**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, 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 {/ \* ... \* /};**

**20.3.2 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

21.1 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 và có khả năng gây mất tập trung rất lớn: đó là một chủ đề thú vị và quan trọng theo đúng nghĩa của nó. Để hiểu về GUI, hãy xem một trong nhiều thư viện C ++ GUI

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 ’’) nhận được giá trị inte-ger từ người dùng. Điều này có thể được thực hiện theo nhiều cách khác nhau. Để cách ly chương trình của chúng tôi khỏi 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.

Ý tưởng là có một lớp Ival\_box (‘‘ hộp nhập giá trị số nguyên ’’) biết nó sẽ chấp nhận phạm vi giá trị đầu vào nào. Một chương trình có thể hỏi Ival\_box về giá trị của nó và yêu cầu nó nhắc người dùng nếu cần. Ngoài ra, một chương trình có thể hỏi Ival\_box nếu người dùng đã thay đổi giá trị kể từ khi chương trình đó xem xét nó:

**User Ival\_box:**

Value

**(via ‘‘system’’) application**

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

Bởi vì có nhiều cách để triển khai ý tưởng cơ bản này, chúng ta phải giả định rằng sẽ có nhiều loại Ival\_box khác nhau, chẳng hạn như thanh trượt, hộp trơn trong đó người dùng có thể nhập số, quay số và tương tác bằng giọng nói.

Cách tiếp cận chung là xây dựng một '' hệ thống giao diện người dùng ảo '' để ứng dụng sử dụng. Hệ thống này cung cấp một số dịch vụ được cung cấp bởi các hệ thống giao diện người dùng hiện có. Nó có thể được thực hiện trên nhiều hệ thống khác nhau để đảm bảo tính di động của mã ứng dụng. Đương nhiên, có những cách khác để cách ly một ứng dụng khỏi hệ thống giao diện người dùng. Tôi chọn cách tiếp cận này vì nó chung chung, vì nó cho phép tôi chứng minh nhiều kỹ thuật và sự cân bằng trong thiết kế, bởi vì những kỹ thuật đó cũng là những kỹ thuật được sử dụng để xây dựng hệ thống giao diện người dùng '' thực '' và - quan trọng nhất - bởi vì những Ngoài việc bỏ qua chủ đề về cách ánh xạ các hành động (sự kiện) của người dùng với các lệnh gọi thư viện, tôi cũng bỏ qua sự cần thiết của việc khóa trong một hệ thống GUI đa luồng.

21.2.1 Kế thừa triển khai

Giải pháp đầu tiên của chúng tôi là một hệ thống phân cấp lớp sử dụng kế thừa triển khai (như thường thấy ở các chương trình cũ hơn).

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());**

**}**

Hầu hết 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. Kiến thức của các lớp chuyên biệt như vậy bị cô lập trong tương đối ít các hàm tạo ra các đối tượng như vậy. Điều này cách ly người dùng khỏi những thay đổi trong việc triển khai các lớp dẫn xuất. Hầu hết các mã có thể bị lãng quên bởi thực tế là có nhiều loại Ival\_box khác nhau.

Tôi sử dụng unique\_ptr (§5.2.1, §34.3.1) để tránh quên xóa ival\_boxes .Để đơn giản hóa cuộc thảo luận, tôi không giải quyết các vấn đề về cách một chương trình chờ đợi đầu vào. Có thể chương trình thực sự đợi người dùng trong get\_value () (ví dụ: sử dụng get () trong tương lai; §5.3.5.1), 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. Những quyết định như vậy rất quan trọng trong việc thiết kế các hệ thống giao diện người dùng.Tuy nhiên, thảo luận về chúng ở đây trong bất kỳ chi tiết thực tế nào sẽ chỉ đơn giản là phân tán sự trình bày của các kỹ thuật lập trình và cơ sở ngôn ngữ. Các kỹ thuật thiết kế được mô tả ở đây và các phương tiện ngôn ngữ hỗ trợ chúng không dành riêng cho giao diện người dùng. Chúng áp dụng cho một loạt các vấn đề.

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 (xem §20.5).

Ngoài Ival\_slider, chúng tôi sẽ xác định các biến thể khác của khái niệm Ival\_box. Chúng có thể bao gồm Ival\_dial, cho phép bạn chọn một giá trị bằng cách xoay một núm; Flashing\_ival\_slider, nhấp nháy khi bạn yêu cầu nó nhắc (); và Popup\_ival\_slider, phản hồi với prompt () bằng cách xuất hiện ở một số nơi nổi bật, do đó khiến người dùng khó có thể bỏ qua. Chúng ta sẽ lấy đồ họa từ đâu? 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

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 tìm kiếm các thiết kế thay thế.

Chúng tôi đã trang bị thêm BBwidget làm cơ sở của Ival\_box. Điều này không hoàn toàn đúng (ngay cả khi phong cách này là com-mon trong hệ thống thế giới thực). Việc sử dụng BBwidget không nằm trong khái niệm cơ bản của chúng tôi về Ival\_box; nó là một chi tiết thực hiện. Việc lấy Ival\_box từ BBwidget đã nâng chi tiết triển khai lên thành quyết định thiết kế cấp đầu tiên. Điều đó có thể đúng. Ví dụ: sử dụng môi trường được xác định bởi ‘‘ Big Bucks Inc. ’’ có thể là một quyết định quan trọng dựa trên cách tổ chức của chúng tôi tiến hành hoạt động kinh doanh của mình. Tuy nhiên, điều gì sẽ xảy ra nếu chúng tôi cũng muốn triển khai Ival\_boxes của mình cho các hệ thống từ ‘‘ Imperial Bananas ’’, ‘‘ Phần mềm được giải phóng ’’ và ‘‘ Compiler Whizzes ’’? Chúng tôi sẽ phải duy trì bốn phiên bản dis-tinct của chương trình của mình:

**class Ival\_box : public BBwidget { /\* ... \*/ }; // BB version**

**class Ival\_box : public CWwidget { /\* ... \*/ }; // CW version**

**class Ival\_box : public IBwidget { /\* ... \*/ }; // IB version**

**class Ival\_box : public LSwindow { /\* ... \*/ }; // LS version**

Có nhiều phiên bản có thể dẫn đến cơn ác mộng kiểm soát phiên bản.

Trong thực tế, chúng ta khó có thể tìm thấy một sơ đồ tiền tố hai chữ cái đơn giản, mạch lạc. Nhiều khả năng, các thư viện từ các bộ cung cấp khác nhau sẽ ở các không gian tên khác nhau và sử dụng các thuật ngữ khác nhau cho các khái niệm tương tự, chẳng hạn như BigBucks :: Widg et, Wizzies :: control và LS :: window. Nhưng điều đó không ảnh hưởng đến cuộc thảo luận về thiết kế phân cấp lớp của chúng ta, vì vậy để đơn giản hóa, tôi bỏ qua các vấn đề về đặt tên và không gian tên.

Một vấn đề khác là mọi lớp dẫn xuất đều chia sẻ dữ liệu cơ bản được khai báo trong Ival\_box. Tất nhiên, dữ liệu đó là chi tiết triển khai cũng được đưa vào giao diện Ival\_box của chúng tôi. Từ quan điểm thực tế, nó cũng là dữ liệu sai trong nhiều trường hợp. Ví dụ: Ival\_slider không cần giá trị được lưu trữ cụ thể. Nó có thể dễ dàng được tính toán từ vị trí của thanh trượt khi ai đó thực hiện get\_value (). Nói chung, việc giữ hai bộ dữ liệu có liên quan, nhưng khác nhau, là yêu cầu khó khăn. Không sớm thì muộn ai đó sẽ khiến chúng không đồng bộ. Ngoài ra, kinh nghiệm cho thấy rằng những người lập trình chuyên nghiệp mới làm quen có xu hướng làm rối tung dữ liệu được bảo vệ theo những cách không cần thiết và gây ra các vấn đề về bảo trì. Các thành viên dữ liệu được giữ kín tốt hơn để người viết các lớp dẫn xuất không thể gây rối với chúng. Vẫn tốt hơn, dữ liệu nên nằm trong các lớp dẫn xuất, nơi nó có thể được định nghĩa để khớp chính xác với các yêu cầu và không thể làm phức tạp tuổi thọ của các lớp dẫn xuất không liên quan. Trong hầu hết các trường hợp, một giao diện được bảo vệ chỉ nên chứa các hàm, kiểu và hằng số.

