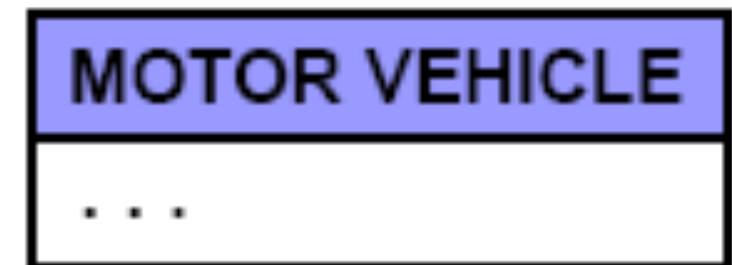
Khái niệm kế thừa

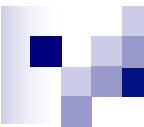
Lớp cơ sở
(Base class)

LỚP CHA
(Super class)

Lớp dẫn xuất
(Derived class)

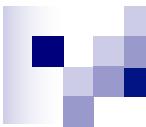
LỚP CON
(Sub class)





Khái niệm kế thừa

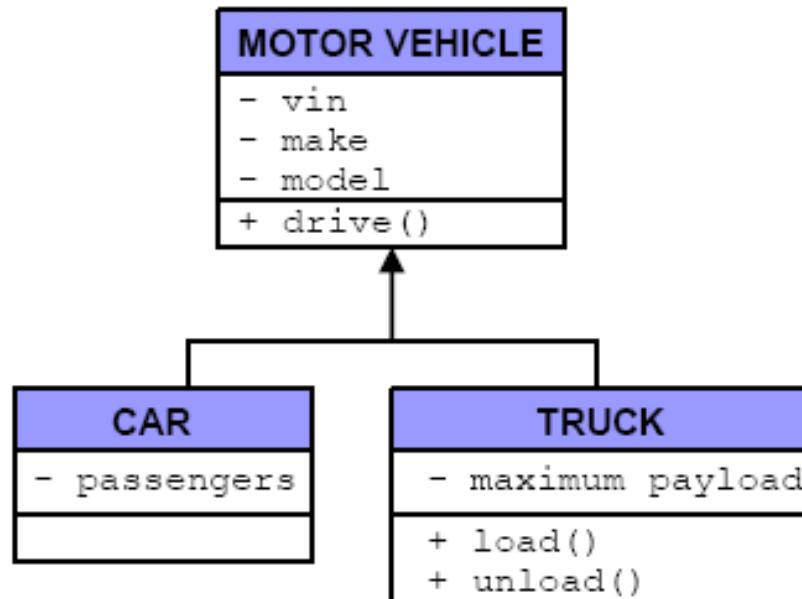
- Lớp cha – superclass (hoặc lớp cơ sở - base class)
 - lớp tổng quát hơn trong một quan hệ “là”
 - các đối tượng thuộc lớp cha có cùng tập thuộc tính và hành vi S
- Lớp con – subclass (hoặc lớp dẫn xuất – derived class)
 - lớp cụ thể hơn trong một quan hệ “là”
 - các đối tượng thuộc lớp con có cùng 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.



Sơ đồ quan hệ đối tượng

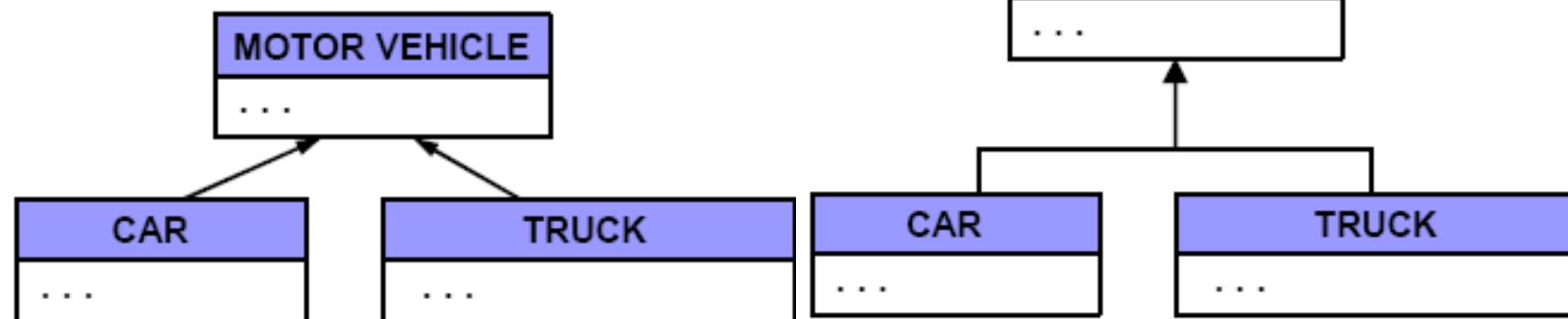
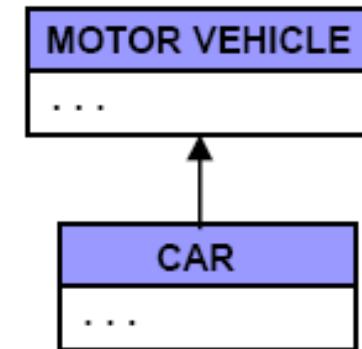
Object Relationship Diagram - ORD

- 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?



