**CHƯƠNG 20 : LỚP DẪN XUẤT**

**1. Giới thiệu**

Từ Simula, C ++ đã vay mượn ý tưởng của các lớp và cấu trúc phân cấp lớp. Ngược lại, việc sử dụng các tính năng ngôn ngữ hỗ trợ cho các ý tưởng thiết kế sẽ phân biệt việc sử dụng C ++ hiệu quả. Một khái niệm (ý tưởng, khái niệm, v.v.) không tồn tại một cách cô lập. Ví dụ, cố gắng giải thích ô tô là gì. Bạn sẽ sớm giới thiệu các khái niệm về bánh xe, động cơ, người lái xe, người đi bộ, xe tải, xe cứu thương, đường, dầu, vé chạy quá tốc độ, nhà nghỉ, v.v. Vì chúng tôi sử dụng các lớp để biểu thị các khái niệm nên vấn đề trở thành cách thể hiện mối quan hệ giữa các khái niệm.

Khái niệm về một lớp dẫn xuất thể hiện các mối quan hệ thứ bậc, nghĩa là, để thể hiện tính chung giữa các lớp. **Ví dụ**, các khái niệm về hình tròn và hình tam giác có liên quan với nhau ở chỗ chúng đều là hình dạng; nghĩa là, họ có chung khái niệm về một hình dạng. Do đó, chúng ta định nghĩa rõ ràng lớp Circle và lớp Triangle để có chung lớp Shape. Trong trường hợp đó, lớp chung Shape, được gọi là lớp cơ sở hoặc siêu lớp và các lớp dẫn xuất từ ​​lớp đó, ở đây Circle và Triangle, được gọi là lớp dẫn xuất hoặc lớp con. Trong trường hợp đó, lớp chung, ở đây Shape, được gọi là lớp cơ sở hoặc siêu lớp và các lớp dẫn xuất từ ​​lớp đó, ở đây Circle và Triangle, được gọi là lớp dẫn xuất hoặc lớp con. Chương này là sự khám phá ý nghĩa của ý tưởng đơn giản này, là cơ sở cho cái thường được gọi là lập trình hướng đối tượng. Các tính năng ngôn ngữ hỗ trợ xây dựng các lớp mới từ những lớp hiện có:

• Kế thừa triển khai: để tiết kiệm nỗ lực triển khai bằng cách chia sẻ các cơ sở được cung cấp bởi một lớp cơ sở

• Kế thừa giao diện: cho phép các lớp dẫn xuất khác nhau được sử dụng thay thế cho nhau thông qua giao diện được cung cấp bởi một lớp cơ sở chung

Kế thừa giao diện thường được gọi là đa hình thời gian chạy (hoặc đa hình động). Ngược lại, việc sử dụng thống nhất các lớp không liên quan đến kế thừa được cung cấp bởi các mẫu, thường được gọi là đa hình thời gian biên dịch (hoặc đa hình tĩnh).

**2 .Lớp Dẫn Xuất**

**Employee**

**↑**

**Manager**

Một lớp dẫn xuất thường được cho là kế thừa các thuộc tính từ cơ sở của nó, còn được gọi là kế thừa. Một lớp cơ sở được gọi là lớp cha và lớp dẫn xuất là lớp con.

Ta có một chương trình định nghĩa cấu trúc nhân viên:

struct Employee{

string name,

char middle\_initial;

}

Và định nghĩa cấu trúc người quảnn lý

struct Manager {

Employee emp;

list<Employee\*> group;

}

**Hoặc sơ đồ:**

|  |
| --- |
| first\_name  family\_name  ... |

**Employee: Manager:**

|  |
| --- |
| first\_n*ame*  *family*\_name  ... |
| group  level  ... |

**3. Hàm thành viên**

Một hàm thành viên của một lớp dẫn xuất có thể truy cập vào các thành viên riêng của lớp cơ sở của nó. Thành viên riêng sẽ trở nên vô nghĩa nếu cho phép một lập trình viên truy cập vào phần riêng của một lớp chỉ đơn giản bằng cách dẫn xuất một lớp mới từ nó

**4. Phân cấp lớp**

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

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

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

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

**5. 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.

**class Employee {**

**public:**

**Employee(const string& name, int dept);**

**virtual void print() const;**

**// ...**

**private:**

**string first\_name , family\_name;**

**short depar tment;**

**// ...};**

Một hàm ảo có thể được sử dụng ngay cả khi không có lớp nào được dẫn xuất từ lớp của nó và một lớp dẫn xuất không cần phiên bản riêng của hàm ảo thì không cần cung cấp một lớp.Ví dụ:

**class Manager : public Employee {**

**public:**

**Manager(const string& name, int dept, int lvl);**

**void print() const;**

**// ...**

**private:**

**list<Employee∗> group;**

**short level;**

**// ...};**

**void Manager::print() const{**

**Employee::print();**

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

**// ...}**

**5. Kiểm soát truy cập**

Thành viên của một lớp có thể là riêng tư, được bảo vệ hoặc công khai:

• Nếu nó là private, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo.

