

CHƯƠNG 5. KẾ THỪA



Visual C++

Nội dung

1

Quan hệ giữa các lớp đối tượng

2

Kế thừa

3

Kế thừa đơn

4

Phạm vi truy xuất trong kế thừa

5

Đa kế thừa

Bài Toán

- Giả sử Công ty có hai loại nhân viên: Nhân viên văn phòng và Nhân viên sản xuất. Viết chương trình quản lý và tính lương cho từng nhân viên của công ty:
- Mỗi nhân viên cần quản lý các thông tin sau: Họ tên, ngày sinh, lương
- Công ty cần tính lương cho nhân viên như sau:
 - Đối với nhân viên sản xuất:
 - $Lương = \text{lương căn bản} + \text{số sản phẩm} * 5.000$
 - Đối nhân viên văn phòng:
 - $Lương = \text{số ngày làm việc} * 100.000$
- Viết chương trình cho phép nhập danh sách các nhân viên, xuất ra các thông tin của nhân viên và tính tổng lương phải trả cho các nhân viên. Yêu cầu: lớp NhanVien có khởi tạo mặc định và khởi tạo có 3 tham số, các lớp NVVP và NVSX có khởi tạo mặc định và khởi tạo với đầy đủ các tham số để khởi tạo cho các thuộc tính.

Bài Toán

- Giả sử Công ty có hai loại nhân viên: Nhân viên văn phòng và Nhân viên sản xuất. Viết chương trình quản lý và tính lương cho từng nhân viên của công ty:
- Mỗi nhân viên cần quản lý các thông tin sau: Họ tên, ngày sinh, lương
- Công ty cần tính lương cho nhân viên như sau:
 - Đối với nhân viên sản xuất:
 - $Lương = \text{lương căn bản} + \text{số sản phẩm} * 5.000$
 - Đối nhân viên văn phòng:
 - $Lương = \text{số ngày làm việc} * 100.000$
- Viết chương trình cho phép hiển thị cửa sổ danh sách nhân viên (họ tên, ngày sinh, lương). Cửa sổ này cho phép tìm kiếm (theo tên, ngày sinh, lương), thêm 1 nhân viên mới. Cửa sổ có nút xem chi tiết thông tin một nhân viên khi người dùng đang chọn 1 dòng tương ứng trong bảng. Cửa sổ mới hiển thị ra thông tin chi tiết nhân viên và cho phép chỉnh sửa thông tin hoặc xóa nhân viên đó.

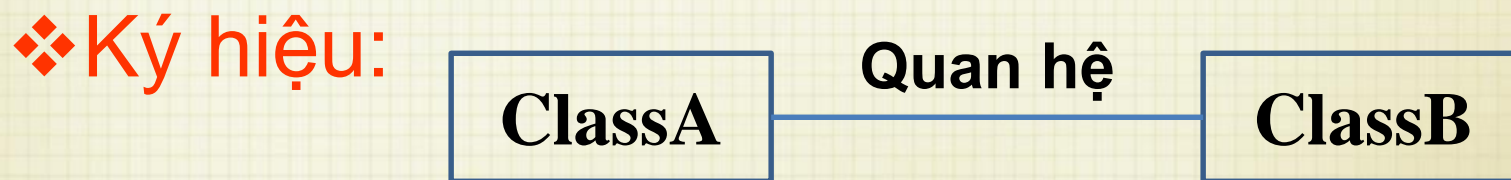
Quan hệ giữa các lớp đối tượng

❖ Giữa các lớp đối tượng có những loại quan hệ sau:

- Quan hệ một một (1-1)
- Quan hệ một nhiều (1-n)
- Quan hệ nhiều nhiều (n-n)
- Quan hệ đặc biệt hóa, tổng quát hóa

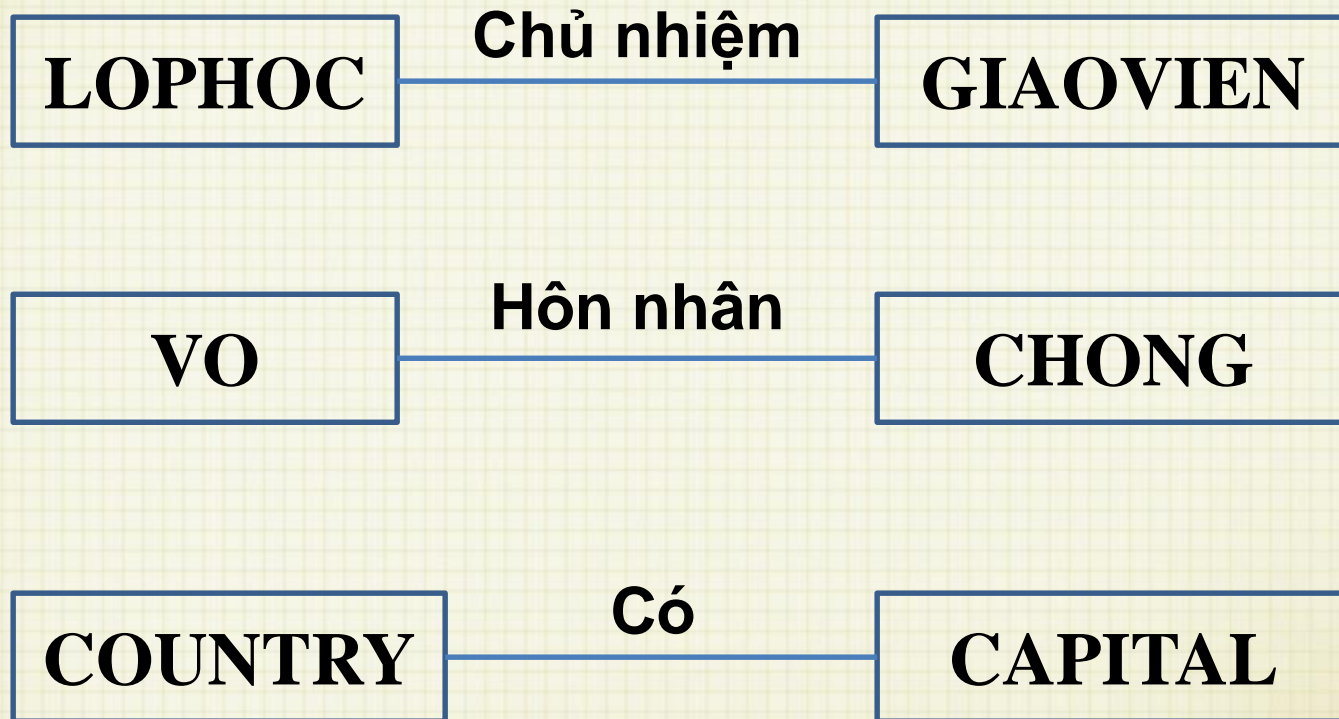
Quan hệ một một (1-1)

❖ **Khái niệm:** Hai lớp đối tượng được gọi là có **quan hệ một-một** với nhau khi một đối tượng thuộc lớp này quan hệ với một đối tượng thuộc lớp kia và một đối tượng thuộc lớp kia có quan hệ duy nhất với một đối tượng thuộc lớp này.