Sơ đồ quan hệ đối tượng

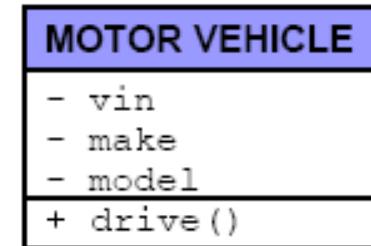
- Biểu diễn một quan hệ thừa kế giữa hai lớp bằng một mũi tên trỏ từ lớp con đến lớp cha
- Có thể biểu diễn quan hệ với nhiều lớp con theo một trong hai kiểu sau:



Biểu diễn sơ đồ quan hệ

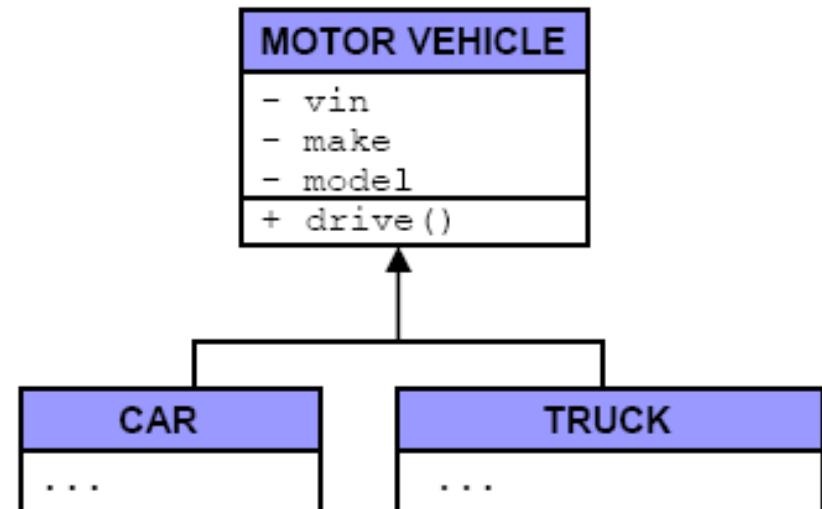
■ Biểu diễn các thuộc tính và hành vi

- Giả sử lớp MotorVehicle có các thuộc tính *vin* (số đăng ký xe), *make* (hãng), *model* (kiểu), và hành vi *drive* (lái)



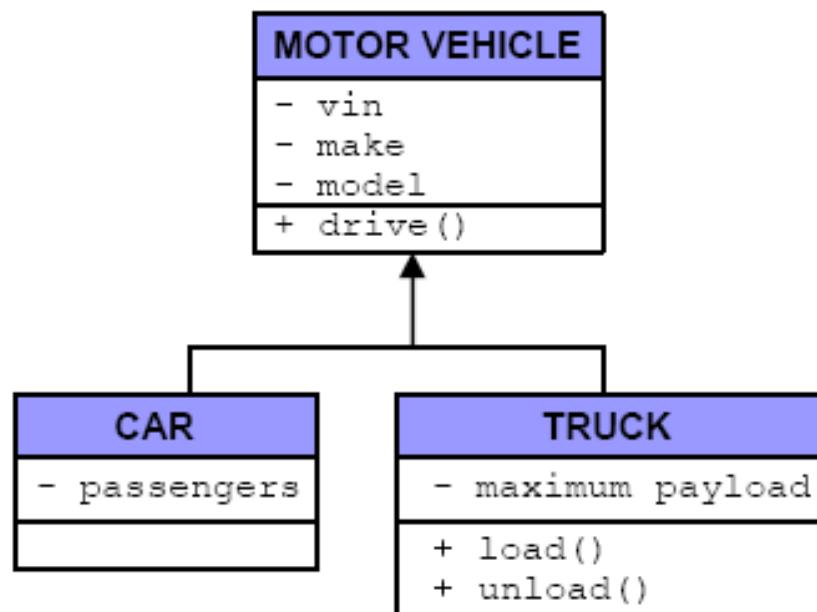
■ Ta có sơ đồ quan hệ

- mọi xe tải, xe ca đều có các thuộc tính *vin*, *make*, *model*, và hành vi *drive*

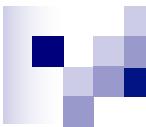


Sơ đồ quan hệ đối tượng

- Ta có sơ đồ

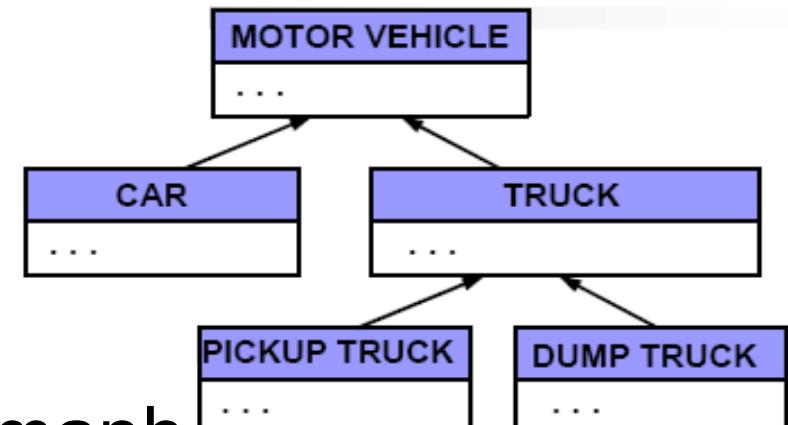


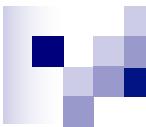
- mỗi xe ca đều có các thuộc tính *vin*, *make*, *model*, và hành vi *drive*, kèm theo thuộc tính *passengers*
- mỗi xe tải đều có các thuộc tính *vin*, *make*, *model*, và hành vi *drive*, kèm theo thuộc tính *maximum payload* và các hành vi *load*, *unload*



Cây thừa kế

- 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.

