**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

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

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 Hàm tạo và hàm 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

**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.

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

Override là việc viết lại một phương thức (method) trong lớp dẫn xuất (derived class) mà ở lớp cơ sở (base class).

Như chúng ta đã biết để lớp dẫn suất override 1 virtual function của base class thì 2 hàm phải có cùng tên, cùng tham số và cùng kiểu dữ liệu trả về. Tuy nhiên trong nhiều trường hợp ta có thể xảy ra nhầm lẫn khi override 1 virtual function.

**20.3.4.2 final**

từ khóa ***final*** khi dùng cho biến trong một phương thức, cũng có ý nghĩa tương tự như khi dùng từ khóa này với thuộc tính của lớp. Nó đều mang giá trị là một hằng số không thể thay đổi được.

**20.3.5 Sử dụng thành viên cơ sở**

Khi bạn khai báo một lớp dẫn xuất, một chỉ định truy cập có thể đứng trước mỗi lớp cơ sở trong danh sách cơ sở của lớp dẫn xuất. Điều này không làm thay đổi các thuộc tính truy cập của các thành viên riêng lẻ của lớp cơ sở như được thấy bởi lớp cơ sở, nhưng cho phép lớp dẫn xuất hạn chế quyền kiểm soát truy cập của các thành viên của lớp cơ sở.

Bạn có thể lấy các lớp bằng cách sử dụng bất kỳ ba mã truy cập nào:

Trong một publiclớp cơ sở, các thành viên công khai và được bảo vệ của lớp cơ sở vẫn là các thành viên công khai và được bảo vệ của lớp dẫn xuất.

Trong một protectedlớp cơ sở, các thành viên công khai và được bảo vệ của lớp cơ sở là các thành viên được bảo vệ của lớp dẫn xuất.

Trong một privatelớp cơ sở, các thành viên công khai và được bảo vệ của lớp cơ sở trở thành các thành viên riêng của lớp dẫn xuất.

Trong mọi trường hợp, các thành viên private của lớp cơ sở vẫn là private. Các thành viên riêng của lớp cơ sở không thể được sử dụng bởi lớp dẫn xuất trừ khi khai báo bạn bè trong lớp cơ sở cấp quyền truy cập rõ ràng cho chúng.

**20.3.5.1 Trình tạo kế thừa**

Khi kế thừa từ một lớp, nếu bạn cần kế thừa các hàm tạo của nó mà không cần khởi tạo thêm, bạn nên using-declarationkế thừa tất cả các hàm tạo của lớp cơ sở thay vì viết chúng bằng tay.

using-declaration đối với hàm tạo kế thừa là một tính năng C ++ 11 làm cho tất cả các hàm tạo của cơ sở hiển thị với độ phân giải quá tải khi khởi tạo lớp dẫn xuất.

Nếu bạn cần thay đổi khả năng truy cập của một trong các hàm tạo được kế thừa, bạn có thể làm điều đó bằng cách giữ using-declarationvà khai báo hàm tạo đó một cách rõ ràng là private

20.4 Các lớp trừu tượng

Các lớp trừu tượng đóng vai trò là biểu thức của các khái niệm chung mà từ đó các lớp cụ thể hơn có thể được dẫn xuất. Bạn không thể tạo một đối tượng thuộc loại lớp trừu tượng. Tuy nhiên, bạn có thể sử dụng con trỏ và tham chiếu đến các kiểu lớp trừu tượng.

Bạn tạo một lớp trừu tượng bằng cách khai báo ít nhất một hàm thành viên ảo thuần túy. Đó là một hàm ảo được khai báo bằng cách sử dụng cú pháp specifier ( ) thuần túy= 0 . Các lớp dẫn xuất từ lớp trừu tượng phải thực hiện hàm ảo thuần túy hoặc chúng cũng là các lớp trừu tượng.

**20.5 Kiểm soát truy cập**

Trong một lớp được sử dụng để đặt khả năng truy cập của các thành viên lớp. Đó là, nó đặt ra một số hạn chế đối với các thành viên lớp không được truy cập trực tiếp bởi các hàm bên ngoài.

Trong C++ có hổ trợ 3 loại access modifier đó là:

* Private
* Public
* Protected

Lưu ý: Nếu chúng ta không chỉ rõ bất kỳ access modifier cho thành viên của lớp, thì mặc định nó là Private.

**20.5.1 Protected**

Protected thường được dùng khi bạn biết chắc là có lớp khác sẽ kế thừa lớp này và những phương thức, thuộc tính đó chỉ được dùng trong lớp kế thừa nó.

Giả sử bạn khai báo lớp Động Vật, trong đó có hàm lưu dữ liệu động vật vào database, hàm này dùng chung cho tất cả các lớp con kế thừa lớp Động Vật. Và để bảo mật nên tôi không muốn ở bên ngoài lớp có thể sử dụng được, vì vậy tôi khai báo là protected.

Đó là những ví dụ cơ bản, chứ thực tế thì cũng tùy vào từng bài toán cụ thể mà bạn lựa chọn

**20.5.2 Quyền truy cập vào các lớp cơ sở**

Cách mà chúng ta có thể lấy được các lớp gọi là các chế độ truy cập. Các chế độ truy cập này có tác dụng:

* Public: nếu một lớp con được khai báo ở chế độ public thì các thành phần của lớp cơ sở được kế thừa bởi lớp con giống như chúng.
* Private: trong trường hợp này, tất cả các thành phần của lớp cơ sở đều trở thành private trong lớp con.
* Protected: các thành phần public của lớp cơ sở trở thành protected trong lớp con.