Bắt nguồn từ BBwidget mang lại lợi ích là làm cho các tiện ích do BBwidget cung cấp có thể hoạt động được cho người dùng Ival\_box. Thật không may, điều đó cũng có nghĩa là những thay đổi đối với lớp BBwidget có thể buộc người dùng phải biên dịch lại hoặc thậm chí viết lại mã của họ để khôi phục từ những thay đổi đó. Đặc biệt, cách hoạt động của hầu hết các triển khai C ++ ngụ ý rằng một sự thay đổi về kích thước của một lớp cơ sở yêu cầu thực hiện thử nghiệm lại tất cả các lớp dẫn xuất.

Cuối cùng, chương trình của chúng tôi có thể phải chạy trong một môi trường hỗn hợp, trong đó các cửa sổ của các hệ thống giao diện người dùng khác nhau cùng tồn tại. Điều này có thể xảy ra do hai hệ thống bằng cách nào đó chia sẻ màn hình hoặc vì chương trình của chúng tôi cần giao tiếp với người dùng trên các hệ thống khác nhau. Việc đặt hệ thống giao diện người dùng của chúng tôi '' có dây '' làm cơ sở duy nhất và duy nhất của giao diện Ival\_box duy nhất của chúng tôi là không đủ linh hoạt để xử lý những tình huống đó.

21.2.2 Kế thừa giao diện

Vì vậy, hãy bắt đầu lại và xây dựng một hệ thống phân cấp lớp mới để giải quyết các vấn đề được trình bày trong điểm mấu chốt của hệ thống phân cấp truyền thống:

[1] Hệ thống giao diện người dùng phải là một chi tiết triển khai bị ẩn với những người dùng không muốn biết về nó.

[2] Lớp Ival\_box không được chứa dữ liệu.

[3] Không cần biên dịch lại mã sử dụng họ lớp Ival\_box sau khi thay đổi hệ thống giao diện người dùng.

[4] Ival\_boxes cho các hệ thống giao diện khác nhau sẽ có thể cùng tồn tại trong chương trình của chúng tôi. Có thể thực hiện một số cách tiếp cận thay thế để đạt được điều này. Ở đây, tôi trình bày một bản đồ rõ ràng sang ngôn ngữ C ++.

Đầu tiên, tôi chỉ định lớp Ival\_box làm giao diện thuần túy:

**class Ival\_box {**

**public:**

**virtual int get\_value() = 0;**

**virtual void set\_value(int i) = 0;**

**virtual void reset\_value(int i) = 0;**

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

**virtual bool was\_chang ed() const = 0;**

**virtual ̃Ival\_box() { }**

**};**

Điều này rõ ràng hơn nhiều so với khai báo ban đầu của Ival\_box. Dữ liệu đã biến mất và các triển khai mô phỏng của các chức năng thành viên cũng vậy. Gone cũng là phương thức khởi tạo, vì không có dữ liệu nào để nó khởi tạo. Thay vào đó, tôi đã thêm một trình hủy ảo để đảm bảo dọn dẹp dữ liệu phù hợp sẽ được xác định trong các lớp dẫn xuất.

Định nghĩa của Ival\_slider có thể giống như sau:

**class Ival\_slider : public Ival\_box, protected BBwidget {**

**public:**

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

**̃Ival\_slider() override;**

**int get\_value() override;**

**void set\_value(int i) override;**

**// ...**

**protected:**

**// ... functions overr iding BBwidget virtual functions**

**// e.g., BBwidget::draw(), BBwidget::mouse1hit() ...**

**private:**

**// ...** **dữ liệu cần thiết cho thanh trượt...**

**};**

Lớp dẫn xuất Ival\_slider kế thừa từ một lớp trừu tượng (Ival\_box) yêu cầu nó triển khai các hàm ảo thuần túy của lớp cơ sở. Nó cũng kế thừa từ BBwidget cung cấp cho nó các phương tiện để làm như vậy. Vì Ival\_box cung cấp giao diện cho lớp dẫn xuất, nên nó được dẫn xuất bằng cách sử dụng public. Vì BBwidget chỉ là một công cụ hỗ trợ triển khai, nó được dẫn xuất bằng cách sử dụng bảo vệ (§20.5.2). Điều này ngụ ý rằng một lập trình viên sử dụng Ival\_slider không thể trực tiếp sử dụng các cơ sở được xác định bởi BBwidget. Giao diện do Ival\_slider cung cấp là giao diện được kế thừa từ Ival\_box, cộng với những gì Ival\_slider tuyên bố rõ ràng. Tôi đã sử dụng dẫn xuất được bảo vệ thay vì dẫn xuất riêng hạn chế hơn (và thường an toàn hơn) để cung cấp BBwidget cho các lớp bắt nguồn từ Ival\_slider. Tôi đã sử dụng tính năng ghi đè rõ ràng vì ‘‘ hệ thống phân cấp tiện ích con ’’ này chính xác là loại hệ thống phân cấp lớn và phức tạp, nơi rõ ràng có thể giúp giảm thiểu sự nhầm lẫn.

Xuất phát trực tiếp từ nhiều hơn một lớp thường được gọi là đa kế thừa (§21.3). Lưu ý rằng Ival\_slider phải ghi đè các hàm từ cả Ival\_box và BBwidget. Do đó, nó phải được xuất phát trực tiếp hoặc gián tiếp từ cả hai. Như được trình bày trong §21.2.1.1, việc lấy Ival\_slider gián tiếp từ BBwidget bằng cách biến BBwidget thành cơ sở của Ival\_box là có thể, nhưng làm như vậy có tác dụng phụ không mong muốn. Tương tự, đặt BBwidget ‘‘ lớp triển khai ’’ làm thành viên của Ival\_box không phải là giải pháp bởi vì một lớp không thể ghi đè các chức năng ảo của các thành viên của nó. Việc đại diện cho win-down của một thành viên BBwidget ∗ trong Ival\_box dẫn đến một thiết kế hoàn toàn khác với một bộ cân bằng riêng biệt.

Đối với một số người, các từ '' đa kế thừa '' cho thấy điều gì đó phức tạp và đáng sợ. các giao diện. Nói cách khác, việc sử dụng lớp trừu tượng Ival\_box gần giống với việc sử dụng một giao diện trong Java hoặc C #.

Điều thú vị là khai báo Ival\_slider này cho phép mã ứng dụng được viết chính xác như trước. Tất cả những gì chúng tôi đã làm là tái cấu trúc các chi tiết triển khai theo hướng hợp lý hơn.

Nhiều lớp yêu cầu một số hình thức dọn dẹp cho một đối tượng trước khi nó biến mất. Vì lớp trừu tượng Ival\_box không thể biết liệu một lớp dẫn xuất có yêu cầu dọn dẹp như vậy hay không, nó phải giả định rằng nó yêu cầu một số. Chúng tôi đảm bảo dọn dẹp thích hợp bằng cách xác định hàm hủy ảo Ival\_box :: ̃Ival\_box () trong cơ sở và ghi đè nó một cách thích hợp trong các lớp dẫn xuất. Ví dụ:

**void f(Ival\_box∗ p)**

**{**

**// ...**

**delete p;**

**}**

Toán tử xóa hủy rõ ràng đối tượng được trỏ tới bởi p. Chúng tôi không có cách nào để biết chính xác đối tượng được trỏ đến bởi p thuộc về lớp nào, nhưng nhờ bộ hủy ảo của Ival\_box, việc dọn dẹp thích hợp như (tùy chọn) được định nghĩa bởi bộ hủy của lớp đó sẽ được thực hiện.

Hệ thống phân cấp Ival\_box hiện có thể được định nghĩa như sau:

**class Ival\_box { /\* ... \*/ };**

**class Ival\_slider**

**: public Ival\_box, protected BBwidget { /\* ... \*/ };**

**class Ival\_dial**

**: public Ival\_box, protected BBwidget { /\* ... \*/ };**

**class Flashing\_ival\_slider**

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

**class Popup\_ival\_slider**

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

**hoặc bằng đồ thị:**

BBwidget Ival\_box Bbwidget

Ival\_slider Ival\_dial

Popup\_\_ival\_slider Flashing\_ival\_slider

Tôi đã sử dụng một đường đứt nét để biểu thị sự kế thừa được bảo vệ (§20.5.1). Người dùng thông thường không thể truy cập các cơ sở được bảo vệ bởi vì họ được (một cách chính xác) được coi là một phần của việc triển khai.