Quan hệ một một (1-1)

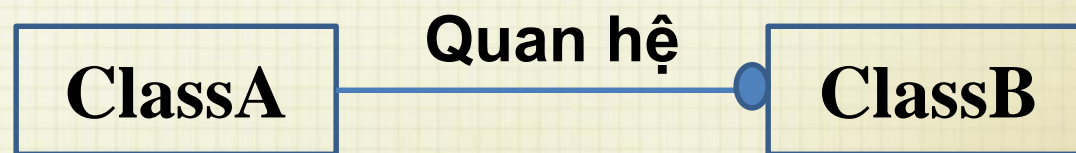
❖ Ví dụ:



Quan hệ một nhiều (1-n)

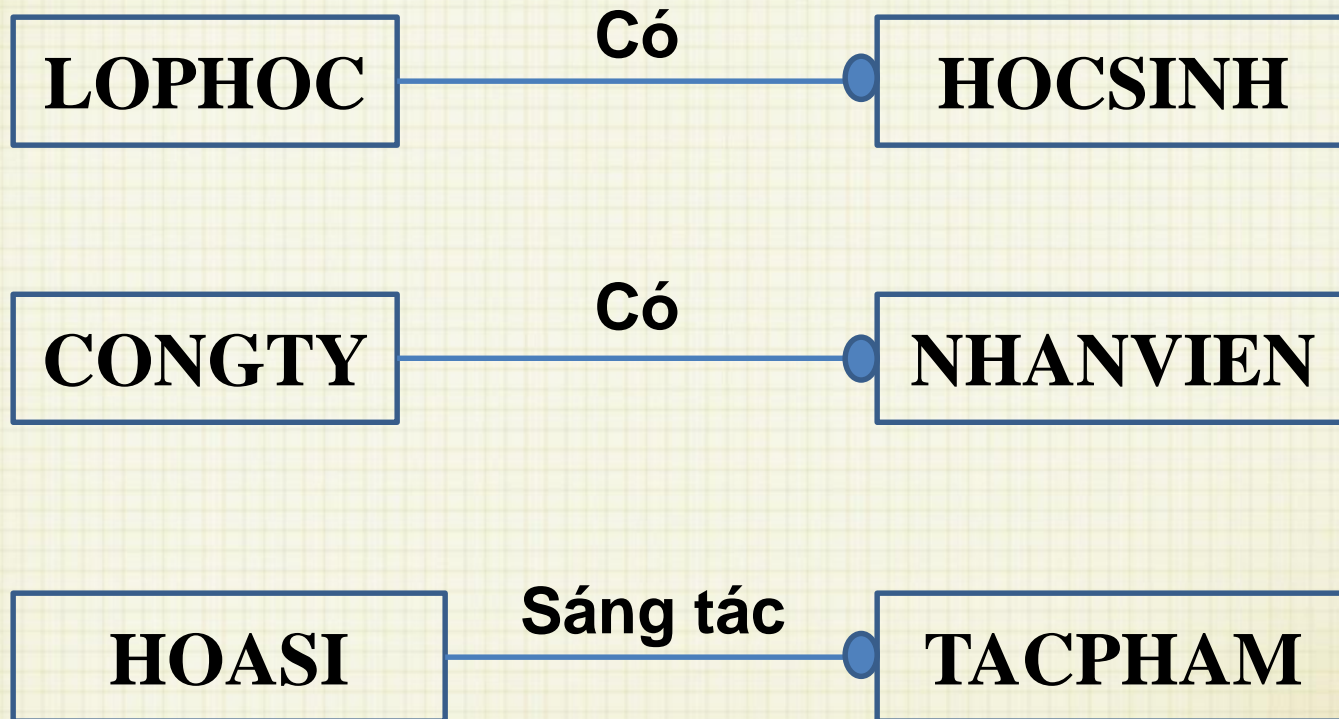
❖ **Khái niệm:** Hai lớp đối tượng được gọi là có **quan hệ một-nhiều** với nhau khi một đối tượng thuộc lớp này quan hệ với nhiều đối tượng thuộc lớp kia và một đối tượng lớp kia có quan hệ duy nhất với một đối tượng thuộc lớp này.

❖ **Kí hiệu:**



Quan hệ một nhiều (1-n)

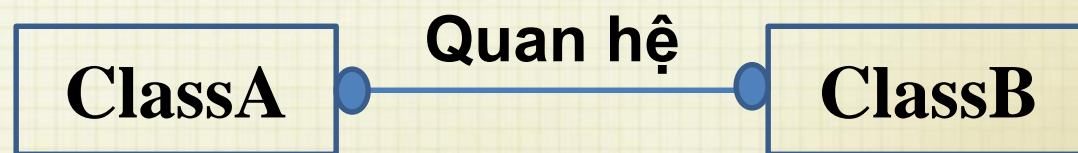
❖ Ví dụ:



Quan hệ nhiều nhiều (n-n)

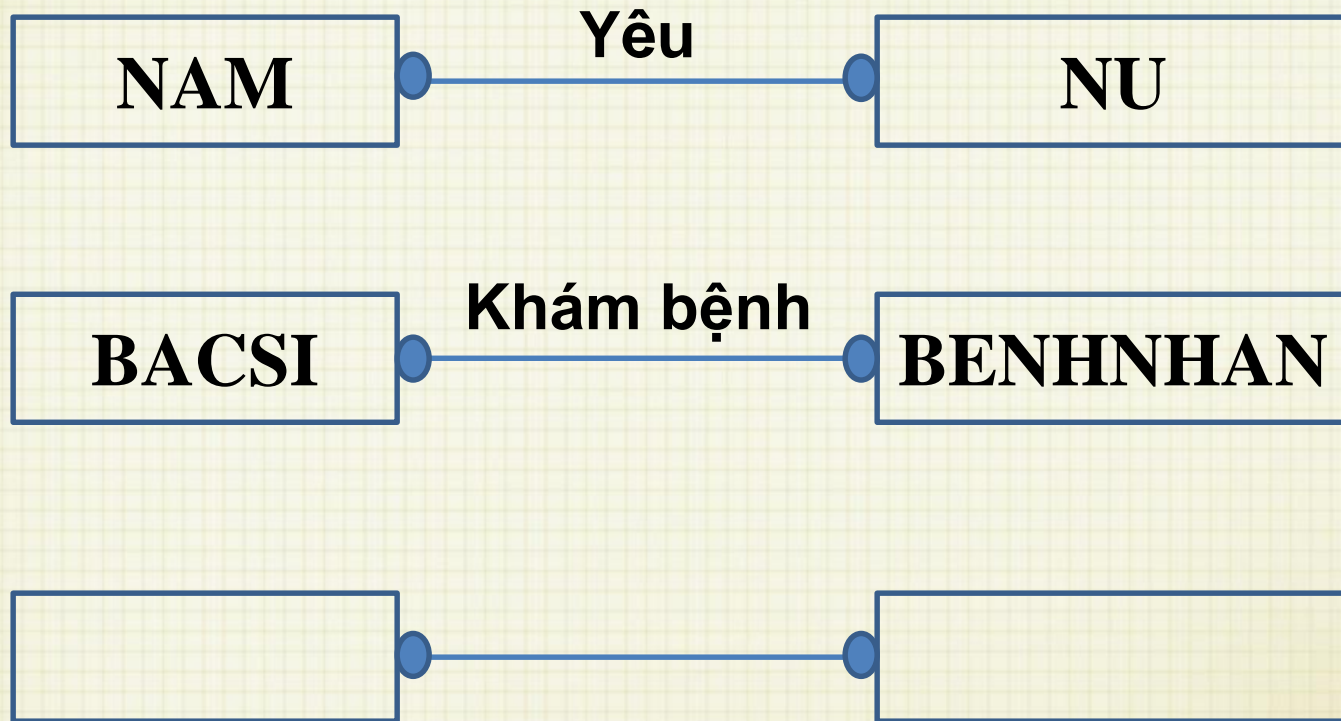
❖ **Khái niệm:** hai lớp đối tượng được gọi là **quan hệ nhiều-nhiều** với nhau khi một đối tượng thuộc lớp này có quan hệ với nhiều đối tượng thuộc lớp kia và một đối tượng lớp kia cũng có quan hệ với nhiều đối tượng thuộc lớp này.

