**Chương 20 : Các lớp dẫn xuất**

**20.2 Các lớp có nguồn gốc**

Cân nhắc xây dựng một chương trình giao dịch với những người được một công ty tuyển dụng. Một chương trình như vậy có thể có một cấu trúc dữ liệu như thế này:

Xây dựng strutc nhân viên :

struct Employee {

string first\_name , family\_name;

char middle\_initial;

Date hiring\_date;

short depar tment;

// ...

};

Xây dựng strutc quản lí:

struct Manager {

Employee emp; // hồ sơ nhân viên quản lí

list<Employee∗> group; // người quản lí

short level;

// ...

};

Nhược điểm của chương trình trên là không có gì cho trình biên dịch và các công cụ khác biết rằng Người quản lý cũng là Nhân viên. Người quản lý ∗ không phải là Nhân viên ∗ .Cách tiếp cận đúng là trình bày rõ ràng rằng Người quản lý là Nhân viên, với một số thông tin được thêm vào:

struct Manager : public Employee {

list<Employee∗> group;

short level;

// ...

};

Trình quản lý có nguồn gốc từ Nhân viên và ngược lại, Nhân viên là một lớp cơ sở cho Trình quản lý. Các

Quản lý lớp có các thành viên của lớp Nhân viên (tên\_người đầu tiên, phòng ban, v.v.) ngoài các thành viên của lớp

các thành viên riêng (nhóm, cấp độ, v.v.).

Khởi tạo thường được biểu diễn bằng đồ thị bởi một con trỏ từ lớp dẫn xuất đến lớp cơ sở của nó

chỉ ra rằng lớp dẫn xuất tham chiếu đến cơ sở của nó (thay vì ngược lại):

Nhân viên -> quản lí

Một lớp dẫn xuất thường được cho là kế thừa các thuộc tính từ cơ sở của nó, vì vậy mối quan hệ còn được gọi là kế thừa Một lớp cơ sở đôi khi được gọi là lớp cha và lớp dẫn xuất là lớp con. Thuật ngữ này, tuy nhiên, gây nhầm lẫn cho những người quan sát rằng dữ liệu trong một đối tượng lớp dẫn xuất là tập siêu dữ liệu của một đối tượng thuộc lớp cơ sở của nó. Một lớp dẫn xuất thường lớn hơn (và không bao giờ nhỏ hơn) so với lớp cơ sở của nó theo nghĩa là nó chứa nhiều dữ liệu hơn và cung cấp nhiều chức năng hơn.

Xuất phát từ Người quản lý từ Nhân viên theo cách này làm cho Người quản lý trở thành một loại phụ của Nhân viên, do đó

Người quản lý có thể được sử dụng ở bất cứ nơi nào Nhân viên có thể chấp nhận được. Ví dụ thể tạo một danh sách

void f(Manager m1, Employee e1)

{

list<Employee∗> elist {&m1,&e1);

// ...

}

Người quản lý (cũng) là một Nhân viên, vì vậy Người quản lý ∗ có thể được sử dụng như một Nhân viên ∗. Tuy nhiên, một Nhân viên không nhất thiết phải là một Người quản lý , vì vậy một Nhân viên ∗ không thể được sử dụng làm Người quản lý ∗. Nói chung, nếu một lớp Derived có một lớp cơ sở công khai

Cơ sở, sau đó một Xuất phát ∗ có thể được gán cho một biến kiểu Cơ sở ∗ mà không cần sử dụng chuyển đổi kiểu rõ ràng. Việc chuyển đổi ngược lại, từ Cơ sở ∗ sang Bắt nguồn ∗, phải rõ ràng. Ví dụ:

void g(Manager mm, Employee ee)

{

Employee∗ pe = &mm; // mọi người quản lí đều là nhân viên

Manager∗ pm = &ee; // không phải mọi nhân viên dều là quản lí

pm−>level = 2; // lỗi : ee hông có cấp độ

pm = static\_cast<Manager∗>(pe); // điểm hoạt động pe

// cho người quản lí mm

pm−>level = 2; // pm trỏ tới Trình quản lý mm có cấp độ

}

Nói cách khác, một đối tượng của một lớp dẫn xuất có thể được coi là một đối tượng của lớp cơ sở của nó khi được thao tác thông qua các con trỏ và tham chiếu.

class Employee; // chỉ khai báo, không có định nghĩa

class Manager : public Employee { //lỗi : Nhân viên không được xác định

// ...