21.2.3 Triển khai Thay thế

Thiết kế này sạch hơn và dễ bảo trì hơn thiết kế truyền thống - và không kém phần hiệu quả.

Tuy nhiên, nó vẫn không giải quyết được vấn đề kiểm soát phiên bản:

**class Ival\_box { /\* ... \*/ }; // common**

**class Ival\_slider**

**: public Ival\_box, protected BBwidget { /\* ... \*/ }; // for BB**

**class Ival\_slider**

**: public Ival\_box, protected CWwidget { /\* ... \*/ }; // for CW**

**// ...**

Không có cách nào để Ival\_slider cho BBwidgets cùng tồn tại với Ival\_slider cho CWwidgets, ngay cả khi hai hệ thống giao diện người dùng có thể cùng tồn tại. Giải pháp rõ ràng là xác định một số lớp Ival\_slider khác nhau với các tên riêng biệt:

**class Ival\_box { /\* ... \*/ };**

**class BB\_ival\_slider**

**: public Ival\_box, protected BBwidget { /\* ... \*/ };**

**class CW\_ival\_slider**

**: public Ival\_box, protected CWwidget { /\* ... \*/ };**

**// ...**

**hoặc bằng đồ thị:**

**Bbwidget Ival\_box Cwwidget**

**BB\_ival\_slider CW\_ival\_slider**

Để cách ly các lớp Ival\_box hướng ứng dụng của chúng ta khỏi các chi tiết triển khai, chúng ta có thể lấy một lớp Ival\_slider trừu tượng từ Ival\_box và sau đó lấy các Ival\_sliders dành riêng cho hệ thống từ điều đó:

**class Ival\_box { /\* ... \*/ };**

**class Ival\_slider**

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

**class BB\_ival\_slider**

**: public Ival\_slider, protected BBwidget { /\* ... \*/ };**

**class CW\_ival\_slider**

**: public Ival\_slider, protected CWwidget { /\* ... \*/ };**

**// ...**

**hoặc bằng đồ thị:**

**Ival\_box**

**Bbwidget Ival\_slider CWwidget**

**BB\_ival\_slider CW\_ival\_slider**

Thông thường, chúng ta có thể làm tốt hơn bằng cách sử dụng các lớp cụ thể hơn trong hệ thống phân cấp triển khai. Ví dụ: nếu hệ thống ‘‘ Big Bucks Inc. ’’ có lớp thanh trượt, chúng ta có thể lấy Ival\_slider của mình trực tiếp từ BBslider:

**class BB\_ival\_slider**

**: public Ival\_slider, protected BBslider { /\* ... \*/ };**

**class CW\_ival\_slider**

**: public Ival\_slider, protected CWslider { /\* ... \*/ };**

**hoặc bằng đồ thị:**

**BBwidget Ival\_box Cwwidget**

**Bbslider Ival\_slider Cwslider**

**BB\_ival\_slider CW\_ival\_slider**

Sự cải tiến này trở nên đáng kể khi - không có gì lạ - những phần tóm tắt của chúng tôi không quá khác biệt so với những phần được cung cấp bởi hệ thống được sử dụng để triển khai. Trong trường hợp đó, việc ghép chương trình được rút gọn thành ánh xạ giữa các khái niệm tương tự. Việc lấy nguồn từ các lớp cơ sở chung, chẳng hạn như BBwidget, hiếm khi được thực hiện.

Hệ thống phân cấp hoàn chỉnh sẽ bao gồm hệ thống phân cấp khái niệm theo hướng ứng dụng ban đầu của chúng tôi gồm các giao diện được biểu thị dưới dạng các lớp dẫn xuất:

**class Ival\_box { /\* ... \*/ };**

**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 { /\* ... \*/ };**

tiếp theo là việc triển khai hệ thống phân cấp này cho các hệ thống giao diện người dùng đồ họa khác nhau, được biểu thị dưới dạng các lớp dẫn xuất:

**class BB\_ival\_slider**

**: public Ival\_slider, protected BBslider { /\* ... \*/ };**

**class BB\_flashing\_ival\_slider**

**: public Flashing\_ival\_slider, protected BBwidget\_with\_bells\_and\_whistles { /\* ... \*/ };**

**class BB\_popup\_ival\_slider**

**: public Popup\_ival\_slider, protected BBslider { /\* ... \*/ };**

**class CW\_ival\_slider**

**: public Ival\_slider, protected CWslider { /\* ... \*/ };**

**// ...**

Sử dụng các chữ viết tắt rõ ràng, hệ thống phân cấp này có thể được biểu diễn bằng đồ thị như sau:

Ival\_box

Ival\_slider Ival\_dial

Ipopup iflash

BBslider Bbslider CWsl CWsl BBb&w CWsl

Bbislider BBipop Cwipop Cwifl BBifl Cwislider

Cấu trúc phân cấp lớp Ival\_box ban đầu dường như không thay đổi được bao quanh bởi các lớp thực thi.

21.2.3.1 Phê bình

Thiết kế lớp trừu tượng linh hoạt và gần như đơn giản để xử lý như thiết kế tương đương dựa trên cơ sở chung xác định hệ thống giao diện người dùng. Trong thiết kế thứ hai, lớp cửa sổ là gốc của cây. Trước đây, cấu trúc phân cấp lớp ứng dụng ban đầu dường như không thay đổi dưới dạng gốc của các lớp cung cấp các triển khai của nó. Theo quan điểm của ứng dụng, các thiết kế này tương đương nhau theo nghĩa mạnh rằng hầu như tất cả các mã đều hoạt động không thay đổi và theo cùng một cách trong hai trường hợp. Trong cả hai trường hợp, bạn có thể xem dòng Ival\_box của các lớp mà không cần bận tâm đến hầu hết thời gian thực hiện chi tiết liên quan đến cửa sổ. Ví dụ: chúng ta sẽ không cần phải viết lại tương tác () từ §21.2.1 nếu chúng ta chuyển từ cấu trúc phân cấp lớp này sang cấu trúc phân cấp khác.

Trong cả hai trường hợp, việc triển khai mỗi lớp Ival\_box phải được viết lại khi giao diện công cộng của hệ thống giao diện người dùng thay đổi. Tuy nhiên, trong thiết kế lớp trừu tượng, hầu như tất cả mã người dùng đều được bảo vệ chống lại những thay đổi đối với hệ thống phân cấp triển khai và không yêu cầu sửa lại sau khi thay đổi như vậy. Điều này đặc biệt quan trọng khi nhà cung cấp hệ thống cấp bậc triển khai đưa ra bản phát hành '' gần như tương thích '' mới. Ngoài ra, người dùng của hệ thống phân cấp lớp trừu tượng ít có nguy cơ bị khóa vào một triển khai độc quyền hơn là người dùng của hệ thống phân cấp cổ điển. Người dùng của hệ thống phân cấp ứng dụng lớp trừu tượng Ival\_box không thể vô tình sử dụng các mối quan hệ từ việc triển khai bởi vì chỉ các cơ sở được chỉ định rõ ràng trong hệ thống phân cấp Ival\_box mới có thể truy cập được; không có gì được thừa kế một cách ngầm định từ một lớp cơ sở triển khai cụ thể.

Kết luận hợp lý của dòng suy nghĩ này là một hệ thống được đại diện cho người dùng như một hệ thống phân cấp của các lớp trừu tượng và được thực hiện bởi một hệ thống phân cấp cổ điển. Nói cách khác:

• Sử dụng các lớp trừu tượng để hỗ trợ kế thừa giao diện (§3.2.3, §20.1).

• Sử dụng các lớp cơ sở với việc triển khai các hàm ảo để hỗ trợ việc triển khai inher-

itance (§3.2.3, §20.1).