❖ **Kí hiệu**



Quan hệ nhiều nhiều (n-n)

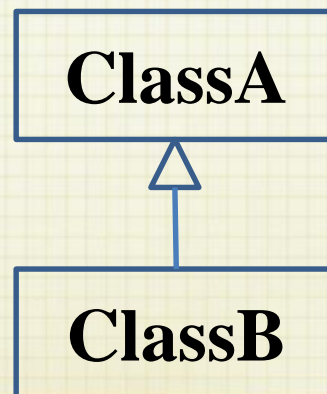
❖ Ví dụ



Quan hệ đặc biệt hóa – tổng quát hóa

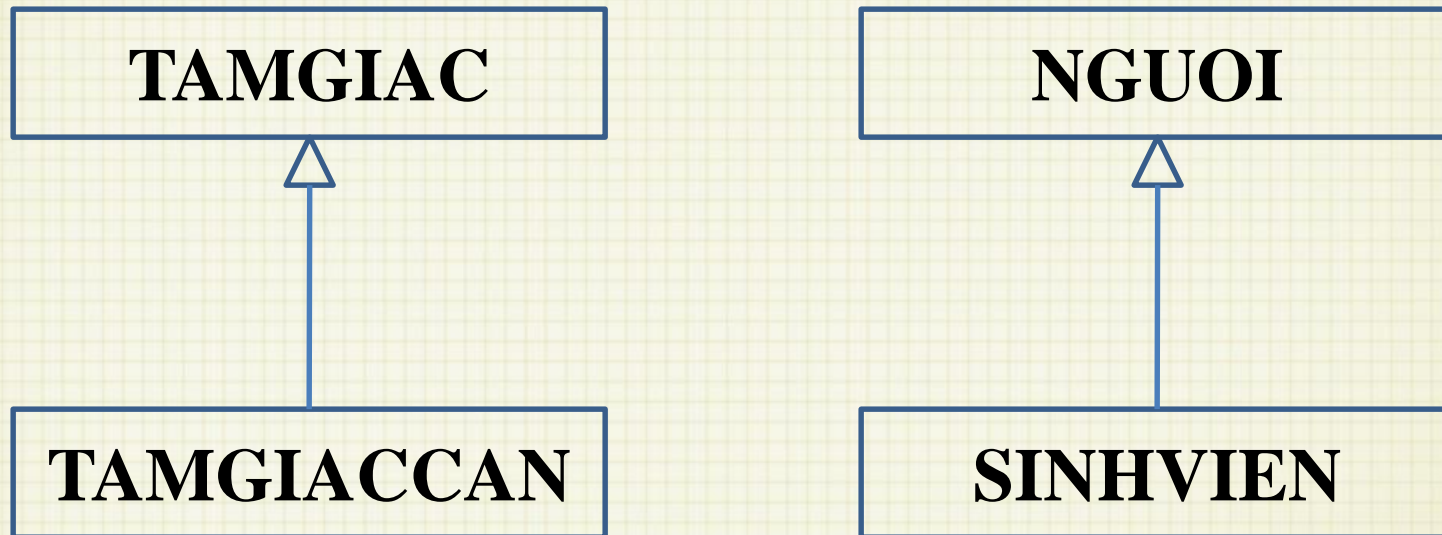
❖ **Khái niệm:** hai lớp đối tượng được gọi là có **quan hệ đặc biệt hóa-tổng quát hóa** với nhau khi lớp đối tượng này là trường hợp đặc biệt của lớp đối tượng kia và lớp đối tượng kia là trường hợp tổng quát của lớp đối tượng này.

❖ **Kí hiệu:**

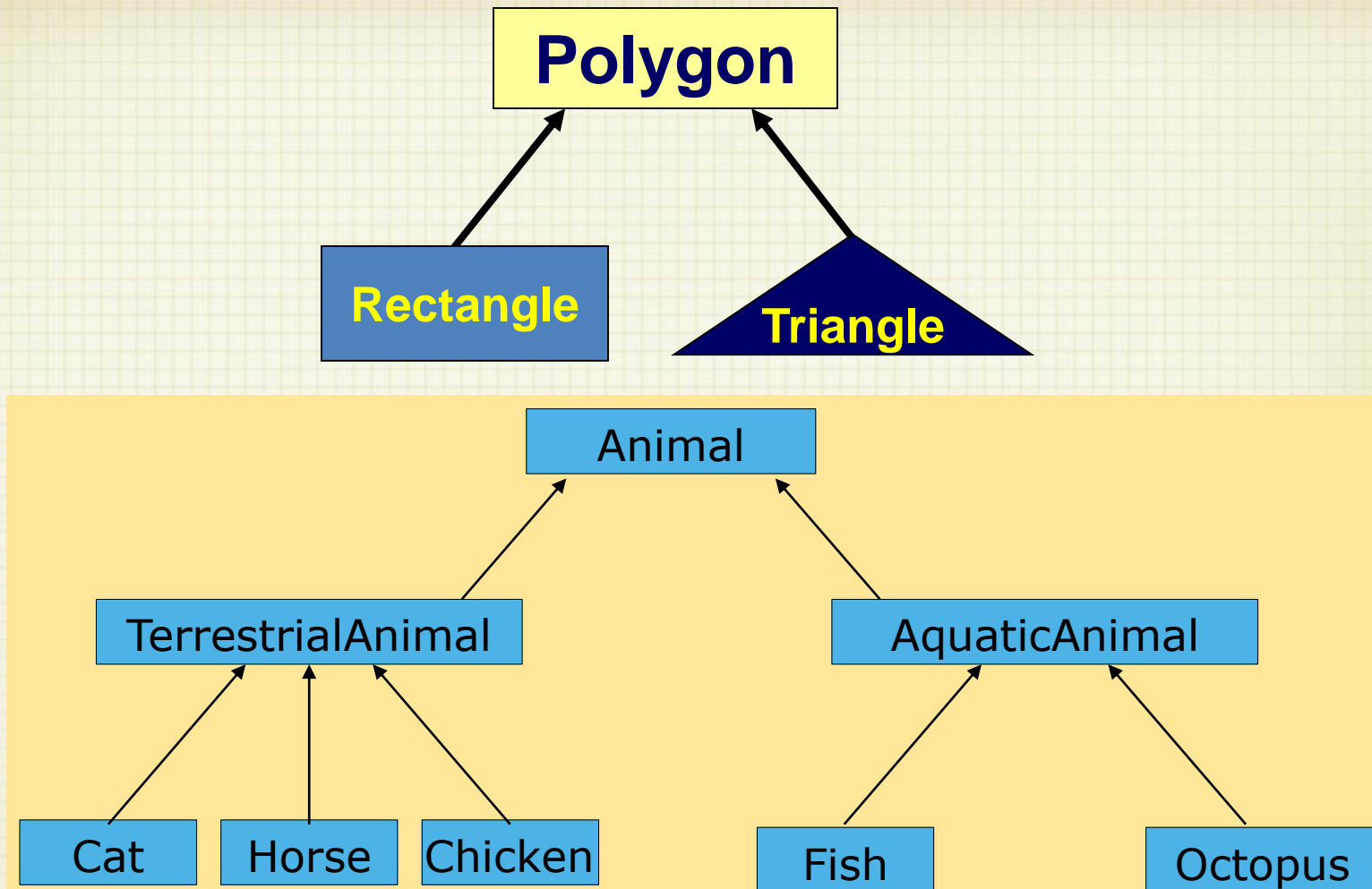


Quan hệ đặc biệt hóa – tổng quát hóa

❖ Ví dụ:



Quan hệ đặc biệt hóa – tổng quát hóa



Kế thừa

- ❖ **Kế thừa** là một đặc điểm của ngôn ngữ dùng để biểu diễn **mối quan hệ đặc biệt hóa – tổng quát hóa giữa các lớp**. Các lớp được trừu tượng hóa và được tổ chức thành một **sơ đồ phân cấp** lớp.
- ❖ Sự kế thừa là một mức cao hơn của **trừu tượng hóa**, cung cấp một cơ chế **gom chung** các lớp có liên quan với nhau thành một mức **khái quát hóa** đặc trưng cho toàn bộ các lớp nói trên.