};

**20.2.1 chức năng thành viên**

Cung cấp một kiểu thích hợp với một tập hợp các thao tác phù hợp và chúng ta cần làm như vậy mà không bị ràng buộc vào các chi tiết của một biểu diễn cụ thể. Ví dụ:

class Employee {

public:

void print() const;

string full\_name() const { return first\_name + ' ' + middle\_initial + ' ' + family\_name; }

// ...

private:

string first\_name , family\_name;

char middle\_initial;

// ...

};

class Manager : public Employee {

public:

void print() const;

// ...

};

Thành viên của một lớp dẫn xuất có thể sử dụng công khai - các thành viên của một cơ sở lớp như thể chúng được khai báo trong chính lớp dẫn xuất. Ví dụ:

void Manager::print() const{

cout << "name is " << full\_name() << '\n';

// ...

}

Tuy nhiên, một lớp dẫn xuất không thể truy cập các thành viên riêng của một lớp cơ sở

Thông thường, giải pháp sạch nhất là dành cho lớp dẫn xuất chỉ sử dụng các thành viên công khai của

lớp cơ sở. Ví dụ:

void Manager::print() const

{

Employee::print(); //in thông tin nhân viên

cout << level; // in thông tin cụ thể về trình quản lí

// ...}

**20.2.2 Người xây dựng và Người phá hủy**

Như thường lệ, hàm tạo và hàm hủy là thiết yếu:

• Các đối tượng được xây dựng từ dưới lên (cơ sở trước thành viên và thành viên trước

dẫn xuất) và hủy từ trên xuống (dẫn xuất trước thành viên và thành viên trước cơ sở);

• Mỗi lớp có thể khởi tạo các thành viên và cơ sở của nó (nhưng không trực tiếp là thành viên hoặc cơ sở của

căn cứ)

• Thông thường, các trình hủy trong một hệ thống phân cấp cần phải là ảo

• Các hàm tạo sao chép của các lớp trong một hệ thống phân cấp nên được sử dụng cẩn thận (nếu có) để tránh bị cắt

• Độ phân giải của một lệnh gọi hàm ảo, một dynamic\_cast hoặc một typeid () trong một phương thức khởi tạo hoặc de-structor phản ánh giai đoạn xây dựng và phá hủy (chứ không phải là kiểu của đối tượng chưa được hoàn thành)

**20.3 Cấu trúc phân cấp lớp**

Bản thân một lớp dẫn xuất có thể là một lớp cơ sở.

class Employee { /\* ... \*/ };

class Manager : public Employee { /\* ... \*/ };

class Director : public Manager { /\* ... \*/ };

20.3.1 Nhập trường

Để sử dụng các lớp dẫn xuất không chỉ là một cách viết tắt thuận tiện trong khai báo. Có bốn giải pháp cơ bản:

1.Đảm bảo rằng chỉ các đối tượng của một kiểu duy nhất được trỏ tới

2.Đặt một trường kiểu trong lớp cơ sở để các chức năng kiểm tra

3.Sử dụng dynamic\_cast

4.Sử dụng các hàm ảo

Sử dụng trường kiểu là một kỹ thuật dễ xảy ra lỗi dẫn đến các vấn đề bảo trì. Các vấn đề gia tăng mức độ nghiêm trọng khi kích thước của chương trình tăng lên vì việc sử dụng trường atype gây ra vi phạm các lý tưởng về tính mô-đun và ẩn dữ liệu. Mỗi hàm sử dụng trường kiểu phải biết về cách biểu diễn và các chi tiết khác về việc triển khai mọi lớp dẫn xuất từ ​​lớp chứa trường kiểu.