21.2.4 Bản địa hóa Tạo đối tượng

Hầu hết một ứng dụng có thể được viết bằng giao diện Ival\_box. Hơn nữa, nếu các mặt liên có nguồn gốc phát triển để cung cấp nhiều tiện ích hơn Ival\_box đơn thuần, thì hầu hết ứng dụng có thể được ghi bằng giao diện Ival\_box, Ival\_slider, v.v.,. Tuy nhiên, việc tạo các đối tượng phải được thực hiện bằng cách sử dụng các tên dành riêng cho việc triển khai như CW\_ival\_dial và BB\_flashing\_ival\_slider. Chúng tôi muốn giảm thiểu số lượng những nơi xuất hiện những cái tên cụ thể như vậy và việc tạo đối tượng khó bản địa hóa trừ khi nó được thực hiện một cách có hệ thống. Như thường lệ, giải pháp là đưa ra một hướng dẫn.

Điều này có thể được thực hiện bằng nhiều cách. Một cách đơn giản là giới thiệu một lớp trừu tượng để đại diện cho tập hợp các thao tác tạo:

**class Ival\_maker {**

**public:**

**virtual Ival\_dial∗ dial(int, int) =0; //thực hiện quay số**

**virtual Popup\_ival\_slider∗ popup\_slider(int, int) =0; // tạo thanh trượt bật lên**

**// ...**

**};**

Đối với mỗi giao diện từ họ lớp Ival\_box mà người dùng nên biết, lớp Ival\_maker cung cấp một hàm tạo một đối tượng. Một lớp như vậy đôi khi được gọi là một nhà máy, và các chức năng của nó (hơi gây hiểu lầm) đôi khi được gọi là các hàm tạo ảo (§20.3.6).

Bây giờ chúng tôi đại diện cho mỗi hệ thống giao diện người dùng bằng một lớp bắt nguồn từ Ival\_maker:

**class BB\_maker : public Ival\_maker { // make BB versions(tao phiên bản BB)**

**public:**

**Ival\_dial∗ dial(int, int) override;**

**Popup\_ival\_slider∗ popup\_slider(int, int) override;**

**// ...};**

**class LS\_maker : public Ival\_maker { // make LS versions**

**public:**

**Ival\_dial∗ dial(int, int) override;**

**Popup\_ival\_slider∗ popup\_slider(int, int) override;**

**// ...**

**};**

Mỗi chức năng tạo ra một đối tượng của giao diện và kiểu triển khai mong muốn. Ví dụ:

**Ival\_dial∗ BB\_maker::dial(int a, int b)**

**{**

**return new BB\_ival\_dial(a,b);**

**}**

**Ival\_dial∗ LS\_maker::dial(int a, int b)**

**{**

**return new LS\_ival\_dial(a,b);**

**}**

Với Ival\_maker, người dùng hiện có thể tạo các đối tượng mà không cần phải biết chính xác hệ thống giao diện người dùng nào được sử dụng. Ví dụ:

**void user(Ival\_maker& im){**

**unique\_ptr<Ival\_box> pb {im.dial(0,99)}; // tạo quay số thích hợp**

**// ...**

**}**

**BB\_maker BB\_impl; // for BB users**

**LS\_maker LS\_impl; // for LS users**

**void driver()**

**{**

**user(BB\_impl); // use BB**

**user(LS\_impl); // use LS**

**}**

Việc truyền các đối số cho '' virtual constructors '' như vậy là một chút khó khăn. Đặc biệt, chúng ta không thể ghi đè các hàm của lớp cơ sở đại diện cho giao diện với các đối số khác nhau trong các lớp dẫn xuất khác nhau. Điều này ngụ ý rằng cần phải có một chút tầm nhìn xa để thiết kế giao diện của lớp nhà máy.

21.3 Thừa kế nhiều

Như được mô tả trong §20.1, kế thừa nhằm mục đích cung cấp một trong hai lợi ích:

• Giao diện dùng chung: dẫn đến ít sao chép mã hơn bằng cách sử dụng các lớp và làm cho mã đó đồng nhất hơn. Điều này thường được gọi là đa hình thời gian chạy hoặc kế thừa giao diện.

• Thực hiện chia sẻ: dẫn đến ít mã hơn và mã thực hiện đồng nhất hơn. Điều này thường được gọi là kế thừa thực thi.

Một lớp có thể kết hợp các khía cạnh của hai kiểu này. Ở đây, chúng ta khám phá cách sử dụng chung hơn của nhiều lớp cơ sở và xem xét các vấn đề kỹ thuật hơn liên quan đến việc kết hợp và truy cập các tính năng từ nhiều lớp cơ sở.

21.3.1 Nhiều giao diện

Một lớp trừu tượng (ví dụ: Ival\_box; §21.2.2) là cách rõ ràng để biểu diễn một giao diện. Đối với một lớp trừu tượng không có trạng thái có thể thay đổi, thực sự có rất ít sự khác biệt giữa cách sử dụng đơn lẻ và sử dụng nhiều lần

của một lớp cơ sở trong một hệ thống phân cấp lớp. Việc giải quyết các điểm mơ hồ tiềm ẩn được thảo luận trong §21.3.3, §21.3.4 và §21.3.5. Trên thực tế, bất kỳ lớp nào không có trạng thái có thể thay đổi đều có thể được sử dụng làm giao diện trong đa

mạng tinh thể thừa kế ple mà không có biến chứng và chi phí đáng kể. Quan sát chính là một lớp không có trạng thái có thể thay đổi có thể được sao chép nếu cần thiết hoặc được chia sẻ nếu điều đó được mong muốn. Việc sử dụng nhiều lớp trừu tượng làm giao diện gần như phổ biến trong các thiết kế hướng đối tượng (trong bất kỳ ngôn ngữ nào có khái niệm về giao diện).

21.3.2 Nhiều lớp triển khai

Hãy xem xét 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. Ví dụ về vệ tinh sẽ là đá, mảnh vỡ từ các phương tiện vũ trụ cũ, vệ tinh liên lạc, và Trạm vũ trụ quốc tế. Các loại vệ tinh này sẽ là đối tượng của các lớp bắt nguồn từ Vệ tinh. Các lớp dẫn xuất như vậy sẽ thêm các thành viên và chức năng dữ liệu và sẽ ghi đè một số chức năng ảo của Vệ tinh thành

điều chỉnh ý nghĩa của chúng cho phù hợp. Bây giờ giả sử rằng tôi muốn hiển thị kết quả của những mô phỏng này bằng đồ thị và tôi đã có sẵn một hệ thống đồ họa sử dụng chiến lược (không hiếm gặp) để dẫn xuất các đối tượng được phát từ một lớp cơ sở chung chứa thông tin đồ họa . Lớp đồ họa này sẽ cung cấp các thao tác để đặt vị trí trên màn hình, chia tỷ lệ, v.v. Để có tính tổng quát, đơn giản và để ẩn các chi tiết của hệ thống đồ họa thực tế, tôi sẽ đề cập đến lớp cung cấp đầu ra đồ họa (hoặc thực tế là phi đồ họa) Trưng bày. Bây giờ chúng ta có thể định nghĩa một lớp vệ tinh truyền thông mô phỏng, lớp Comm\_sat:

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

**public:**

**// ...**

**};**

hoặc bằng đồ thị:

Satellite Displayed

Comm\_sat

Ngoài bất kỳ hoạt động nào được định nghĩa cụ thể cho Comm\_sat, sự kết hợp của các hoạt động trên Vệ tinh và Hiển thị có thể được áp dụng. Ví dụ:

**void f(Comm\_sat& s)**

**{**

**s.draw(); // Displayed::draw()**

**Pos p = s.center(); // Satellite::center()**

**s.transmit(); // Comm\_sat::transmit()**

**}**

Tương tự, một Comm\_sat có thể được chuyển cho một hàm mong đợi một Vệ tinh và một hàm mong đợi Hiển thị. Ví dụ:

**void highlight(Displayed∗);**

**Pos center\_of\_gravity(const Satellite∗);**

**void g(Comm\_sat∗ p)**

**{**

**highlight(p); // chuyển một con trỏ đến par t được hiển thị củaComm\_sat**

**Pos x = center\_of\_gravity(p); // chuyển một con trỏ đến phần Satellite the 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; // overr ide Displayed::draw()**

