NGÔN NGỮ LẬP TRÌNH C++

SLIDE 6.2: Kế thừa và đa hình

Giảng viên: Th.S Bùi Văn Kiên

NỘI DUNG

- 1. Khái niệm kế thừa
- 2. Hàm khởi tạo và hàm hủy trong kế thừa
- 3. Hàm bạn, lớp bạn
- 4. Đa hình
- 5. Đa kế thừa
- 6. Lớp trừu tượng

1. Khái niệm kế thừa

Lập trình hướng đối tượng có hai đặc trưng cơ bản:

- Đóng gói dữ liệu, được thể hiện bằng cách dùng khái niệm lớp để biểu diễn đối tượng với các thuộc tính private, chỉ cho phép bên ngoài truy nhập vào thông qua các phương thức get/set.
- Dùng lại mã, thể hiện bằng việc thừa kế giữa các lớp. Việc thừa kế cho phép các lớp thừa kế (gọi là lớp dẫn xuất) sử dụng lại các phương thức đã được định nghĩa trong các lớp gốc (gọi là lớp cơ sở)

1. Khái niệm kế thừa

- Cú pháp khai báo một lớp kế thừa từ một lớp khác như sau:

class <Tên lớp dẫn xuất>: <Từ khóa dẫn xuất> <Tên lớp cơ sở>{

… // Khai báo các thành phần lớp

};

- Tên lớp dẫn xuất: là tên lớp được cho kế thừa từ lớp khác. Tên lớp này tuân thủ theo quy tắc đặt tên biến trong C++.
- Tên lớp cở sở: là tên lớp đã được định nghĩa trước đó để cho lớp khác kế thừa. Tên lớp này cũng tuân thủ theo quy tắc đặt tên biến của C++.
- Từ khóa dẫn xuất: là từ khóa quy định tính chất của sự kế thừa. Có ba từ khóa dẫn xuất là private, protected và public.

1. Khái niệm kế thừa

- Ví dụ:

```
// Lớp cơ bản (Base class)
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

// Lớp dẫn xuất (Derived class) kế thừa từ lớp Animal
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();     // Kết quả: Animal is eating
    myDog.bark();    // Kết quả: Dog is barking

    return 0;
}
```

1. Khái niệm kế thừa

- Từ khóa dẫn xuất

Kiểu dẫn xuất	Tính chất lớp cơ sở	Tính chất lớp dẫn xuất
private	private protected public	không truy cập được private private
protected	private protected public	không truy cập được protected protected
public	private protected public	không truy cập được protected public

2. Hàm khởi tạo trong kế thừa

- Khi khai báo một đối tượng có kiểu lớp được dẫn xuất từ một lớp cơ sở khác. Thông thường, trong hàm khởi tạo của lớp dẫn xuất phải có hàm khởi tạo của lớp cơ sở.

<Tên hàm khởi tạo dẫn xuất>([<Các tham số>]): <Tên hàm khởi tạo cơ sở>([<Các đối số>]){

… // Khởi tạo các thuộc tính mới bổ sung của lớp dẫn xuất

};

2. Hàm khởi tạo trong kế thừa

- Nếu định nghĩa hàm khởi tạo bên ngoài phạm vi lớp thì phải thêm tên lớp dẫn xuất và toán tử phạm vi "::" trước tên hàm khởi tạo.
- Giữa tên hàm khởi tạo của lớp dẫn xuất và hàm khởi tạo của lớp cơ sở, chỉ có môt dấu hai chấm ":", nếu là hai dấu "::" thì trở thành toán tử phạm vi lớp.

```
class Bus: public Car{
    // Khai báo các thành phần
    Bus():Car(){
        // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
    }
};
```

2. Hàm khởi tạo trong kế thừa

- Nếu không chỉ rõ hàm khởi tạo của lớp cơ sở sau dấu hai chấm ":" chương trình sẽ tự động gọi hàm khởi tạo ngầm định hoặc hàm khởi tạo không có tham số của lớp cơ sở nếu hàm đó được định nghĩa tường minh trong lớp cơ sở.

```
class Bus: public Car{
    // Khai báo các thành phần
    Bus(){ // Gọi hàm khởi tạo không tham số của lớp Car
        // Khởi tạo các thuộc tính mới bổ sung của lớp Bus
    };
}
```

