



UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

KẾ THỪA (TT)

C++



Microsoft®

Visual Studio®

Nội dung

- ❖ Phạm vi truy xuất trong kế thừa
- ❖ Đa kế thừa

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 {  
    public:  
        int a;  
        void f();  
    protected:  
        int b;  
        void g();  
    private :  
        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.

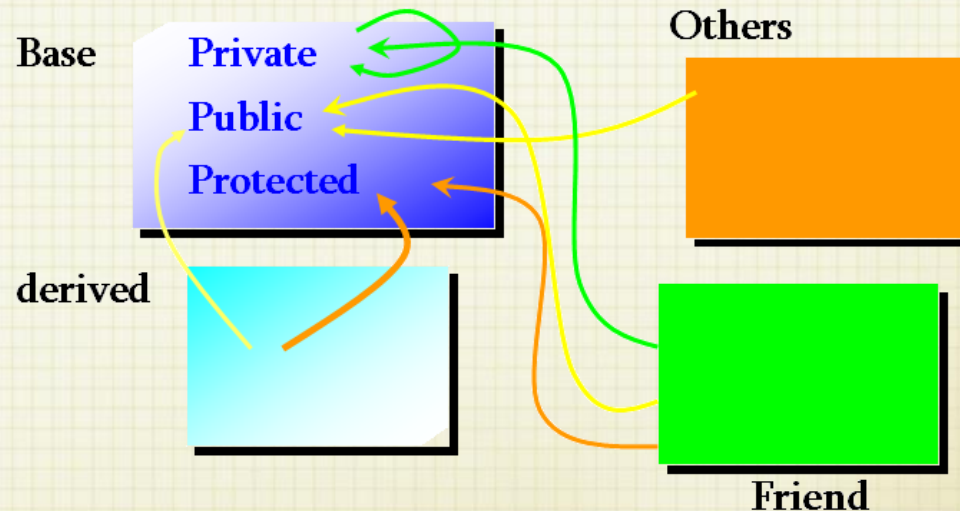
❖ Thuộc tính private: Thành phần có thuộc tính private

- Là **riêng tư của lớp đó**
- Chỉ có **hàm thành phần** của lớp đó 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 **lớp 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 Ngươi::HoTen va  
// Ngươi::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 xuất, Nguoi::HoTen, Nguoi::NamSinh  
}  
  