**// ...**

**};**

Điều này đảm bảo rằng Comm\_sat :: center () và Displayed :: draw () sẽ được gọi cho một Comm\_sat được coi là Comm\_sat và Displayed, tương ứng. Tại sao tôi không giữ các phần Vệ tinh và Hiển thị của Comm\_sat hoàn toàn tách biệt? Tôi có thể đã xác định Comm\_sat để có một thành viên Vệ tinh và một thành viên Hiển thị. Ngoài ra, tôi

có thể đã định nghĩa Comm\_sat để có một thành viên Satellite ∗ và một thành viên Displayed ∗ và để hàm tạo của nó thiết lập các kết nối thích hợp. Đối với nhiều vấn đề thiết kế, tôi chỉ làm như vậy. Tuy nhiên, hệ thống lấy cảm hứng từ ví dụ này được xây dựng dựa trên ý tưởng về một lớp Vệ tinh với các chức năng ảo và một lớp Hiển thị (được thiết kế riêng) với các chức năng ảo. Bạn đã cung cấp các vệ tinh của riêng mình và các đối tượng được hiển thị của riêng bạn thông qua tính năng dẫn xuất. Đặc biệt, bạn phải ghi đè các chức năng thành viên ảo Satellite và các chức năng thành viên ảo Hiển thị để chỉ định hành vi của các đối tượng của riêng bạn. Đó là tình huống khó tránh khỏi tình trạng đa kế thừa các lớp cơ sở với trạng thái và sự thực thi. Các giải pháp thay thế có thể khó khăn và khó duy trì. Về cơ bản, nó ngăn lập trình viên viết rất nhiều hàm chuyển tiếp (để bù đắp cho thực tế là chúng ta chỉ có thể ghi đè các hàm được định nghĩa trong base). Kỹ thuật này không ảnh hưởng đáng kể đến thiết kế tổng thể của một chương trình và đôi khi có thể mâu thuẫn với mong muốn giữ các chi tiết triển khai được ẩn. Tuy nhiên, một kỹ thuật không cần phải thông minh để trở nên hữu ích. Tôi thường thích có một hệ thống phân cấp triển khai duy nhất và (nếu cần) một số lớp trừu tượng cung cấp giao diện. Điều này thường linh hoạt hơn và dẫn đến các hệ thống dễ phát triển hơn. Tuy nhiên, không phải lúc nào bạn cũng hiểu được điều đó - đặc biệt nếu bạn cần sử dụng các lớp hiện có mà bạn không muốn sửa đổi (ví dụ: vì chúng là một phần của thư viện của người khác). Lưu ý rằng với kế thừa đơn (chỉ), lập trình viên các lựa chọn để triển khai các lớp Hiển thị, Vệ tinh và Comm\_sat sẽ bị hạn chế. Comm\_sat có thể là Vệ tinh hoặc Được hiển thị, nhưng không phải là cả hai (trừ khi Vệ tinh được bắt nguồn từ Hiển thị hoặc ngược lại). Một trong hai phương án thay thế sẽ làm mất tính linh hoạt.

Tại sao mọi người lại muốn có một lớp Comm\_sat? Trái với phỏng đoán của một số người, ví dụ về Satellite là có thật. Thực sự đã có - và có thể vẫn còn - một chương trình được xây dựng dọc theo các dòng được sử dụng để mô tả sự kế thừa đa phương thức triển khai ở đây. Nó được sử dụng để nghiên cứu thiết kế các hệ thống thông tin liên lạc liên quan đến vệ tinh, trạm mặt đất, vv Thực tế, Satellite được hình thành từ một khái niệm ban đầu về một nhiệm vụ đồng thời. Với mô phỏng như vậy, chúng tôi có thể trả lời các câu hỏi về luồng lưu lượng thông tin liên lạc, xác định phản ứng thích hợp đối với trạm mặt đất đang bị mưa bão chặn, xem xét sự cân bằng giữa các kết nối vệ tinh và kết nối liên kết với Trái đất, v.v.

**21.3.3 Độ phân giải mơ hồ**

Các lớp cơ sở 2 có thể có các hàm thành viên trùng tên. Ví dụ:

**class Satellite { // vũ trụ**

**public:**

**virtual Debug\_info get\_debug();**

**// ...**

**};**

**class Displayed {**

**public:**

**virtual Debug\_info get\_debug();**

**// ...**

**};**

Khi sử dụng Comm\_sat, các chức năng này phải được phân loại. Điều này có thể được thực hiện đơn giản bằng cách đặt tên thành viên đủ điều kiện theo tên lớp của nó:

**void f (Comm\_sat & cs){**

**Debug\_info di = cs.g et\_debug (); // error: mơ hồ**

**di = cs.Satellite :: get\_debug (); // OK**

**di = cs.Displayed :: g et\_debug (); // OK**

**}**

Tuy nhiên, việc phân định rõ ràng là lộn xộn, vì vậy tốt nhất là giải quyết các vấn đề như vậy bằng cách định nghĩa một hàm mới trong lớp dẫn xuất:

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

**public:**

**Debug\_info get\_debug() // overr ide Comm\_sat::get\_debug() and Displayed::get\_debug()**

**{**

**Debug\_info di1 = Satellite::get\_debug();**

**Debug\_info di2 = Displayed::g et\_debug();**

**return merge\_info(di1,di2);**

**}**

**// ...**

**};**

Một hàm được khai báo trong lớp dẫn xuất sẽ ghi đè tất cả các hàm có cùng tên và kiểu trong các lớp cơ sở của nó. Thông thường, đó chính xác là điều phải làm vì nói chung là một ý tưởng tồi nếu sử dụng cùng một tên cho các phép toán có ngữ nghĩa khác nhau trong một lớp duy nhất. Lý tưởng cho ảo là một cuộc gọi có cùng tác dụng độc lập với giao diện nào được sử dụng để tìm hàm (§20.3.2).

Trong việc thực hiện một hàm ghi đè, thường cần phải xác định rõ ràng tên để có được phiên bản phù hợp từ một lớp cơ sở. Một tên đủ điều kiện, chẳng hạn như Telstar :: draw, có thể đề cập đến một trận hòa được khai báo trong Telstar hoặc trong một trong các lớp cơ sở của nó. Ví dụ:

**class Telstar : public Comm\_sat {**

**public:**

**void draw()**

**{**

**Comm\_sat::draw(); // tìm thấy Displayed::draw**

**// ... nội dung riêng ..**

**}**

**// ...**

**};**

Hoặc đồ thị:

**Satellite Displayed**

**Comm\_sat**

**Telstar**

Nếu Comm\_sat :: draw không giải quyết được lệnh draw được khai báo trong Comm\_sat, trình biên dịch sẽ tìm kiếm một cách đệ quy trong các lớp cơ sở của nó; nghĩa là, nó tìm kiếm Satellite :: draw và Displayed :: draw, và nếu cần thiết sẽ tìm kiếm trong các lớp cơ sở của chúng. Nếu chính xác một kết quả phù hợp được tìm thấy, tên đó sẽ được sử dụng. Nếu không, không tìm thấy Comm\_sat :: draw hoặc không rõ ràng.

Nếu, trong Telstar :: draw (), tôi đã nói đơn giản draw (), thì kết quả sẽ là một lệnh gọi đệ quy ‘‘ vô hạn ’’ của Telstar :: draw (). Tôi có thể đã nói Displayed :: draw (), nhưng bây giờ mã sẽ bị hỏng một cách tinh vi nếu ai đó thêm Comm\_sat :: draw (); nói chung tốt hơn là tham chiếu đến một lớp cơ sở trực tiếp hơn là một lớp cơ sở gián tiếp.

Tôi có thể đã nói Comm\_sat :: Displayed :: draw (), nhưng điều đó sẽ là thừa. Nếu tôi nói Satellite :: draw (), kết quả sẽ là một lỗi vì quá trình rút đã kết thúc trên nhánh Hiển thị của cấu trúc phân cấp lớp.