Kế thừa

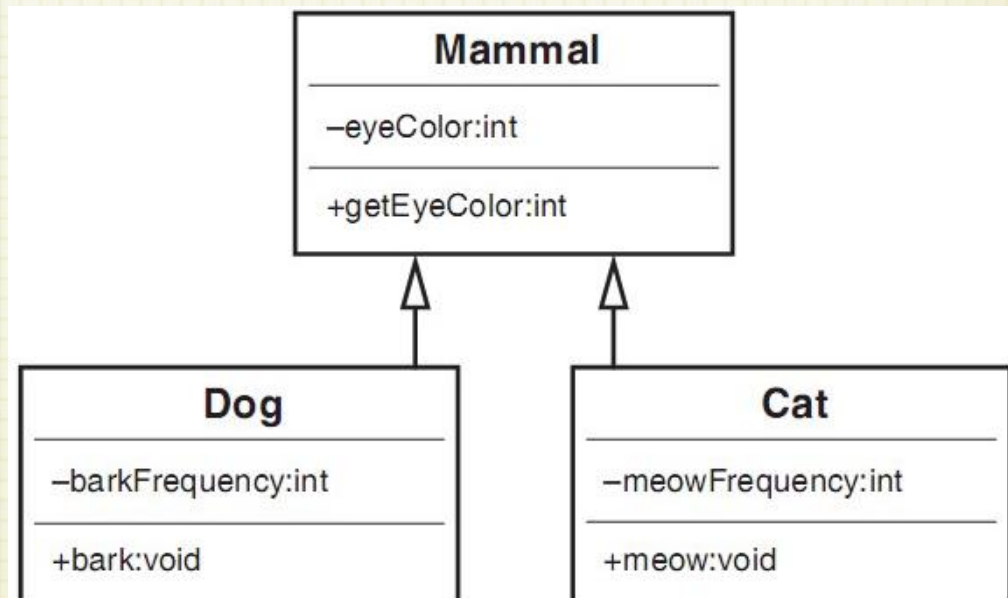
- ❖ Các lớp với các đặc điểm tương tự nhau có thể được tổ chức thành một **sơ đồ phân cấp kế thừa (cây kế thừa)**.
- ❖ Quan hệ “là 1”: Kế thừa được sử dụng thông dụng nhất để biểu diễn quan hệ “là 1”.
 - Một sinh viên là một người
 - Một hình tròn là một hình ellipse
 - Một tam giác là một đa giác
 - ...

Lợi ích kế thừa

- ❖ Kế thừa cho phép **xây dựng lớp mới từ lớp đã có**.
- ❖ Kế thừa cho phép tổ chức các lớp **chia sẻ mã chương trình chung**, nhờ vậy có thể dễ dàng sửa chữa, nâng cấp hệ thống.
- ❖ Trong C++, kế thừa còn định nghĩa **sự tương thích**, nhờ đó ta có cơ chế chuyển kiểu tự động.

Đặc tính Kế thừa

- ❖ Cho phép định nghĩa lớp mới từ lớp đã có.
 - Lớp mới gọi là **lớp con (subclass)** hay **lớp dẫn xuất (derived class)**
 - Lớp đã có gọi là **lớp cha (superclass)** hay **lớp cơ sở (base class)**.

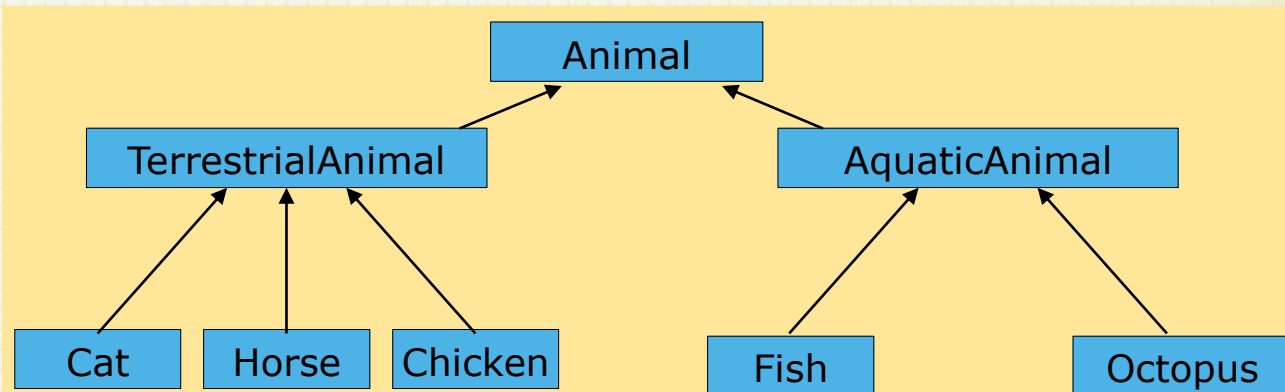


Đặc tính Kế thừa

❖ Thừa kế cho phép:

- Nhiều lớp có thể dẫn xuất từ một lớp cơ sở
- Một lớp có thể là dẫn xuất của nhiều lớp cơ sở

❖ Thừa kế không chỉ giới hạn ở một mức: Một lớp dẫn xuất có thể là lớp cơ sở cho các lớp dẫn xuất khác

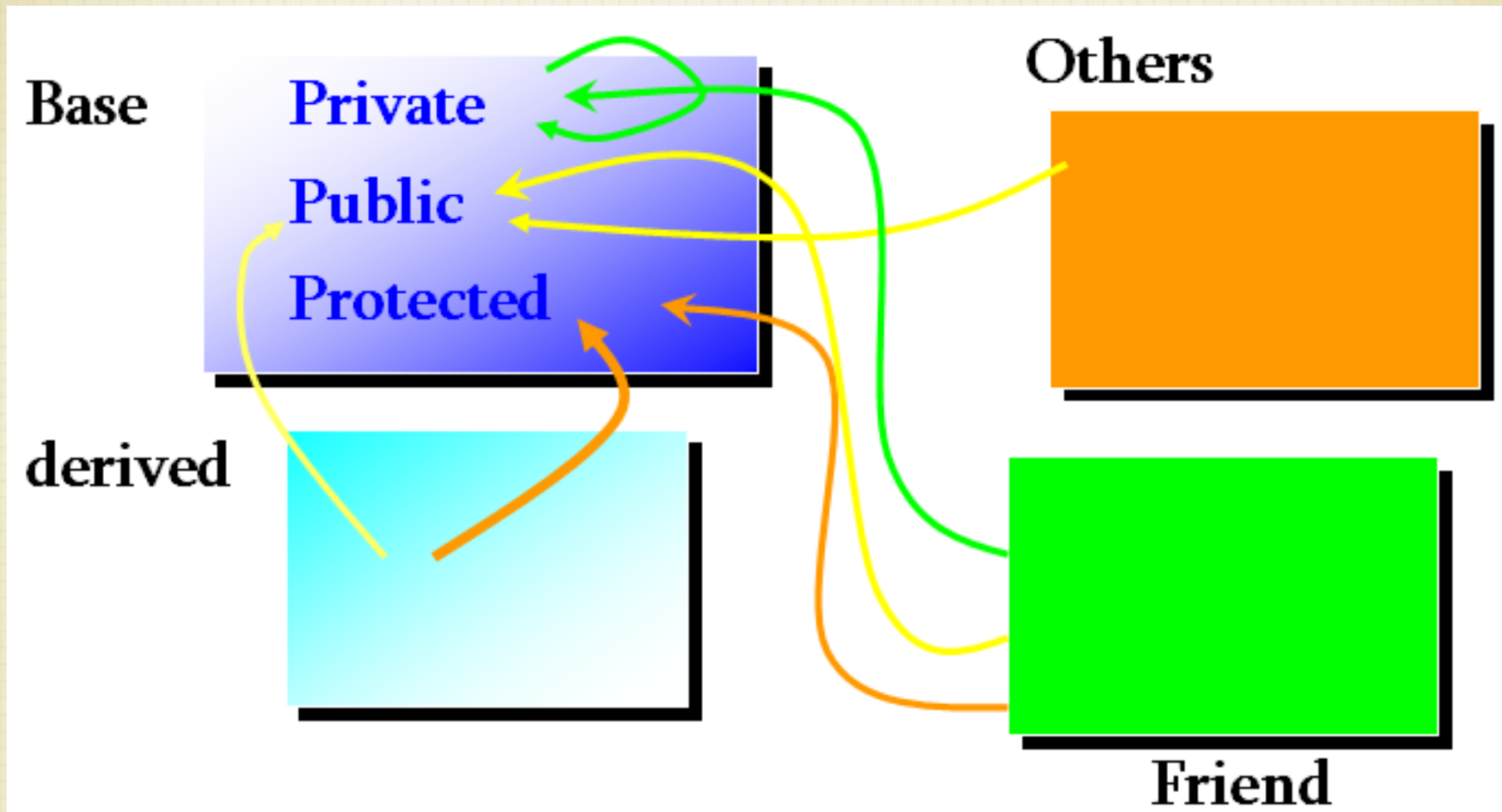


Cú pháp khai báo kế thừa

```
class SuperClass{  
    //Thành phần của lớp cơ sở  
};
```

```
class DerivedClass : [public/protected/private]  
    SuperClass{  
    //Thành phần bổ sung của lớp dẫn xuất  
};
```