Có vẻ như bất kỳ dữ liệu phổ biến nào có thể truy cập được từ mọi lớp dẫn xuất, chẳng hạn như trường kiểu, sẽ thúc giục mọi người thêm nhiều dữ liệu như vậy. Do đó, cơ sở chung trở thành kho lưu trữ tất cả các loại ‘‘ thông tin hữu ích ’’. Điều này, đến lượt nó, giúp việc triển khai các lớp cơ sở và lớp dẫn xuất được đan xen vào nhau theo những cách không mong muốn nhất. Trong một hệ thống phân cấp lớp lớn, dữ liệu có thể truy cập (không phải riêng tư) trong một lớp cơ sở chung sẽ trở thành '' biến toàn cục '' của hệ thống phân cấp.

Để thiết kế sạch sẽ và bảo trì đơn giản hơn, chúng tôi muốn tách biệt các vấn đề riêng biệt và tránh phụ thuộc lẫn nhau.

**20.2.3 Chức năng ảo**

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép lập trình viên khai báo các hàm trong một lớp cơ sở có thể được định nghĩa lại trong mỗi lớp dẫn xuất. Trình biên dịch và trình liên kết sẽ đảm bảo sự tương ứng chính xác giữa các đối tượng và các chức năng được áp dụng cho chúng.

Để khai báo hàm ảo, bạn thêm từ khóa virtual trước tên hàm. Hãy theo dõi ví dụ minh họa sau để hiểu hơn cách hàm ảo làm việc.

#include <iostream>

using namespace std;

class Sun

{

public:

~Sun()

{

cout << "This is Sun\*" << endl;

}

};

class Sun1 : public Sun

{

public :

~Sun1()

{

cout << "This is Sun1" << endl;

}

};

int main()

{

Sun\* sun = new Sun1();

delete sun;

return 0;

}

Kết quả:This is Sun\*

Ở ví dụ trên hàm hủy của Sunclass không được định nghĩa là virtual nên khi ta hủy con trỏ sun hàm hủy của Sun1 không được gọi. Nên ta định nghĩa hàm hủy của Sun class là virtual.

**20.3.3 Ép kiểu tường minh**

Gọi một hàm bằng toán tử phân giải phạm vi, ::, như được thực hiện trong Manager :: print () đảm bảo rằng cơ chế ảo không được sử dụng:

void Manager::print() const

{

Employee::print(); // not a virtual call

cout << "\tlevel " << level << '\n';

// ...

}

Nếu không, Manager :: print () sẽ phải chịu một đệ quy vô hạn. Việc sử dụng một tên đủ điều kiện có một hiệu quả mong muốn khác. Nghĩa là, nếu một hàm ảo cũng nội tuyến (không phải là hiếm), thì phép thay thế nội tuyến có thể được sử dụng cho các lệnh gọi được chỉ định bằng cách sử dụng ::. Điều này cung cấp cho người lập trình một cách hiệu quả để xử lý một số trường hợp đặc biệt quan trọng trong đó một hàm ảo gọi một hàm ảo khác cho cùng một đối tượng. Hàm Manager :: print () là một ví dụ về điều này. Bởi vì kiểu của đối tượng được xác định trong lệnh gọi Manager :: print (), nó không cần được xác định động lại cho lệnh gọi kết quả của Employee :: print ().

**20.3.4 Ghi đè hàm**

Nếu bạn khai báo một hàm trong lớp dẫn xuất có cùng tên và kiểu với hàm ảo trong lớp cơ sở, thì hàm trong lớp dẫn xuất sẽ ghi đè hàm trong lớp cơ sở.