Các thành phần private của lớp cơ sở luôn là private trong lớp con.

**20.6 Trỏ tới thành viên**

Con trỏ đến các thành viên cho phép bạn tham chiếu đến các thành viên không cố định của các đối tượng lớp. Bạn không thể sử dụng con trỏ tới thành viên để trỏ đến thành viên lớp tĩnh vì địa chỉ của thành viên tĩnh không được liên kết với bất kỳ đối tượng cụ thể nào. Để trỏ đến một thành viên lớp tĩnh, bạn phải sử dụng một con trỏ bình thường.

**Chương 21: Phân cấp lớp**

**21.1 Giới thiệu**

**21.2 Thiết kế cấu trúc phân cấp lớp**

Hãy xem xét một vấn đề thiết kế đơn giản: Cung cấp một cách để một chương trình (‘‘ một ứng dụng ’’) có được một tương tác giá trị mầm từ người dùng. Điều này có thể được thực hiện theo nhiều cách khác nhau. Để cách ly chương trình của chúng tôi từ sự đa dạng này và cũng để có cơ hội khám phá các lựa chọn thiết kế khả thi, chúng ta hãy bắt đầu bằng cách xác định mô hình chương trình của chúng tôi về thao tác nhập đơn giản này.

**21.2.1 Triển khai kế thừa**

Một lớp có thể được kế thừa từ hơn một lớp khác, nghĩa là, nó có thể kế thừa dữ liệu và hàm từ nhiều lớp cơ sở. Để định nghĩa một lớp kế thừa (Derived Class), chúng ta sử dụng một danh sách để xác định các lớp cơ sở. Danh sách này liệt kê một hoặc nhiều lớp cơ sở và có form sau:

class lop\_ke\_thua: access\_modifier lop\_co\_so

ví dụ :

#include <iostream>

using namespace std;

// lop co so: Hinh

class Hinh

{

public:

void setChieuRong(int rong)

{

chieurong = rong;

}

void setChieuCao(int cao)

{

chieucao = cao;

}

protected:

int chieurong;

int chieucao;

};

// day la lop ke thua: HinhChuNhat

class HinhChuNhat:

public Hinh

{

public:

int tinhDienTich()

{

return chieurong \* chieucao;

}

};

int main(void)

{

HinhChuNhat Hcn;

Hcn.setChieuRong(14);

Hcn.setChieuCao(30);

// in dien tich cua doi tuong.

cout << "Tong dien tich la: " << Hcn.tinhDienTich() << endl;

return 0;

}

**21.2.1.1 Phê bình**

Thiết kế này hoạt động tốt theo nhiều cách, và đối với nhiều vấn đề, kiểu phân cấp này là một giải pháp tốt . Tuy nhiên, có một số chi tiết khó xử có thể khiến chúng tôi phải tìm kiếm các thiết kế thay thế.

**21.2.2 Kế thừa giao diện**

Một cách sử dụng đa kế thừa không gây tranh cãi liên quan đến *kế thừa giao diện* . Trong C ++, tất cả kế thừa là kế thừa *thực thi* , bởi vì mọi thứ trong lớp cơ sở, giao diện và sự thực thi, đều trở thành một phần của lớp dẫn xuất. Không thể chỉ kế thừa một phần của một lớp (chẳng hạn như giao diện một mình). Như Chương 14 của Tập 1 đã giải thích, kế thừa riêng và được bảo vệ có thể hạn chế quyền truy cập vào các thành viên được kế thừa từ các lớp cơ sở khi được sử dụng bởi các máy khách của một đối tượng lớp dẫn xuất, nhưng điều này không ảnh hưởng đến lớp dẫn xuất; nó vẫn chứa tất cả dữ liệu lớp cơ sở và có thể truy cập tất cả các thành viên lớp cơ sở không riêng tư.

**21.3 Đa kế thừa**

Multiple Inheritance là một tính năng của C ++ trong đó một lớp có thể kế thừa từ nhiều hơn một lớp.

Các hàm tạo của các lớp kế thừa được gọi theo thứ tự mà chúng được kế thừa. Ví dụ, trong chương trình sau, phương thức khởi tạo của B được gọi trước phương thức khởi tạo của A.

Một lớp trong C++ có thể kế thừa các thành viên từ nhiều lớp, và đây là cú pháp:

class lop\_ke\_thua: access\_modifier lop\_co\_so\_1, access\_modifier lop\_co\_so\_2 ...

#include<iostream>

using namespace std;

class A

{

public:

A() { cout << "A's constructor called" << endl; }

};

class B

{

public:

B() { cout << "B's constructor called" << endl; }

};

class C: public B, public A // Note the order

{

public:

C() { cout << "C's constructor called" << endl; }

};

int main()

{

C c;

return 0;

}

Đầu ra:

Hàm tạo của B được gọi là

Hàm tạo của A được gọi là

Hàm tạo của C được gọi là

Các hàm hủy được gọi theo thứ tự ngược lại của các hàm tạo.

-Truy cập các thành phần lớp trong đa kế thừa :  
Việc truy nhập đến các thành phần của các lớp trong đa kế thừa được dựa trên các nguyên tắc sau:

* Việc truy nhập từ đối tượng lớp dẫn xuất đến các thành phần của mỗi lớp cơ sở được tuân theo quy tắc phạm vi tương tự như trong đơn kế thừa.
* Trong trường hợp các lớp cơ sở đều có các thành phần cùng tên, việc truy xuất đến thành phần của lớp nào phải được chỉ rõ bằng toán tử phạm vi: “<Tên lớp>::” đối với thành phần lớp cơ sở đó.