Truy cập thành viên của lớp

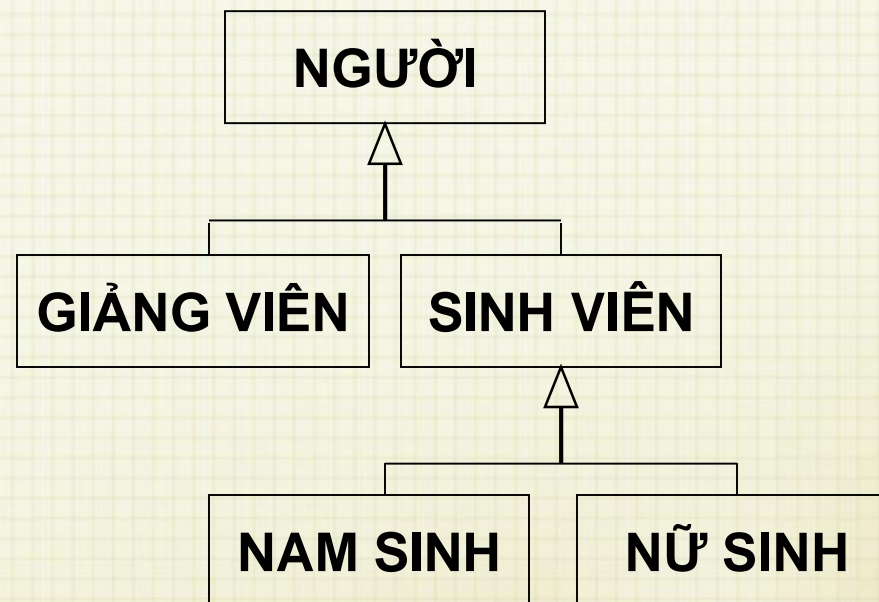


Kế thừa đơn

- ❖ Xét hai khái niệm **Người** và **Sinh viên** với mối quan hệ tự nhiên: *Một Sinh viên là một Người*. Trong C++, ta có thể biểu diễn khái niệm trên, một sinh viên là một người có thêm một số thông tin và một số thao tác (riêng biệt của sinh viên).
- ❖ Như vậy, ta tổ chức lớp **Sinh viên kế thừa từ lớp Người**.

Kế thừa đơn

- ❖ Ta có thể tổ chức hai lớp **Nam sinh** và **Nữ sinh** là hai lớp con (lớp dẫn xuất) của lớp Sinh viên. Trường hợp này, lớp Sinh viên trở thành lớp cha (lớp cơ sở) của hai lớp trên.



Kế thừa đơn – Ví dụ

```
class Nguoi {  
    char *HoTen;  
    int NamSinh;  
public:  
    Nguoi();  
    Nguoi( char *ht, int ns):NamSinh(ns) {HoTen=strdup(ht);}  
    ~Nguoi() {delete [ ] HoTen;}  
    void An() const { cout<<HoTen<<" an 3 chen com \n";}  
    void Ngu() const { cout<<HoTen<<" ngu ngay 8 tieng \n";}  
    void Xuat() const;  
    friend ostream& operator << (ostream &os, Nguoi& p);  
};
```

Kế thừa đơn – Ví dụ

```
class SinhVien : public Nguoi {  
    char *MaSo;  
public:  
    SinhVien();  
    SinhVien( char *ht, char *ms, int ns) : Nguoi(ht,ns) {  
        MaSo = strdup(ms);  
    }  
    ~SinhVien() {  
        delete [ ] MaSo;  
    }  
    void Xuat() const;  
};
```

Kế thừa đơn – Ví dụ

```
void Nguoi::Xuat() const
{
    cout << "Nguoi, ho ten: " << HoTen;
    cout << " sinh " << NamSinh;
    cout << endl;
}

void SinhVien::Xuat() const {
    Nguoi::Xuat();
    cout << "Sinh vien, ma so: " << MaSo;
    //cout << ", ho ten: " << HoTen;
    //cout << ", nam sinh: " << NamSinh;
    cout << endl;
}
```


Kế thừa đơn – Ví dụ

```
void main() {  
    Ngươi p1("Le Van Nhan",1980);  
    SinhVien s1("Vo Vien Sinh", "200002541",1984);  
    cout << "1.\n";  
    p1.An();           s1.An();  
    cout << "2.\n";  
    p1.Xuat();         s1.Xuat();  
    s1.Ngươi::Xuat();  
    cout << "3.\n";  
    cout << p1 << "\n";  
    cout << s1 << "\n";  
}
```

Kế thừa đặc tính của lớp cha

❖ Khai báo

```
class SinhVien : public Nguoi {  
    //...  
};
```

- Cho biết lớp **Sinh viên** kế thừa từ lớp **Người**. Khi đó Sinh viên *thừa hưởng các đặc tính* của lớp Người.

❖ **Về mặt dữ liệu:** Mỗi đối tượng Sinh viên *tự động có thành phần dữ liệu* họ tên, năm sinh của người.

Kế thừa đặc tính của lớp cha

- ❖ Về mặt thao tác: Lớp Sinh viên được tự động kế thừa các thao tác của lớp cha. Đây chính là khả năng sử dụng lại mã chương trình.
- ❖ Riêng phương thức thiết lập không được kế thừa.
- ❖ Khả năng thừa hưởng các thao tác của lớp cơ sở có thể được truyền qua “vô hạn mức”.

Định nghĩa lại thao tác ở lớp con

- ❖ Ta có thể định nghĩa lại các đặc tính ở lớp con đã có ở lớp cha, việc định nghĩa chủ yếu là thao tác.

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

Ràng buộc ngữ nghĩa ở lớp con

- ❖ Có thể áp dụng quan hệ **kế thừa mang ý nghĩa ràng buộc**, đối tượng ở lớp con là đối tượng ở lớp cha nhưng có dữ liệu bị ràng buộc:
 - Hình tròn là Ellipse với ràng buộc bán kính ngang dọc bằng nhau.
 - Số ảo là số phức với ràng buộc phần thực bằng 0
 - ...
- ❖ Lớp số ảo sau đây là một ví dụ minh họa.

Ví dụ

```
class Complex {  
    friend ostream& operator <<(ostream&, Complex);  
    friend class Imag;  
    double re, im;  
public:  
    Complex( double r = 0, double i = 0):re(r), im(i){ }  
    Complex operator +(Complex b);  
    Complex operator -(Complex b);  
    Complex operator *(Complex b);  
    Complex operator /(Complex b);  
    double Norm() const { return sqrt(re*re + im*im);}  
};
```


Ví dụ

```
class Imag: public Complex {  
public:  
    Imag(double i = 0):Complex(0, i){ }  
    Imag(const Complex &c) : Complex(0, c.im){ }  
    Imag& operator = (const Complex &c){  
        re = 0; im = c.im;  
        return *this;  
    }  
    double Norm() const {  
        return fabs(im);  
    }  
};
```

Ví dụ

```
void main()
{
    Imag i = 1;
    Complex z1(1,1)
    Complex z3 = z1 - i;    // z3 = (1,0)
    i = Complex(5,2);      // i là số ảo (0,2)
    Imag j = z1;           // j là số ảo (0,1)
    cout << "z1 = " << z1 << "\n";
    cout << "i = " << i << "\n";
    cout << "j = " << j << "\n";
}
```

Ràng buộc ngữ nghĩa ở lớp con

- ❖ Trong ví dụ trên, lớp số ảo (Imag) kế thừa hầu hết các thao tác của lớp số phức (Complex).
- ❖ Tuy nhiên, ta muốn ràng buộc mọi đối tượng thuộc lớp số ảo đều phải có phần thực bằng 0. Vì vậy, phải định nghĩa lại các hàm thành phần có thể vi phạm điều này.
- ❖ Ví dụ phép toán gán phải được định nghĩa lại để đảm bảo ràng buộc này.

Phạm vi truy xuất

- ❖ Khi thiết lập quan hệ kế thừa, ta vẫn phải quan tâm đến **tính đóng gói** và **che dấu thông tin**.
- ❖ Điều này ảnh hưởng đến phạm vi truy xuất của các thành phần của lớp.
- ❖ Hai vấn đề được đặt ra là:
 - Truy xuất theo chiều dọc
 - Truy xuất theo chiều ngang

Phạm vi truy xuất

❖ Truy xuất theo chiều dọc:

- Hàm thành phần của lớp con có quyền truy xuất các thành phần của lớp cha hay không?