2. Hàm khởi tạo trong kế thừa

- Hàm khởi tạo có tham số:

```
class Car{
    int speed;
    char mark[20];
    float price;
public:
    Car();
    Car(int, char[], float);
};
```

```
/* Định nghĩa lớp Bus kế thừa từ lớp Car */
class Bus: public Car{
    int label; // Số hiệu tuyến xe
public:
    Bus(); // Khởi tạo không tham số
    Bus(int, char[], float, int); // Khởi tạo đủ tham số
};

Bus::Bus():Car(){
    label = 0;
}

// Khởi tạo đủ tham số
Bus::Bus(int sIn, char mIn[], float pIn, int lIn):Car(sIn, mIn, pIn){
    label = lIn;
}
```

2. Hàm hủy trong kế thừa

- Khi một đối tượng lớp dẫn xuất bị giải phóng khỏi bộ nhớ, thứ tự gọi các hàm hủy bỏ ngược với thứ tự gọi hàm thiết lập: gọi hàm hủy bỏ của lớp dẫn xuất trước khi gọi hàm hủy bỏ của lớp cơ sở.
- Vì mỗi lớp chỉ có nhiều nhất là một hàm hủy bỏ, nên ta không cần phải chỉ ra hàm hủy bỏ nào của lớp cơ sở sẽ được gọi sau khi hủy bỏ lớp dẫn xuất. Do vậy, hàm hủy bỏ trong lớp dẫn xuất được khai báo và định nghĩa hoàn toàn giống với các lớp thông thường:

<Tên lớp>::~<Tên lớp>([<Các tham số>]){

… // giải phóng phần bộ nhớ cấp phát cho các thuộc tính bổ sung

}

2. Ghi đè (overide)

- Một phương thức của lớp cơ sở được gọi là override nếu ở lớp dẫn xuất cũng định nghĩa một phương thức:
  - có cùng tên.
  - có cùng số lượng tham số.
  - có cùng kiểu các tham số (giống nhau từng đôi một theo thứ tự).
  - có cùng kiểu dữ liệu trả về.

3.1 Hàm bạn

- Trong C++, friend function là một hàm được khai báo bên ngoài lớp nhưng có quyền truy cập vào các thành viên private và protected của lớp đó. Điều này giúp hàm có thể truy cập các thuộc tính và phương thức của lớp mà không cần phải thông qua các phương thức công khai (public methods).
- Một điểm quan trọng là friend function không phải là thành viên của lớp, nhưng vẫn có quyền truy cập trực tiếp vào các thành viên private và protected của lớp.

3.1 Hàm bạn

- Ví dụ:

```
class Box {
private:
    double length;

public:
    Box(double l) : length(l) {}

    // Khai báo friend function
    friend double calculateVolume(Box& b);
};

double calculateVolume(Box& b) {
    return b.length * b.length * b.length;
}

int main() {
    Box b(3.0);
    cout << "Volume of Box: " << calculateVolume(b) << endl;
    return 0;
}
```

3.1 Hàm bạn

Ưu điểm của friend function:

- Friend function cho phép truy cập trực tiếp vào các thành viên private và protected của lớp, làm cho việc thao tác với dữ liệu của lớp trở nên dễ dàng hơn trong một số tình huống nhất định.
- Friend function hữu ích khi cần thực hiện các phép toán hoặc hoạt động liên quan đến nhiều lớp, mà không cần phải tạo các phương thức công khai trong từng lớp.

Lưu ý:

- Friend function không phải là thành viên của lớp, vì vậy nó không thể truy cập vào các phương thức hoặc thuộc tính của lớp qua đối tượng this như các phương thức thành viên.
- Việc sử dụng friend function cần phải cẩn thận vì có thể vi phạm nguyên tắc đóng gói (encapsulation) của OOP nếu không sử dụng hợp lý.

3.2. Lớp bạn

- friend class là một lớp được phép truy cập các thành viên private và protected của một lớp khác, dù chúng không phải là thành viên của lớp đó. Điều này cho phép một lớp "bạn" (friend) có thể sử dụng các thuộc tính và phương thức private của lớp chủ mà không cần thông qua các phương thức công khai.