• Nếu nó được bảo vệ, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo và bởi các hàm thành viên và bạn bè của các lớp dẫn xuất từ lớp này .

• Nếu nó là công khai, tên của nó có thể được sử dụng bởi bất kỳ chức năng nào.

người dùng chung

các hàm thành viên và bạn bè của lớp dẫn xuất

chức năng thành viên riêng và bạn bè

public:

protected:

private:

**6.Quyền truy cập vào các lớp cơ sở**

Giống như một thành viên, một lớp cơ sở có thể được khai báo là riêng tư, được bảo vệ hoặc công khai. Ví dụ:

class X : public B { /\* ... \*/ };

class Y : protected B { /\* ... \*/ };

class Z : private B { /\* ... \*/

Các chỉ số truy cập khác nhau phục vụ các nhu cầu thiết kế khác nhau:

• công khai dẫn xuất làm cho lớp dẫn xuất trở thành một kiểu con của cơ sở của nó. Ví dụ, X là một loại B. Đây là dạng dẫn xuất phổ biến nhất.

• Các cơ sở riêng hữu ích nhất khi xác định một lớp bằng cách giới hạn giao diện cho một cơ sở để có thể cung cấp các đảm bảo mạnh mẽ hơn. Ví dụ, B là một chi tiết triển khai của Z .

• Các cơ sở được bảo vệ rất hữu ích trong các cấu trúc phân cấp lớp trong đó dẫn xuất thêm là tiêu chuẩn. Giống như dẫn xuất riêng, dẫn xuất được bảo vệ được sử dụng để biểu diễn các chi tiết triển khai.. Ví dụ:

class XX : B { /\* ... \*/ }; // B is a private base

struct YY : B { /\* ... \*/ }; // B is a public base

**Chương 21: CẤU TRÚC PHÂN CẤP LỚP**

**1. Giới thiệu**

Trọng tâm chính của chương này là các kỹ thuật thiết kế, hơn là các tính năng ngôn ngữ. Đề thi được lấy từ thiết kế giao diện người dùng, nhưng tôi tránh chủ đề về lập trình hướng sự kiện thường được sử dụng cho các hệ thống giao diện người dùng đồ họa (GUI). Một cuộc thảo luận về chính xác cách một hành động trên màn hình được chuyển thành lời gọi của một hàm thành viên sẽ thêm ít vào các vấn đề của thiết kế phân cấp lớp. Để hiểu về GUI, hãy xem một trong nhiều thư viện C ++ GUI

**2. Thiết kế cấu trúc phân cấp lớp**

**User Ival\_box:**

Value

**(via ‘‘system’’) application**

**set\_value() get\_value()**

**3. Kế thừa triển khai**

Lớp Ival\_box xác định giao diện cơ bản cho tất cả các Ival\_box và chỉ định triển khai mặc định mà các loại Ival\_box cụ thể hơn có thể ghi đè bằng các phiên bản của riêng chúng. Ngoài ra, chúng tôi khai báo dữ liệu cần thiết để triển khai khái niệm cơ bản:

**class Ival\_box {**

**protected:**

**int val;**

**int low, high;**

**bool chang ed {false}; // thay đổi bởi người dùng set\_value()**

**public:**

**Ival\_box(int ll, int hh) :val{ll}, low{ll}, high{hh} { }**

**virtual int get\_value() { chang ed = false; return val; } // cho ứng dụng**

**virtual void set\_value(int i) { chang ed = true; val = i; } // cho người dùng**

**virtual void reset\_value(int i) { chang ed = false; val = i; } // cho ứng dụng**

**virtual void prompt() { }**

**virtual bool was\_chang ed() const { return chang ed; }**

**virtual ̃Ival\_box() {};**

**};**

Việc triển khai mặc định của các chức năng là khá cẩu thả và được cung cấp ở đây chủ yếu để đánh lừa ngữ nghĩa dự định. Ví dụ, một lớp thực tế sẽ cung cấp một số kiểm tra phạm vi.

Một lập trình viên có thể sử dụng '' các lớp ival '' như thế này:

**void interact(Ival\_box∗ pb)// tương tác**

**{**

**pb−>prompt(); // aler t user**

**// ...**

**int i = pb−>get\_value();**

**if (pb−>was\_chang ed()) {**

**// ... new value; do something ...// giá trị khác , làm viễ gì đó**

**}**

**else {**

**// ... do something else ...// làm việc khác**

**}}**

**void some\_fct()**

**{**

**unique\_ptr<Ival\_box> p1 {new Ival\_slider{0,5}}; // Ival\_slider bắt nguồn Ival\_box**

**interact(p1.get());**

**unique\_ptr<Ival\_box> p2 {new Ival\_dial{1,12}};**

**interact(p2.get());**

**}**

Mã ứng dụng được viết dưới dạng (con trỏ tới) Ival\_boxes thuần túy như cách tương tác (). Bằng cách đó, ứng dụng không cần biết về số lượng lớn các biến thể tiềm năng của khái niệm Ival\_box. Điều này cách ly người dùng khỏi những thay đổi trong việc triển khai các lớp dẫn xuất.