Đó là một quy tắc đơn giản và hiệu quả. Tuy nhiên, đối với các cấu trúc phân cấp lớp lớn hơn, có thể khó để chắc chắn rằng bạn thực sự ghi đè chức năng mà bạn định ghi đè

**20.3.4.1 từ khóa override**

Vì dụ về cách sử dụng từ khóa override:

class B

{

public:

virtual void f(short)

{

std::cout << "B::f" << std::endl;

}

};

class D : public B

{

public:

virtual void f(short) override

{

std::cout << "D::f" << std::endl;

}

};

Một lợi thế rất quan trọng nữa khi sử dụng từ khóa override là: xét ví dụ ở trên, nếu bạn hoặc một đồng nghiệp khác của bạn vô tình hoặc cố ý sửa tên hàm hoặc tham số đầu vào của hàm f ở class B thì lúc biên dịch sẽ bị lỗi, điều này chỉ ra cho chúng ta thấy nếu muốn sửa một phương thức có tên là f ở lớp cha mà lớp con đã viết lại bằng từ khóa override thì phải sửa luôn cả phương thức f ở lớp con.

**20.3.4.2 final**

Từ khóa final ở sau một hàm muốn chỉ ra rằng tất cả class con kế thừa từ class này KHÔNG được phép viết lại hàm này. Nếu chúng ta cố tình viết lại hàm có từ khóa final thì lúc biên dịch sẽ bị lỗi.

Ví dụ về cách sử dụng từ khóa final:

class B

{

public:

virtual void f(int) final

{

std::cout << "B::f" << std::endl;

}

};

class D : public B

{

public:

virtual void f(int)

{

std::cout << "D::f" << std::endl;

}

};

Với ví dụ trên, chúng ta thấy rằng hàm f ở class B đã có từ khóa final, nhưng class D vẫn cố tình viết lại hàm f, lúc này khi biên dịch sẽ bị lỗi.

**20.3.5 Sử dụng thành viên cơ sở**

Các hàm không quá tải trên các phạm vi . Ví dụ:

struct Base {

void f(int);

};

struct Derived : Base {

void f(double);

};

void use(Derived d)

{

d.f(1); // gọi Derived::f(double)

Base& br = d

br.f(1); // gọi Base::f(int)

}

Điều này có thể khiến mọi người ngạc nhiên và đôi khi chúng tôi muốn quá tải để đảm bảo rằng

chức năng thành viên được sử dụng. Đối với không gian tên, sử dụng khai báo có thể được sử dụng để thêm một hàm vào

phạm vi. Ví dụ:

struct D2 : Base {

using Base::f; // đưa tất cả fs từ Base vào D2

void f(double);

};

void use2(D2 d)

{

d.f(1); // call D2::f(int), that is Base::f(int)

Base& br = d

br.f(1); // gọi Base::f(int)

}

Đây là một hệ quả đơn giản của một lớp cũng được coi là một không gian tên

Một số khai báo sử dụng có thể mang tên từ nhiều lớp cơ sở. Ví dụ:

struct B1 {

void f(int);

};

struct B2 {

void f(double);

};

struct D : B1, B2 {

using B1::f;

using B2::f;

void f(char);

};

void use(D d)

{

d.f(1); // gọi D::f(int), that is, B1::f(int)

d.f('a'); // gọi D::f(char)

d.f(1.0); // call D::f(double), that is, B2::f(double)

}

Chúng ta có thể đưa các hàm tạo vào một phạm vi lớp dẫn xuất. Tên được đưa vào phạm vi lớp dẫn xuất bởi khai báo using có quyền truy cập của nó được xác định bởi vị trí của khai báo using. Chúng ta không thể sử dụng các chỉ thị using để đưa tất cả các thành viên của một lớp cơ sở vào lớp nguồn gốc.

**20.3.5.1 Trình tạo kế thừa Inheriting Constructors**