3.2. Lớp bạn

- Ví dụ:

```
class Box {
private:
    double length;

public:
    Box(double l) : length(l) {}
    friend class FriendClass;
};

class FriendClass {
public:
    // FriendClass có quyền truy cập vào các thành viên private của Box
    void display(Box& b) {
        cout << "Length of Box: " << b.length << endl;
    }
};

int main() {
    Box box(5.0);
    FriendClass fc;
    fc.display(box);

    return 0;
}
```

3.2. Lớp bạn

Lý do sử dụng friend class:

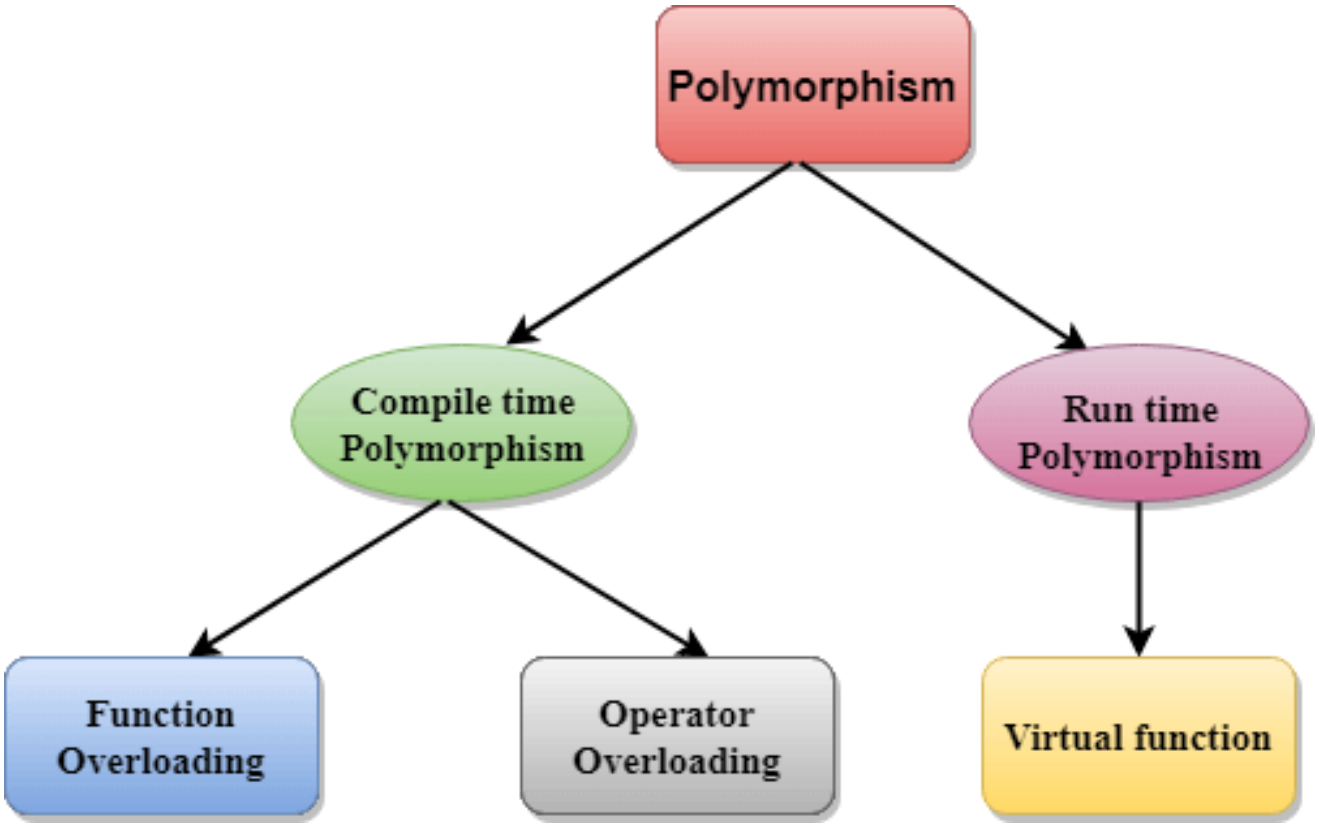
- Tăng tính linh hoạt: Khi bạn cần cho phép một lớp bên ngoài (không phải là thành viên của lớp) truy cập vào các thuộc tính private của lớp.
- Tổ chức mã hiệu quả: Thường được sử dụng trong các thiết kế dữ liệu phức tạp hoặc khi một lớp cần làm việc trực tiếp với một lớp khác mà không muốn sử dụng getter/setter.