Ví dụ get\_debug () về cơ bản giả định rằng ít nhất một số phần của Satellite và Displayed đã được thiết kế cùng nhau. Việc vô tình có được một kết hợp chính xác về tên, kiểu trả về, kiểu đối số và ngữ nghĩa là điều cực kỳ khó xảy ra. Nhiều khả năng chức năng tương tự được hỗ trợ theo nhiều cách khác nhau để có thể hợp nhất nó thành một thứ có thể được sử dụng cùng nhau. Ban đầu, chúng tôi có thể đã được giới thiệu với hai lớp SimObj và Widget mà chúng tôi không thể sửa đổi, không cung cấp chính xác những gì chúng tôi cần và nơi họ cung cấp những gì chúng tôi cần, đã làm như vậy thông qua các giao diện không tương thích. Trong trường hợp đó, chúng tôi có thể đã thiết kế Vệ tinh và Hiển thị dưới dạng các lớp giao diện của mình, cung cấp '' lớp ánh xạ '' cho các lớp cấp cao hơn của chúng tôi sử dụng:

**class Satellite : public SimObj {**

// ánh xạ các cơ sở của SimObj đến một thứ dễ sử dụng hơn cho mô phỏng vệ tinh

**public:**

**virtual Debug\_info get\_debug(); // call SimObj::DBinf() and extract infor mation**

**// ...**

**};**

**class Displayed : public Widget {**

// ánh xạ các cơ sở Widget sang thứ gì đó dễ sử dụng hơn để hiển thị kết quả mô phỏng Vệ tinh

**public:**

**virtual Debug\_info get\_debug(); // read Widget data and compose Debug\_info**

**// ...**

**};**

Hoặc đồ thị :

**SimObj Widget**

**Satellite Displayed**

**Comm\_sat**

**Telstar**

Interestingly enough, this is exactly the technique we would use to disambiguate in the unlikely case where two base classes provided operations with exactly the same name, but with different semantics: add an interface layer. Consider the classical (but mostly hypothetical/theoretical) example of a class of a draw() member function in a video game involving cowboys:

**class Window {**

**public:**

**void draw(); //** hình ảnh hiển thị

**// ...**

**};**

**class Cowboy {**

**public:**

**void draw(); //** rút súng từ bao da

**// ...**

**};**

**class Cowboy\_window : public Cowboy, public Window {**

**// ...**

**};**

Làm cách nào để ghi đè Cowboy :: draw () và Window :: draw ()? Hai hàm này có ý nghĩa hoàn toàn khác nhau (ngữ nghĩa) nhưng giống hệt nhau về tên và kiểu; chúng ta cần ghi đè chúng bằng hai hàm riêng biệt. Không có giải pháp ngôn ngữ trực tiếp nào cho vấn đề này (vấn đề kỳ lạ, nhưng việc thêm các lớp trung gian sẽ làm được:

**struct WWindow : Window {**

**using Window::Window; //** kế thừa các hàm tạo

**virtual void win\_draw() = 0; //** buộc lớp dẫn xuất ghi đè

**void draw() override final { win\_draw(); } //** hình ảnh hiển thị

**};**

**struct CCowboy : Cowboy{**

**using Cowboy::Cowboy; //** kế thừa các hàm tạo

**virtual void cow\_draw() = 0; //** buộc lớp dẫn xuất ghi đè

**void draw() override final { cow\_draw(); } //** rút súng từ bao da

**};**

**class Cowboy\_window : public CCowboy, public WWindow {**

**public:**

**void cow\_draw() override;**

**void win\_draw() override;**

**// ...**

**};**

**Hoặc đồ thị:**

Window Cowboy

Wwindow CCowboy

Cowboy\_window

Nếu người thiết kế Window cẩn thận hơn một chút và chỉ định draw () là const, thì toàn bộ vấn đề sẽ tan thành mây khói. Tôi **thấy** điều đó khá điển hình.

**21.3.4 Sử dụng lặp lại một lớp cơ sở**

Khi mỗi lớp chỉ có một lớp cơ sở trực tiếp, cấu trúc phân cấp lớp sẽ là một cây và một lớp chỉ có thể xuất hiện một lần trong cây. Khi một lớp có thể có nhiều lớp cơ sở, một lớp có thể xuất hiện nhiều lần trong cấu trúc phân cấp kết quả. Hãy xem xét một lớp cung cấp phương tiện để lưu trữ trạng thái trong tệp (ví dụ: để ngắt, gỡ lỗi thông tin hoặc tính liên tục) và khôi phục nó sau:

**struct Storable {** // lưu trữ liên tục

**virtual string get\_file() = 0;**

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

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

**virtual ̃Storable() { }**

**};**

Một lớp hữu ích như vậy đương nhiên sẽ được sử dụng ở một số nơi trong hệ thống phân cấp lớp. Ví dụ:

**class Transmitter : public Storable {**

**public:**

**void write() override;**

**// ...**

**};**

**class Receiver : public Storable {**

**public:**

**void write() override;**

**// ...**

**};**

**class Radio : public Transmitter, public Receiver {**

**public:**

**string get\_file() override;**

**void read() override;**

**void write() override;**

**// ...**

**};**

Hoặc đồ thị:

Storable

Receiver Transmitter

Radio

So sánh sơ đồ này với bản vẽ của đối tượng Stocking trong §21.3.4 để thấy sự khác biệt giữa kế thừa thông thường và kế thừa ảo. Trong một đồ thị kế thừa, mọi lớp cơ sở của một tên nhất định được chỉ định là ảo sẽ được biểu diễn bằng một đối tượng duy nhất của lớp đó. Mặt khác, mỗi lớp cơ sở không được chỉ định ảo sẽ có subobject riêng đại diện cho nó. Tại sao ai đó lại muốn sử dụng cơ sở ảo chứa dữ liệu? Tôi có thể nghĩ ra ba cách rõ ràng để hai lớp trong hệ thống phân cấp lớp chia sẻ dữ liệu:

[1] Làm cho dữ liệu trở nên phi địa phương (bên ngoài lớp dưới dạng biến toàn cục hoặc không gian tên).

[2] Đặt dữ liệu trong một lớp cơ sở.

[3] Phân bổ một đối tượng ở đâu đó và cấp cho mỗi lớp một con trỏ.

Tùy chọn [1], dữ liệu phi địa phương, thường là một lựa chọn kém vì chúng tôi không thể kiểm soát mã nào truy cập dữ liệu và cách thức. Nó phá vỡ mọi quan niệm về sự đóng gói và cục bộ.

Tùy chọn [2], đặt dữ liệu trong một lớp cơ sở, thường là đơn giản nhất. Tuy nhiên, đối với thừa kế đơn, giải pháp đó làm cho dữ liệu hữu ích (và các chức năng) '' nổi lên '' thành một lớp cơ sở chung; thường nó '' bub-bles '' đến tận gốc của cây thừa kế. Điều này có nghĩa là mọi thành viên của lớp hier-archy đều có quyền truy cập. Điều đó về mặt logic rất giống với việc sử dụng dữ liệu phi địa phương và gặp phải những vấn đề tương tự. Vì vậy, chúng ta cần một cơ sở chung không phải là gốc của cây - tức là một cơ sở ảo.

Tùy chọn [3], chia sẻ một đối tượng được truy cập thông qua con trỏ, có ý nghĩa. Tuy nhiên, sau đó (các) construc-tor cần dành bộ nhớ cho đối tượng được chia sẻ đó, khởi tạo nó và cung cấp các con trỏ đến đối tượng được chia sẻ cho các đối tượng cần truy cập. Đó đại khái là những gì mà các hàm tạo làm để triển khai một cơ sở ảo.

Nếu bạn không cần chia sẻ, bạn có thể thực hiện mà không cần cơ sở ảo và mã của bạn thường tốt hơn và thường đơn giản hơn cho nó. Tuy nhiên, nếu bạn thực sự cần chia sẻ trong một hệ thống phân cấp lớp chung, về cơ bản bạn có sự lựa chọn giữa việc sử dụng cơ sở ảo và chăm chỉ xây dựng các biến thể ý tưởng của riêng bạn.

Chúng ta có thể biểu diễn một đối tượng của một lớp với cơ sở ảo như sau:

Receiver

Transmitter



Radio



Storable

Các '' con trỏ '' đến đối tượng được chia sẻ đại diện cho cơ sở ảo, Stocking, sẽ là phần bù và thường một trong số đó có thể được tối ưu hóa bằng cách đặt St Lovely ở một vị trí cố định liên quan đến Máy thu hoặc Subobject của Máy phát. Mong đợi tổng chi phí lưu trữ là một từ cho mỗi cơ sở ảo.