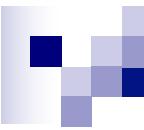




Định nghĩa thừa kế

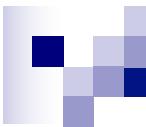
```
class MyDerivedClass :<keyword> MyBaseClass {  
    private:  
        // Khai báo thuộc tính mới của MyDerivedClass.  
    public:  
        // Khai báo phương thức mới của MyDerivedClass.  
    };  
};
```

- Keyword: là kiểu kế thừa:
 - public, private, protected.



Định nghĩa thừa kế

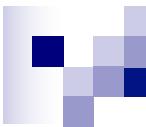
- Mô tả một lớp con cũng giống như biểu diễn nó trong ORD, ta chỉ tập trung vào những điểm khác với lớp cha
- Ích lợi
 - đơn giản hóa khai báo lớp,
 - hỗ trợ nguyên lý đóng gói của hướng đối tượng
 - hỗ trợ tái sử dụng code (sử dụng lại định nghĩa của các thành viên dữ liệu và phương thức)
 - việc che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



Định nghĩa thừa kế

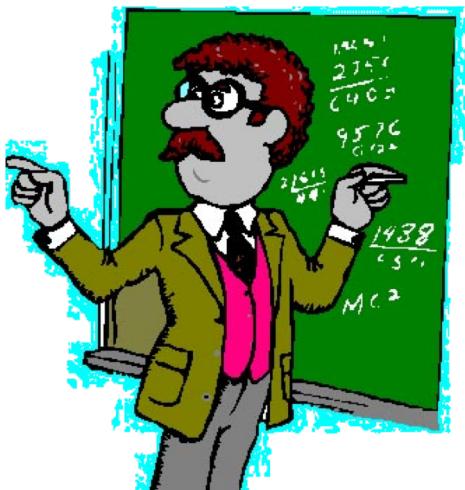
Ví dụ: is_a

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



Định nghĩa thừa kế

■ Ví dụ:



Giáo viên

Thông tin:
Họ tên.
Mức lương.
Số ngày nghỉ.

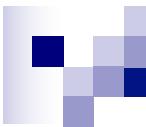
Công việc:
Giảng dạy.
Tính lương.

Thông tin:
Họ tên.
Mức lương.
Số ngày nghỉ.
Lớp chủ nhiệm.

Công việc:
Giảng dạy.
Tính lương.
Sinh hoạt chủ nhiệm.



GVCN



Định nghĩa thừa kế

■ Ví dụ:

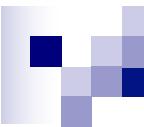
```
class GiaoVien
{
private:
    char *m_sHoTen;
    float m_fMucLuong;
    int m_iSoNgayNghi;
public:
    GiaoVien(char *sHoTen,
              float fMucLuong,
              int iSoNgayNghi);
    void giangDay();
    float tinhLuong();
```

Lớp kế thừa

```
class GVCN : public GiaoVien
{
private:
    char *m_sLopCN;
public:
    GVCN(char *sHoTen,
          float fMucLuong,
          int iSoNgayNghi,
          char *sLopCN);
    void sinhHoatCN();
};
```

Lớp cơ sở

GVCN thừa hưởng **TẤT CẢ**
thuộc tính và phương thức
của **GiaoVien**



Định nghĩa thừa kế

■ Ví dụ:

```
void main()
```

```
{
```

```
    GiaoVien gv1("Minh", 500000, 5);
```

```
    gv1.giangDay();
```

```
    float fLuong1 = gv1.tinhLuong();
```

```
    GVCN gv2("Hanh", 700000, 3);
```

```
gv2.giangDay();
```

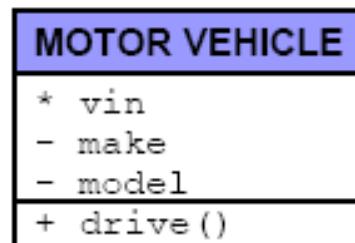
```
    gv2.sinhHoatCN();
```

```
    float fLuong2 = gv2.tinhLuong();
```

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;
}

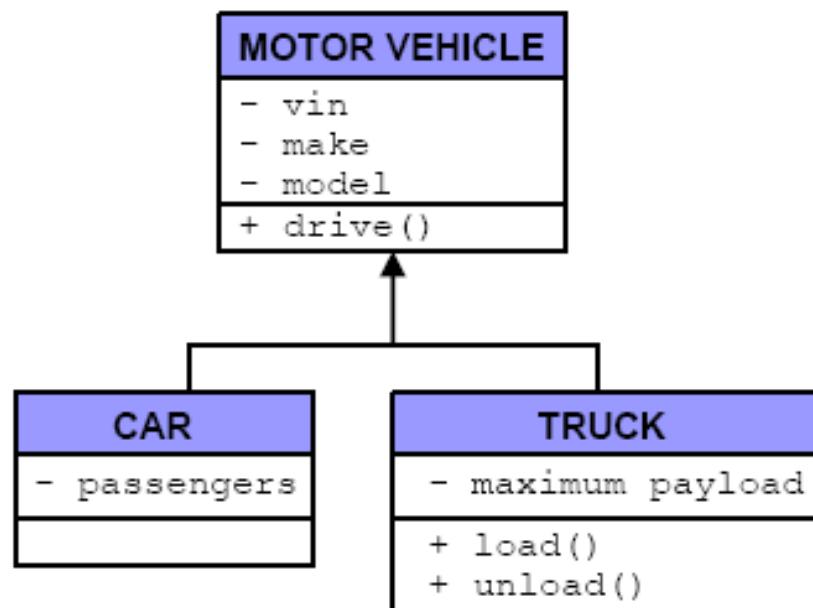
// We could actually use // the default destructor
MotorVehicle::~MotorVehicle() { /*...*/ }

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

- 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.