Lưu ý:

- Việc sử dụng friend class có thể vi phạm nguyên tắc encapsulation (đóng gói) trong OOP vì nó cho phép lớp "bạn" truy cập vào các thành viên private.
- Bạn chỉ nên sử dụng friend class khi thực sự cần thiết, và cần phải cân nhắc về tính bảo mật và tính linh hoạt của mã nguồn.

4. Đa hình trong C++

Phân loại:



4.1 Function overloading

- Một số phương thức trong cùng một lớp được coi là overload nếu:
  - có cùng tên.
  - khác số lượng tham số.

```
void add(int a, int b)
{
   cout << "sum = " << (a + b);
}

void add(double a, double b)
{
     cout << endl << "sum = " << (a + b);
}
```

4.2 Operator overloading

Overload operator

- Một số toán tử như +, -, *, /, >>, <<, … có thể overload được
- Ví dụ:

```
PhanSo operator+(PhanSo ps) {
        PhanSo kq;
        kq.tu = this->tu * ps.mau + ps.tu * this->mau;
        kq.mau = this->mau * ps.mau;
        kq.rutGon();
        return kq;
}
```

→ Khi đó có thể sử dụng ps1 + ps2

4.2 Operator overloading

Overload operator

- Overload toán tử >> để sử dụng với cin

Ví dụ:

```
friend istream& operator>> (istream &is, PhanSo &ps)
{
        return is >> ps.tu >> ps.mau;
}
```

→ Khi đó có thể sử dụng cin >> ps1;

4.2 Operator overloading

Overload operator

- Overload toán tử << để sử dụng với cout
- Ví dụ:

```
friend ostream& operator<< (ostream &os, PhanSo ps)
{
        return os << ps.tu << "/" << ps.mau;
}
```

→ Khi đó có thể sử dụng cout << ps1;

4.3 Virtual function

- Sự kế thừa trong C++ cho phép có sự tương ứng giữa lớp cơ sở và các lớp dẫn xuất trong sơ đồ thừa kế:
- Một con trỏ có kiểu lớp cơ sở luôn có thể trỏ đến địa chỉ của một đối tượng của lớp dẫn xuất.
- Tuy nhiên, khi thực hiện lời gọi một phương thức của lớp, trình biên dịch sẽ quan tâm đến kiểu của con trỏ chứ không phải đối tượng mà con trỏ đang trỏ tới; phương thức của lớp mà con trỏ có kiểu được gọi chứ không phải phương thức của đối tượng mà con trỏ đang trỏ tới được gọi.

4.3 Virtual function

```
class Person{
        public:
        void show(){
            cout << "person" << endl;
        }
};
class Student: public Person{
    public:
    void show(){
        cout << "student" << endl;
    }
};
```

```
Student st;
Person *pSt = &st;
pSt->show(); //print person
```

4.3 Virtual function

```
class Person{
        public:
        virtual void show(){
            cout << "person" << endl;
        }
};
class Student: public Person{
    public:
    void show(){
        cout << "student" << endl;
    }
};
```

```
Student st;
Person *pSt = &st;
pSt->show(); //print student
```

4.3 Virtual function

```
class Person{
        public:
        virtual void show() = 0;
};

class Student: public Person{
    public:
    //void show(){
    //    cout << "student" << endl;
    //}
};
```

```
Student st;
Person *pSt = &st;
pSt->show(); //error
```

4.3 Virtual function

```
class Person{
        public:
        virtual void show() = 0;
};

class Student: public Person{
    public:
    void show(){
        cout << "student" << endl;
    }
};
```

```
Student st;
Person *pSt = &st;
pSt->show(); //print student
```