❖ Truy xuất theo chiều ngang:

- Các thành phần của lớp cha, sau khi kế thừa xuống lớp con, thì thế giới bên ngoài có quyền truy xuất thông qua đối tượng của lớp con hay không?

Truy xuất theo chiều dọc

- ❖ Lớp con có quyền truy xuất các thành phần của lớp cha hay không, hoàn toàn **do lớp cha quyết định**. Điều đó được xác định bằng **thuộc tính kế thừa**.
- ❖ Trong trường hợp lớp Sinh viên kế thừa lớp Người, Sinh viên có quyền truy xuất họ tên của chính mình (được khai báo ở lớp Người) hay không?

Phạm vi truy xuất

```
class A{  
private:  
    int a;  
    void f();  
protected:  
    int b;  
    void g();  
public:  
    int c;  
    void h();  
};
```

```
void A::f()  
{  
    a = 1;    b = 2;    c = 3;  
}  
void A::g()  
{  
    a = 4;    b = 5;    c = 6;  
}  
void A::h(){  
    a = 7;    b = 8;    c = 9;  
}
```

Phạm vi truy xuất

❖ Ví dụ: Cho biết trong đoạn chương trình sau câu lệnh nào đúng, câu lệnh nào sai.

```
void main()
```

```
{
```

```
    A x;
```

```
    x.a = 10;
```

```
    x.f();
```

```
    x.b = 20;
```

```
    x.g();
```

```
    x.c = 30;
```

```
    x.h();
```

```
}
```

Phạm vi truy xuất

❖ Thuộc tính public:

- Thành phần nào có thuộc tính public thì có thể truy xuất từ bất cứ nơi nào.

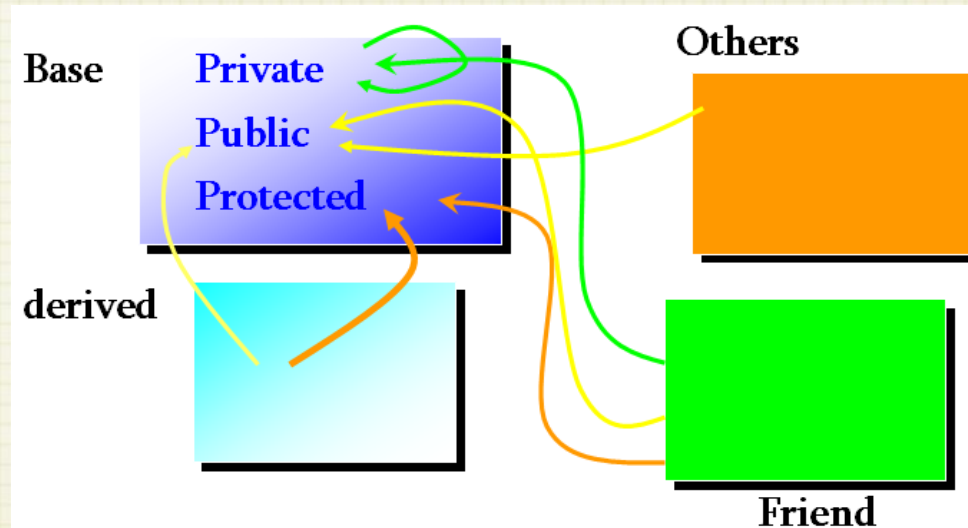
❖ Thuộc tính private: Thành phần có thuộc tính private

- Là **riêng tư của lớp đó**
- Chỉ có **hàm thành phần** của lớp và ngoại lệ các **hàm bạn** được phép truy xuất.
- Các lớp con cũng không có quyền truy xuất

Phạm vi truy xuất

❖ Thuộc tính protected:

- Cho phép qui định một vài thành phần nào đó của lớp là **bảo mật**, theo nghĩa thế giới bên ngoài không được phép truy xuất, nhưng tất cả các lớp con, cháu... đều được phép truy xuất.



Ví dụ Thuộc tính private

```
class Nguoi {  
    char *HoTen;  
    int NamSinh;  
public:  
    //...  
};  
class SinhVien : public Nguoi {  
    char *MaSo;  
public:  
    //...  
    void Xuat() const;  
};
```

Thuộc tính private

- ❖ Trong ví dụ trên, không có hàm thành phần nào của lớp SinhVien có thể truy xuất các thành phần **HoTen**, **NamSinh** của lớp Nguoi.
- ❖ Ví dụ, đoạn chương trình sau đây sẽ gây ra lỗi:

```
void SinhVien::Xuat() const {  
    cout << "Sinh vien, ma so: "<<MaSo<<",ho  
    ten:"<<HoTen;  
}
```


Thuộc tính private

- ❖ Ta có thể khắc phục lỗi trên nhờ khai báo lớp SinhVien là bạn của lớp Nguoi như trong ví dụ ban đầu:

```
class Nguoi {  
    friend class SinhVien;  
    char *HoTen;  
    int NamSinh;  
public:  
    //...  
};
```

Thuộc tính private

- ❖ Khai báo lớp bạn như trên, lớp SinhVien có thể truy xuất các thành phần private của lớp Nguoi.
- ❖ Cách làm trên chỉ giải quyết được nhu cầu của người sử dụng khi muốn tạo lớp con có quyền truy xuất các thành phần dữ liệu private của lớp cha.
- ❖ Tuy nhiên, cần phải sửa lại lớp cha và tất cả các lớp ở cấp cao hơn mỗi khi có một lớp con mới.

Thuộc tính private

```
class Nguoi {  
    friend class SinhVien;  
    friend class NuSinh;  
    char *HoTen; int NamSinh;  
public:  
    //...  
    void An() const { cout << HoTen << " an 3 chen com";}  
};  
class SinhVien : public Nguoi {  
    friend class NuSinh;  
    char *MaSo;  
public:  
    //...  
};
```

Thuộc tính protected

- ❖ Trong ví dụ trước, khi cài đặt lớp **NuSinh** ta phải thay đổi lớp cha **SinhVien** và cả lớp cơ sở **Nguoi** ở mức cao hơn.

```
class Nguoi {  
    protected:  
        char *HoTen;  
        int NamSinh;  
    public:  
        //...  
};
```


Thuộc tính protected

```
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;  
};
```

Thuộc tính protected

```
class NuSinh : public SinhVien {  
public:  
    NuSinh(char *ht, char *ms, int ns) : SinhVien(ht,ms,ns){  
    }  
    void An() const {  
        cout << HoTen << " ma so " << MaSo << " an 2 to pho";  
    }  
};  
  
// Co the truy xuat Nguoi::HoTen va  
// Nguoi::NamSinh va SinhVien::MaSo
```

Thuộc tính protected

```
void Nguoi::Xuat() const {  
    cout << "Nguoi, ho ten: " << HoTen << " sinh " << NamSinh;  
}  
  
void SinhVien::Xuat() const {  
    cout << "Sinh vien, ma so: " << MaSo << ", ho ten: " <<  
    HoTen;  
    // Ok: co quyen truy xuat, Nguoi::HoTen, Nguoi::NamSinh  
}  
  
void SinhVien::Xuat() const {  
    cout << "Sinh vien, ma so: " << MaSo  
    cout << ", ho ten: " << HoTen;  
}
```