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

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

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:

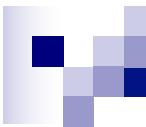
```
Car::Car(int vin, string make, string model, int passengers)
        :MotorVehicle(vin,make,model)
{
    this->passengers = passengers;
}
```

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**

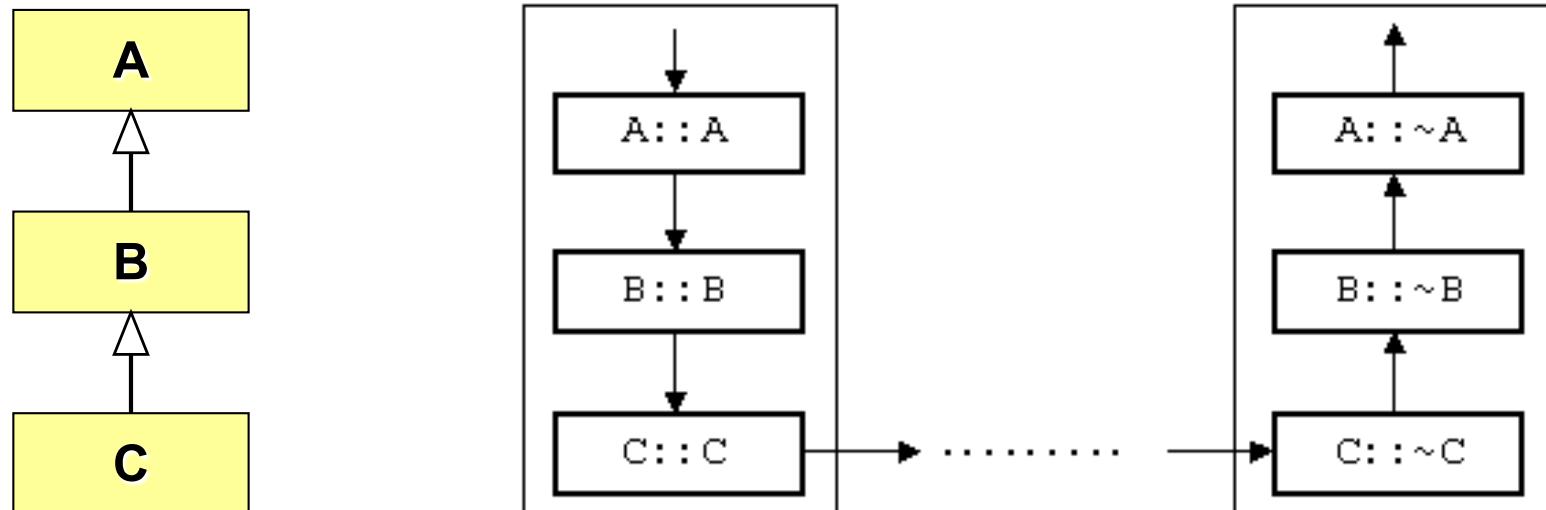
Định nghĩa lớp dẫn xuất

- 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;
};
```

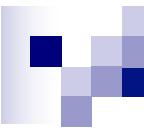
- HinhTron r;
- HinhTron t(200, 200, 50);

Quyền truy cập (Access privilege)

- 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

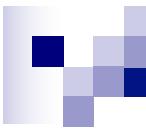
```
class Car : public MotorVehicle
{
    public:
        Car (...);
        ~Car();
    private:
        int passengers;
};
```

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



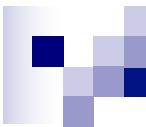
Quyền truy nhập

- 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 nhập

- 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.

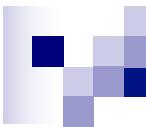


Quyền truy nhập

- 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 thẻ 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**

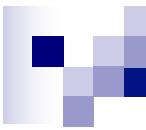
```
class MotorVehicle
{
    //...
private:
    int vin;
    string make;
    string model;
};

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



Quyền truy nhập

- Tuy quy định quyền truy nhập như trên có vẻ kỳ cục, nhưng nó đảm bảo tính đóng gói
 - nếu không, ta có thể lấy được quyền truy nhập vào cấu trúc bên trong của một lớp chỉ bằng cách tạo một lớp dẫn xuất của lớp đó
- Trường hợp nếu ta xây dựng lớp cơ sở và cố ý muốn “cấp” cho lớp dẫn xuất quyền truy nhập tới một số thành viên/phương thức,
 - từ khoá `protected` quy định quyền truy nhập cho các thành viên/phương thức sẽ được sử dụng bởi lớp dẫn xuất



Quyền truy nhập

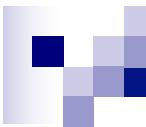
- Từ khoá **protected** dùng để cấp quyền truy nhập tới thành viên/phương thức của một lớp cho các lớp dẫn xuất (và chỉ các lớp dẫn xuất) của lớp đó
- Đối với mọi đối tượng khác của C++, các thành viên/phương thức **protected** được coi như **private** (chúng đều không thể truy nhập được)
- Tuy tiêu chí đóng gói khuyên nên để mọi thứ **private**, nhưng khi tạo các cây thừa kế, người ta thường hay cấp quyền truy nhập **protected**

Quyền truy nhập

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

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

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



Quyền truy nhập

- Quay lại việc sử dụng các từ khoá truy nhập để quy định “kiểu” thừa kế

```
class MotorVehicle { ... };
```

```
class Car : protected MotorVehicle { ... };
```

- Khi dùng kiểu thừa kế `protected` cho lớp con `Car`, quan hệ “`Car` thừa kế `MotorVehicle`” sẽ được nhìn thấy từ
 - mọi phương thức bên trong `Car`,
 - mọi phương thức thuộc các lớp con của `Car`
- Tuy nhiên, mọi đối tượng khác của C++ không nhìn thấy quan hệ này