21.3.5.1 Xây dựng cơ sở ảo

Sử dụng các đế ảo, bạn có thể tạo các mạng phức tạp. Đương nhiên, chúng tôi muốn giữ các mạng đơn giản, nhưng dù phức tạp đến đâu chúng tôi làm chúng, ngôn ngữ này đảm bảo rằng một phương thức khởi tạo của một cơ sở ảo được gọi chính xác một lần. Hơn nữa, hàm tạo của một cơ sở (dù là ảo hay không) được gọi trước các lớp dẫn xuất của nó. Bất cứ điều gì khác sẽ gây ra hỗn loạn (nghĩa là, một đối tượng có thể được sử dụng trước khi nó được khởi tạo). Để tránh sự hỗn loạn như vậy, phương thức khởi tạo của mọi cơ sở ảo được gọi (ngầm định hoặc rõ ràng) từ phương thức khởi tạo cho đối tượng hoàn chỉnh (phương thức khởi tạo cho lớp dẫn xuất nhất). Đặc biệt, điều này đảm bảo rằng một cơ sở ảo được xây dựng chính xác một lần ngay cả khi nó được đề cập ở nhiều nơi trong hệ thống phân cấp lớp. Ví dụ:

**struct V {**

**V(int i);**

**// ...**

**};**

**struct A {**

**A(); //** nhà xây dựng mặc định

**// ...**

**};**

**struct B : virtual V, vir tual A {**

**B() :V{1} { /\* ... \*/ }; //** nhà xây dựng mặc định ; phải khởi tạo cơ sở V

**// ...**

**};**

**class C : virtual V {**

**public:**

**C(int i) : V{i} { /\* ... \*/ }; //** phải khởi tạo cơ sở V

**// ...**

**};**

**class D : virtual public B, virtual public C {**

**//** ngầm định lấy cơ sở ảo V từ B và C

**//** ngầm nhận cơ sở ảo A từ B

**public:**

**D() { /\* ... \*/ } //** error: không có hàm tạo mặc định cho C hoặc V

**D(int i) :C{i} { /\* ... \*/ }; //** error: không có hàm tạo mặc định cho V

**D(int i, int j) :V{i}, C{j} { /\* ... \*/ } // OK**

**// ...**

**};**

Lưu ý rằng D có thể và phải cung cấp bộ khởi tạo cho V. Việc V không được đề cập rõ ràng như một cơ sở của D là không thích hợp. Kiến thức về một cơ sở ảo và nghĩa vụ khởi tạo nó '' nổi lên '' đối với lớp dẫn xuất nhất. Một cơ sở ảo luôn được coi là một cơ sở trực tiếp của lớp dẫn xuất nhất của nó. Thực tế là cả B và C đều khởi tạo V là không liên quan vì trình biên dịch không biết cái nào trong hai bộ khởi tạo đó thích hơn. Do đó, chỉ bộ khởi tạo được cung cấp bởi lớp dẫn xuất nhất mới được sử dụng.

Hàm tạo cho một cơ sở ảo được gọi trước các hàm tạo cho các lớp dẫn xuất của nó.

Trong thực tế, điều này không hoàn toàn được bản địa hóa như chúng tôi mong muốn. Đặc biệt, nếu chúng ta dẫn xuất một lớp khác, DD, từ D, thì DD phải thực hiện công việc khởi tạo các cơ sở ảo. Trừ khi chúng ta có thể kế thừa các hàm tạo của D (§20.3.5.1) một cách đơn giản, điều đó có thể gây phiền toái. Điều đó sẽ khuyến khích chúng ta không lạm dụng các lớp cơ sở ảo.

Vấn đề logic này với các hàm tạo không tồn tại đối với các hàm hủy. Chúng đơn giản được gọi theo thứ tự xây dựng ngược lại (§20.2.2). Đặc biệt, một hàm hủy cho một cơ sở ảo được gọi chính xác một lần.

**21.3.5.2 Gọi một thành viên lớp học ảo Chỉ một lần**

Khi định nghĩa các hàm cho một lớp có cơ sở ảo, người lập trình nói chung không thể biết liệu cơ sở đó có được chia sẻ với các lớp dẫn xuất khác hay không. Đây có thể là một vấn đề khi triển khai một dịch vụ yêu cầu một hàm lớp cơ sở được gọi chính xác một lần cho mỗi lần gọi hàm dẫn xuất. Khi cần, lập trình viên có thể mô phỏng lược đồ được sử dụng cho các hàm tạo bằng cách gọi một hàm lớp cơ sở ảo chỉ từ lớp dẫn xuất nhất. Ví dụ: giả sử chúng ta có một lớp Window cơ bản biết cách vẽ nội dung của nó:

**class Window {**

**public:**

**//** vật liệu cơ bản

**virtual void draw();**

**};**

Ngoài ra, chúng tôi có nhiều cách khác nhau để trang trí cửa sổ và thêm tiện nghi:

**class Window\_with\_border : public virtual Window {**

//nội dung đồng hồ

**protected:**

**void own\_draw(); //** hiển thị mặt và kim đồng hồ

**public:**

**void draw() override;**

**};**

Hoặc đồ thị:

**Window**

**Window\_with\_border Window\_with\_menu**

**Clock**

Các hàm draw () bây giờ có thể được định nghĩa bằng cách sử dụng các hàm own\_draw (), để một người gọi bất kỳ draw () nào được Window :: draw () gọi chính xác một lần. Điều này được thực hiện độc lập với loại Cửa sổ mà draw () được gọi:

**void Window\_with\_border::draw()**

**{**

**Window::draw();**

**own\_draw(); //** hiển thị đường viền

**}**

**void Window\_with\_menu::draw()**

**{**

**Window::draw();**

**own\_draw(); //** hiển thị menu

**}**

**void Clock::draw()**

**{**

**Window::draw();**

**Window\_with\_border::own\_draw();**

**Window\_with\_menu::own\_draw();**

**own\_draw(); //** hiển thị mặt và kim đồng hồ

**}**

Lưu ý rằng một cuộc gọi đủ điều kiện, chẳng hạn như Window :: draw (), không sử dụng cơ chế gọi ảo. Thay vào đó, nó trực tiếp gọi hàm được đặt tên rõ ràng, do đó tránh được đệ quy vô hạn khó chịu. Truyền từ một lớp cơ sở ảo sang một lớp dẫn xuất được thảo luận trong §22.2.

**21.3.6 Cơ sở sao chép so với Cơ sở ảo**

Việc sử dụng đa kế thừa để cung cấp các triển khai cho các lớp trừu tượng đại diện cho các giao diện thuần túy ảnh hưởng đến cách một chương trình được thiết kế. Lớp BB\_ival\_slider (§21.2.3) là một ví dụ:

**Class BB\_ival\_slider**

**: public Ival\_slider, //** interface

**BBslider được bảo vệ** // triển khai

**{**

// triển khai các chức năng theo yêu cầu của Ival\_slider và BBslider, sử dụng các tiện ích từ BBslider

**};**

Trong ví dụ này, hai lớp cơ sở đóng các vai trò khác nhau về mặt logic. Một cơ sở là lớp trừu tượng công khai cung cấp giao diện và cơ sở còn lại là lớp cụ thể được bảo vệ cung cấp ‘‘details ’’ việc triển khai. Những vai trò này được phản ánh trong cả kiểu của các lớp và trong kiểm soát truy cập (§20.5) được cung cấp. Việc sử dụng đa kế thừa là điều cần thiết ở đây vì lớp dẫn xuất cần ghi đè các chức năng ảo từ cả giao diện và quá trình thực thi. Ví dụ, hãy xem xét lại các lớp Ival\_box từ §21.2.1. Cuối cùng (§21.2.2), tôi đã làm cho tất cả các lớp Ival\_box trở nên trừu tượng để phản ánh vai trò của chúng như là giao diện thuần túy. Làm điều đó cho phép tôi đặt tất cả các chi tiết triển khai trong các lớp triển khai cụ thể. Ngoài ra, tất cả việc chia sẻ chi tiết triển khai đều được thực hiện theo hệ thống phân cấp cổ điển của hệ thống cửa sổ được sử dụng để triển khai. Khi sử dụng một lớp trừu tượng (không có bất kỳ dữ liệu được chia sẻ nào) làm giao diện, chúng ta có một lựa chọn:

• Sao chép lớp giao diện (một đối tượng cho mỗi đề cập trong hệ thống phân cấp lớp).