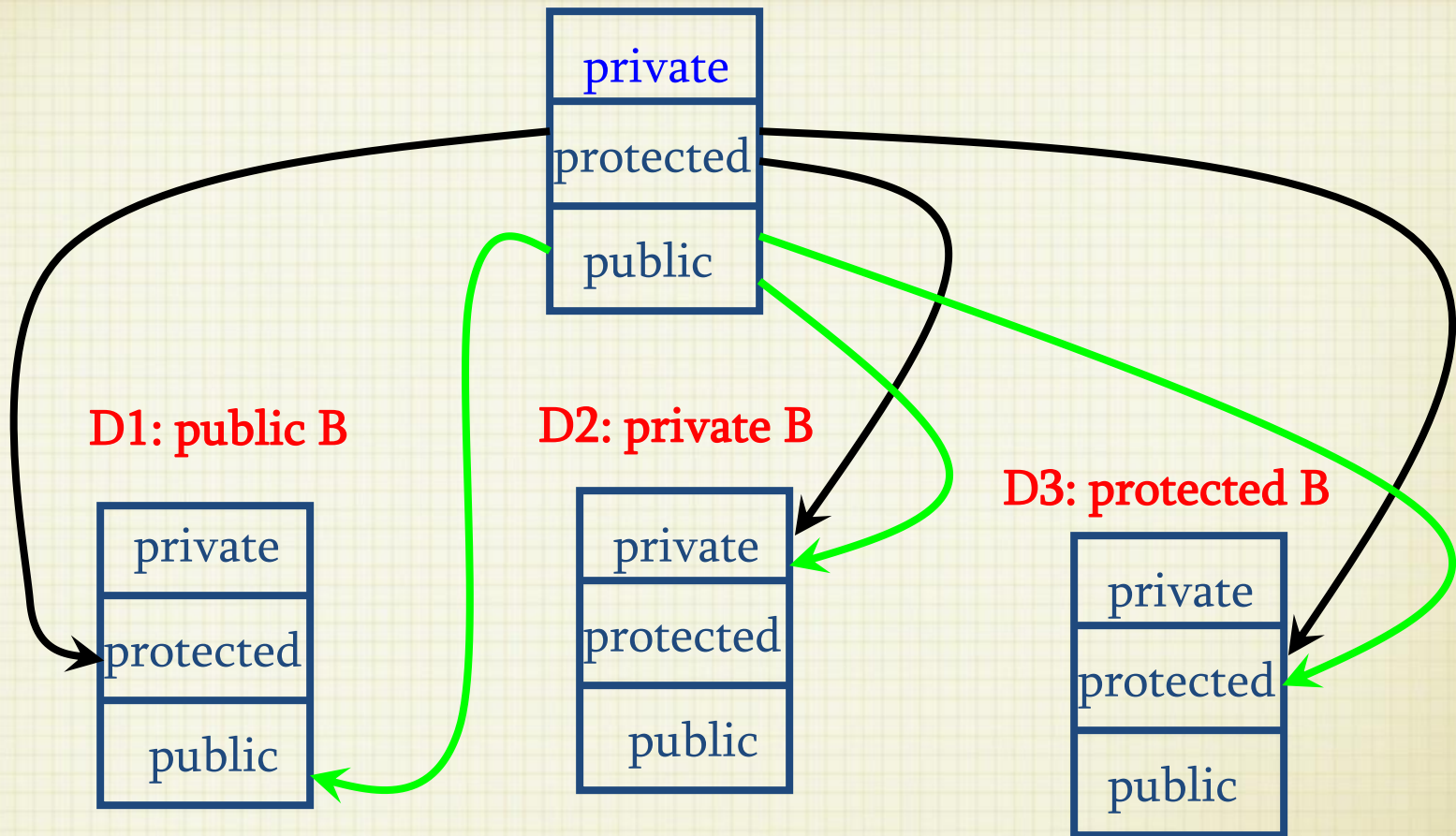
Thuộc tính protected

- ❖ Là cách để tránh phải sửa đổi lớp cơ sở khi có lớp con mới hình thành → Đảm bảo tính đóng gói.
- ❖ Thông thường ta dùng thuộc tính protected cho thành phần dữ liệu và public cho thành phần phương thức.
- ❖ Tóm lại, thành phần có thuộc tính protected chỉ cho phép những lớp con kế thừa được phép sử dụng.

Truy xuất theo chiều ngang

- ❖ Thành phần **protected** và **public** của lớp khi đã kế thừa xuống lớp con thì thế giới **bên ngoài** có **quyền truy xuất** thông qua đối tượng thuộc lớp con hay không?
 - Điều này hoàn toàn do lớp con quyết định bằng **phạm vi kế thừa**: Kế thừa **public**, Kế thừa **protected**, Kế thừa **private**

Phạm vi truy xuất trong kế thừa



Phạm vi truy xuất trong kế thừa

Type of Inheritance

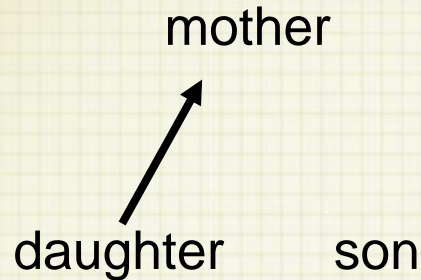
Access Control for Members

	private	Protected	public
private			
protected	Private	Protected	Protected
public	Private	Protected	Public

Phạm vi truy xuất trong kế thừa

Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
Public	public in derived class. Can be accessed directly by any non- static member functions, friend functions and non-member functions.	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	private in derived class. Can be accessed directly by all non- static member functions and friend functions.
Protected	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	private in derived class. Can be accessed directly by all non- static member functions and friend functions.
Private	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.

Ví dụ 1



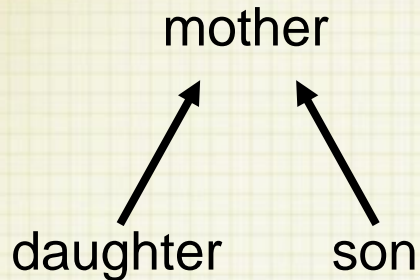
```
class mother{  
    protected:  
        int x, y;  
    public:  
        void set(int a, int b);  
    private:  
        int z;  
};
```

```
class daughter : public mother{  
    private:  
        double a;  
    public:  
        void foo ( );  
};
```

```
void daughter :: foo ( ){  
    x = y = 20;  
    set(5, 10);  
    cout<<"value of a "<<a<<endl;  
    z = 100;  
}
```

daughter can access 3 of the 4 inherited members

Ví dụ 2



```
class mother{  
    protected:  
        int x, y;  
    public:  
        void set(int a, int b);  
    private:  
        int z;  
};
```

```
class son : private mother{  
    private:  
        double b;  
    public:  
        void foo ( );  
};
```

```
void son :: foo ( ){  
    x = y = 20;  
    set(5, 10);  
    cout<<"value of b "<<b<<endl;  
    z = 100;  
}
```

Phương thức thiết lập

- ❖ Phương thức thiết lập của lớp cơ sở **luôn luôn được gọi** mỗi khi có một đối tượng của lớp dẫn xuất được tạo ra.
- ❖ Nếu mọi phương thức thiết lập của lớp cơ sở đều đòi hỏi phải cung cấp tham số thì lớp con bắt buộc phải có phương thức thiết lập để cung cấp các tham số đó

Phương thức thiết lập

❖ Ví dụ 1:

```
class A {  
    public:  
    A ()  
    { cout<< "A:default"<<endl; }  
    A (int a){  
        cout<<"A:parameter"<<endl;  
    }  
};
```

```
class B : public A{  
    public:  
    B (int a){  
        cout<<"B"<<endl;  
    }  
};
```

B test(1);

output:

**A:default
B**

Phương thức thiết lập

❖ Ví dụ 2:

```
class A {  
    public:  
    A ()  
    { cout<< "A:default"<<endl; }  
    A (int a){  
        cout<<"A:parameter"<<endl;  
    }  
};
```

```
class C : public A  
{  
    public:  
    C (int a) : A(a){  
        cout<<"C"<<endl;  
    }  
};
```

C test(1);

output: A:parameter
C

Định nghĩa các thành phần riêng

- ❖ Ngoài các thành phần được kế thừa, lớp dẫn xuất có thể định nghĩa thêm các thành phần riêng

```
class HìnhTron : Diem {  
    double r;  
public:  
    HìnhTron( double tx, double ty, double rr) : Diem(tx, ty){  
        r = rr;  
    }  
    void Ve(int color) const;  
    void TinhTien( double dx, double dy) const;  
};  
HìnhTron t(200,200,50);
```

Định nghĩa các thành phần riêng

- ❖ Lớp dẫn xuất cũng có thể **override** các phương thức đã được định nghĩa ở trong lớp cha.

```
class A {  
    protected:  
        int x, y;  
    public:  
        void print () {  
            cout<<"From A"<<endl;  
        }  
};
```

```
class B : public A  
{  
    public:  
        void print () {  
            cout<<"From B"<<endl;  
        }  
        void test()  
        {  
            print();  
            A::print();  
        }  
}
```

Truy cập phương thức

```
class Point{  
    protected:  
        int x, y;  
    public:  
        void set(int a, int b)  
            { x=a; y=b; }  
        void foo ();  
        void print();  
};
```

```
Point A;  
A.set(30,50);   ???  
A.print();
```

```
class Circle : public Point{  
    private: double r;  
    int x;  
    public:  
        void set (int a, int b, double c) {  
            Point ::set(a, b); //same name function call  
            r = c;  
        }  
        void print() { //.. }  
};
```

```
Circle C;  
C.set(10,10,100);   ???  
C.foo ();           ???  
C.print();          ???
```


Phương thức hủy bỏ

- ❖ Khi một đối tượng bị hủy đi, phương thức hủy bỏ của nó sẽ được gọi. Sau đó, các phương thức hủy bỏ của lớp cơ sở sẽ được gọi một cách tự động.
- ❖ Vì vậy, lớp con không cần và cũng không được thực hiện các thao tác dọn dẹp cho các thành phần thuộc lớp cha.