Quyền truy nhập

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  
    .....  
};
```

Overriding

■ Kế thừa một phần:

- Không kế thừa “máy móc” tất cả.
- Lớp kế thừa có thể thay đổi những gì đã kế thừa!!
 - Định nghĩa lại phương thức đã kế thừa.
 - gọi là **Overiding**

Lớp kế thừa thừa hưởng **TẤT CẢ** thuộc tính và phương thức của lớp cơ sở **TRÙ** nhưng phương thức được định nghĩa lại!!



Overriding

- 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:

- GVCN kế thừa từ GiaoVien.
- GVCN tính lương khác GiaoVien.
 - Lương GV = Mức lương – Số ngày nghỉ * 10000.
 - Lương GVCN = Lương GV + Phụ cấp 50000.

→ Viết lại phương thức tinhLuong() cho lớp GVCN.

Overiding

■ Ví dụ:

```
class GiaoVien
{
private:
    char    *m_sHoTen;
    float   m_fMucLuong;
    int     m_iSoNgayNghi;
public:
    GiaoVien(char *sHoTen, float fMucLuong, int iSoNgayNghi);
    void giangDay();
    float tinhLuong()
    {
        return m_fMucLuong – m_iSoNgayNghi * 10000;
    }
};
```

Overiding

■ Ví dụ:

```
class GVCN : public GiaoVien
{
private:
    char    *m_sLopCN;
public:
    GVCN(char *sHoTen,
          float fMucLuong,
          int iSoNgayNghi,
          char *sLopCN);
    void sinhHoatCN();
    float tinhLuong()
    {
        return GiaoVien::TinhLuong() + 50000;
    }
};
```

```
void main()
{
    GiaoVien  gv1("Minh", 500000, 5);
    gv1.giangDay();
    float fLuong1 = gv1.tinhLuong();

    GVCN  gv2("Hanh", 700000, 3);
    gv2.giangDay();
    float fLuong2 = gv2.tinhLuong();
}
```

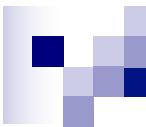
Overiding – Ví dụ

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

```
class Ellipse{  
    //...  
    public:  
        //...  
        void rotate(double rotangle) { //... }  
};
```

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

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



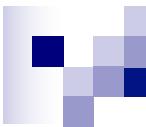
Quan hệ IS-A và HAS-A

■ Quan hệ IS-A:

- Lớp A quan hệ **IS-A** với lớp B
 - A là một trường hợp đặc biệt của B.
 - A cùng loại với B.

■ Ví dụ:

- GVCN là một GiaoVien đặc biệt.
- HinhVuong là một HinhChuNhat đặc biệt.
- ConMeo là một ConVat đặc biệt.



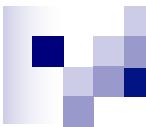
Quan hệ IS-A và HAS-A

■ Quan hệ HAS-A:

- Lớp A quan hệ **HAS-A** với lớp B
 - A bao hàm B.
 - A chứa B.
 - B là một bộ phận của A.

■ Ví dụ:

- ChiecXe chứa BanhXe.
- QuyenSach chứa ChuongSach.



Quan hệ IS-A và HAS-A

■ Dr. Guru khuyên: luật xây dựng lớp.

- A có quan hệ IS-A với B.
→ Cho A thừa kế B.
- A có quan hệ HAS-A với B.
→ Cho B là một thuộc tính của A.

■ Ví dụ:

```
class ConMeo : public ConVat { };  
class ChiecXe  
{  
private:  
    BanhXe *m_pBanhXe;  
};
```

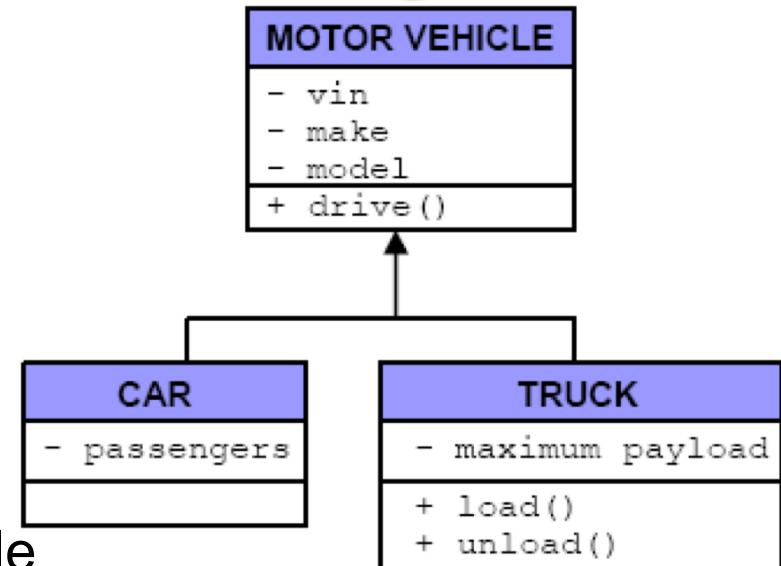


Các đối tượng được thừa kế trong C++

- 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 thừa kế 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 (về tổng quát)
 - 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 car. Nhưng trong cây thừa kế của ta, không có gì đòi hỏi rằng mọi xe chạy bằng máy đều là xe car



Các đối tượng được thừa kế 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 thừa kế 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 thừa kế trong C++

- Trong cây thừa kế **MotorVehicle**, giả sử mọi thành viên dữ liệu đều được khai báo **protected**, và 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 thừa kế 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 thừa kế trong C++

- Ta có thể khai báo các thể hiện của các lớp đó và sử dụng chúng như thế nào?
 - Thí 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 thừa kế 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
 - Thí dụ:

```
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
```

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 con trỏ (hoặc tham chiếu) tới lớp cơ sở
 - 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 thuộc lớp cơ sở để trỏ tới một thể hiện của lớp dẫn xuất, trình biên dịch sẽ chỉ cho ta xem thể hiện này như là đối tượng thuộc lớp cơ sở
 - Do đó, 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 trả 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

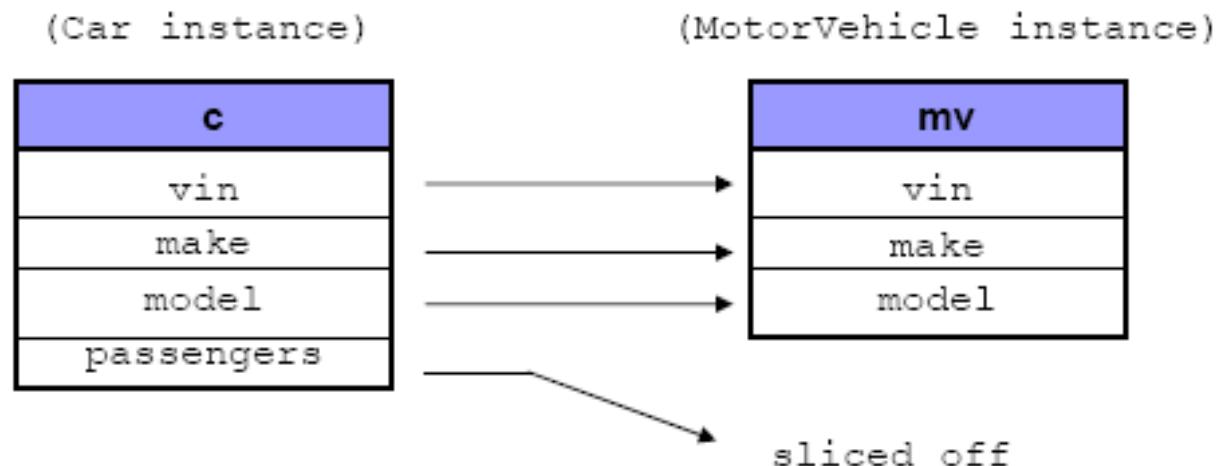
Slicing

■ 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 tương 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

- Nếu ta biết chắc chắn rằng một con trỏ lớp cơ sở quả thực đang trỏ tới một lớp con, ta có thể tự đổi kiểu cho con trỏ lớp cơ sở bằng cách sử dụng **static_cast**

```
Car* cPointer = new Car(10,"Honda","S2000",2);
MotorVehicle *mv=cPointer; //Upcast
Car* cPointer2;
cPointer2 = static_cast<Car *> ( mv ); // (car*) mv
```

- Ta có thể thấy mối nguy hiểm khi làm việc này - chuyện gì xảy ra nếu đối tượng ta đang cố đổi kiểu thực ra không thuộc lớp mà ta nghĩ?

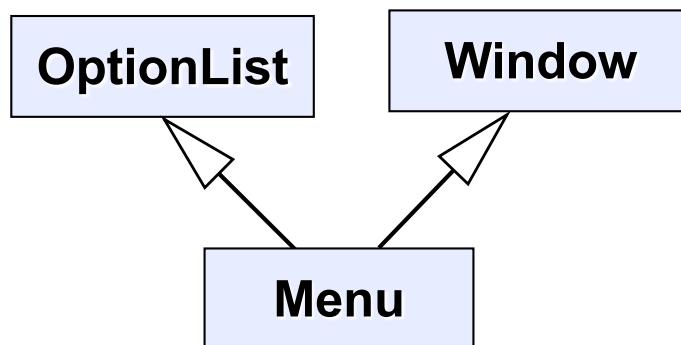
```
Truck* tPointer = new Truck(10, "Toyota", "Tacoma", 5000);
MotorVehicle* mv = tPointer; // Upcast
Car* cPointer2;
cPointer2 = static_cast<Car*>(mv); // Explicit downcast
```

Downcast

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

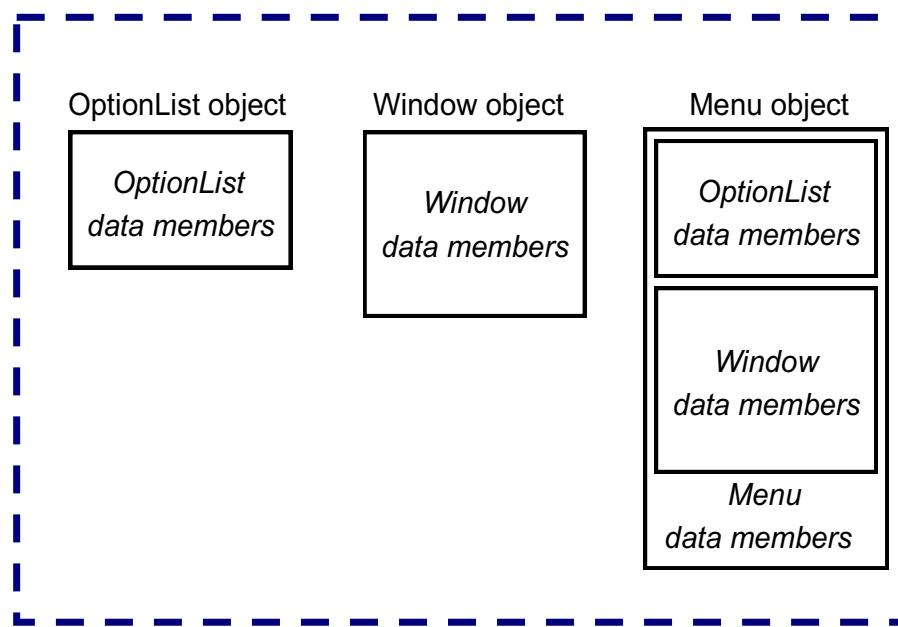
- Đoạn mã trên hoàn toàn hợp lệ và sẽ được trình biên dịch chấp nhận
- Tuy nhiên, nếu chạy đoạn trình trên, chương trình có thể bị lỗi vỡ (thường là khi lần đầu truy nhập đến thành viên/phương thức được định nghĩa của lớp dẫn xuất mà ta đổi mới)
- Ví dụ: Point-Circle

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 mờ hồ trong đa thừa kế

```
class OptionList {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Window {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Menu : public OptionList,  
           public Window  
{ ..... };
```

Hàm cùng tên

Chỉ rõ hàm
của lớp nào

Gọi
hàm
của lớp
nào?