void SinhVien::Xuat() const {  
    cout << "Sinh vien, ma so: " << MaSo  
    cout << ", ho ten: " << HoTen;  
}
```

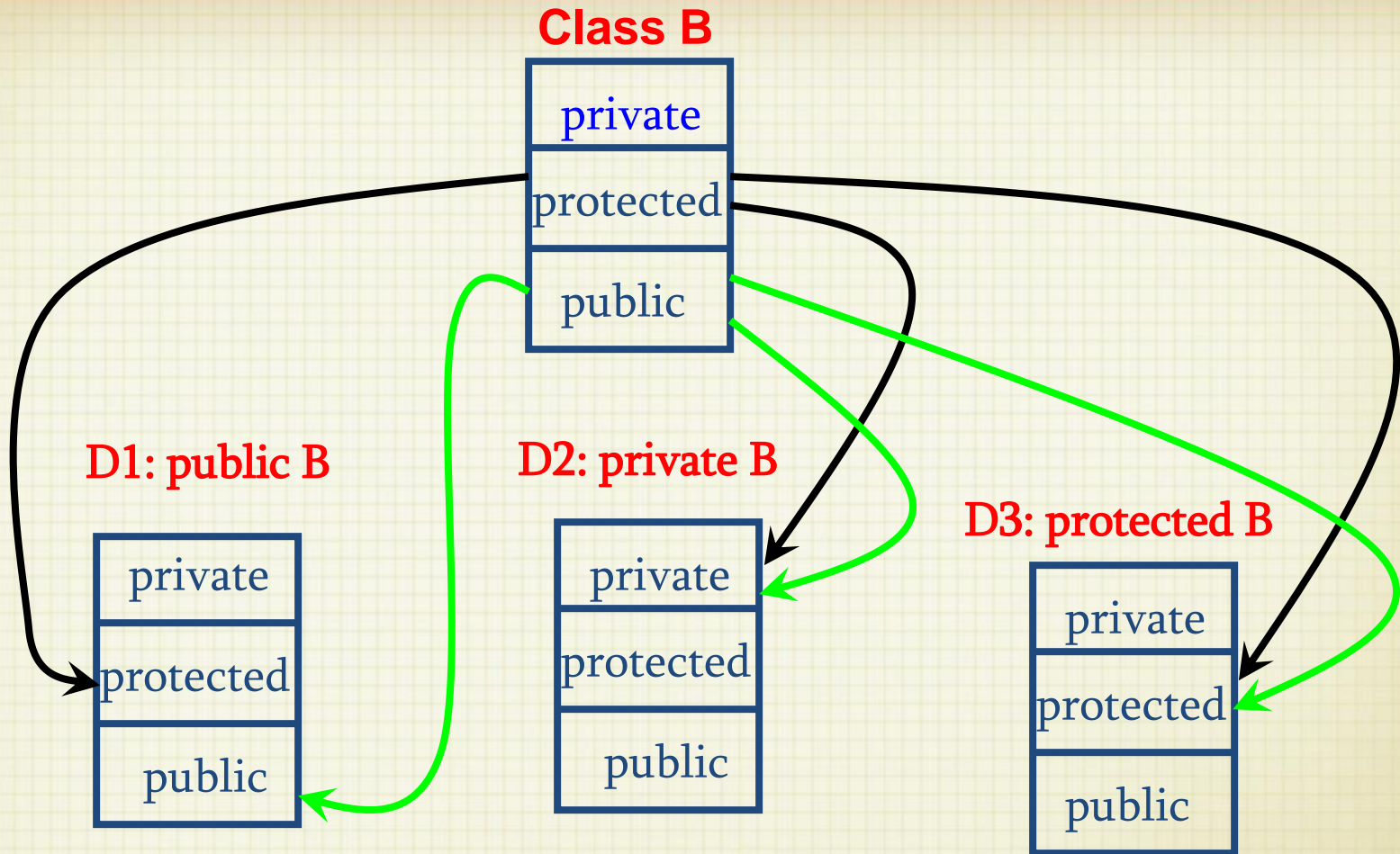
Thuộc tính protected

- ❖ 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: public, protected, private ?**

Phạm vi truy xuất trong kế thừa



Phạm vi truy xuất trong kế thừa

Type of Inheritance

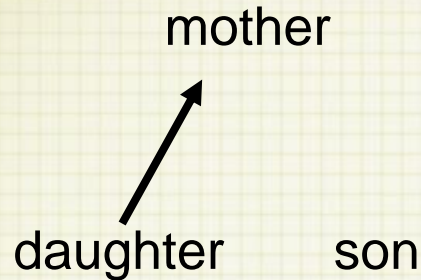
Access Control for Members

	private	Protected	public
private	?	?	?
protected	?	?	?
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



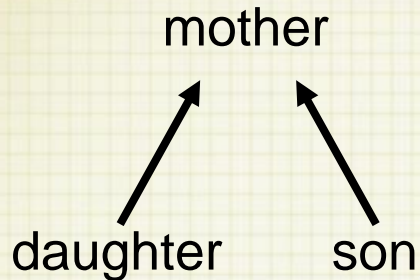
```
class mother{  
    protected:  
        int x, y;  
    public:  
        void set(int a, int b);  
    private:  
        int z;  
};
```

```
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;  
        }  
};
```



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

```
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, ...? *(theo thứ tự ưu tiên: pt khởi tạo từ trái sang phải, pt hủy thì ngược lại)*
- ❖ 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 *(được giải quyết khi sử dụng lớp ảo, phần sau sẽ trình bày)*

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();
```

```
}
```

C. BASE_A :: f = g;

C. BASE_A :: a = 1;

C. BASE_B :: g();

The end