Có thể chương trình thực sự đợi người dùng trong get\_value () (ví dụ: sử dụng get () trong tương lai, có thể chương trình liên kết Ival\_box với một sự kiện và chuẩn bị trả lời một lệnh gọi lại, hoặc có thể chương trình tạo ra một luồng cho Ival\_box và sau đó sẽ hỏi về trạng thái của luồng đó.

Các loại Ival\_box khác nhau được định nghĩa là các lớp bắt nguồn từ Ival\_box. Ví dụ:

**class Ival\_slider : public Ival\_box {**

**private:**

**// ...nội dung đồ họa để quyết định thanh trượt trông như thế nào, etc. ...**

**public:**

**Ival\_slider(int, int);**

**int get\_value() override; // lấy hàm từ người dùng và ứng dụng gửi vào val**

**void prompt() override;// ghi đè };**

Các thành viên dữ liệu của Ival\_box đã được khai báo được bảo vệ để cho phép truy cập từ các lớp dẫn xuất. Do đó, Ival\_slider :: g et\_value () có thể gửi một giá trị vào Ival\_box :: val. Thành viên được bảo vệ có thể truy cập được từ các thành viên của chính một lớp và từ các thành viên của các lớp dẫn xuất, nhưng không phải đối với người dùng thông thường .

Hầu hết các hệ thống giao diện người dùng cung cấp một lớp xác định các thuộc tính cơ bản của một thực thể trên màn hình. Vì vậy, nếu chúng tôi sử dụng hệ thống từ ‘‘ Big Bucks Inc. ’’, chúng tôi sẽ phải làm cho mỗi lớp Ival\_slider, Ival\_dial, v.v., của chúng tôi trở thành một loại BBwid-get. Điều này đơn giản nhất sẽ đạt được bằng cách viết lại Ival\_box của chúng tôi để nó bắt nguồn từ BBwidget. Theo cách đó, tất cả các lớp của chúng ta kế thừa tất cả các thuộc tính của một BBwidget. Ví dụ: mọi Ival\_box đều có thể được đặt trên màn hình, tuân theo các quy tắc kiểu đồ họa, được thay đổi kích thước, được kéo xung quanh, v.v., phù hợp với tiêu chuẩn do hệ thống BBwidget thiết lập. Hệ thống phân cấp lớp của chúng ta sẽ trông như thế này:

**class Ival\_box : public BBwidget { /\* ... \*/ }; // viết lại người dùng để sử dụng Bbwidget**

**class Ival\_slider : public Ival\_box { /\* ... \*/ };**

**class Ival\_dial : public Ival\_box { /\* ... \*/ };**

**class Flashing\_ival\_slider : public Ival\_slider { /\* ... \*/ };**

**class Popup\_ival\_slider : public Ival\_slider { /\* ... \*/ };**

**hoặc bằng đồ thị**:

BBwidget

Ival\_box

Ival\_slider Ival\_dial

Popup\_ival\_slider Flashing\_ival\_slider

**4.Nhiều lớp triển khai**

Một mô phỏng của các thiên thể quay quanh Trái đất, trong đó các vật thể quay xung quanh được thể hiện như đối tượng của lớp Vệ tinh. Một đối tượng Vệ tinh sẽ chứa quỹ đạo, kích thước, hình dạng, albedo, tham số mật độ, v.v. và cung cấp các hoạt động để tính toán quỹ đạo, sửa đổi thuộc tính, v.v.

**class Comm\_sat : public Satellite, public Displayed {**

**public: // ...**

**};**

**hoặc bằng đồ thị:**

Satellite Displayed

Comm\_sat

Việc thực hiện điều này rõ ràng liên quan đến một số kỹ thuật biên dịch (đơn giản) để đảm bảo rằng các chức năng mong đợi một Vệ tinh nhìn thấy một phần khác của Comm\_sat so với các chức năng mong đợi một Hiển thị. Các chức năng ảo hoạt động như bình thường. Ví dụ:

**class Satellite {**

**public:**

**virtual Pos center() const = 0; // center of gravity**

**// ...};**

**class Displayed {**

**public:**

**virtual void draw() = 0;**

**// ...};**

**class Comm\_sat : public Satellite, public Displayed {**

**public:**

**Pos center() const override; // overr ide Satellite::center()**

**void draw() override; // over ide Displayed::draw()**

**// ...};**