Phương thức hủy bỏ - Ví dụ

```
class SinhVien : public Nguoi {  
    char *MaSo;  
public:  
    SinhVien( char *ht, char *ms, int ns) : Nguoi(ht,ns){  
        MaSo = strdup(ms);  
    }  
    SinhVien(const SinhVien &s) : Nguoi(s){  
        MaSo = strdup(s.MaSo);  
    }  
    ~SinhVien() {delete [ ] MaSo;}  
    //...  
};
```

Con trở và kế thừa

❖ Con trở trong kế thừa hoạt động theo nguyên tắc sau:

- Con trở trở đến đối tượng thuộc lớp cơ sở thì có thể trở đến các đối tượng thuộc lớp con.
- Nhưng con trở trở đến đối tượng thuộc lớp con thì không thể trở đến các đối tượng thuộc lớp cơ sở.
- Có thể ép kiểu để con trở trở đến đối tượng thuộc lớp con có thể trở đến đối tượng thuộc lớp cơ sở. Tuy nhiên thao tác này có thể nguy hiểm.

Đa kế thừa

- ❖ Đa kế thừa cho phép một lớp có thể là dẫn xuất của nhiều lớp cơ sở.

```
class A : public B, public C {  
    ...  
};
```

- ❖ Các đặc điểm của kế thừa đơn vẫn đúng cho trường hợp đa kế thừa.

Đa kế thừa

- ❖ Làm thế nào biểu thị **tính độc lập** của **các thành phần cùng tên** bên trong một lớp dẫn xuất?
- ❖ Các phương thức thiết lập và hủy bỏ được gọi như thế nào: thứ tự, truyền thông tin, ...?
- ❖ Làm thế nào giải quyết tình trạng **thừa kế xung đột** trong đó, lớp D dẫn xuất từ B và C, và cả hai cùng là dẫn xuất của A

Đa kế thừa – Ví dụ

```
class BASE_A{  
    public:  
        int a;  
        int f( ){  
            return 0;  
        }  
        int g( ){  
            return 0;  
        }  
        int h( ) { return 0;}  
};
```

```
class BASE_B  
{  
    public:  
        int a;  
        int f( ){  
            return 0;  
        }  
        int g( ){  
            return 0;  
        }  
};
```

Đa kế thừa – Ví dụ

```
class ClassC : public BASE_A, public BASE_B{  
    //...  
};
```

```
void main(){  
    ClassC C;  
    C.f = g;    //Lỗi mơ hồ  
    C.a = 1;    //Lỗi mơ hồ  
    C.g();      //Lỗi mơ hồ  
    C.h();  
}
```

BÀI TẬP 1

- Giả sử Công ty có hai loại nhân viên: Nhân viên văn phòng và Nhân viên sản xuất. Viết chương trình quản lý và tính lương cho từng nhân viên của công ty:
- Mỗi nhân viên cần quản lý các thông tin sau: Họ tên, ngày sinh, lương
- Công ty cần tính lương cho nhân viên như sau:
 - Đối với nhân viên sản xuất:
 - $\text{Lương} = \text{lương căn bản} + \text{số sản phẩm} * 5.000$
 - Đối nhân viên văn phòng:
 - $\text{Lương} = \text{số ngày làm việc} * 100.000$
- Nhập vào N nhân viên sản xuất và M nhân viên văn phòng. Xuất ra thông tin các nhân viên vừa nhập.
- Xuất ra tổng số nhân viên công ty đang quản lý
- Xuất ra thông tin nhân viên có lương cao nhất trong từng loại
- Xuất ra các nhân viên có họ là "Le"
- Xuất ra tổng lương của từng loại nhân viên và tổng lương phải trả của công ty

BÀI TẬP 2

- Giả sử Trường ĐH CNTT TP.HCM đào tạo *sinh viên theo 2 hệ là hệ cao đẳng và hệ đại học*. Thông tin cần quản lí của một *sinh viên cao đẳng* bao gồm: mã số sinh viên, họ tên, địa chỉ, tổng số tín chỉ, điểm trung bình, điểm thi tốt nghiệp. Thông tin cần quản lí của một *sinh viên đại học* bao gồm: mã số sinh viên, họ tên, địa chỉ, tổng số tín chỉ, điểm trung bình, tên luận văn, điểm luận văn. Cách xét tốt nghiệp của sinh viên mỗi hệ là khác nhau:
- Sinh viên hệ cao đẳng tốt nghiệp khi có tổng số tín chỉ từ 120 trở lên, điểm trung bình từ 5 trở lên và điểm thi tốt nghiệp phải đạt từ 5 trở lên.
- Sinh viên hệ đại học tốt nghiệp khi có tổng số tín chỉ từ 170 trở lên, điểm trung bình từ 5 trở lên và phải bảo vệ luận văn với điểm số đạt được từ 5 điểm trở lên.
- Bạn hãy đề xuất thiết kế các lớp đối tượng cần thiết để quản lý danh sách các sinh viên của Trường và hỗ trợ xét tốt nghiệp cho các sinh viên theo tiêu chí đặt ra như trên. Hãy viết chương trình bằng C++ cho phép thực hiện các yêu cầu sau:
 - Nhập vào danh sách sinh viên, có thể sử dụng string cho các chuỗi kí tự.
 - Cho biết số lượng sinh viên đủ điều kiện tốt nghiệp?
 - Cho biết sinh viên đại học nào có điểm trung bình cao nhất?

BÀI TẬP 3

Hành tinh Babilon có 3 loại robot : Pedion, Zattacker và Carrier. Cả 3 loại robot đều có một trọng lượng nhất định M . Pedion thuộc loại robot xây dựng, có $M = 20$ kg và một độ linh hoạt F ($1 \leq F \leq 5$). Zattacker là robot có khả năng tấn công với trọng lượng $M = 50$ kg có sức mạnh P ($20 \leq P \leq 30$). Carrier là robot mang theo năng lượng tiếp tế cho 2 loại còn lại, có trọng lượng $M = 30$ kg và kho năng lượng vận chuyển E ($50 \leq E \leq 100$). Năng lượng tiêu thụ khi robot đi 1 quãng đường S km là:

- a. Pedion: $M * S + (F + 1) * S / 2$.
- b. Zattacker: $M * S + P * P * S$.
- c. Carrier: $M * S + 4 * E * S$.

Một hôm, tổng chỉ huy Babilon điều động một đoàn robot gồm A Pedion, B Zattacker và C Carrier (A, B, C là do người dùng nhập vào) đi khai phá những vùng đất còn bỏ hoang. Các robot có trọng lượng cố định theo loại, nhưng F, P và E thì có giá trị ngẫu nhiên trong miền giá trị của nó. Giả sử tất cả robot đều đi được một đoạn đường dài 10km, hãy xuất ra:

- a. Loại robot, các thông tin của nó và số năng lượng nó đã sử dụng.
- b. Cho biết loại robot tiêu thụ năng lượng nhiều nhất trong đoàn.
- c. Xuất tổng năng lượng đã tiêu thụ của cả đoàn robot.

Yêu cầu áp dụng phương pháp lập trình hướng đối tượng để giải quyết bài toán trên.

Q & A