```
void main() {  
    Menu m1(...);  
    m1.Highlight(10);  
    ....  
}
```

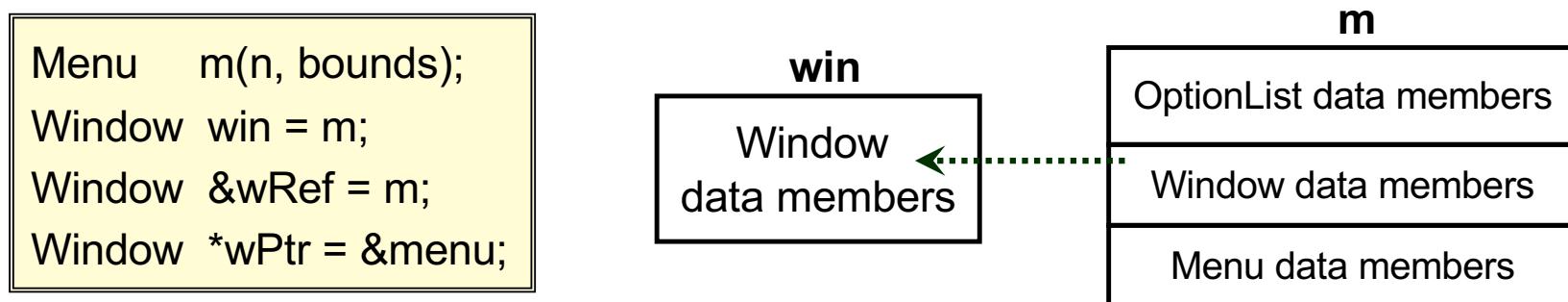
xử lý