• Làm cho lớp giao diện ảo để chia sẻ một đối tượng đơn giản giữa tất cả các lớp trong hệ thống phân cấp đề cập đến nó.

Sử dụng Ival\_slider làm cơ sở ảo cung cấp cho chúng tôi:

**class BB\_ival\_slider**

**: public virtual Ival\_slider, protected BBslider { /\* ... \*/ };**

**class Popup\_ival\_slider**

**: public virtual Ival\_slider { /\* ... \*/ };**

**class BB\_popup\_ival\_slider**

**: public virtual Popup\_ival\_slider, protected BB\_ival\_slider { /\* ... \*/ };**

Hoặc đồ thị:

Ival\_slider BBslider

Popup\_ival\_slider BB\_ival\_slider

BB\_popup\_ival\_slider

Có thể dễ dàng hình dung các giao diện khác bắt nguồn từ Popup\_ival\_slider và các lớp triển khai khác bắt nguồn từ các lớp đó và BB\_popup\_ival\_slider.

Tuy nhiên, chúng tôi cũng có giải pháp thay thế này bằng cách sử dụng các đối tượng Ival\_slider được sao chép:

**class BB\_ival\_slider**

**: public Ival\_slider, protected BBslider { /\* ... \*/ };**

**class Popup\_ival\_slider**

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

**class BB\_popup\_ival\_slider**

**: public Popup\_ival\_slider, protected BB\_ival\_slider { /\* ... \*/ };**

Hoặc đồ thị:

Ival\_slider Ival\_lider BBslider

Popup\_ival\_slider BB\_ival\_slider

BB\_popup\_ival\_slider

Đáng ngạc nhiên là không có lợi thế cơ bản về thời gian chạy hoặc không gian đối với thiết kế này so với thiết kế kia. Trong thiết kế Ival\_slider được sao chép, không thể chuyển đổi hoàn toàn BB\_popup\_ival\_slider thành Ival\_slider (vì điều đó sẽ không rõ ràng):

**void f(Ival\_slider∗ p);**

**void g(BB\_popup\_ival\_slider∗ p)**

**{**

**f(p); // error : Popup\_ival\_slider ::Ival\_slider or BB\_ival\_slider ::Ival\_slider?**

**}**

Mặt khác, có thể xây dựng các tình huống hợp lý trong đó việc chia sẻ ngụ ý trong thiết kế cơ sở ảo gây ra sự không rõ ràng cho các phôi từ lớp cơ sở (§22.2). Tuy nhiên, những tính cách xung quanh như vậy rất dễ bị đối phó.

Làm cách nào để chúng ta chọn giữa các lớp cơ sở ảo và các lớp cơ sở được sao chép cho các giao diện của chúng ta? Tất nhiên, chúng ta thường không có lựa chọn vì chúng ta phải tuân theo một thiết kế hiện có. (vì không cần cấu trúc dữ liệu hỗ trợ chia sẻ) và chúng ta thường lấy các đối tượng giao diện của mình từ '' các hàm tạo ảo '' hoặc '' các hàm gốc '' (§21.2.4). Ví dụ:

**Popup\_ival\_slider ∗ popup\_slider\_factory (args){**

**// ...**

**trả về BB\_popup\_ival\_slider mới (args);**

**// ...**

**}**

Không cần chuyển đổi rõ ràng để chuyển từ triển khai (tại đây, BB\_popup\_ival\_slider) sang

giao diện trực tiếp (tại đây, Popup\_ival\_slider).

**21.3.6.1 Ghi đè các hàm cơ sở ảo**

Một lớp dẫn xuất có thể ghi đè một hàm ảo của lớp cơ sở ảo trực tiếp hoặc gián tiếp của nó. Đặc biệt, hai lớp khác nhau có thể ghi đè các chức năng ảo khác nhau từ cơ sở ảo. Theo cách đó, một số lớp dẫn xuất có thể đóng góp triển khai cho giao diện được trình bày bởi một lớp cơ sở ảo. Ví dụ, lớp Window có thể có các hàm set\_color () và prompt (). Trong trường hợp đó, Window\_with\_border có thể ghi đè set\_color () như một phần của việc kiểm soát lược đồ màu và Window\_with\_menu có thể ghi đè prompt () như một phần kiểm soát các tương tác của người dùng:

**class Window {**

**// ...**

**virtual void set\_color(Color) = 0;** // đổi màu nền

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

**};**

**class Window\_with\_border : public virtual Window {**

**// ...**

**void set\_color(Color) override;** // kiểm soát màu nền

**};**

**class Window\_with\_menu : public virtual Window {**

**// ...**

**void prompt() override; //** kiểm soát các tương tác của người dùng

**};**

**class My\_window : public Window\_with\_menu, public Window\_with\_border {**

**// ...**

**};**

Điều gì sẽ xảy ra nếu các lớp dẫn xuất khác nhau ghi đè cùng một chức năng? Điều này được cho phép nếu và chỉ khi một số lớp ghi đè có nguồn gốc từ mọi lớp khác ghi đè hàm. Đó là, một chức năng phải ghi đè lên tất cả các chức năng khác. Ví dụ: My\_window có thể ghi đè prompt () để cải thiện những gì Win- dow\_with\_menu cung cấp:

**class My\_window : public Window\_with\_menu, public Window\_with\_border {**

**// ...**

**void prompt() override; //** không để các tương tác của người dùng đến cơ sở

**};**

**Hoặc đồ thị:**

**Window { set\_color(), prompt() }**

**Window\_with\_border { set\_color() } Window\_with\_menu { prompt() }**

**My\_window { prompt() }**

Nếu hai lớp ghi đè một hàm của lớp cơ sở, nhưng không ghi đè lên lớp kia, thì cấu trúc phân cấp lớp là một lỗi. Lý do là không một hàm duy nhất nào có thể được sử dụng để cung cấp một ý nghĩa nhất quán cho tất cả các lệnh gọi một cách độc lập với lớp mà chúng sử dụng làm giao diện. Hoặc, bằng cách sử dụng thuật ngữ triển khai, không có bảng hàm ảo nào có thể được xây dựng vì lệnh gọi hàm đó trên đối tượng hoàn chỉnh sẽ không rõ ràng. Ví dụ, có Radio trong §21.3.5 không được khai báo write (), khai báo write () trong Máy thu và Máy phát sẽ gây ra lỗi khi xác định Đài phát thanh. Như với Radio, xung đột như vậy được giải quyết bằng cách thêm một hàm ghi đè vào lớp dẫn xuất nhất.

Một lớp cung cấp một số - nhưng không phải tất cả - việc triển khai cho một lớp cơ sở ảo thường được gọi là mixin.

21.4 Lời khuyên

[1] Sử dụng unique\_ptr hoặc shared\_ptr để tránh quên xóa các đối tượng được tạo bằng new; §21.2.1.

[2] Tránh các thành viên ngày tháng trong các lớp cơ sở dùng làm giao diện; §21.2.1.1.

[3] Sử dụng các lớp trừu tượng để thể hiện các giao diện; §21.2.2.

[4] Cung cấp cho một lớp trừu tượng một hàm hủy ảo để đảm bảo việc dọn dẹp thích hợp; §21.2.2.

[5] Sử dụng ghi đè để làm cho việc ghi đè trở nên rõ ràng trong các cấu trúc phân cấp lớp lớn;§21.2.2.

[6] Sử dụng các lớp trừu tượng để hỗ trợ kế thừa giao diện; §21.2.2.

[7] Sử dụng các lớp cơ sở với các thành viên dữ liệu để hỗ trợ kế thừa triển khai; §21.2.2.

[8] Sử dụng đa kế thừa thông thường để thể hiện sự kết hợp của các tính năng; §21.3.

[9] Sử dụng đa kế thừa để tách việc triển khai khỏi giao diện; §21.3.

[10] Sử dụng cơ sở ảo để biểu diễn một cái gì đó chung cho một số, nhưng không phải tất cả, các lớp trong phân cấp; §21.3.5.