5. Đa kế thừa

- C++ cho phép đa kế thừa, tức là một lớp có thể được dẫn xuất từ nhiều lớp cơ sở khác nhau, với những kiểu dẫn xuất khác nhau
- Đa kế thừa được khai báo theo cú pháp:

class <Tên lớp dẫn xuất>: <Từ khoá dẫn xuất> <Tên lớp cơ sở 1>, <Từ khoá dẫn xuất> <Tên lớp cơ sở 2>, …<Từ khoá dẫn xuất> <Tên lớp cơ sở n>{

… // Khai báo thêm các thành phần lớp dẫn xuất

};

- Ví dụ:

```
class Bus: public Car, public PublicTransport{
        … // Khai báo các thành phần bổ sung
};
```

6. Lớp trừu tượng

- Sự cho phép đa kế thừa trong C++ dẫn đến một số hậu quả xấu, đó là sự xung đột giữa các thành phần của các lớp cơ sở, khi có ít nhất hai lớp cơ sở lại cùng được kế thừa từ một lớp cơ sở khác.
- Để tránh các vấn đề này, C++ cung cấp một khái niệm là kế thừa từ lớp cơ sở trừu tượng
- Việc chỉ ra một sự kế thừa trừu tượng được thực hiện bằng từ khoá virtual khi khai báo lớp cơ sở:

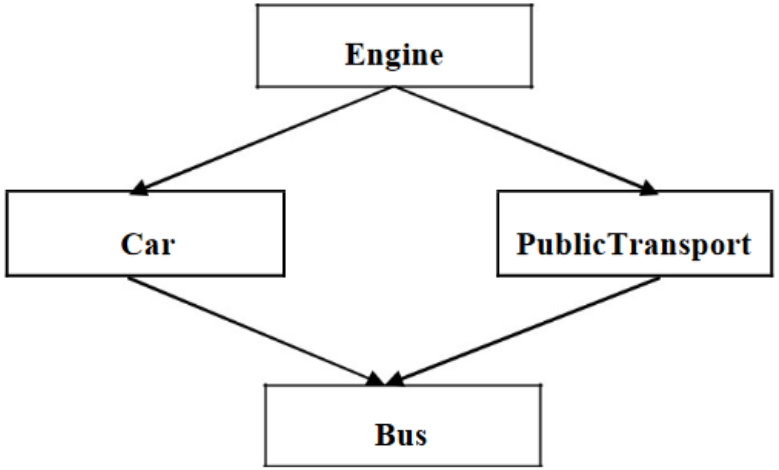
class <Tên lớp dẫn xuất>: <Từ khoá dẫn xuất> virtual <Tên lớp cơ sở>{

… // Khai báo các thành phần bổ sung

};

6. Lớp trừu tượng

- Các thành phần dữ liệu của lớp Engine bị lặp lại trong lớp Bus hai lần: Bus::Car::Engine và Bus::PublicTransport::Engine.
- Hàm khởi tạo của lớp Engine cũng được gọi hai lần: từ Car và từ PublicTransport.
- Hàm huỷ của lớp Engine cũng sẽ bị gọi tới hai lần.



6. Lớp trừu tượng

- Để tránh các vấn đề này, C++ cung cấp một khái niệm là kế thừa từ lớp cơ sở trừu tượng.
- Khi đó, ta cho các lớp Car và PublicTransport kế thừa trừu tượng từ lớp Engine. Bằng cách này, các thành phần của lớp Engine chỉ xuất hiện trong lớp Bus đúng một lần. Lớp Engine được gọi là lớp cơ sở trừu tượng của các lớp Car và PublicTransport.

6. Lớp trừu tượng

- Ví dụ:

```
class Engine{
// Các thành phần lớp Engine
};
class Car: public virtual Engine {
// Khai báo các thành phần bổ sung
};
class PublicTransport: public virtual Engine{
// Khai báo các thành phần bổ sung
};

class Bus: public Car, public PublicTransport{…};
```

Bài tập 1

1. Viết chương trình xử lý nhập ngày tháng năm, in ra ngày tháng năm sử dụng class.

2. Viết chương trình nhập vào 2 điểm trong mặt phẳng tọa độ Oxy.

- a) Nhập vào tọa độ cho từng điểm sử dụng cin
- b) Sử dụng cout để in ra tọa độ điểm
- c) Tính khoảng cách giữa 2 điểm

QUESTIONS & ANSWERS