```
void main() {  
    Menu m1(...);  
    m1.OptionList::Highlight(10);  
    m1.Window::Highlight(20);  
    ....  
}
```

Chuyển kiểu

- Có sẵn 1 phép chuyển kiểu không tường minh:

- Đổi tượng lớp cha = Đổi tượng lớp con;
 - Áp dụng cho cả đối tượng, tham chiếu và con trỏ.



- Không được thực hiện phép gán ngược:

- Đổi tượng lớp con = Đổi tượng lớp cha; // **SAI**

Nếu muốn thực hiện
phải tự định nghĩa
phép ép kiểu

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

};

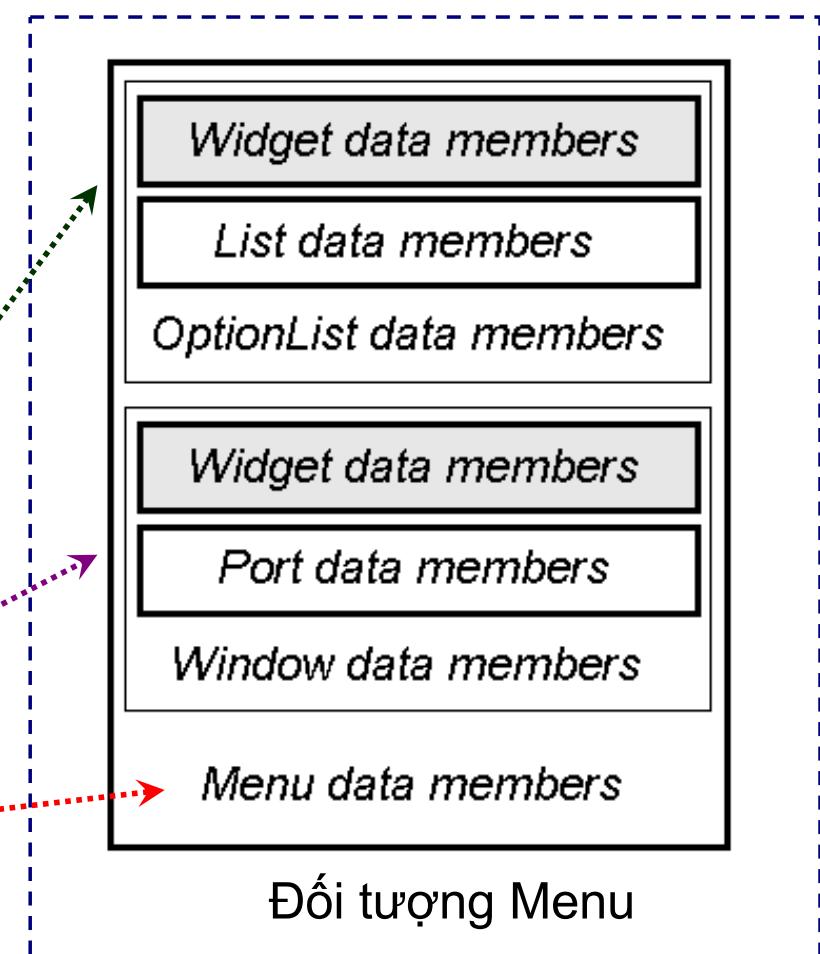
Lớp cơ sở ảo

■ Sự mơ hồ - dữ thừa dữ liệu

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

class Window
    : public Widget, Port
{ /* ... */ };

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



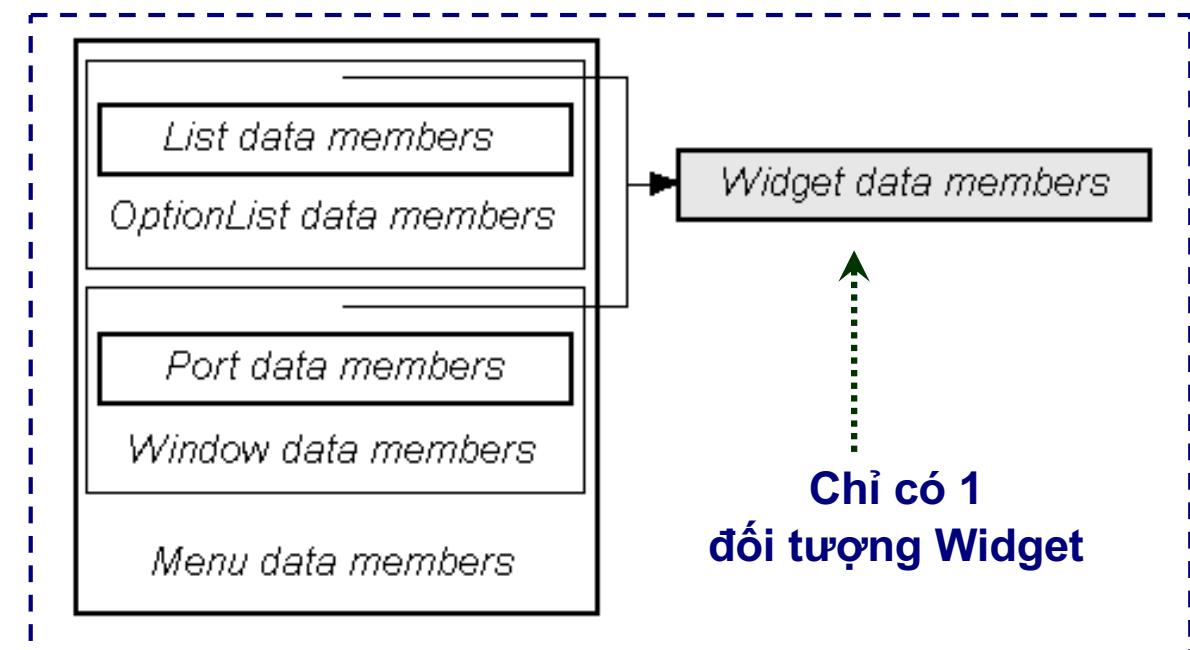
Lớp cơ sở ảo (tt)

■ Cách xử lý: dùng lớp cơ sở ảo.

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

class Window
    : virtual public Widget,
      public Port
{ /*...*/};

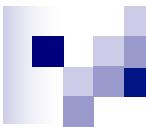
class Menu
    : public OptionList,
      public Window
{ /*...*/};
```



```
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 xây dựng sao chép:
$$Y::Y \text{ (**const Y&**)}$$
 - Phép gán:
$$Y\& Y::\text{operator } = \text{ (**const Y&**)}$$
- Nếu không định nghĩa, sẽ tự động có hàm xây 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**.



BÀI TẬP



Bài tập

■ Bài tập 5.1:

Những cặp đối tượng sau có quan hệ IS-A hay HAS-A?

Khai báo lớp cho từng cặp thể hiện quan hệ giữa chúng.

- Hình vuông / Hình chữ nhật.
- Đa giác / Cạnh.
- Giám đốc / Nhân viên.
- Hình tròn / Hình Ellipse.
- Máy bay / Động cơ.
- Câu / Tùy.
- Mỹ phẩm / Hàng hóa.
- Cây lúa / Cây lương thực.
- Thư viện / Sách.
- Phim hoạt hình / Phim ảnh.

Bài tập

■ Bài tập 5.2: Cho lớp **TaiKhoan**:

```
class TaiKhoan
{
private:
    float    m_fSoDu = 0;
public:
    float baoSoDu() { return m_fSoDu;  }
    void napTien(float fSoTien) {  m_fSoDu += fSoTien;  }
    void rutTien(float fSoTien)
    {
        if (fSoTien <= m_fSoDu)
            m_fSoDu -= fSoTien;
    }
};
```

Bài tập

■ Bài tập 5.2:

Dựa trên lớp **TaiKhoan**, xây dựng lớp **TaiKhoanTietKiem** như sau:

- Có thêm thông tin:
 - Kỳ hạn gửi.
 - Lãi suất.
 - Số tháng đã gửi.
- Khi nạp tiền, số tháng đã gửi được tính lại từ đầu.
- Chỉ được rút tiền khi đến kỳ hạn.
- Cho phép tăng số tháng đã gửi.
- Tính số dư tại thời điểm hiện tại.

Bài tập

■ Bài tập 5.3:

Một chiếc xe máy chạy 100km tốn 2lit xăng, cứ chở thêm 10kg hàng xe tốn thêm 0.1lit xăng.

Một chiếc xe tải chạy 100km tốn 20lit xăng, cứ chở thêm 1000kg hàng xe tốn thêm 1lit xăng.

Dùng kế thừa xây dựng lớp **XeMay** và **XeTai** cho phép:

- Chất một lượng hàng lên xe.
- Bỏ bớt một lượng hàng xuống xe.
- Đổ một lượng xăng vào xe.
- Cho xe chạy một đoạn đường.
- Kiểm tra xem xe đã hết xăng chưa.
- Cho biết lượng xăng còn trong xe.